Ruby

Total Estimated Time: 100-200 hrs

!! This section is in the "Collecting Resources" phase !!

Top level table of contents

Table of Contents

Intro

In this unit you will learn Ruby, the language designed specifically with programmer happiness in mind. It's a healthy chunk of learning but, by the end of it all, you'll have built some pretty sweet games including Tic Tac Toe, Minesweeper, Checkers, and Chess. You'll be able to put together a Twitter spambot (that really spams!), save and open files, test out your code, separate your spaghetti code into nice modular classes, and even reproduce some basic algorithms and data structures for solving complex problems. Basically, you're going to start feeling a whole lot more like a real programmer and that feeling will be justified.

Some people believe you can just dive right into Rails and start firing out websites. Rails is a framework built using Ruby and every piece of code in it is Ruby. When (not *if*) something in your project breaks, you'd better be able to debug it. And what happens when you want to stretch your wings and do something just a bit beyond what the vanilla tutorials show you how to do? The amount of time you'd spend googling your error messages and staring blankly at help docs was better spent learning Ruby.

As you may gather, this is also where the real project work begins. Some of the early material will be fairly straightforward and will rely on simple exercises to help reinforce understanding. As we get further along and into some of the more

advanced topics, we'll be learning less and doing more... just the way it should be. Buckle up, strap in, and let's get learning!

How this will work:

Ruby's a big language so it's been broken up into smaller chunks to make it more digesible. In each section, you'll first be asked to do readings, watch videos, or otherwise view content. We'll provide a "A Brief Summary" of the material but it's not a replacement for actually doing the reading. At the end of each section or group of sections will be programming exercises which are best done in pairs.

Our free resources:

The goal here is to provide as much of this curriculum as possible using free resources. If you've done the prep work from Web Development 101 then you should have a good handle on the basics but these resources are important to help you really understand the material. * The staple book: Zed Shaw's Learn Code the Hard Way, an extension of his wildly popular Learn Python the Hard Way into Ruby. * For the crazies: Why's Poignant Guide to Programming (check it out... if it jives with your learning style, you may have found the match you never thought you'd find) * If there's anything you need to brush up on still: The Ruby User's Guide has sections on many topics you might want to dive back into for a deeper look.

Other good resources: * Peter Cooper's Beginning Ruby is a solid introduction to Ruby that covers pretty much the breadth of the language as you need to understand it. * Brian Marick's Everyday Scripting with Ruby takes a pragmatic approach to learning Ruby to help with the kinds of problems you might face in a variety of different real-world work scenarios.

TODO: Walkthrough of a real Ruby program

Basic Ruby

Intro

Ruby shouldn't be anything new to you by now... you should have completed the preparatory readings, Ruby Monk, and the test-first exercises as part of the Ruby 101 section of the Web Development 101 Unit that you completed prior to jumping into this. If you haven't, go back and work on that before getting started here. It's expected that you have a pretty good handle on the basics of the Ruby language and an idea of what you're still shaky on.

In this section on Basic Ruby, we're going to make sure you really do understand all the building blocks of the language and of programming in general. It's a lot of stuff... this section is going to have the most to absorb right off the bat. If there's something that you still just don't quite understand, track it down via the Additional Resources section or Google for it on your own. *How* you learn

isn't as important as making sure you're comfortable with your understanding of everything that will be covered here.

Basic Data Types and Other Basic Stuff

Basic data types are the building blocks of computer programs. Understanding the basic data types and what you can do with them is like knowing your ABC's. Get well acquainted with Numbers, Strings, Arrays, Hashes, Objects, Methods, and Dates/Times. You're populating your developer's toolkit.

The exercises should help you hone in on what things you understand well and which ones you need to dig deeper on. The larger exercises towards the end will round things out a bit more wholistically.

Numbers and Operators and Expressions

Goals This is pretty straightforward stuff. The goal here is to familiarize yourself with all basic data types and how they interact in arithmetic expressions

Thought Questions

- What's the difference between an `Integer` and a `Float?`
- Why should you be careful when converting back and forth between integers and floats?
- What's the difference between `=`, `==`, and `===`?
- How do you do exponents in Ruby?
- What is a `range`?
- How do you create a range?
- What's the difference between `(1..3)` and `(1...3)`?
- What are three ways to create a range?

Do These First

- Read Learn Ruby the Hard Way Chapters 1-5
- Read Learn to Program Chapter 1 (You should already have completed this)
- Read http://rubylearning.com/satishtalim/numbers_in_ruby.html

A Brief Summary When doing mathematical operations, Ruby expects the result to be the same type as the inputs, so dividing two integers by each other will produce an integer... whether you want to or not:

```
> 5 / 3
=> 1
```

To fix this, you need to make one of the inputs a different data type that can handle decimals, like a *floating point* number (float):

```
> 5.0 / 3               # as long as one of them is a float...
=> 1.6666666666666667   # ... the result is a float
```

Converting between integers and floats is easy – just use `.to_i` and `.to_f` respectively:

```
> 5.0234.to_i
=> 5
> 5.to_f
=> 5.0
```

Because Ruby is so flexible, it lets you do some quirky things like multiplying completely different data types together in a way that you sort of think you *should* be able to but never expected to actually be able to do:

```
> "hi" * 3
=> "hihihihi"
```

These types of operations work the same way with variables:

```
> my_word = "howdy"
=> "howdy"
> my_word * 3
=> "howdyhowdyhowdy"
```

A **Range** is just a continuous sequence and we represent it in a shorthand way. If we want to say `3,4,5,6,7,8,9,10,11`, it's much easier to just write it the short way `(3..11)`, meaning "all the integers beween 3 and 11, including both 3 and 11". If we wrote it `(3...11)`, it would actually exclude `11`. You can also create a range using `Range.new(start, finish)`, though the shorthand notation is more conventional.

For equality: * `=` is for assignment, so it assigns a value to a variable as in `> name = "Erik"` * `==` is for checking that two things are equal but don't have to be identical instances. You'll use this in most cases, especially when working with conditional statements. `> 1 + 1 == 2` returns '=> true'. When you start creating your own classes (like an "Animal" class), you'll need to tell Ruby how to compare two animal instances by writing your own version of this method

(it's easy). * `===` can actually mean some different things (you can overwrite it easily). You will probably be sticking with `==` in most situations, but it's good to understand `===` as well. `===` typically asks whether the thing on the right is a member or a part or a type of the thing on the left. If the thing on the left is the range `(1..4)` and we want to know if 3 is inside there:

```
> (1..4) === 3
=> true
```

This also works for checking whether some object is an instance of a class:

```
> Integer === 3
=> true
```

See http://stackoverflow.com/questions/4467538/what-does-the-operator-do-in-ruby for a more detailed explanation.

Exercises

- Play around in IRB and just try multiplying and dividing and equating and comparing things to each other. Be creative until you have a good handle on things.

Objects and Methods

Intro and Goals "Everything in Ruby is an Object" is something you'll hear rather frequently. "Pretty much everything else is a method" could also be said. The goal here is for you to see the Matrix... that everything in Ruby is an Object, every object has a class, and being a part of that class gives the object lots of cool methods that it can use to ask questions or do things. Being incredibly object-oriented gives Ruby lots of power and makes your life easier.

Hopefully you've already picked most of this up from the prep-work.

Thought Questions

- What is an object?
- What is a class?
- What is a method?
- How can you print out an object or class's methods?
- What is inheritance?
- How many classes can a class inherit from in Ruby?

- What's the top-level ancestor class that all others inherit from in Ruby?
- What are Reflection Methods?
- What are a method's Side Effects?
- How do methods take inputs?
- Do all methods have outputs?
- What is Method Chaining and how does it work?
- What's the syntax for writing your own methods?
- What is returned by a method?
- What is an implicit return?
- What are default inputs?
- What does `self` mean?

Do These First

- Ruby Inheritance

A Brief Summary Think of every "thing" in Ruby as a having more than meets the eye. The number `12` is more than just a number... It's an **object*** and Ruby lets you do all kinds of interesting things to it like adding and multiplying and asking it questions like `> 12.class` or `> 12+3`.

Ruby gives all objects a bunch of neat **methods**. If you ever want to know what an object's methods are, just use the `.methods` method! Asking `> 12345.methods` in IRB will return a whole bunch of methods that you can try out on the number 12345. You'll also see that the basic operators like `+` and `-` and `/` are all methods too (they're included in that list). You can call them using the dot `> 1.+2` like any other method or, luckily for you, Ruby made some special shortcuts for them so you can just use them as you have been: `> 1+2`.

Some methods ask true/false questions, and are usually named with a question mark at the end like `.is_a?`, which asks whether an object is a type of something else, e.g. `1.is_a?Integer` returns `true` while `"hihi".is_a?Integer` returns `false`. You'll get used to that naming convention. Methods like `.is_a?`, which tell you something about the object itself, are called **Reflection Methods** (as in, "the object quietly reflected on its nature and told me that it is indeed an Integer"). `.class` was another one we saw, where the object will tell you what class it is.

What is a method? A method is just a function or a black box. You put the thing on the left in, and it spits something out on the right. *Every method returns something*, even if it's just `nil`. Some methods are more useful for their **Side Effects** than the thing they actually return, like `puts`. That's why when you say `> puts "hi"` in IRB, you'll see a little `=> nil` down below... the method prints out your string as a "side effect" and then returns `nil` after it's done. When you write your own methods, if you forget to think about the return statement,

sometimes you'll get some wierd behavior so always think about what's going in and what's coming out of a method.

Methods can take **inputs** too, which are included in parentheses to the right of the method name (though they can be omitted, as you do with `> puts("hi")` becoming `> puts "hi"`... it's okay to be lazy, as long as you know what you're doing). Going back to the addition example, `> 1+2==3` is asking whether 1+2 will equal 3 (it returns `=> true`), but it can more explicitly be written `> 1.+(2).==(3)`. So, in this case, you can see there's more going on than meets the eye at first.

That example also shows **Method Chaining**, which is when you stick a bunch of methods onto each other. It behaves like you'd expect – evaluate the thing on the left first, pass whatever it returns to the method on the right and keep going. So `> 1+2==3` first evaluates `1+2` to be `3` and then evaluates `3==3` which is `true`. This is great because it lets you take what would normally be many lines of code and combine them into one elegant expression.

Bang Methods are finished with an exclamation point `!` like `.sort!`, and they actually modify the original object. Remember, when you run a normal method in IRB, it will output whatever the method returns but it will preserve the original object. Bang methods save over the original object:

```
> my_numbers = [1,5,3,2]
=> [1, 5, 3, 2]
> my_numbers.sort
=> [1, 2, 3, 5]
> my_numbers
=> [1, 5, 3, 2]          # still unsorted
> my_numbers.sort!
=> [1, 2, 3, 5]
> my_numbers
=> [1, 2, 3, 5]          # overwrote the original my_numbers object!
```

Let's answer the question, "Where did all those methods come from?" **Classes** are like umbrellas that let us give an object general behaviors just based on what it is. An object is an instance of a class – you (yes, you) are an instance of the `Person` class. There are lots of behaviors (methods) that you can do just by virtue of being a `Person`... `.laugh`, `.jump`, `.speak("hello")`. This is really useful in programming because you often need to create lots of instances of something and it's silly to have to rewrite all the methods you want all of them to have anyway, so you write them at the class level and all the instances get to use them.

Instances of a class get to **inherit** the behaviors of that class. Inheritance works for classes too! Your class `Person` has lots of methods but many of them are inherited just by virtue of you also being a `Mammal` or even just a `LivingThing`. You get to use all the methods of your ancestor classes

An interesting exercise to try in Ruby is to use the method `.superclass` to ask a class what its parent is. If you just keep on going and going, you'll see that everything eventually inherits from `BasicObject`, which originates most of the methods you have access to in the original object:

```
> 1.class.superclass.superclass.superclass
=> BasicObject
> BasicObject.methods
=> # giant list of methods
```

Random Note: Running the `.methods` method on a class only returns the class methods, whereas `.instance_methods` will return all methods available to any instance of that class (so `String.methods` will return a list of class methods, while `"hello".methods` will return a longer list that is the same as `String.instance_methods`).

Other Random Note: Use `.object_id` to see an object's id, and this can be useful if you're running into odd errors where you thought you were modifying and object but it's not changing. If you debug and look at the id's along the way, you may find that you're actually only modifying a COPY of that object.

To **Write Your Own Methods**, just use the syntax `def methodname(argument1, argument2)`, though the parentheses around the arguments are optional. The method will return ("spit out") either whatever follows the `return` statement or the result of the last piece of code that was evaluated (an **Implicit Return** statement). You call the inputs by whatever name you defined them at the top.

You can write methods in IRB... it will let you use multiple lines if it detects that you have unfinished business (a `def` without an `end` or unclosed parentheses):

```
> def speak(words)
>    puts words
>    return true
> end
=> nil              # ignore this
> speak("hello!")
hello!
=> true
```

What if you want to assume that the input to a method is a particular value if there hasn't been any supplied? That's easy, just specify the **Default Input** by assigning it to something where it's listed as an input:

```
> def speak(words="shhhhh")
>     puts words
> end              # implicitly returns what puts returns... nil!
```

```
=> nil              # ignore this
> speak             # no input
shhhhh
=> nil
```

What is `self`? It's a word that you see a whole lot in Ruby and it's actually pretty simple... it refers to whatever object the current method was called on (i.e. the "caller"). So if I called `current_user.jump`, inside the definition of the `jump` method, `self` would refer to the current_user.

That is incredibly useful because we create methods that could be called by any number of different objects so we need a way inside of that method to dynamically refer to whatever object called the method so we can do stuff to it. You may see something like this:

```
> def full_name
>   "#{self.first_name} #{self.last_name}"
> end   # Remember, this implicitly returns the string "firstname"
```

Exercises

Additional Resources

Strings

Intro and Goals Strings are a huge part of web programming and you'll run into them everywhere from variable names to user input to giant gobs of HTML text to handling big dictionary files. They're actually pretty simple at the core but, for being just a jumble of characters, strings have some pretty cool properties in Ruby and you can do a whole lot to manipulate them.

This section should give you an appreciation for the ways you can mess with strings and some of the handy methods that Ruby gives you along the way to make your life easier.

Thought Questions

- What's the difference between single and double quotes?
- What is string interpolation?
- What are escape characters?
- What are line breaks?
- How do you make other things into strings?

- How do you concatenate strings?
- How do you access a specific character or substring?
- How do you split up strings into arrays?
- How are strings an arrays similar?
- How do you get and clean up user input on the command line?
- What does it mean that strings are "mutable" and why care?
- What is a symbol?
- How is a symbol different from a string?
- What is a Regular Expression (RegEx)?

Check These Out First

- Chris Pine on Strings (was part of the prep work)
- A list of Escape Characters in Ruby
- Read through (and watch the video) for this Regular Expressions in Ruby explanation.
- A great little Regex Tutorial and the example problems (should only take an hour or so)

A Brief Summary Strings are just made up of individual characters and denoted with quotes. `> I confuse Ruby and probably throw an error` but `> "I do not because I have quotes"`.

Double Quotes are often interchangeable with **Single Quotes**... there's almost no difference and you're free to use either. Two cases make the distinction important:

A) When you want to show quotes inside a string:

> my_long_string = "And she said, 'Cool program!' " => "And she said, 'Cool program!' "

Note that you can accomplish the same type of thing by escaping the quote characters (see below).

B) When you want to use string interpolation and when you want to show quotes within a string.

String Interpolation occurs when you want to plug something else into a string, like a variable. You'll find yourself using this a lot, for instance, when you make websites with dynamic text content that needs to change depending on who is logged in. Simply use the pound symbol and curly braces `#{}` to do so, and the output of whatever is within those curly braces will be converted to a string and, presto! You've got a new string:

```
> my_name = "Tiny Tim"
=> "Tiny Tim"
> my_string = "My name is #{my_name}!"
=> "My name is Tiny Tim!"
```

The key point here, though, is that interpolation *only works inside DOUBLE quotes*. Keep the interpolated stuff brief or your code won't be very legible. Single quotes will simply escape every special character in the string and treat it like a normal letter (so the pound-curly-braces has no special meaning):

```
> my_name = "Neo"
=> "Neo"
> my_string = 'My name is #{my_name}!'
=> "My name is \#{my_name}!"  # Hey! That's not what we wanted!
```

Escaping characters just means telling the output program to not treat them specially at all (like the pound symbol, which has special meaning before the curly braces). You do so with a back slash \ before whatever you want to escape. Sometimes you'll see what looks like a strange jumble of characters in your output, with lots of those " floating around, and you'll know that you've got some escaping going on.

```
> now = "RIGHT NOW"
=> "RIGHT NOW"
> puts "interpolating #{now} but not \#{now}"
"interpolating RIGHT NOW but not #{now}"
=> nil            # Remember, puts returns nil!
```

IRB shows you the backslashes, but they'll be hidden in your `puts` output.

As you can imagine, this could get pretty tedious if you're trying to output a blog post or some other long batch of text that contains lots of mixed characters and don't want to manually or programmatically replace special characters, so later we'll see some simple convenience methods to use to take care of those issues for you.

There are some special characters that are actually denoted using the backslash and you'll want to know the key ones, which will probably pop up again and again:
`\n` will output a new line * `\r` is a newline too (carriage return) * `\t` will output a tab

```
> puts "let's put a bunch of newlines between this\n\n\nand this."
```

```
and this.
=> nil
```

`to_s` is a method that will try to convert anything into a string.

```
> 12345.to_s
=> "12345"
```

This method gets called a LOT behind the scenes later on... basically anytime Ruby or especially Rails is outputting or rendering something (like a variable), it will call `to_s` on it to make it a nice friendly string first.

Fun fact: If you've created your own object, you may need to (or GET to!) write your own `to_s` method for it to display properly in some settings. For example (looking ahead), if you've got a Person object and want to display its first name whenever you tried to `puts` it, you'd want to write the `to_s` method to do so.

Combining Strings without using interpolation can be done using "concatenation", or basically just adding them together:

```
> my_name = "Billy Bob"
> "hello" + " world" + ", say I, the great " + my_name + "!"
# => "hello world, say I, the great Billy Bob!"
```

Instead of adding them with a plus `+`, you can also use the friendly shovel operator `<<` to append to a string (just like with arrays...):

```
> "howdy " << "fella!"
=> "howdy fella!"
```

To **Access a Specific Character or Substring** of a string, just treat it like an array! A string acts like a zero-indexed array that ends at -1, so use `[0]` to access the first letter, `[-1]` to access the last letter, and `[n..m]` to pluck a substring:

```
> s = "hello"
=> "hello"
> s[0]
=> "h"
> s[-1]
=> "o"
> s[-2]
=> "l"
> s[2..4]
```

```
=> "llo"
> s[1..-2]
=> "ell"
```

Break a String into Pieces using `.split`, which creates an array of substrings that are broken up based on whatever character you passed in:

```
> list = "eggs, milk, cheese and crackers"
=> "eggs, milk, cheese and crackers"
> list.split(", ")
=> ["eggs", "milk", "cheese and crackers"]
> list.split(" ")
=> ["eggs,", "milk,", "cheese", "and", "crackers"]
```

You can also split based on individual characters by passing either a blank string or a blank regular expression (denoted by `//`):

```
> list.split("")      # or also list.split(//)
=> ["e", "g", "g", "s", ",", " ", "m", "i", "l", "k", ",", " ", "c", "h", "e", "e", "s", "e'
```

When you write your Ruby programs, you'll probably want to ask for **User Input**... which is easy with `gets`, which then waits for the user to type something. You'll want to store whatever the user types into a variable and be sure to trim off the extra line break (from when the user hit the `enter` key) using `.chomp`:

```
> player1 = gets
Erik                # this was typed in manually
=> "Erik\n"         # woah, let's get rid of that \n
> player1 = gets.chomp
Erik
=> "Erik"           # better.
```

`.chomp` will cut off a space or newline at the END of the string (and can take an optional input so you can specify what exactly to chomp off). `.strip` will remove ALL spaces and newlines from both the beginning and end of the string:

```
> " dude \n".chomp
=> " dude "         # still have the extra spaces
> " dude \n".strip
=> "dude"           # clean as a whistle.
```

Of course, it's up to you to figure out if your user has entered something illegal or harmful, but at least you have an easy job removing extraneous spaces and returns.

Other Helpful String Methods include: * `.length` to get the length of the string * `.downcase` to convert `"ALL THIS"` to `"all this"` * `.upcase` to convert `"all this"` back to `"ALL THIS"` * `.reverse` to convert `"hello"` to `"olleh"`

Fun fact: Strings made with the backtick characters ` (which is usually located on the same key as the tilde ~) are actually interpolated and run by your operating system, so in IRB if you type `> puts ‘ls‘` on a mac, it will actually output your directory contents!

What about all the times you may want to **Search For or Replace Within Strings**? For that, you need to begin understanding **Regular Expressions**, or "RegEx"'s. There's a handy method for strings called `.gsub(pattern, replace_with_this)`, which finds any occurrances of that pattern and replaces it with whatever you want:

```
> "hello".gsub("l","r")
=> "herro"
```

But what if you want to go looking for more advanced patterns than just simple letters? Pretty much anytime you've got a function that needs to go mucking through a string looking for patterns, you can employ a Regular Expression.

Regular Expressions are really just a special syntax that is used to find things (and not just in Ruby, they're used all over the place). It's beyond the scope of this summary for sure, but I hope you've tried them out at RegexOne. Once you know how to match stuff, you'll feel ready to take on any big dictionary files or big batches of questionable user input that needs to be standardized. But be careful, it can be too tempting to go hog-wild with your expressions. It's something you should at least know the basics of but probably will not be applying all that often "in the wild".

The last thing to cover is **Symbols**, which start creeping up all over the place when you get into Rails and even Hashes. Symbols are denoted with the colon before the name, e.g. `:my_symbol` instead of `"my_string"`. A symbol is basically like a string without any depth... string lite, if you will. A string is **Mutable**, meaning it can be added to, reversed, and generally messed with in a hundred different ways. Whenever you have text that you want to play around with or use as, well, text, just stick with strings.

But sometimes all you want is a name, like when you're using a hash key. In that case, it makes sense to use symbols. Symbols are immutable, so they don't change. They are also only stored in memory in one place, wherease strings have a new place in memory each time you declare one:

```
> "hello".object_id
=> 70196107337380
> "hello".object_id
=> 70196107320960       # different
> :hello.object_id
=> 461128
> :hello.object_id
=> 461128               # same!
```

While you're learning, just stick with strings until you see the examples using symbols, which will mostly be with hash keys.

Exercises

- TODO: IRB reversing, shouting stuff, find the ith letter...

See Below

Arrays

Intro and Goals We saw how strings are a lot like arrays... so it's probably a good time to dive into what arrays are and how flexible they are in Ruby. Arrays are almost as ubiquitous as strings. You'll be working with them all the time to help store data for you, everything from the names of all your users to coordinates on a game board. An array is an all-purpose bucket into which you can put pretty much anything.

Here, you'll learn the basics of creating arrays, how to manipulate them in a dozen different ways, and some best practices for working with arrays. Note that we'll be learning even more about how to dig around inside of arrays in the future section on iterators, so if you're excitedly waiting to better understand .each, .map and others like them, we're almost there! If not... you will be.

Thought Questions

- What are three ways to create an array?
- How do you prepopulate the array with default data?
- How do you access items in an array?
- How do you modify the items in an array?
- How do you combine arrays?
- How do you find the values in one array that aren't in another?
- How do you find values in both arrays?

- What is the difference between `push`/`pop` and `shift`/`unshift`?
- What is the shovel operator?
- How is `> arr.pop` different from `> arr[-1]`?
- How is `push`ing or `<<`ing another array into your array different from just adding them together?
- How do you delete items in an array?
- Why should you be careful deleting items in an array?
- How can you convert arrays to strings?
- How can you convert from other data types to arrays?
- How can you figure out if an array contains a particular value?
- How do you find the biggest item in an array?
- How do you find the smallest item in an array?
- How to you find out how big an array is?
- How do you put an array in order?
- What are the naming conventions for arrays?
- What should you store in arrays?

Check These Out First

- Do Codecademy's section on arrays for some practice with them

A Brief Summary Arrays begin life as empty containers waiting to be filled with objects or data. As items are added, they stay in whatever spot you put them, which is good because then you know exactly where to find them later. You can put anything in an array! Numbers, strings, objects, symbols, haikus...

Creating an Array can happen in many different ways. You can either create it empty, specify how many spaces it should have (still empty), or even **fill it with default values**:

```
> a = Array.new     # 1
=> []               # see, it's empty
> b = []
=> []               # still empty
> c = Array[]
=> []
> empty_a = Array.new(5)
=> [nil, nil, nil, nil, nil]
> full_a = Array.new(3, "hi")
=> ["hi", "hi", "hi"]
```

And remember, you can store pretty much anything in there, even other arrays:

```
> full_b = [1, 4, 8, "hello", a]
=> [1, 4, 8, "hello", []]  # Cool, arrays inside arrays!
```

. . . *but don't do that!* It's best to keep only ONE type of thing in your arrays or you'll have many headaches down the road because you'll almost always assume that there's only one type of thing in there. Forget that you ever learned that arrays can hold different values.

Accessing Items is super easy, just start from `0` like you did with strings. Just like with strings, you can start from the end of the array using negative numbers from `-1` and you can even grab ranges of values (which are themselves arrays!):

```
> arr = [1, 3, 5, 7, 2]   # favorite way to declare an array
=> [1, 3 ,5 ,7 ,2]
> arr[0]
=> 1
> arr[-1]
=> 2
> arr[1..3]
=> [3, 5, 7]          # this returned an array!
> arr[1..200000]
=> [3, 5, 7, 2]       # no error... silently cuts off at the end
```

Modifying Items is as simple as accessing them is. . . just set them equal to a value:

```
> arr[0] = 42
=> 42
> arr
=> [42, 3, 5, 2]      # changed it!
> arr[0..2] = 99
=> 99
> arr
=> [99, 2]            # wiped out several values, oops...
```

Adding Arrays is also done similarly to strings, by just mashing one onto the end of the other:

```
> first = [1,2,300]
=> [1,2,300]
> second = [7,8,9]
=> [7,8,9]
> combined = first + second
=> [1,2,300,7,8,9]      # this is a NEW array
```

Subtracting Arrays is a bit different... think of the minus sign as saying "take away any and all values that are in the right array from the left array". The only values remaining will be those from the left that were not included in the right side at all:

```
> [1,2,3] - [2,3,4]
=> [1]                # the 4 did nothing
> [2,2,2,2,2,3,4] - [2, 5, 7]
=> [3,4]              # it killed ALL the 2's
```

You'll find yourself adding arrays a lot more frequently than subtracting them but it's good to know both.

If you want to find values in **Both** arrays, check their union using the ampersand `&`:

```
> [1,2,3]&[2,4,5]
=> [2]
```

What if you only want to add or subtract one single value? That's a very common operation with arrays, and Ruby has provided four handy methods that let you either pluck away or add onto the front or back of the array. First, the more common is to add or remove stuff from the END of the array, using **`.push` or `.pop`**:

```
> my_arr = [1,2,3]
=> [1,2,3]
> my_arr.push(747)
=> [1, 2, 3, 747]
> my_arr
=> [1, 2, 3, 747]     # warning: we modified my_arr!!!
> my_arr.pop
=> 747
> my_arr.pop
=> 3
> my_arr
=> [1, 2]          # warning: pop also modified my_arr
```

What if you want to take the item off the FRONT of the array? This is less common. For that, use the similar `.shift` and `.unshift` methods:

```
> my_arr = [1,2,3]
=> [1,2,3]
> my_arr.shift
```

```
=> 1
> my_arr
=> [2,3]          # warning: shift also modified my_arr!
> my_arr.unshift(999)
=> [999, 2, 3]
> my_arr
=> [999, 2, 3]    # warning: unshift... yep, modified my_arr.
```

So the `push`/`pop`/`shift`/`unshift` methods should take you wherever you realistically need to go. Although there's another handy method you should be aware of, the **Shovel Operator**, aka `<<`. This method is *almost* identical to push, since it just jams whatever's to its right into the array:

```
> my_arr = [1,2,3]
=> [1,2,3]
> my_arr << 3
=> [1, 2, 3, 3]
> my_arr << [4,5]
=> [1, 2, 3, 4, [4, 5]]   # Array within array alert!
```

For now, just think of it as the cool way of pushing onto an array. But note that `<<` is often overridden (like in Rails), and so it pays to be mindful of exactly what flavor of `push`ing you're doing.

Deleting Items from an array should be done carefully because, if you're deleting items inside a loop or something like that, it will change the index of the other items and you'll need to anticipate this or live to regret it. Delete an item at a specific index using `.delete_at`, which is sort of like `pop`ing but from anywhere you want:

```
> my_arr = [1,2,3]
=> [1, 2, 3]
> my_arr.delete_at(1)
=> 2
> my_arr
=> [1,3]
```

See if an array **includes an item** AT ALL by using `.include?`, which, as you should see from the `?` at the end, returns true or false:

```
> my_arr.include?(3)
=> true
> my_arr.include?(132)
=> false
```

To find WHERE a specific item lives in the array, use `.index` but note that it only returns the FIRST instance of this (and then gives up. Lazy method.):

```
> my_arr.index(3)
=> 2
> [1,2,3,4,5,6,7,3,3,3,3,3].index(3)
=> 2                  # Just the index of the FIRST match
> my_arr.index(132)
=> nil                # Not an error, just nil
```

A few useful and commonly used methods: * `.max` to find the **biggest value** of an array * `.min` to find the **smallest value** of an array * `.size` to find out **how big the array is** * `.shuffle` will mess up your whole array by putting it in random order * `.sort` will clean it up again for you by putting your array **in order**. Though `.sort` is pretty self-explanatory in the simple case, it can actually take parameters to let you decide if you want to sort things using a different (or reverse) methodology. * `.sample` picks out a totally random value from the array... good for gambling games! * `.first` gives you the first item (but doesn't remove it, so it's same as `[0]`) but can be more descriptive of your code's intent. * `.last` is same as `[-1]`

Do as I say and not as I do: name your arrays with the plural form (because it has a bunch of things in it, like `colorful_bugs` instead of `colorful_bug`) and be descriptive. No one likes to try and figure out what `array1` or `a` contains... stick with `colorful_bugs`. I just kept them short here because they're tiny examples. Someone should rename them all.

Strings are a lot like arrays... so much so that we can even **Convert an Array into a String**! Just use `.join` and tell it what, if anything, you want in between each element (the "separator"):

```
> ["he", "llo"].join
=> "hello"
> colorful_bugs = ["caterpillar", "butterfly", "ladybug"]
=> ["caterpillar", "butterfly", "ladybug"]
> "I found a #{colorful_bugs.join(' and a ')} in the yard!"
=> "I found a caterpillar and a butterfly and a ladybug in the yard!"
```

Want to know a cool way to make an array? Create it from a `Range`(which you learned in the first section) and just **Convert it to an Array**:

```
> my_awesome_array = (1..6).to_a
=> [1,2,3,4,5,6]
```

Advanced stuff (you don't need to know this right now): Remember how we could create a new array and fill it up with stuff using `Array.new(5, "thing")`?

`Array.new` also takes an optional argument that is a block and it will run that block every time it needs to populate a new element. Woah! We got a bit ahead of ourselves, but it's a cool feature to have floating in the back of your head.

```
> Array.new(5){|item_index| item_index ** 2}
=> [0, 1, 4, 9, 16]    # It squared each index to populate the array!
```

Exercises

Hashes

Intro and Goals Hashes may be a bit intimidating at first but they're actually pretty similar to arrays. They're basically just containers for data, like arrays, but instead of storing data based on a numeric indices, you use "keys" which can be strings or symbols. This makes hashes more appropriate for storing data with a bit more depth to it.

Thought Questions

- What is a hash?
- What are keys and values?
- How is a hash similar to an Array?
- How is a hash different from an Array?
- What are 3 ways to create a hash?
- What is the hash rocket?
- How do you access data in a hash?
- How do you change data in a hash?
- What are options hashes?
- How do you delete data from a hash?
- How do you add hashes together?
- How do you list out all the keys or values?
- How do you see if the hash contains a key or value?
- What is a set?

Check These Out First

- Treehouse's intro to Hashes video, and don't worry about the awesome_print gem, it's not required.
- Codecademy's section on Hashes for the basics.
- Codecademy's hashes and symbols section to bring together what we talked about in the strings section.

- Go back and do Ruby Monk's Hashes section if you didn't do it during the Web Dev 101 section. Shame on you for not doing it before >:o

A Brief Summary A *Hash* is just a container for data where each piece of data is mapped to a *Key*. The data is called the *Value*. Keys can be either strings or symbols. Values can be anything, just like with arrays. A hash looks almost like an array, but with squiggly braces `{}` instead of hard ones `[]`. There's no order to a hash (unlike an array)... you're accessing your data using strings anyway so it doesn't matter which order they're in.

Make a new hash using several methods:

```
> my_hash = Hash.new
=> {}
> my_hash = {}        # easier way
=> {}
```

You store data in a hash by matching a key with a value. Use the **Hash Rocket** `=>` (not to be confused with the same symbol in our IRB examples which denotes the IRB output) if you're creating the hash, or just index into it like an array using hard brackets `[]`.

```
> favorite_colors = { "eyes" => "blue", "hair" => "blonde"}
=> {"eyes"=>"blue", "hair"=>"blonde"}
> favorite_colors["eyes"]
=> "blue"
```

Change Data in a hash just like an array, by indexing into it and assigning a new value. Unlike an array, to create a new data item, just pretend it already exists and assign it a value:

```
> favorite_colors["eyes"] = "green"       # Changing an item
=> "green"
> favorite_colors
=> {"eyes"=>"green", "hair"=>"blonde"}
> favorite_colors["skin"] = "suburned"    # Adding a new item
=> "sunburned"
> favorite_colors
=> {"eyes"=>"blue", "hair"=>"blonde", "skin"=>"sunburned"}
```

Hashes are useful for lots of reasons behind the scenes, but it should be immediately obvious that you can handle more nuanced data than you can with arrays. How about a dictionary of words? Just store the words as keys and the meanings as values, if you so choose.

You see hashes all the time in Rails, including as a way of passing options or parameters to a method (since they can store all kinds of different things and be variably sized), and these are often called **Options Hashes**. Methods are often defined along the lines of `def method_name arg1, arg2, arg3, options_hash`, allowing the user to specify any number of different parameters for that method.

Note that, when calling a method, if a hash is the last argument you are entering, *you can skip the squiggly braces*. It's convenient, but can be a real head-scratcher for beginners who are trying to read code and wondering why there are methods being called with strangely mixed inputs and no braces:

```
> some_object.some_method argument1, argument2, :param1 => value1, :param2 => value2
```

Or, for a real version in Rails that creates a link on the webpage and can optionally assign it an ID (among other things):

```
> link_to "click here!", "http://www.example.com", :id => "my-special-link"
```

If you recall our discussion from Strings, we use symbols as keys for hashes more often than not.

```
> favorite_smells = { :flower => "daffodile", :cooking => "bacon" }
=> { :flower => "daffodile", :cooking => "bacon" }
```

Delete from a hash by just setting the value to `nil` or by calling the `.delete` method:

```
> favorite_smells[:flower] = nil
=> nil
> favorite_smells
=> {:cooking => "bacon" }             # one deleted...
> favorite_smells.delete(:cooking)
=> "bacon"
> favorite_smells
=> {}                                 # ...and the other.
```

That's all pretty straightforward. What if we want to **add two hashes together**? Just use `.merge`. If there are any conflicts, the incoming hash (on the right) overrides the hash actually calling the method.

```
> favorite_beers = { :pilsner => "Peroni" }
=> { :pilsner => "Peroni" }
> favorite_colors.merge(favorite_beers)
=> {"eyes"=>"blue", "hair"=>"blonde", "skin"=>"sunburned", :pilsner => "Peroni"}       # oka
```

If you want to know what **All the Keys** are (more common) or **All the Values*** are (less common) in a hash, just use the aptly named `.keys` and `.values` methods to spit them out as an array:

```
> favorite_colors.keys
=> ["eyes", "hair", "skin", :pilsner]
> favorite_colors.values
=> ["blue", "blonde", "sunburned", "Peroni"]
```

A simpler kind of hash is called a **Set**, and it's just a hash where all the values are either True or False. It's useful because your computer can search more quickly through this than an array trying to store the same information due to the way it's set up behind the scenes. You'll encounter them in some of the exercises later.

Exercises

Numbers, Strings and Arrays and Hashes Exercises (Paired)

- sort
- reverse

* * towers of hanoi *

Dates and Times

Intro and Goals When you're building a website, you'll inevitably come into contact with dates and times. When was that submitted? Show only posts created after this time. How long has that user been registered?

All languages have conventions for how they keep track of dates and times and, of course, Ruby is no different... just a bit easier than the rest. In general, computers keep track of time in terms of seconds since a specified point in time. Someone decided a long time ago that Time shall begin at midnight on January 1st, 1970, and so that's typically the "0th" second.

Ruby uses the `Time` class to let you work with dates and times, giving you some handy methods to find out about specific parts (like what day of the week it is) and to allow you to display them in a user-friendly fashion. You probably won't need to dive too deeply into this stuff until you start working with Rails but you do need to understand the basics (as laid out below).

Thought Questions

- How do you get the current date and time?
- How do you find just the Year? Month? Hour? Second? Weekday?
- How do you create a `Time` specifically for 12/25/2013?

- How do you find how many days have passed between two `Time`'s?
- What's the difference between UTC and GMT and Local times?
- How would you find out the time that was 100 seconds ago? 10 days ago?
- TODO: Datetime

Check These Out First

- The Ruby Date and Time explanation from TutorialsPoint. No need to memorize all the Time Formatting Directives, just know what they are and where to find them.

A Brief Summary To **Get Current Time** you just create a new `Time` object with no parameters or use `Time.now`, which is the same thing:

```
> Time.new
=> 2013-07-10 17:04:10 -0700
> Time.now
=>2013-07-10 17:04:11 -0700
```

`Time` gives you some handy methods to ask it questions. Almost all of them are very intuitive, so the general rule is "if you think the method should exist, it probably does":

```
> my_time = Time.now
=> 2013-07-10 17:04:10 -0700
> my_time.year
=> 2013
> my_time.month
=> 7
> my_time.day
=> 10
> my_time.wday
=> 0               # the day of the week, starting Sunday
> my_time.hour
=> 17
> my_time.min
=> 4
```

```
> my_time.sec
=> 10
```

`Time` also takes inputs if you want to create a specific time, from year to time zone:

```
Time.new(year, month, day, hour, min, sec, time_zone_offset_from_utc)
```

```
> Time.new(2012,2,14)
=> 2012-02-14 00:00:00 -0800
```

You can add and subtract times just like they were numbers (because, remember, they basically are... just the number of seconds since 1970):

```
> vday = Time.new(2012,2,14)    # Valentine's Day!
=> 2012-02-14 00:00:00 -0800
> vday+3600                     # 1 hour's worth of seconds
=> 2012-02-14 01:00:00 -0800
> xmas = Time.new(2013,12,25)
=> 2013-12-25 00:00:00 -0700    # Xmas!
> ( xmas - Time.now )/60/60/24.to_i
=> 167                          # That's too long...
```

What if you want to display a date in a pretty way, like on your website or for your user's benefit? There are a couple of baked in methods and then a "build-your-own-adventure" way to specify:

```
> nownow = Time.now
=> 2013-07-10 17:37:27 -0700
> nownow.ctime                  # a standard display type
=> "Wed Jul 10 17:38:10 2013"
> nownow.UTC                    # Remove the time zone
=> 2013-07-11 00:38:10 UTC
> nownow.strftime("%Y-%m-%d %H:%M:%S")
=> "2013-07-11 00:38:10"
```

Wait, what were all those `%Y` characters? They just tell the `.strftime` method what components of the `Time` to output and how you'd like them formatted. There's a long list of them back at the TutorialsPoint site. You don't need to remember them since you can just google for them when you decide to output a string, but know that they give you the flexibility to output a date or time in pretty much any way you could imagine.

Extra Stuff: Time Zones and Local Time What's that trailing `-0800` in `2012-02-14 00:00:00 -0800`? It's because that time was created on my local

system, which is many hours "earlier" in the day from the Coordinated Universal Time (called UTC... no, it doesn't match up but here's why) which is used by computers around the world as the standard benchmark time (so two computers communicating about times will always be talking about the same exact one and not have to worry about time zones).

I prefer to think of UTC as "Universal Time Code" because reasons. UTC is the new GMT... Greenwich Mean Time. You'll start thinking of things in terms of "how many hours away am I from England?" when you run into time zone bugs somewhere down the road.

Back to the point, the `-0800` above says that we created a new time for midnight on Valentine's Day but only from the perspective of someone on the West Coast of the USA... it was really 8am in Greenwich, England and according to every other computer in the world. You'll forget this stuff until you need it and that's fine.

Use `.localtime` to display the `Time` object in whatever your local system time is (if it was created in UTC it will be different).

Exercises (in IRB)

- How many days until your birthday?
- How many days, hours, minutes, and seconds until Christmas?
- Display `"January 10, 2001 10:00AM"` in IRB

Miscellaneous Issues

Intro and Goals Here are some useful things that don't really fit nicely into another section.

Thought Questions

- What is `nil`?
- How do you check if something is `nil`?
- What's the difference between `nil` and `blank` and `empty`?
- Are the following `nil` or `empty`?
  - `" "`, `""`, `[]`, `[""]`, `{}`
- What's the difference between `puts` and `p` and `print`?
- What does `.inspect` do?
- What do `+=`, `-=`, `*=` and `/=` do?
- What is parallel assignment?
- What's the easiest way to swap two variables?

Check These Out First

- Nil vs Empty vs Blank
- p vs puts in Ruby

A Brief Summary So **What is `nil`?** It represents nothing... literally. Before you assign a value to something, it starts as `nil`, for instance an item in an array or a variable:

```
> my_arr = []
=> []
> my_arr[3]
=> nil          # hasn't been assigned yet
```

Sometimes you want to know if a value or variable is `nil` before doing something (because otherwise the operation would throw a bunch of errors at you). Use the method `.nil?` to ask whether it's nil or not beforehand.

```
> nil.nil?
=> true
> [].nil?
=> false        # Waitasecond....
```

Why is `[]` not nil? The array itself exists so it isn't nil... it just happens to contain no values yet so it's empty. If we asked for the first value of that array using `[][0].nil?`, that would be `true`.

If you try to run a method on something that is `nil`, which you will inevitably do many many times by accident, you'll get the familiar `NoMethodError`:

```
> user_i_looked_up_but_was_not_found_so_is_nil.empty?
=> NoMethodError: undefined method `empty?' for nil:NilClass
```

`.blank?` and `.empty?` are similar – both basically check if the object has nothing in it – but `.blank?` will also ignore any whitespace characters. *Note that `.blank?` is a method provided by Rails and is not available in Ruby.*

We've seen lots of `puts` so far but you've probably also run across `p`. **What's the Difference?** `p` will give you some more information because it runs the `.inspect` method on the object while `puts` runs the `.to_s` method. `.inspect` is meant to be informative where `puts` is "pretty". The difference may not be readily apparent while you're only working with simple objects like strings and arrays, but you'll notice it when you start creating your own objects and you want to see what's inside (without typing out `puts my_object.inspect`).

`=` is an **Assignment Operator** but there are a few others that are interesting and common shorthands as well: * `a += b` is the same as `a = a + b` * `a -= b` is the same as `a = a - b` * `a *= b` is the same as `a = a * b` * `a /= b` is the same as `a = a / b` * `a %= b` is the same as `a = a % b` * `a **= b` is the same as `a = a ** b`

Parallel Assignment is when you assign the values of more than one variable at a time (though it works for arrays as well!):

```
> a, b = 1, "hi"
=> [1, "hi"]      # ignore this output
> a
=> 1
> b
=> "hi"
> my_array = [1,2,3,4]
=> [1,2,3,4]
> my_array[1], my_array[3] = 100, 200
=> [100,200]      # ignore
> my_array
=> [1,100,3,200]
```

It's also a great way to **Swap Two Variables**:

```
> a = 10
> b = 20
> a,b = b,a
> a
=> 20
> b
=> 10
```

Exercises

Tutorial

Exercises (Paired) >> PUT LATER AFTER METHODS / BLOCKS!

- Rewrite the following methods from Enumerable. You may not use the real method... which should be obvious. You MAY use #each in the following methods once you've created it yourself the first time:

- **each**

  each_with_index

- **map**

  select

*

Conditionals and Flow Control

Intro and Goals Now you've got an understanding of what tools you can use and it's time to start thinking about how the Ruby interpreter moves through your code. Sometimes you want to execute a certain chunk of code, other times you don't. In this section, you'll see the different ways of controlling the flow of your program.

Thought Questions

- What is a "boolean"?
- What are "truthy" values?
- Are `nil`, `0`, `"0"`, `""`, `1`, `[]`, `{}` and `-1` considered true or false?
- When do you use `elsif`?
- When do you use `unless`?
- What does `<=>` do?
- Why might you define your own `<=>` method?
- What do `||` and `&&` and `!` do?
- What is returned by `puts("woah") || true`?
- What is `||=`?
- What is the ternary operator?
- When should you use a `case` statement?

Check These Out First

- Github Gist on Truthiness
- Do the Codecademy Control Flow Course (all sections)
- See these answers on the Spaceship Operator

A Brief Summary You'll need to understand which types of things Ruby considers "true" and which ones it considers "false". **"Truthiness" and "Falsiness"** are ways of saying "what evaluates to true?"" and "what evaluates to false"? In many languages, there is some nuance to that question. In Ruby, it's simple: `nil` and `false` are false and that's it. Everything else is "truthy".

An `if` statement is pretty straightforward too. Supply a condition and, if it's true (or truthy!), the code will be executed. If you supply an `else` clause, then that code is executed otherwise. Everything proceeds normally from that point forward. Remember that the parentheses are implicit.

```
if some_variable.is_a?(String)
    # do some code if some_variable is a string
else
    # this code will not run unless the variable is NOT a String
end
```

`unless` is the opposite of `if` (which should make sense from the english of it). So it will jump into the included code... UNLESS the statement is `true`. You can also have an `else` clause, though it's less common because then you need to scratch your head to think about it so your code gets a bit less readable.

```
unless home_team.won_the_super_bowl?
    puts "I need to drown my sorrows in ice cream"
end
```

Want some more Ruby awesomeness? Put your `if` and `unless` statements in a single line, and in any order you want:

```
if current_user.is_a?(Vampire) dispatch_vampire_hunters
```

... works just as well as:

```
dispatch_vampire_hunters if current_user.is_a?(Vampire)
```

Use **Comparison Operators** as the building blocks to construct your conditional statements. There are some simple ones that you should already be familiar with: `==, <, >, >=, and <=`. `!=` is "not equal".

The **Spaceship Operator** `<=>` is a special one that comes up because it actually gives three different possible outputs depending on whether the left side is greater than, less than, or equal to the right side.

```
> 1 <=> 1000
=> -1
> 1 <=> 1
=> 0
> 1 <=> -1000
=> 1
```

The Spaceship can be useful because, like basically everything else, it's actually a method and you can override it in your own classes. It's most commonly used in sorting methods. Imagine that you created a `Person` class and you wanted to sort an array of `Person` objects. You first have to teach Ruby how to compare two `Person`s by defining the `<=>` method for the `Person` class:

```
def Person
  def <=> (other_person)  # to compare two people, use last names
    self.last_name <=> other_person.last_name
  end
end                 # now we can run people_array.sort, woohoo!
```

Logical Operators go one step beyond simple comparisons and let you start chaining together several comparisons into a single statement. That lets you build more interesting and complex `if` statements. The most common are: * `&&` aka `and`, meaning both sides must be true for the full expression to evaluate to true * `||` (the pipe symbol, usually on the same key as the backslash) aka `or`, meaning that if EITHER of the two sides is true, the expression is true (else false) * `!` aka `not`, which reverses the expression from true to false or false to true

"In the wild" you'll probably see some complex or odd looking `if` statements. The trick is to start breaking everything that looks like a conditional piece into what is evaluates to... either `true` or `false`. So what do you evaluate first? Ruby logical expressions use a similar order of operations to normal math: left to right unless there are parentheses.

```
> ( false || true ) && !(true && true )
=> false
```

Ruby is Lazy which means two things here: 1. It will only evaluate far enough to determine that the expression is definitively true or false.
2. It will return whatever is returned by the last part of the expression to get evaluated (instead of just a simple `true` or `false` it relies on that returned thing being truthy or falsey).

That's important because we can actually use methods as part of our logical chains. It means that methods on the left side of the expression get executed but the ones on the right may never get executed at all... a fact that many programmers (including you later) utilize to make their code nice and compact.

```
> puts("this isn't important") || puts("THIS IS IMPORTANT!!!")
"this isn't important"
=> nil
```

What happened? Ruby knows that it only needs one side of the `&&` to be false for the whole thing to be false, and since `puts` on the left already returned `nil` which is falsey, Ruby stopped evaluating the expression and the really important message never got displayed.

It returned `nil` instead of `false` because `&&` and `||` and the like don't just return `true` or `false`... they return the result of the last expression to get evaluated, which was the `nil` from the `puts`. For another example, `7 || nil` returns `7` (which behaves truthy) instead of `true` and `7 && nil` returns `nil` (which behaves falsey) instead of `false`. If this seems a bit much to swallow right off the bat, keep it in the back of your mind until you first see it in action then it will click.

`||=` is a sneaky expression that takes advantage of Ruby's natural laziness – it basically expands to `thing_a || thing_a = thing_b`. So if `thing_a` hasn't yet been assigned to anything, it is `nil` so the Ruby checks the right side of the || to see if that might be true.becomes `thing_b`. If it has already been assigned a value, it just keeps that value like normal. This is another sneaky trick used by programmers in situations like when you want to , you should be able to see that

You may have seen some oddly compact and strange looking statements that appeared to be `if` statements under the hood. That's probably because they use the **Ternary Operator**, which is a shorthand notation for a simple `if` that separates the different parts using the `?` and `:` like:

```
condition ?  do_this_if_true :  do_this_if_false
```

So:

```
> true ? puts "I like truth" : puts "not gonna happen"
"I like truth"
=> nil
```

You can nest `if` statements inside one another but sometimes it gets a little crazy and you find yourself 6 levels deep (and probably needing to rethink your strategy). For those situations where you're really just checking to see if something equals any one of a number of clear but different options, a `case` statement can be a good substitute. It basically lets you construct a chain of logic that says "if `x` equals `option_a`, do this, if it equals `option_b`, do this, if it equals `option_c`, do this... and otherwise do this."

```
case current_user.energy   # Assume it's an value 1-3
```

```
when 3
    puts "Go run a marathon!"
when 2
    puts "Go for a walk."
when 1
    puts "Go take a nap"
else
    puts "You're only supposed to have energy of 1,2 or 3..."
```

Additional Resources

- ||= on Stack Overflow

*

Exercises

1. TODO: What will these return? (check using IRB)
    - true/false
    - actual returns

Iteration

Intro and Goals You can assemble code, tell the program which parts of it to execute, and wrap it all up in a method. There's still something missing... what if you want to make something happen a whole bunch of times? You certainly don't just run the method again and again manually. Luckily we've got several standard ways of iterating through a piece of code until we tell the program to stop.

You should understand the basic iterators `for` and `while` and understand how to use `.each` and `.times`. We'll talk more about blocks and the other Ruby iterators like `.map` and `.select` in the next sections, so it should be more obvious how `.each` and `.times` work after reading that.

Do This First

- Codecademy's loops track
- Codecademy's loops 'project'
- If you want a bit more, check out Skork's entry on Ruby looping and iterators

Thought Questions

- What does `loop` do?
- What are the two ways to denote a block of code?
- How do you print out each item of a simple array `[1,3,5,7]` with:
  - `loop`?
  - `while`?
  - `for`?
  - `.each`?
  - `.times`?
- What's the difference between `while` and `until`?
- How do you stop a loop?
- How to you skip to the next iteration of a loop?
- How would you start the loop over again?
- What are the (basic) differences between situations when you would use `while` vs `times` vs `each`?

A Brief Summary `loop` is the most basic way to loop in Ruby and it's not used all that much because the other ways to loop are much sexier. `loop` takes a block of code, denoted by either `{ ... }` or `do ... end` (if it's over multiple lines). It will keep looping until you tell it to stop using a `break` statement:

```
> loop { puts "this will not stop until you press CTRL+c" }
this will not stop until you press CTRL+c
this will not stop until you press CTRL+c
... and so on

> i=0
> loop do
>   i+=1
>   print "#{i} "
>   break if i==10
> end
1 2 3 4 5 6 7 8 9 10 => nil
```

`while` performs a similar function but in a much more compact way, by allowing you to specify the condition that must be true to keep looping, and you'll find yourself using it much more in your own code. It doesn't actually take a formal block of code, just runs everything until it reaches its `end`. Just remember to declare the variable(s) you'll be using (or they'll get reset with each iteration) and to increment at some point (or you'll get stuck in an infinite loop [use ctrl+c to break in Terminal]):

```
> i=1
> while i < 5
>   print "#{i} "
>   i+=1
> end
1 2 3 4 => nil
```

`until` is almost identical to `while` but, instead of running as long as the specified condition is `true`, it runs as long as the condition is `false`.

`for` is a looping mechanism present in lots of other languages but it gets de-emphasized in Ruby and you don't see it used much. A common use is to loop over every number in a range. Whatever you name the first variable is how you call that number inside the loop:

```
> for a_number in (1..3)
>   print "#{a_number} "
> end
1 2 3 => 1..3
```

Things get more interesting when you realize that most of your loops will probably involve looping over each element in either an array or a hash of items. Ruby knows this and made it super easy for you by specifying the `.each` method that you call directly on that array or hash. It will automatically pass whichever item it is currently on into your code block. That item will be named whatever name you specify inside the pipes `| name_goes_here |`:

```
> guys = ["Bob", "Billy", "Joe"]
> guys.each do |current_name|    # better to call it just "name"
>   print "#{current_name}! "
> end
Bob! Billy! Joe! => ["Bob", "Billy", "Joe"]  # returns original array
```

Many other loops are just you trying to do something a certain number of times (which was the case in our `for` loop example). In that case, Ruby has the simplest possible method for you, `.times`. If you pipe in an argument, it will be the current iteration starting from zero:

```
> 5.times do |jump_num|
>     print "Jump #{jump_num}!"
> end
Jump 1!Jump 2!Jump 3!Jump 4!Jump 5!=> 5
```

A couple other methods with similar purposes to `times` that you see less frequently: * `.upto`, as in `1.upto(4) { |current_number| ...some code...`

}, just like `.times` but you choose the starting and ending point instead of always starting at zero. * `.downto`, similar to `upto` but... down... to....

Your best friends early on will be `while` for anything that needs to run until a certain condition is reached (like winning the game), `each` for any time you want to do stuff with every item in an array or hash, and `times` for the simple cases when you just want to do something a fixed number of times.

Because you may want some additional control over your loops, use these statements to jump in and out of them for certain abnormal conditions: * `break` will **stop the current loop**. Often used with an `if` to specify under what conditions to do that. * `next` will **jump to the next iteration**. Also usually used with an `if` statement. * `redo` will let you restart the loop (without evaluating the condition on the first go-through), again usually with some condition * `retry` works on most loops (not `while` or `until`) similarly to `redo` but it *will* re-evaluate the condition before running through the loop again (hence *try* instead of *do*). * NOTE: Do NOT use `return` to exit a loop, since that will exit the whole method that contains it as well!

Additional Resources

- A lecture by Avi Flombaum on iteration in Ruby that shows you the nuts and bolts of it (esp. starting minute 16), including some of Khan Academy's new visualizations. Solid stuff.

Exercises

Writing your Own Methods

- call
- self
- scripts
- running scripts on the command line

Blocks, Procs, and Lambdas

Intro and Goals One of the most confusing parts of learning basic Ruby is understanding what blocks are and how they work, mostly because it's something you probably haven't ever seen before. It shouldn't be, because they're actually pretty simple. You've already seen them before, whether it's using them yourself during the prep work or most recently in the section on iteration, where they came up as inputs to some of the iterators.

Here, you'll learn more about blocks and also about their lessor known cousins, procs and lamdas. By the end, you should be comfortable working with blocks and writing your own methods that take them.

Do These First

- Codecademy's Methods and Blocks path
- Codecademy's deeper dive into Blocks path

Thought Questions

- How is a block like a function?
- How is a block different from a function?
- What are the two ways to declare a block?
- How do you return data from a block?
- What happens if you include a `return` statement in a block?
- Why would you use a block instead of just creating a method?
- What does `yield` do?
- How do you pass arguments to a block from within a method?
- What is a proc?
- What is a lambda?

A Brief Summary Blocks are just chunks of code that you can pick up and drop into another method as an input. They're often called anonymous functions because they have no name but behave just like functions. They're like helper functions... you don't find blocks just hanging around without some method (like `.each`) using them.

You **declare a block** using squiggly braces `{}` if it's on one line or `do ... end` if it's on multiple lines (by convention... you can use either one if you really want):

```
> [1,2,3].each { |num| print "#{num}! " }
1! 2! 3! =>[1,2,3]
> [1,2,3].each do |num|
>    print "#{num}!"
> end
1! 2! 3! =>[1,2,3]         # Identical to the first case.
```

Just like methods, some blocks take inputs, others do not. Some return important information, others do not. Blocks let you use the implicit return (whatever's on the last line) but NOT `return`, since that will actually return you from whatever method actually called the block.

Blocks are used as arguments to other functions (like `.each`), just like the normal arguments that you see between the parentheses... they just happen to always be listed last and on their own because they tend to take up multiple lines. Don't

think of them as anything too special. The `.each` function is built to accept a block as an argument.

How does `.each` take a block then? Through the magic of the `yield` keyword, which basically says "run the block right here". When you write your own methods, you don't even need to specially declare that you'd like to accept a block. It will just be there waiting for you when you call `yield` inside your method. `yield` can pass in parameters to your block as well. See this version of the `.each` method to get an idea of what's happening under the hood:

```
def my_each
    i = 0
    while i < self.size
        yield(self[i])
    end
    self
end
```

As you can see, we iterate over the array that our `my_each` method was called on (which can be grabbed using `self`). Then we call the block that got passed to `my_each` and pipe in whatever member of the original array we are currently on. Last, we just return the original array because that's what `each` does. We would run it just the same way as `each`:

```
> [1,2,3].my_each { |num| print "#{num}!" }
1! 2! 3! => [1,2,3]
```

which operates just like: PASTE CODE

You've seen them used as inputs to `.each`

yield stuff, pass the argument in #### Blocks are Very Ruby-ish

Ways to declare a Block

Block Return Statements

Where Blocks are Used

Examples

Procs

Additional Resources

- Procs, Lambdas and Closures in Ruby by Peter Cooper (video)
- Blocks vs procs vs lambdas free screencast
- Understanding Blocks Procs and Lambdas
- Blocks explained by Alex Chaffee (video)

Which of these resources were most helpful to getting you that "aha!" moment?

Exercises

Enumerable

Intro and Goals You've learned about `Array` and `Hash` but only got half the story... they each have their own methods for adding and deleting and accessing data but what makes them really powerful in Ruby is the ability to use `Enumerable` methods as well as the basic ones you've just learned.

"Enumerable" is actually a `module`, which means it is just a bunch of methods packaged together that can (and do) get "mixed in", or included, with other classes (like `Array` and `Hash`. That means that Ruby programmers don't have to write all those methods many different times - they just write them once, package them up as `Enumerable`, and tell `Array` and `Hash` to include them.

In this case, `Enumerable` contains really useful methods like `map` and `each` and `select` that you'll use again and again so our goal with this section is to get well acquainted with them. You'll need to start becoming familiar with code blocks as well, which are used by all these methods.

Thought Questions

- What is a module?
- Why are modules useful?
- What are blocks?
- What does a block return?
- What does `each` do?
- What does `each` return?
- What does `map` do?
- What does `map` return?
- What is the difference between `map` and `collect`?
- What does `select` do?
- What does `select` return?

Check These Out First

- Codecademy's section on iterating over Arrays and Hashes

A Brief Summary `Enumerable` gives you lots of useful ways of doing something to every element of an array or hash, which is a very common type of need when you're building programs.

What if I want to keep only the even numbers that are in an array? The traditional way would be to build some sort of loop that goes through the array, checks whether each element is even, and starts populating a temporary array that we will return at the end. We haven't covered iterators yet, but it might look something like:

```
my_array = [1,2,3,4,5,6,7,8,100]
#

TOO ADVANCED?  PUT IN LATER SECTION???
```

Exercises

Writing Your Own Methods

Intro and Goals

Check These Out First

A Brief Summary

Additional Resources

Exercises

Style

https://github.com/bbatsov/ruby-style-guide

Projects

- Running Ruby scripts from the command line

Intermediate Ruby

Classes, inheritance (use .methods), OO design, refactoring, naming, recursion, scope, regex, Modules, metaprogramming (reflection?) http://www.codecademy.com/tracks/ruby for oop

Ruby and the Web

File I/O and Serialization

Testing with RSpec and Test Driven Development

http://guides.rubyonrails.org/testing.html https://www.relishapp.com http://betterspecs.org

Basic Data Structures

Basic Algorithms

Final Projects

Finish

Additional Resources

- Why Ruby and Python are different