The Ruby Book!

Step by Logical Step

By Nicholas Johnson

Document Version: 0.9.0 - Beta

Last Updated: 2017

Why you should care about Ruby

Hello and Welcome to this super exciting little introduction to Ruby, the language that aims to make developers fall in love with programming again.

Ruby is an expressive, open language that lets you do an awful lot in next to no time at all.

It's the language that powers a good fraction of the world's tech startups. Why? Because it lets small teams of crack developers build incredible applications that would have taken months, or even years using traditional techniques. It's a language that enables MVPs and lightening iteration.

- **On the web**, frameworks like Rails, Sinatra and many more allow you to create web applications in stupidly short timespans.
- **In the mobile space**, Ruby Motion lets you compose mobile apps using native components.
- **On the Desktop**, Ruby allows you to create desktop apps that leverage your operating system's native UI.
- **At Enterprise scale**, JRuby runs on the Java Virtual Machine, giving it ready access to other JVM based languages such as Java and Scala.

And what's in it for you? Ruby is a friendly community. People want to help you, people want to pay it forwards. And once you're good, you can earn very significant money. A good Ruby developer in London will earn around £500 per day. This is a language which is worth your time.

Wow your clients with your awesome productivity. Tackle side projects with ambition and alacrity.

Feeling excited yet?

Here are some of the great things about Ruby:

- Very high level - a little bit goes a long long way.
- Super light, clean, flexible, some would say poetic syntax.
- Genuinely fully object oriented. Integers are objects. Even methods are objects.
- Introspection is built in, not tacked on. Powerful metaprogramming lets you write code to write code.
- Operators overloading is super simple. Define your own types and do maths with them.
- Not written a method yet? With MethodMissing, Ruby can write it for you, while your code is running.
- Interpreted, so no build, compile, debug cycle to monkey up your flow, just write and refresh.
- Awesome frameworks: Rails, Sinatra and many more.
- Massive library of Gems to call on. You rarely have to reinvent the wheel.
- Incredibly friendly community.
- Super charming.

…And more nice little touches than you can shake a stick at.

Now are you excited? You're about to learn a language, designed specifically to make programmers happy, that doesn't make you jump through hoops, that gives you the respect you deserve, and which, if you master it, can make you rich. Welcome to Ruby!

Welcome to Ruby!

To get started you're going to need some basics. I'm going to assume here that you have a copy of Ruby, a terminal, and a text editor of some description. You'll need access to irb, the interactive ruby interpreter.

The following things need to be true:

1. You can get up a command line or terminal of some description, ideally bash or similar.
2. If you type ruby at the command line, this needs to work and not throw an error
3. If you type irb at a command line, this too needs to work (type exit to get back out again)

If these things are not all true for you, please see me.

Hello IRB

IRB is the interactive ruby interpreter. If you have access to a command line and a copy of Ruby, you will have IRB. It's a testbed. It lets you try ideas out quickly. Here we're going to use it to write our first Ruby program.

At a command line, type

```
irb;
```

to enter interactive Ruby. Got it up? Good. Now type:

```
  puts "Hello Ruby you charming little thing"
```

Now press enter. See what you did there? Puts stands for put string. It just appends a string to the current output stream, which here is the terminal. Notice as well what you didn't do. You typed no semi-colons and no braces.

Now lets go a step further

In IRB, type:

```
  puts "Hello Ruby " * 5
```

See what happened? Multiplication works on strings. This might seem odd at first, but it's actually consistent with the way that Ruby treats operators as methods. We'll come to this later.

IRB is tremendously useful for trying things out. Because Ruby is so expressive, you can do a huge amount just on one line. Ruby loves it when you write clean, concise, and above all, expressive code, and IRB can help you do this. You can tackle many of the exercises in this book using IRB.

IRB Exercise

Throughout this book you’ll find lots fun and exciting exercises. I’d encourage you to try them out, but be pragmatic. If something is really simple and obvious, you can skip it, I won’t mind. Here’s a simple one for starters:

Write a line of code in IRB which prints our “Hello Ruby”. Use puts to output a string.

You can create a simple loop using the times method:

```
10.times { puts 'cheese and crackers' }
```

Make irb print Hello World 50 times, 1000 times, 100000 times.

Editor Exercise

In this exercise we’re going to create a simple ruby app and run it from the command line. The goal is just to make sure that our environment works and we can use it.

1. Fire up your editor (Sublime, Brackets, Vim, Notepad, etc) and create a file called hello_world.rb
2. Create a hello world app in the editor and call it from the command line.

Further Editor Exercise

In this exercise we’ll create a simple program that reads from the command line.

1. Get a value from the command line using a = gets
2. Write a little program that asks the user if they like cheese.
3. If they reply yes, hand them some gouda. If they reply no, give them a stern telling off.

An if statement in Ruby looks like this:

```
if a
  puts('a was true')
else
  puts('a was false')
end
```

Variables and Constants

Let’s talk about Variables. I’ll assume here that you know what a variable is. Compared to some languages, Ruby is very free and easy about variables, it cuts you a lot of slack and assumes for the most part that you know what you are doing.

Variables are duck typed

Variables in Ruby are **duck typed**. If the contents of a variable can quack like a duck, it will be allowed to swim on the pond. The interpreter will not do any type checking for you in advance. This might upset people coming from a .Net, C++ or Java background, and it's one of the reasons Ruby is suited to small teams of crack developers, since you can't enforce an interface, and can't easily prevent people from passing silly parameters to your methods.

You just have to trust to your basic intelligence. Scary?

It's also one of the reasons that Ruby is so fabulously productive, since you don't need to interact with the type system. Polymorphism is assumed. You don't need to do any work to enable it. Productivity wise, this is an enormous win, provided you can trust yourself and your co-workers.

Declaring variables

Variables come into existence when they are first declared. There is no need to define them.

For example:

```
hi = "Hello Ruby"
some_big_number = 1000000
```

This is pretty sensible. A variable can hold anything you like, and the same variable can be re-purposed to hold something else entirely:

```
a = 10
a = "red"
```

A little best practice: naming conventions

There are naming conventions governing variable names in Ruby. These conventions are not enforced, but you should stick by them if you want people to like you, since Ruby is case sensitive.

Variable names

Variable names always start with a lower case letter. By convention they are all lower case with optional underscores (snake case) eg:

```
number_of_people
user_name
height_of_the_eiffel_tower
```

Constants

Constants start with an upper case letter and by convention are CAPITALISED_SNAKE_CASE:

```
MAX_NUMBER_OF_PEOPLE = 20
NUMBER_OF_DAYS_IN_A_LEAP_YEAR = 364
```

Note that constants are not actually constant, you can redefine them if you really, really need to. You'll get a warning, but it won't break. Ruby is like this, it assumes you're clever. Yes, Ruby is a language that gives you the respect you DESERVE.

Clever Tricks

There are lots of little tricks you can do with variables that are useful, and can help a lot when trying to appear clever.

Assignment chaining

Assignments can be chained saving typing eg:

```
x = y = z = 4
puts x + y + z
  => 12
```

This works because the output of the assignment is the value that is being defined, so the output of z = 4 is 4.

Parallel assignment

Ruby also supports parallel assignment allowing you to assign multiple different variables on one line, eg:

```
a,b = 5,6

# a => 5
# b => 6
```

You can exploit this to swap the values of two variables in one line:

```
a,b = b,a

# a => 6
# b => 5
```

Integers (Fixnums)

Everything in Ruby is an object including integers, or Fixnums as they are known in Ruby. Integers therefore have methods.

```
x = 1
x.to_s
  => '1'
```

Basic Maths

You can do all the maths you would like with integers. It all works as you would expect:

```
1 + 2
  => 3

5 - 1
  => 4

3 * 4
  => 12

4 / 2
  => 2
```

You also have the common maths shortcuts you find in other languages. Again, these work as you would expect:

```
x = 2
  => 2

x += 5
  => 7

x *= 2
  => 14
```

Use ** for exponentiation should you feel that way inclined:

```
2 ** 2
  => 4
2 ** 3
  => 8
2 ** 4
  => 16
```

Note: There is no Incrementation operator

A surprise here. Ruby has no pre/post increment/decrement operator. For instance, x++ or x– will fail to parse. Use x += 1 instead. There are some relatively technical and quite good reasons for this to do with consistency.

Matz dislikes this because the ++ incrementation is the only operator with an implied second value, so we don't have it. Use a +=1 instead.

Looping with Integers

Remember how everything in Ruby is an object? This lets us do some reasonably clever things. For example, we can it to perform looping, like so:

```
5.times {puts "hello"}

hello
hello
hello
hello
hello
```

This might look strange. The little bit of code between the curly brackets {puts “hello”} is called a block. A block is an inline function. We are actually passing this block (or function) to the Fixnum times method, which is then executing it 5 times. This may seem a little odd at first, but it's actually really rather good and will soon become second nature, as we shall see when we reach the section on blocks.

Comparison

As you would expect, greater than or less than signs to do comparison. == tests for equality.

```
1 > 2
  => false

2 >= 2
  => true

1 < 2
  => true

1 == 1
  => true
```

The Spaceship Operator

A tremendously useful one this, the spaceship operator returns 1, 0 or-1 depending on whether the first operand is larger, the same as or smaller than the second. This is excellent, as this type of comparison forms the basis of all known sorting algorithms.

To make any class sortable, all you need to do is to implement the spaceship operator on it. Telling a custom class how to respond to an operator is called operator overloading. More on operator overloading later.

4 <=> 3 => 1

4 <=> 4 => 0

2 <=> 10 => -1

Large integers

Underscores can be included in integers to make them more legible. We don’t do this often, but it’s a fun trick to know:

```
x = 100_000_000
y= 100000000
puts x == y
  => true
```

Floats

Ruby also has floating point numbers. Floats are useful for dealing with very large numbers, or high precision numbers.

Declare a float just by including a decimal point, like so:

```
a = 0.5
  => 0.5

a = 1.0
  => 1.0
```

You can convert integers to floats using the to_f method like so:

```
15.to_f
  => 15.0
```

Integers are not implicitly converted to floats, so:

```
3 / 2
  => 1
```

rather than 1.5

However, if one of the operands is already a float, the output will be a float so:

```
3.0 / 2
  => 1.5

3 / 2.0
  => 1.5
```

Infinity

Floats have the rather handy ability of extending to infinity, like so:

```
1.0 / 0
  => Infinity

(1.0 / 0).infinite?
  => 1

(-1.0 / 0).infinite?
  => -1
```

If you need to say that something, say a stack, can contain an unlimited number of values, you might find a use for infinity.

Exercise - do a little maths

1. Get a value from the command line using a = gets
2. Write a little program that asks the user how old they are in years. You will need to use years.to_i
3. Output the number of weeks they have lived for.
4. Assume a lifespan of 79 years. Output the number of weeks they have left.

Exercise - repeat yourself

Write a one line program that writes "Hello World" 250 times to the screen

Exercise - exponents

Use times to write a program which outputs powers of 2, up to some maximum value, like this:

```
1
2
4
8
16
32
...
```

Strings

Ruby has an unusually rich and detailed String manipulation toolkit. String creation is similar to other languages

```
string_1 = "Hello Ruby"
string_2 = 'Hello Everyone'
puts string_1
  => Hello Ruby
puts string_2
  => Hello Everyone
```

You are free to use single or double quotes around your string.

Strings are Objects

Strings are real objects and have methods:

```
s = "Hello Ruby"
puts s.reverse
  => "ybuR olleH"
```

String Literals are allowed

As you might expect, you can create strings, and call methods on them directly.

```
"Hello Ruby".upcase
  => "HELLO RUBY"
```

Embedding Variables right in Strings

Use double quotes and the #{} syntax if you want Ruby to look for variables embedded in a string:

```
name = "Derrick"
puts "Hello, my name is \#{name}"
  => "Hello, my name us Derrick"
```

Nice, simple, inline, readable.

Escaping with \

You can include escape characters in your string with a backslash:

```
"Ruby\'s great! \\n Oh yes it is!"
  => "Ruby's great"
  => "Oh yes it is!"
```

‘Single’ vs “double” quoted strings

A question I get asked all the time is whether we should prefer single or double quotes.

Double quotes will hunt for variables and escape characters, so are technically slightly slower.

For this reason you should use single quotes, except where you need an embedded variable or escape character.

String concatenation

Add two strings together using simple arithmetic:

```
"Hello " + "Ruby"
  => "Hello Ruby"
"Hello " << "Everybody"
  => "Hello Everybody"
"Hello " * 5
  => "Hello Hello Hello Hello Hello"
```

More likely you will want to do a join on an array, something more like this:

```
["Hello", "Ruby"].join(" ");
```

Conversion to other types

Strings can be converted to lots of other types using the casting methods. There are lots of these built in, but feel free to write your own as well. The “to_i” method converts to an integer.

```
"5".to_i
  => 5
"99 Red Balloons".to_i
  => 99
```

Conversion from other types

Strings can be created from any other type using the to_s method. For example, if you have an integer, and you need it to be a string, do it like this:

```
5.to_s
  => "5"
```

-# ## Exercise - The String API

-# The String API is rich and deep

Exercise - Working with Strings

Review the above and attempt the following

1. Create a string and assign it to a variable.
2. Reverse the string using the reverse method.
3. Permanently reverse the string using the reverse! method. Methods that end in an ! are destructive, they affect the original variable.
4. Use gets to populate a variable which contains your name. Now use the #{} syntax to create a greeting string that incorporates this variable.

Exercise GSub

The gsub String method gives us is global substitution. Read about it here:

http://ruby-doc.org/core-2.1.4/String.html#method-i-gsub

I have an issue where I commonly type a instead of e. It's a terrible problem for me, but you can help. Write code to replace all instances of the letter 'a' in a string with the letter 'e'.

Exercise - The String API

Check out the string API here:

http://ruby-doc.org/core-2.2.2/String.html

In my application I have a string like this:

```
email = "   dave@davely.com ";
```

I need rid of that whitespace. Please clean it up for me.

Now I have a string like this:

```
first_name = "  dave";
```

I need it to be "Dave"

Please tidy it so we can save it to the database.

Exercise - Casting

1. Write code which receives a number and a name from the command line, say "Hal" and "123".
2. Now output "I'm sorry Dave, I can't do that" 123 times

Exercise - strftime

The Time class will allow you to easily format a Time object as a String using strftime.

Get the current time using Time.now

Now review the strftime api here http://ruby-doc.org/core-2.2.2/Time.html#method-i-strftime

Now output the time in this format:

“20 Jan 1946 at 12:45”

Functions

Functions are declared using the def keyword:

```
def greeting
  puts "Hello Ruby"
end

greeting()
  => Hello Ruby
```

Accepting parameters

Functions can accept parameters as you would expect. We pass them like this:

```
def greet(name)
  puts "hello #\{name}"
end

greet("dave")
  => "hello dave"
```

Optional braces

When calling a function, the braces are optional.

```
greet "dave"
  => "hello dave"
```

This is a really nice syntax, and comes into it’s own when we start writing methods.

Returning a Value

Functions can return a value. We pass back a value using the return statement, like so:

```
def say_hello_to(name)
  return "hello #\{name}"
end

puts say_hello_to "dave"
  => "hello dave"
```

get_greeting_for “dave” **evaluates** to the string “hello dave”. This string is received by puts, which then outputs it to the screen.

Optional return statements

The return statement is also optional. If it’s omitted the function will return the last evaluated expression, so:

```
def get_greeting_for(name)
  "hello #\{name}"
end

puts get_greeting_for "dave"
  => "hello dave"
```

This is a clean and useful syntax for short methods such as getters.

Default Values

We can set the default value of an argument, so if no value is passed, our function will still work:

```
def get_greeting_for(name="anonymous")
  return "hello #\{name}"
end

puts get_greeting_for
  => "hello anonymous"
```

Note that if we have several arguments, and some are missing, they will be filled in from left to right, so the last ones will take their default values.

Upshot

- Functions in Ruby are created using the def keyword (short for define).
- Functions that exist in an object are typically called methods.
- Functions and methods are the same, except one belongs to an object.
- Objects are created from classes using the .new method

Exercise - Meet and greet

Write a simple function that greets a person by name. It should receive a name and return a string.

If it is called without parameters it should say “Hello anonymous”

Exercise - A simple function

Write a function which receives a value and outputs a string containing all the numbers up to and including that value.

Integrate this into a command line app.

Flow Control

Ruby loves it when you tell it what to do.

if/elsif/else/end

If statements are present. They work as you’d expect. Notice that no braces are required.

```
bacon = true
fish = false
if fish
  puts 'I like fish'
elsif bacon
  puts 'I like bacon'
else
  puts "I don't like fish or bacon"
end

  => 'I like bacon'
```

unless

The unless keyword is the opposite of the if keyword. It’s equivalent to !if (not if). It can make your code more readable.

```
bacon = false
unless bacon
  puts 'fish'
else   puts 'bacon'
end

  => "fish"
```

Statement Modifiers

Ruby loves it when you write things concisely. The if and unless keywords can also be placed after the line of code you may or may not want to execute. When used in this way they are called statement modifiers:

```
user = "dave"

puts "Hello \#{user}" if user
puts 'please log in' unless user
```

The Ternary operator

Like most languages Ruby includes a ternary operator. It works as you'd expect. If the first portion evaluates to true the first result is given, otherwise the second result is given:

```
bacon = true
puts bacon ? 'bacon' : 'fish'
  => "bacon"
```

This is equivalent to:

```
bacon = true
if bacon
  puts 'bacon'
else
  puts 'fish'
end

  => "bacon"
```

Booleans, Logic, nil and undefined

Booleans in Ruby are logical and clear

What counts as false?

Only Nil and False are false in Ruby. If it exists it's true. That includes zeros, empty strings, etc. This is because 0 is an object in Ruby, as is the empty string ""

We can test this with a short function, that determines if the parameter evaluates to true or false, like so:

```
def true?(value)
  if (value)
    true
  else
    false
  end
```

```
end

true?(false)     # => false
true?(nil)        # => false
true?(0)           # => true
true?("")          # => true
true?(true)       # => true
true?(15)          # => true
true?([0,1,2])   # => true
true?('a'..'z') # => true
true?("pears")   # => true
true?(:bananas) # => true
```

A variable is nil if the variable has been declared but doesn't point to anything (remember Ruby is fully object oriented so all variables are pointers to objects)

Nil

A variable is nil if it has been declared, but holds no value. Nil exists as a type, and has methods. For example:

```
a = nil
a.to_s
  => ""
a.nil?
  => true
```

Nil is a very useful thing to be able to return. It means "no value".

Undefined

A variable is undefined if it has not been declared. You can test for this using the defined? operator.

```
a = 1
defined? a
  => "local-variable"
defined? b
  => nil
```

Boolean Algebra

You can do all the standard things using Boolean algebra.

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

```
12 == 12
  => true
```

are all supported.

Special uses for logical OR ||

There are useful things that can be done with the OR || command. The second part is only evaluated if the first part returns false (nil or false evaluate to false), and the return value is the last value calculated. Rails exploits this letting you do neat things like this:

```
name = nil;
user_name = name || "Anonymous Coward";
```

Here we have a default value. If name is nil, anonymous coward will be used instead.

Exercise - Student Simulator

A simple one to start with. I have a boolean variable called hungry:

```
hungry = true;
```

1. If hungry is true output “Make toast”.
2. If it is not, output the word: “Go to sleep”.

Exercise - Cat name picker

I want to choose a name for my cat, but for personal and ideological reasons I am only interested in cat names which start with the letter R.

1. Make a function that receives a cat name.
2. If a cat starts with the letter “R”, return the name, else return nil. (Remember you do not need to explicitly return nil, just don’t return.)

You can get the first letter of a string using the square bracket syntax:

```
"hello"[0]
# => "h"
```

Optional extension

Employ a while loop to loop over the function until an acceptable name is suggested. A while loop in ruby looks like this:

```
x = 0
while x < 5
  x += 1
  puts x
end
```

Further extension

Catch the case where I accidentally type “Ruby the Cat” instead of “ruby the Cat”. The easiest way to do this is with a downcase.

http://ruby-doc.org/core-2.1.0/String.html#method-i-downcase

Case Statements

Of course Ruby also has case statements. You almost never see these in the wild as case statements are a bit, well, 1990s, but if you should decide you need one, here’s how they work.

```
refreshment_type_required = "biscuit"
suggest_you_eat = case refreshment_type_required
  when "pastry" : "cinnamon danish whirl"
  when "hot drink" : "mocha with sprinkles"
  when "biscuit" : "packet of bourbon creams"
  else "glass of water"
end
```

A case statement will break automatically if you hit a matching term, you don’t need to tell it to break as with some other languages.

Blocks Rock

A Block is an unnamed function, a lambda, which is received by a function, and which can tell it what to do.

Blocks are where the fun really starts. A block in Ruby is a sort of unnamed function that can be passed to a method using a special syntax. It can be as long or as complicated as we like, and can span many lines. Here’s a simple example:

```
5.times {puts "Ruby Ruby"}

Ruby Ruby
Ruby Ruby
Ruby Ruby
Ruby Ruby
Ruby Ruby
```

The block here is the bit of code: puts “Ruby Ruby”. This piece of code is passed to the times method. We use blocks for everything in Ruby, most notably when looping, but in plenty of other ways too. Come with me now as we enter the world of loops…

Why is this cool?

This is super great because it means that the number 5 knows in itself how to iterate up to itself. We have perfect encapsulation. We don’t need a for loop. We don’t need to make any assumptions about implementation. It’s all hidden under the surface.

Passing parameters to a block

We can have our function pass parameters to it’s block. We do this using the | | “chute” syntax. For example:

```
5.times{|i| puts i}

0
1
2
3
4
```

Here the times method passes a number to the block, which is available in the variable i. This is one of the many ways in which we do iteration in Ruby.

Blocks are better than loops

One of the most common and programmer friendly applications of blocks is in looping. Many objects, most notably, Arrays and Hashes will accept a block then apply that block to each of their members in turn. All the looping code is encapsulated, meaning we don’t need to worry about the internal structure of the array.

```
people = ["jim","harry","terrence","martha"]
people.each { |person| puts person }

  => "jim"
  "harry"
  "terrence"
  "martha"
```

As we saw earlier, the Fixnum.times method works in a similar way:

```
5.times { puts "Ruby" }

  => Ruby
  Ruby
  Ruby
```

```
    Ruby
    Ruby
```

If we wanted to iterate over some specific numbers, we might use the upto method to create an Enumerable object, and then iterate over that. Observe:

```
5.upto(10) {|i| puts i}

  => 5
  6
  7
  8
  9
  10
```

We could also iterate over a Range object. more on Ranges shortly:

```
(6..8).each {|i| puts i}
  => 6
  7
  8
```

The most common use of a loop, iterating over an array is totally covered, and in fact, when writing Ruby code, we almost never write the sort of looping constructs you might be used to. Blocks have them covered

each_with_index

If for some reason we need t get the index of an array, we can do this too using Array.each_with_index.

This method accepts a block with two parameters, the value and an index. We can use it like so:

```
people = ["jim","harry","terrence","martha"]
people.each_with_index { |person, i|   puts "person: #\{i} is called #\{person}" }
  => person: 0 is called jim
  person: 1 is called harry
  person: 2 is called terrence
  person: 3 is called martha
```

Short and Curlies

We can declare a block in two ways. We can use the curly braces syntax, or the do/end syntax.

The difference between them is that the curly braces syntax can only take a single line of code, whereas the do/end syntax can take as much code as you like, even an entire web page template if needed.

Here is an example of the curly braces syntax:

```
favourable_pets = ["kittens","puppies","hamsters"]

favourable_pets.each_with_index { |pet, i| puts pet; puts i }
  => kittens
  puppies
  hamsters
```

And here is an example of the do/end syntax. Notice the code is more spread out. This is more readable for large blocks of code.

```
favourable_pets.each_with_index do |pet, i|
  puts pet
  puts i
end

  => kittens
  0
  puppies
  1
  hamsters
  2
```

Exercise - Gsub accepts a block

The String.gsub method will find and replace substrings in text. It’s terribly useful, but sometimes we need more, we need to find and manipulate strings in text.

Say you have a string containing URLs, maybe culled from twitter. You could replace all the urls like this:

```
tweet_text = "hello http://www.google.com hi"

tweet_text.gsub(/http:\\/\\/[^ ]*/, "A URL was here.")

  => "hello A URL was here. hi"
```

This is OK, but what if we wanted to replace the URL with a functioning link. The gsub method will optionally accept a block. The block receives a parameter which contains the match. The block must then return a value which is used to replace the match.

1. Create a string containing one or more URLs. Now write code using gsub to hyperlink all the urls.
2. Assume your string is a tweet. Now write code to hyperlink all the hash tags. Hashtags start with a # and end with a space or newline.
3. Finally write code to hyperlink all the user names. User names start with @.

Map - Reversing the letters

We can split a string about an array using the split function, like this:

‘hello world’.split(’ ’) => [‘hello’, ‘world’]

Use string.split and Array#map method to take a sentence and reverse the letters of all the words, like so.

```
"Hello there" # => "olleH ereht"
```

First split, then map to reverse.

For Bonus points, capitalise the first letter to produce:

```
"Olleh ereht"
```

Inject - Adding all the numbers

Inject is a ridiculously handy function that will allow you to inject a block between each element of an array. The block will in turn receive the next element in the array and the output of the previous call to the block.

I have an array of numbers like this:

```
[4,9,6,3]
```

1. Use inject to add up all the numbers in an array.
2. Use inject to multiply all the numbers in the array.

Filing - save to the file system

Filing also uses a block. We call File.open and pass it a block. Within the block we have access to the file object.

```
File.open("path/to/file", 'wb') do |file|
  file.write('hey there!', :ruby)
end
```

Write code which saves a random string to a file.

RSpec

There are many test harnesses for Ruby. RSpec is the most popular. It has inspired many imitators such as Jasmine for JavaScript and PHPSpec for PHP.

Getting RSpec

Install rspec using

```
gem install rspec
```

or if you prefer add it to your Gemfile and bundle.

You now have a new terminal command, rspec. Type:

```
rspec;
```

in a terminal to test your installation.

Writing a spec

RSpec is very easy to setup. Create a folder called spec in the same directory as your code. This is where your specs will live.

In the spec folder create a file called test_spec.rb. Place the following code in it:

```
describe "an example spec" do
  it "passes" do
    expect(true).to be(true)
  end
end
```

run your specs with:

```
rspec spec/*
```

You should see a message like this, telling you the specs have passed:

```
Finished in 0.00105 seconds (files took 0.11919 seconds to load)
1 example, 0 failures
```

Testing some actual code

Let’s write an alarm clock function to wake us up in the morning. When we call it it’s going to give us a message: “Beep beep beep”. This will help us wake up.

```
def wake_me_up
  "Beep beep beep"
end
```

Save this in a file called clock.rb

Now let's write a spec for it. First we need to import our clock, then we call the code, then we say what the result should be:

```
require_relative '../clock'

describe "alarm clock" do
  it "beeps" do
    expect(wake_me_up).to eq('Beep beep beep')
  end
end
```

Run this and verify it works

be vs eq

Note that we have used to eq above, rather than to be. Be tests for the same object. Eq tests for object equality.

When you make two strings, those strings are not the same object, but they do contain the same characters. Be will fail, eq will pass.

Test First

This is all well and good, but what about Test Driven Development (TDD). In this methodology we write the test first, watch it fail, then write code to make it pass.

Some-days I feel sleepy and I'd like to be woken with more force. I'd like my alarm clock to accept an optional parameter and wake me up more strongly.

First we write the spec

```
require_relative '../clock'

describe "alarm clock" do
  it "beeps" do
    expect(wake_me_up).to eq('Beep beep beep')
  end
  it "beeps louder" do
    expect(wake_me_up(6)).to eq('Beep beep beep beep beep beep')
  end
end
```

Now we run the spec and we see red:

```
1) alarm clock beeps
   Failure/Error: expect(wake_me_up(6)).to eq('Beep beep beep beep beep beep')
   ArgumentError:
     wrong number of arguments (1 for 0)
   # ./clock.rb:5:in `wake_me_up'
  # ./spec/clock_spec.rb:16:in `block (2 levels) in <top (required)>'

Finished in 0.00161 seconds (files took 0.14819 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./spec/clock_spec.rb:15 # alarm clock beeps
```

Notice it's telling us exactly what and where the problem is. Let's write code to make our test pass.

```
def wake_me_up(i = 3)
  ("beep " * i).strip.capitalize
end
```

Run the test again and success, we have a passing test.

Many developers say they live for these moments of green. You may or may not love testing as much as this but regardless, a good suite of passing tests can give you piece of mind at the weekend and help you sleep better at night, knowing you have done a good job.

Exercise - Write some tests

Review the RSpec matcher documentation here:

https://www.relishapp.com/rspec/rspec-expectations/docs/built-in-matchers

In the section on blocks you wrote code to reverse the individual words in a string, but not the string itself, such that:

```
"Hello World";
```

becomes:

```
"olleH dlroW";
```

1. Extract this code into a function and write a spec for it.
2. Do the same for the gsub and inject exercises.

Arrays

Like most languages Ruby allows you to create Arrays to hold multiple values. Arrays in Ruby are one dimensional, though you can declare an array of arrays if you feel the need.

Creating Arrays

Create an array using square brackets like so:

```
cakes = ["florentine", "lemon drizzle", "jaffa"]
```

Access an array using square brackets:

```
cakes[0]
  => "florentine"
```

Zero Indexed

Arrays are zero indexed, so 0 is the first element.

Polymorphic

As you might expect from Ruby, arrays can hold any type of element. This is allowed:

```
["hello", 1, 0.5, false]
```

Manipulating arrays

Arrays can be manipulated in a huge variety of ways For example:

Adding

```
[1,2,3] + [4,5,6]
  => [1, 2, 3, 4, 5, 6]
```

Subtracting

```
[1,2,3] - [1,3]
  => [2]
```

Appending

```
[1,2,3] << 4
  => [1, 2, 3, 4]
```

Multiplication

```
[1,2,3] * 2
  => [1, 2, 3, 1, 2, 3]
```

Handy dandy array API

There are also a raft of array methods we can use, such as:

```
[1,2,3].reverse
  => [3, 2, 1]

[1,2,3].include? 2
  => true
```

You might notice here that this method name has a question mark in it? This is a Ruby convention. Methods with a question mark return true or false.

```
[4,9,1].sort
  => [1, 4, 9]
```

Array splitting

Parallel assignment works when pulling values out of a array.

```
array_of_numbers = [1,2,3,4]
a,b = array_of_numbers
a
  => 1
b
  => 2
```

We can also pull the first element, and return the rest of the array should we wish to:

```
arr = [1,2,3]
a,*arr = arr
a
  => 1
arr
  => [2, 3]
```

This is a clever trick to know as it impresses people and make you look brainy.

Creating an array using the to_a method

The to_a method allows many objects to be converted to arrays. For example an array can be created from a range as follows

```
(1..10).to_a
  => [1,2,3,4,5,6,7,8,9,10]
```

Ranges

Ranges are useful objects that can be used to represent a sequence. Ranges are defined using the .. or … syntax. For example:

```
1..10
```

represents the sequence of numbers between 1 and 10.

```
1...10
```

represents a range that excludes the high value. In this case the numbers 1 to 9

Ranges in Memory

Ranges are compact. Every value in the range is not held in memory, so for example the range:

```
1..100000000
```

…takes up the same amount of memory as the range:

```
1..2
```

Conversion to arrays

Ranges can be converted to arrays using to_a like so

```
(1..10).to_a
  => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Note we need to put braces around the range here to disambiguate the dots.

Character Arrays

We can also declare arrays of characters like this:

```
'a'..'f'
```

Testing if a range contains a value.

We can use the equality === operator to test if a range contains a value. For example:

```
('a'..'f') === 'e'
  => true

('a'..'f') === 'z'
  => false
```

Exercise - Iterating over an array

We can pass a block to an array to be executed on each item in that array. For example:

```
['fork','knife','spoon'].each {|table_item| puts table_item}
```

Our code doesn't need to know the internal details of the array, it just passes a block, and lets the array sort itself out.

1. here is an array of strings. Pass a block to Array.each printing them out one by one.

```
['cats','hats','mats','caveats']
```

2. The each_with_index method accepts a block with two parameters, the value, and the index. Use it to print out the strings in the array preceded by their index in the array like this:

```
1. cats
2. hats
3. mats
4. caveats
```

3. Repeat the exercise above, but now only print out every string with an odd number index.
4. Investigate the remove method. Modify your code to only print out only strings which contain the letter 'c'
5. Functions which return nothing are hard to test. Modify your code so that instead of putting to the screen it returns a string. Use RSpec to test it.

Exercise - Join

Use the join method to join the array together with commas, you should get something like this:

```
Fluffy, Hammy, Petunia
```

Array manipulation

Reverse the order of the array, output something like this:

```
Petunia, Hammy, Fluffy
```

Append two arrays to each other

Create an array of dogs. append it to the first. using the plus operator, then output both arrays together, like this:

```
Fluffy
Hammy
Petunia
```

Exercise - Map

Use Map to capitalise each element in the array prior to output, like so:

```
FLUFFY
HAMMY
PETUNIA
```

Now capitalise and reverse each element in the array.

```
PETUNIA
HAMMY
FLUFFY
```

Exercise - Sort and reverse

Use sort to sort the array alphabetically like so:

```
BARKY
FLUFFY
GOMEZ
HAMMY
PETUNIA
WOOFY
```

And sort the array in reverse order:

```
WOOFY
PETUNIA
HAMMY
GOMEZ
FLUFFY
BARKY
```

Sorting an Array Using a Block

By default array.sort will sort any array of values using the <=> spaceship method of the value. This method as we have seen returns -1, 0 or 1 depending on whether the value is lower, equivalent or greater than another value. The String class implements the <=> method by comparing the two values alphabetically, so by default array.sort sorts an array of strings alphabetically like so:

```
array = ["James", "Derek", "Stuart", "Thomas"]
puts array.sort
  => Derek
  James
  Stuart
  Thomas
```

If we want to override this, there are two easy ways to do it. First we can override the spaceship operator (coming soon). Alternately, we can pass Array.sort a block which can be used instead. For example the following code sorts an array in order of the second letter:

```
array = ["James", "Derek", "Stuart", "Thomas"]
puts array.sort {|a,b| a[1] <=> b[1]}
  => James
  Derek
  Thomas
  Stuart
```

Nice and Simple.

Exercise - Sort according to the length of the string.

You can get the length of the string using “string”.length.

```
BARKY
WOOFY
HAMMY
GOMEZ
FLUFFY
PETUNIA
```

Modifying each element of an array with map

Map is an insanely useful array function that lets you modify each element of an array using a block.

Say you have an array of strings:

```
['kittens', 'puppies', 'hamsters']
```

Lets say you want to convert this to an array of symbols, you could do something like

```
['kittens', 'puppies', 'hamsters'].map {|i| i.to_sym}
```

Exercise - Random Alphanumeric with Map

Use the Array#map method to generate a random string. You can generate a random letter by making an array of acceptable characters like so:

```
n = (0..9).to_a + ('a'..'z').to_a + ('A'..'Z').to_a + %w(_ -)
```

You can then pull a random one out like so:

```
n.sample
```

Now use map to generate a 128 character random string.

Hashes and Symbols

Hashes are a big deal in Ruby. We use them a lot, particularly when passing bits of data around. Symbols are tiny lightweight Ruby placeholder objects. They are often used in conjunction with hashes. In this section we will look at hashes, symbols, and some of their common uses.

Symbols are not magic runes

Symbols often seem like magic runes to Ruby newcomers. In fact they're just little objects that have some special features and syntax surrounding them to make them a little easier to use and a little lighter on the computers memory.

Symbol syntax

Symbols are defined using the colon operator or the to_sym method:

```
:price
:length
:outrageousness
"My Price".to_sym
```

Features of Symbols

- Symbols are tiny little objects
- Symbols don't have a value, they are placeholders, not variables.
- There can only ever be one symbol with a particular name, this is managed by Ruby. Because of this they are not processor or memory intensive.
- Symbols can potentially represent a memory leak if you create a lot of them.

There can only be one

A symbol will only exist once in memory, no matter how many times it is used. If for example, you create two symbols in different places both called :name for example, only one object would be created. This object would persist for as long as the Ruby interpreter was running.

Uses of Symbols

Symbols are most commonly used as placeholders in Hashes. We have used symbols already in Rails, for example the Rails params hash associates any the values of any parameters passed in by the user (from a form or in the url) with a symbol representing the name of that value.

Hashes

Hashes are objects that associate lists of arbitrary objects with each other. For example:

```
animals = Hash.new
animals[:tall] = "giraffe"
animals[:minute] = "kitten"
puts animals.inspect
```

Hash Assignment Shorthand

```
animals = {:tall => "giraffe", :minute => "kitten"}
puts animals[: minute]
=> kitten
```

Setting Default Values in a Hash

A hash will return nil if it can't find any matching keys. You can set a default value that a hash will return in this instance should you so desire.

```
animals = Hash.new("monkey")
puts animals[:funny]
```

```
=> "monkey"
```

You can also set this value using the Hash.default method

```
animals.default = "star mole"
puts animals[:odd]=> star mole
```

Any Objects can be used as a key

Any Object can be used as a key. Here we use the number 45 as a key, and store the root directory of the local file system as a value:

```
animals[45] = Dir.new '/'
puts animals.inspect # {45 => #<Dir:0x284a394>
```

It's all allowed.

One caveat to bear in mind, if you are using Ruby 1.8.6 or less, you can't use a hash as a key in another hash. This is an issue that was addressed in Ruby 1.8.7 and up.

Exercise - Hashes and Symbols

Create a hash to hold a list of feelings and foods. It should look like something like this, only longer:

```
food_hash = {
  :happy => "ice cream",
  :pensive => "witchetty grub"
}
```

Write a function that allows a user to type in a feeling and have it return the corresponding food (Tip: use to_sym to convert the user's input into a symbol)

Functions that receive a Hash

A function can receive a hash of values. This is tremendously useful, and we do this all the time.

```
def get_greeting_for(args={})
  name = args[:name] || "anonymous"
  return "hello #\{name}"
end

puts get_greeting_for :name => "Fat Tony"
```

```
=> "hello Fat Tony"

puts get_greeting_for
=> "hello anonymous"
```

Here our function receives a parameter we've called args. The default value of the args parameter is an empty hash. Any key value pairs we pass will go into args, and can be pulled out.

On the second line we do this:

```
name = args[:name] || "anonymous"
```

Here we set the value of name to be either the value stored in args under the :name key, or if this evaluates to nil (and therefore false) we set it to anonymous. This is tremendously useful, since we can create functions that accept multiple arguments, in any order, with any defaults that make sense.

You should get used to writing configurable, extensible methods that receive a hash. This is a real rubyism.

Exercise - receive a hash

Extend your food finder function so it can receive a hash. You should be able to call it like this:

```
food mood: :pensive
```

Write RSpec

Write RSpec to verify it does in fact work.

Review

- Symbols are singleton strings. They are very lightweight, and can be used anywhere you need a unique object, such as in a hash.
- Hashes are dictionary objects, sometimes called hash tables. They associate keys with values.
- You can use any object as a key, and store any object as a value.
- We commonly use symbols as keys in hashes
- A special syntax exists which lets us pass a hash to a function.

Hard exercise - A substitution cypher

A substitution cypher is one in which each letter is changed for another, so 'a' might map to 'z' and 'b' might map to 'y'.

Here we will create a substitution cypher in a very few words.

We can use zip to combine two arrays

```
[1,2,3].zip [4,5,6]
  => [[1,4],[2,5],[3,6]]
```

we can compose an array of characters from a range like this:

(‘a’..‘z’).to_a

We can create a Hash from arrays of arrays, like so:

```
Hash[[[1,4],[2,5],[3,6]]]
  => {1 => 4, 2 => 5, 3 => 6}
```

You can rotate an array using rotate, like this

```
[1,2,3].rotate 1
  => [2,3,1]
```

Use these techniques to create a substitution cypher hash, something like this:

```
{'a' => 'b', 'b' => 'c', c => 'd' ...}
```

1. Now write a function that can accept a string.
2. Use split to break the string into an array.
3. Use map with a block to rewrite each element, so a becomes z, etc…
4. Now use join to turn the array back into a string.

If you have get everything right you should be able to pass the string back though, and it will go back to how it was.

For bonus points, write the function body in two lines of code.

Exercise - Receiving a hash

Modify your substitution cypher so it can receive a hash of values.

I want to be able to call it like this

```
substitution_cypher "Hello Ruby", rotate: 3
```

Classes and Objects

Ruby is Object Oriented. Many languages claim to be object oriented, but most fall short in some area or another. When Ruby says it's object oriented, it really, really means it.

We have used lots of Ruby objects so far, when we have done things like:

```
"Abracadabra".reverse.downcase
```

and

```
Time.now
```

Lets look now at how we can define our own objects.

Classes

Ruby has a class based object model. This means we define classes, then use them to stamp out as many objects as we like. Of course, this being Ruby, classes are objects and have methods in their own right, but we'll get to this soon enough.

We define a class like so:

```
class Pet
end
```

Here we have defined a very simple class, called Pet. By convention, classes in Ruby always start with a capital letter. Now lets create a pet:

```
flopsy = Pet.new
```

This is great, we have created a new instance of the Pet class. Flopsy is an instance of Pet.

Note that all objects get a new method for free. This they inherit from the Object superclass. More on inheritance in a bit.

Convention in Ruby

Because Ruby is such a relaxed language, conventions become very important. snake_case for variables, CapitalLetters for classes.

These things are not mandated by the language but they will help you to write more consistent, legible and inter-operable code.

Finding the class of an object

We can go in reverse, to find the class of an instance like this

```
flopsy.class
  => Pet
```

Ruby doesn’t shy away from introspection, it comes baked in, as we shall see later.

Methods - Giving Flopsy Abilities

This is all very nice, but flopsy is not very interesting, she can’t walk the tightrope or play chess, or really do anything much. To make Flopsy more interesting, we need a method:

```
class Pet
  def play_chess
    puts "Now playing chess"
  end
end
```

Here see now. We have added a play chess method to flopsy. We can now write:

```
flopsy.play_chess
```

…and she will, after a fashion. She is only a housepet after all.

Naming Conventions

There are a few things to bear in mind when naming methods in Ruby if you want to appear cool and down with the kids.

First, use snake case for all function names, like this.

```
each_with_index
```

Second, if your method returns a boolean, and is a question, frame it as such. Use a question mark, like so:

```
['toast','jam','honey'].include? 'ham'
person.has_name?
password.valid?
```

Third, if your method modifies the original object in place, rather than returning a new object, and is therefore destructive, indicate this with an exclamation mark, like so:

```
['toast',['jam','honey']].flatten!
  => ['toast','jam','honey']
```

Instance variables - Giving flopsy some superpowers

Flopsy is still a little dull. It would be great to be able to store some data about her, maybe give her some custom attributes.

In Ruby we save an instance variable using the @ syntax. Instance variables are @ variables. All instance variables are private, so to get at them, we need to write methods called getters and setters to access them. Lets have a look now:

```
class Pet

  def super_powers=(powers)
    @super_powers = powers
  end

  def super_powers
    @super_powers
  end

end
```

Here we have given flopsy two methods, a getter and a setter. The first is a setter. The super_powers= method receives a parameter and stores it in an instance variable called @super_powers.

The second is a getter. It simply returns the @super_powers instance variable that was previously set.

We can now set flopsy's super power like this:

```
flopsy.super_powers = "Flight"
```

and retrieve it like this:

```
flopsy.super_powers
  => "Flight"
```

Note we don't have to declare the @super_powers variable anywhere. We can just set it, and that's fine.

This is great because...

Getters and setters give us a clean way to provide an interface onto our object. It insulates us from implementation details. We are free to store the data in any way we wish, as a variable, as a file, in a database, in an encrypted hash, or as a combination of other variables.

This is how active record works when using Rails. Values can be got from the database as though we were accessing object attributes.

Just like Flopsy, the boundary between attributes and methods is far more fuzzy than in most languages. This is partly because of Ruby's optional parentheses, which make it look as though we are accessing attributes, when in fact we are always accessing methods.

We can have read only attributes by only creating a getter, and write only attributes by only creating a setter. An example of a write only attribute would be a password, which might get set, and then encrypted with a one way hash, never to be read again.

The attr method

Since class variables are so common, ruby defines shortcuts for creating them and their associated getters and setters. The attr method creates an attribute and its associated getter. If the second parameter is true a setter is created too. The attr_reader and attr_writer methods create getters and setters independently.

Initialising flopsy using the initialize method.

The initialize method is called by the new method and it is here that we put any code we need to initialize the object.

When flopsy's sidekick mopsy was first created, she didn't have any powers at all, observe:

```
mopsy = Pet.new
mopsy.super_powers
  => nil
```

Poor mopsy. We can remedy this situation by giving mopsy a basic superpower when she is initialised. Lets do this now.

```
class Pet
  def initialize(args = {})
    @super_powers = args[:power] || "Ability to eat toast really really quickly"
  end
end
```

Now when we recreate mopsy, she comes already tooled up

```
mopsy = Pet.new
mopsy.super_powers
  => "Ability to eat toast really really quickly"
```

We can also do this:

```
mopsy = Pet.new :power => "none worth mentioning"
mopsy.super_powers
  => "none worth mentioning"
```

If you need to see a list of all mopsy's attributes you can do so using inspect like so:

```
mopsy.inspect
  => "#<Pet:0x102f87fe8 @super_powers=\"Ability to eat toast really really quickly\">"
```

Upshot

- Functions in Ruby are created using the def keyword (short for define).
- Functions that exist in an object are typically called methods.
- Functions and methods are the same, except one belongs to an object.
- Objects are created from classes using the .new method

Exercise - Create your own fluffster

Everything is an object, so it's important that we get a whole lot of practice in with making them. In this series of exercises we will create a class, implement some methods, overload some operators and create some virtual attributes.

Define your class

1. Define a class either for a lethal warship or a fluffy animal, depending on your mood today.
2. Add methods so you can set a name for your gunship/fluffster.
3. Add methods to get and set an age in milliseconds.
4. Write a method that retrieves the age in days. You should use the same age attribute as in 3, just modify it before returning it.
5. Write a method that sets the age in days. Again, use the same age attribute, just monkey with it a bit before setting it.
6. Write a getter that returns the age in weeks. (age in days / 7)
7. Write getters and setters that return a string representing a standard action, e.g. "wriggle cutely", or "blow things up with extravagant firepower"

Exercise - Receiving a hash

Extend the warship / flufster class you created earlier so it can receive a hash of initial values.

Write an initializer method that accepts a hash and sets default values appropriately. You should be able to call it like this:

```
Fluffster.new age: 2, name: "Floppy"
```

Exercise - Read only virtual attributes

1. Add x and y position attributes, with getters, but no setters. Initialise the position attributes to be zero.
2. Add move north, south, east and west methods. When these methods are called they should modify the x and y positions.
3. Add a distance_from_home method that computes the current distance from home.
4. Add an at_home? method that returns true if the person is at home, i.e. if their x and y coordinates are zero.

Virtual attributes are particularly useful for things like passwords where you don’t actually want to store the password anywhere, or allow retrieval.

Monkey Patching for Great Justice

Because Ruby is an interpreted language objects are open and can be modified at runtime. Classes can be reopened at any time.

We can give mopsy new methods, even after she has already been created. Observe:

```
class Pet
  def play_chess
    puts "now playing chess"
  end
end

class Pet
  def shoot_fire
    puts "activating primary weapon"
  end
end

mospy.shoot_fire
  => activating primary weapon
```

Mopsy can still play chess. The Pet class was added to, not overwritten

```
mopsy.play_chess
  => Now playing chess
```

Modifying an Existing Class

As we’ve mentioned before existing classes can be extended. This includes built in Ruby classes. This is a feature that can be used both for good, and for evil:

For good

```
class String
  def put_times(n)
    for i in (1..n)
      puts self
    end
  end
end
```

```
"Marmalade Toast".put_times 5
```

For Evil

```
class Fixnum
  def *(num)
    self + num
  end
end

puts 5*4
  => 9
```

Yes, Ruby lets you do this. Be careful and do things and your code will read like liquid sunlight.

Monkey Patching

Reopening code in this way is often known as monkey patching. We can modify or extend any existing class at runtime, even built in classes like strings and arrays. This can be used to great effect, for example Rails Fixnum date extensions, which allow you to type things like:

```
Date.today + 5.days
```

Here Fixnum has been monkey patched with an function that allows it to work with dates and times. This is a nice syntax, although it makes some people cross as it appears to break encapsulation.

Monkey patching is fun, but use it with care, otherwise you'll end up with a twisted mess on the floor.

Metaprogramming

Monkey patching is the first step towards meta-programming - writing code which writes code. More on this soon.

Exercise - Seconds

Extend Fixnum with a method .seconds which returns the number * 1000

You can now call Time.now + 60.seconds to get the time in one minute.

For bonus points, also create minutes, hours, days and weeks methods. You can now call Time.now + 1.week.

Exercise - Green Bottles

Extend the FixNum class with a green_bottles method that returns the lyrics for the popular song.

I want to be able to say:

5.green_bottles

and get back:

```
"5 green bottles sitting on the wall\n
4 green bottles ..."
```

If you are running a Macintosh, turn the sound up and try this. Note the backticks which you can find above the alt key:

```
`say #\{5.green_bottles}`
```

Bonus

For bonus points, make it accept a block that receives the song line by line. I want to be able to call:

```
5.green_bottles {|song_line| puts song_line}
```

Exercise - Every Other ()

Extend the Array class with a method that iterates over every other element. Call it like this:

```
names.every_other {|name| puts name}
```

You will need to explicitly receive the block, filter the array, then pass it to the each method.

Operator Overloading

Did I mention that in Ruby everything is an object? This extends to operators, such as +, -, * and /. Operators in Ruby are actually methods, and we can define and redefine them, like so:

```
class Pet
  def +(pet)
    p = Pet.new
    p.super_powers = self.super_powers + " and also " + pet.super_powers
```

```
      return p
    end
  end
```

Here we have defined a plus method that receives another pet. This simply creates a new pet with the combined superpowers of it's two parents and returns it. Observe the offspring of Mopsy and Flopsy:

```
cottontail = mopsy + flopsy
cottontail.super_powers
  => "Ability to hop really really quickly and also Flight"
```

Overriding the + Operator Exercise

In this section you will extend your warship / flufster class with some operator overloading.

Implement the + operator. Have it return a new object with the names and standard actions concatenated.

Sorting your custom class with the <, > and <=> (spaceship).

A slightly more useful one this time.

1. Define a method on your class which specifies an attack power.
2. Now define the < operator, which returns true if the object has a lower attack power.
3. Follow up with the > and == operators.

Now for a spaceship

1. Implement the spaceship operator on your class. The spaceship operator returns either -1, 0 or 1 depending on whether the first operand is less than, the same as, or greater than the second one.
2. Now create an array of objects based on your class, and sort them.
3. Pass Array.sort a block, and sort the array based on different criteria.

You can now call the .sort method on an array of objects, and it will be sorted by attack power.

Further Exercise (If you finish first)

Implement the * operator.

Have it return an array containing multiple instances of the object. (either copies, or several variables pointing to the same object, free choice.) You can duplicate an object using the .dup method.

Inheritance

Rails supports single object inheritance. This means a class can have one parent class and will inherit all the methods and attributes belonging to that class. We define inheritance relationships using the < operator.

For example, let's say we'd like to define a particular type of pet, say a small and fluffy kitten. Let's create a kitten class that can inherit from our Pet class:

```
class Kitten < Pet
  def play_tennis
    puts "I am now playing tennis"
  end
end
```

Our kitten now has all the attributes of a pet. It can shoot fire from it's eyes, and play some good chess, but in addition it can also play tennis:

```
tiger = Kitten.new
tiger.play_tennis
  => I am now playing tennis
tiger.shoot_fire
  => now shooting fire
```

Exercise - Subclasses

Extend your Fluffster / Warship exercise from before. Create a subclass of warship, perhaps a frigate, that has different abilities.

Eigenclasses (Static Methods)

In Ruby, all methods exist within a class. When you create an object, the methods for that object exist within it's class. Methods can be public, private or protected, but there is no concept of a **static method**. Instead, we have singleton classes, commonly referred to as eigenclasses.

No static methods

Static methods are class methods. they belong to the class, not the instance. This would break Ruby's simple object structure, since classes are instances of class Class, adding methods to Class, would make them available everywhere, which is not what we want.

Singletons

Instead, Ruby lets us define an unnamed singleton class that sits in the inheritance tree directly above any object. Lets do this now and create a static method.

```
class Kitten
  class << Kitten
    def max_size
      8
    end
  end
end
```

The class << Kitten syntax opens up the eigenclass and pops the max_size method within it. We can then access it like this

```
puts Kitten.max_size
```

Notice that we are talking to the Kitten class as an object here.

Shorthand

We use the class << self syntax to explicitly open an object's eigenclass. We can accomplish the same thing using the shorthand syntax:

```
def Kitten.max_size
  8
end
```

This adds a method to the Kitten eigenclass. We can also add a method to the eigenclass of any other object, like so:

```
fluffy = Kitten.new
popsy = Kitten.new

def popsy.deploy_wheels
  @wheels = :deployed
end

def popsy.launch_scouter
  @scouter = :launched
end
```

Here we have added a method to popsy's eigenclass, allowing her to deploy wheels.

Eigenclasses in the Inheritance Hierarchy

The eigenclass sits directly above the object in the inheritance hierarcy, below the class of the object. It provides a handy place to put methods hat we want to apply directly to the object, rather than to every instance of that object. It feels technical, but once you get it, it’s actually rather nice.

Exercise - Adding a method directly to an object

In irb (or in a ruby file) create 3 instances of your warship/pet class. Add a different method to each of them. Verify that only the instance you added the method to it to can call it.

You’re writing to an eigenclass. Feels natural doesn’t it?

Modules (Mixins)

Modules can be used to add reusable functionality to a class. They are sometimes known as Mixins. A module consists of a whole bunch of methods. By importing it into a class, we gain access to all those methods. This is a handy way to get around the restrictions of single object inheritance, since we may import as many modules as we like.

Defining Shared Functionality

Lets teach flopsy how to make an omelette. Then she will be able to help out in the kitchen. It would be nice if our omelette could accept a few options, so lets allow that too.

```
module CookOmelette
  def cook_omelette(args={})
    number_of_eggs = args[:number_of_eggs] || args[:eggs] || 2
    cheese = args[:cheese] ? "cheese" : nil
    ham = args[:ham] ? "ham" : nil
    mushrooms = args[:mushrooms] ? "mushrooms" : nil
    ingredients = [cheese,ham,mushrooms].delete_if{ |ingredient| ingredient
    ingredients = ingredients.join(" & ")
    "#\{ ingredients } omelette with #\{number_of_eggs} eggs".strip
  end
end
```

Now include the mixin in the Pet class.

```
class Pet
  include CookOmelette
end
```

All our pets can now make delicious omelettes. Observe:

```
mopsy.cook_omelette
  => "omelette with 2 eggs"
```

```
mopsy.cook_omelette :ham => true, :cheese => true, :eggs => 4
  => "cheese & ham omelette with 4 eggs"
```

Module Inheritance

Including a mixin in a class adds those methods to that class as though they had been defined within that class.

Methods added to a class by a module are inherited by subclasses of that class. For example, by including the CookOmelette mixin in the Pet class the Kitten subclass and all its instances also gain that method.

Exercise - Extracting Common Functionality

This exercise extends the lethal warship of fluffy kitten class .

1. Lets Create a module now. Extract the age and age_in_weeks methods into a module that can be included elsewhere. Name the module sensibly. Now remove these from your class, and instead import the module. You now have the ability to make anything have an age, and to query it's age sensibly.

2. Write a stereo module that allows your class to play some cool random sounds (really strings). Add it to your class.

Extend vs. Include

Include and Extend allow us to take methods from a module and add them to an object. They work slightly differently from each other though. Let's take a look…

Extend adds methods directly to an object

Extend adds methods to an object. It extends that object by adding new features to it.

```
class Hamster
end

module PetSkills
  def snuggle;end
end

Hamster.extend PetSkills
```

If you extend a class, you create a class method.

```
h = Hamster.new
Hamster.methods.include? :snuggle
# => true
h.methods.include? :snuggle
# => false
```

If you extend an instance of a class, you create an instance method, but only on that instance. You can extend any object like this.

```
h.extend PetSkills
h.methods.include? :snuggle
# => true

i = Hamster.new;
i.methods.include? :snuggle
# => false
```

You can call extend on any object to add methods to that object alone.

Include adds instance methods to a class

Include takes a more traditional approach. If you include a module in a class, the methods in the module will be added as instance methods, and will be available to all instances of that class.

```
class Gerbil
  include PetSkills
end

g = Gerbil.new
Gerbil.methods.include? :snuggle
# => false
g.methods.include? :snuggle
# => true
```

Upshot

Extend will add methods to an object, and only to that object. If we extend a class we get class methods.

Include will include methods from a module into a class, those methods become instance methods for objects of that type.

Exception Handling

Exception handling in Ruby is very similar to other languages.

Raising an Exception

Raising an exception in Ruby is trivially easy. We use raise.

```
raise "A Error Occurred"
```

This will raise the default RuntimeException.

Raising a Specific Exception

We can also raise a specific type of exception:

```
value = "Hi there"
raise TypeError, 'Expected a Fixnum' if value.class != Fixnum
```

Rescuing Exceptions

We can rescue exceptions easily. Put the code that might raise an exception in a begin, rescue end block. If an exception occurs, control will be passed to the rescue section.

```
begin
  raise "A problem occurred"
rescue => e
  puts "Something bad happened"
  puts e.message
end
```

Rescuing Specific Exceptions

We can rescue different types of exceptions

```
value = "Hi there"

begin
  raise TypeError, 'Expected a Fixnum' if value.class != Fixnum
  raise "A problem occurred"
rescue TypeError => e
  puts "A Type Error Occurred"
  puts e.message
rescue => e
  puts "an unspecified error occurred"
end
```

The Ruby Exception Hierarchy

Here are the built in exceptions available in Ruby:

```
Exception;
NoMemoryError;
ScriptError;
LoadError;
NotImplementedError;
SyntaxError;
SignalException;
Interrupt;
StandardError;
ArgumentError;
IOError;
EOFError;
IndexError;
LocalJumpError;
NameError;
NoMethodError;
RangeError;
FloatDomainError;
RegexpError;
RuntimeError;
SecurityError;
SystemCallError;
SystemStackError;
ThreadError;
TypeError;
ZeroDivisionError;
SystemExit;
fatal;
```

Defining Your Own Exception

You can define your own exceptions like so:

```
class MyNewError < StandardError
end
```

You can then raise your new exception as you see fit.

Exception Exercises

Try these exercises to get a feel for exception handling in Ruby.

Raising an Argument Error

Extend your kitten class from yesterday. Lets assume your kitten needs an age (0 will not do)

Raise an argument error if age is not set in the initialiser

Raise a Type Error

Your kittens age must be a Fixnum. Check for this, if it is not, throw a Type Error

Catching a Division By Zero Error

Can your kitten do maths? If not, write a divide function now that accepts two values and divides them. Catch the division by zero error, and if it occurs, return nil.

Writing methods which accept blocks

So how does a method yield control to a block? Well, we use a keyword called yield. For example, the following (very simple and not terribly useful) function accepts a block of code and then simply runs it once:

```
def do_once
  yield
end

do_once {puts "hello"}
  => "hello"
do_once {10+10}
  => 20
```

You can call yield as many times as you like. This is how the Array.each method works, by iterating over the array and calling yield for each item, passing the item as a parameter. Here's a silly example:

```
def do_thrice
  yield
  yield
  yield
end

do_thrice {puts "hello"}
  => "hello hello hello"

do_thrice {puts "hello".upcase}
  => "HELLO HELLO HELLO"
```

Passing Parameters to a Block

A block is an unnamed function and you can easily pass parameters to it. The following example accepts an array of names and for each one sends a greeting to the block. The block in this case just puts the greeting to the screen.

```
def greet(names)
  for name in names
    yield("Hi There #\{name}!")
  end
end
```

```
greet(["suzie","james","martha"]) { |greeting| puts greeting }
  => Hi There suzie!
  Hi There james!
  Hi There martha!
```

Upshot

We can tell our function to receive a block in two ways:

1. Explicitly, by simply receiving a block parameter
2. Implicitly, by calling yield within our block

We can check if a block was passed in using the block_given? method.

Re-implement the Fixnum#times method Exercises

This calls the block a specified number of times. It allows you to write code like this:

```
12.times_over { puts "starfish" }
```

Recreate this method in your own words.

Extend it so you can call it like this:

```
12.times_over_with_index { |i| puts "\#{i} - starfish" }
```

Re-implement the Array#each_with_index method.

This method calls its block once for each element in an array. You call it like this:

```
[1,2,'cats'].each_with_index { |el, i| puts "\#{i} - \#{el}" }
```

Further Exercise - Real world block code

Check out the Rails link_to method. Like many helpers it receives an optional block.

Read through the codebase and see if you can see how it does it:

https://github.com/rails/rails/blob/433ad334fa46623f4b218d3bb34e3f63d5481c18/actionview/lib/action_view/hel

Creating a Gem

A gem is a zip file containing code, typically, though not always Ruby code, plus a little metadata to help RubyGems and Bundler to manage it nicely. Gems contain a gemspec manifest file, which contains metadata, and usually a lib directory containing the code itself.

Because a gem contains Ruby code, it can do whatever you like. It can monkeypatch Rails. It can delare new modules and classes. It can extend existing objects, etc.

A gem can be stored locally, or on a gem host like rubygems.org. If you publish your gem to rubygems it will be available for anyone to download and use. If you keep it in your project, you can still use bundler, you just provide a path.

http://asciicasts.com/episodes/245-new-gem-with-bundler

In this section we're going to create a simple gem using Bundler.

Creating a gem using Bundler

We are going to create a summarise gem which will extend the string class with methods to create a summary, and to check whether the string can be summarised, like so:

```
class String
  def summarise(l=200)
    i = 0
    self.split.map{ |word| word if (i += word.length) < l}.compact.join(' ')
  end

  def summarisable?(length=200)
    return self.summarise(length) != self
  end
end
```

first of all create the gem:

```
bundle gem summarise
```

We now have a new summarize directory containing a lib directory for code, a gemspec file for metadata, and a few other files.

We also have an empty git repository initialised for us.

Gemspec

The gemspec file is the heart of your gem, it contains all the metadata about your gem. Your generated gemspec will look something like this:

```
# coding: utf-8
lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require 'summarise/version'

Gem::Specification.new do |spec|
  spec.name          = "summarise"
```

```
    spec.version       = Summarise::VERSION
    spec.authors       = ["Nicholas Johnson"]
    spec.email         = ["email@domain.com"]
    spec.description   = %q{TODO: Write a gem description}
    spec.summary       = %q{TODO: Write a gem summary}
    spec.homepage      = ""
    spec.license       = "MIT"

    spec.files         = `git ls-files`.split($/)
    spec.executables   = spec.files.grep(%r{^bin/}) { |f| File.basename(f) }
    spec.test_files    = spec.files.grep(%r{^(test|spec|features)/})
    spec.require_paths = ["lib"]

    spec.add_development_dependency "bundler", "~> 1.3"
    spec.add_development_dependency "rake"
  end
```

Notice how spec.files is set by getting the files currently included in git.

Also notice how the version is taken from the Summarize::VERSION constant. This is defined in lib/summarise/version.rb. The version can be updated by editing this file.

Gem Contents

We now define the gem in the lib directory.

lib/sumarise.rb

This file simply imports the rest of the code.

```
require "summarise/version"
require "summarise/string_extensions"
require "summarise/string"

module Summarise
end
```

lib/summarise/string_extensions.rb

This declares the methods we want to add.

```
module Summarise
  module StringExtensions
    def summarise(l=200)
      i = 0
      self.split.map{ |word| word if (i += word.length) < l}.compact.join(' ')
    end

    def summarisable?(length=200)
      return self.summarise(length) != self
    end
  end
end
```

lib/summarise/string.rb

This extends the String class.

```
class String
  include Summarise::StringExtensions
end
```

Building the gem

Because the gemspec uses Git to discover which files to include, we must first commit your updated files.

```
git add .
git commit -a -m "first commit"
```

Build the gem using the gem build command:

```
gem build summarise.gemspec
```

You will create a file called something like: summarise-0.0.1.gem

Local Private Gems

If you want to keep your gem private, you can deploy it directly into your vendor/gems directory, like so:

First unpack it into vendor/gems:

```
gem unpack summarise-0.0.1.gem --target /path_to_rails_app/vendor/gems/.
```

Now declare it in your Gemfile:

```
gem 'summarise', :path => "#\{File.expand_path(__FILE__)}/../vendor/gems/summarise-0.0.1"
```

Finally install it into Gemfile.lock

```
bundle install
```

This is a good way to develop a gem, as you can deploy it locally and work on it in situ.

Uploading your gem to RubyGems.org

If you'd like to share your gem with the community, you can also push it to RubyGems.org

You'll need a RubyGems account:

https://rubygems.org/users/new

Now simply push the packaged gem:

```
gem push summarise-0.0.1.gem
```

Exercise - creating a gem

Create a random string gem. I want to be able to call something like:

String.random(6)

to get back a random alphanumeric string.

Send

Send allows you to call a method identified by a symbol.

My app has classes for people and events. Each class has different attributes.

```
class Person
  attr_accessor :name
end

class Event
  attr_accessor :title
end
```

I also have a module which sets up the objects.

```
module Setup
  def init(name_or_title)

  end
end
```

Exercise - Send a little love

Include the Setup module in the Person and Item classes. Now write the init method. use respond_to? and send to initialise either the name or title.

Start with an array [:name, :title], then iterate over it, checking if the object responds to the methods, then call the method when you get a result.

Define Method

We use define method to create a method on the fly given a string as a name.

```
class A
  define_method :c do
    puts "Hey!"
  end
end

A.new.b

A.new.c(1,2)
```

Exercise - Green Hamsters

Hamsters come in red, green, blue and orange.

A hamster class looks like this:

```
class Hamster
  attr_accessor :colour
end
```

for now we’ll just assume that only these colours are allowed.

Given an instance of hamster, I would like to be able to call something like:

```
hammy = Hamster.new
hammy.colour = :red
hammy.is_red?
```

and have it return true or false

Bonus marks

Rather than defining a limited set of methods in advance, use method_missing to catch when a method is not defined, then define it on the fly. If I call is_taupe? method_missing should catch that, define a Hanster#is_taupe? method, then call it.

Exercise - drying up code

The following real code is full of duplication. How might you use define_method to dry it up?

```
class Widget
  def product
    product = Product.find_by_slug(object_slug)
    if !product
      product = Product.first
    end
    product
  end

  def poem
    poem = Poem.find_by_slug(object_slug)
    if !poem
      poem = Poem.first
    end
    poem
  end
end
```

If you don't want to create a whole Rails instance, you can use a simple scaffold like this:

```
class Product
  class << self
    def find_by_slug
      return "success"
    end
  end
end
class Poem < Product; end
```

Method Missing

Trying to call a method that doesn't exist in Ruby is an exception rather than a language error. We can catch it, or we can simply implement a method_missing function which will be called when no matching method is found.

```
class A

  def ary
    [:a,:b,:c]
  end
```

```
    def method_missing(method, *args)
      puts ary.include?(method)
    end
  end

  a = A.new

  a.b
  a.d
```

Exercise - Create a Spaceship.

Assuming you have built a mighty warship, give your class an array of abilities like this: [:warp, :photon_torpedos, :holodeck] etc.

Now we're going to use method_missing to allow us to query our space ship. We will be able to call methods like enterprise.has_holodeck?, and voyager.can_warp? and get back a true or a false value.

Here's a regex that should help

```
FEATURE_REGEX = /^(?:has|can)_(\w*)\?$/
if find = method.to_s.match(FEATURE_REGEX)
  feature = find[1]
end
```

Real world example

For a real world example, read and understand the string_enquirer codebase here:

https://github.com/rails/rails/blob/d71d5ba71fadf4219c466c0332f78f6e325bcc6c/activesupport/lib/active_suppo

Instance_Eval

We use instance eval to evaluate a block in the context of a Ruby object.

```
class A
  def initialize
    @b = 123
  end
end
puts A.new.instance_eval { puts @b }
```

We can use this to construct a DSL (Domain specific language, like this:)

```
class Review
  attr_accessor :stars, :title, :content
  def initialize &block
    self.stars = 0
    instance_eval &block
  end

  def set_title t
    self.title = t
  end
end

r = Review.new do |review|
  set_title "new Macbook"
end
```

Exercise - Instance Eval

Use instance_eval to define a DSL for creating web pages.

```
Page.new do
  set_title "My page"
  set_content "Page Content"
end
```

Extension

Extend your DSL so you can also create sub-pages, like so:

```
Page.new do
  set_title "My homepage"
  set_content "Page Content"
  sub_page do
    set_title "My sub page"
    set_content "Page Content"
  end
end
```

D3 Course
JavaScript for Programmers Course
Learn to Program with JavaScript!
Maths For Machine Learning (for programmers)
The MongoDB Course
The NodeJS Course
Python for Machine Learning Course
The Rails Course
The React Course
Responsive Design Course
The Ruby Course
The HTML/CSS Course

Webdev