Week 1 Notes

Installing Ruby (and Rails)

Mac

On a Mac, ruby comes pre-installed, but you'll really want to update it to the latest version (Ruby 1.9.3) and install a few other programs we'll need. The easiest way to do that is to follow the instructions below for your version of OS X:

- Mountain Lion: Link (Skip XQuartz and all of step #4 'Install MongoDB').
- Lion and earlier: Link (Skip the last section, 'Create an Example Rails App').
  - You'll also want to install homebrew, follow the instructions at the bottom of the page here

Windows

On Windows, the best way to get started is to install using the RailsInstaller.

Command Line

General Concepts

The command line (or 'terminal') is an inteface for issuing commands to your computer, and viewing the results. It is entirely text-based.

Commands are almost always a single line in length. When you hit 'enter' after typing a command, the computer runs the command, and any output is displayed.

Commands consist of the *command*, followed by any *arguements* that modify the commands behavoir. For example, on a Mac/Linux, I can issue the following two commands

- **ls**
- **ls -a**

In both cases, I'm running the command **ls**, which lists the files in the current directory. But in the second case, I've added the arguement *-a*, which tells **ls** to show all files, even hidden ones.

When you're using the command line, your always working in the context of a directory on your computer. In example above, that's important, so my computer knows which directoy to list the contents of. If I want to see the contents of a different directory, I need to *change directory* (see command list below)

Common Commands (Mac / Win / Linux)

- **cd *some_directory*** - change current working directory to *some_directory* (which must be in the current directory)
- **cd ..** - '..' has a special meaning, the parent directory of the current one. So this takes you up one level.
- **mkdir *new_folder_name*** - make a new directory in the current directory

Mac/Linux Specific

- **pwd** - list the current directory
- **ls** - list files and folders in the current directory
- **rm *some_file*** - deletes *some_file* (add *-r* to delete folder, i.e. **rm -r *some_folder***)

Windows Specific

- In windows, just look at the command prompt to see the current directory you're in.
- **dir** - list files and folders in the current directory

A note on IDEs

There are programs availabe called Integrated Development Environments (IDEs) which make it easy to manage and run your code. A popular one is *RubyMine*. It's ok to use these, but I've chosen not to highlight them in the course, becuase most people in the ruby community write how-to's assuming you're using a command line, and it's an important skill to master.

Text Editors

If you have a favorite text editor to program in, feel free to keep on using it.

If you don't have a favorite text editor, I recommend Sublime Text 2.

Runing ruby programs

When writing your programs, make sure to save it as a file ending in .rb (i.e. lolcat.rb), and in a convenient directory. It's best not to use spaces in your folder or file names, as these spaces are difficult to deal with in the command line.

To run the program, *cd* to the apporpriate directory and use the *ruby* command on your file: cd path/to/code ruby lolcat.rb

If you want to play with Ruby without saving and running files, you can use *irb* (interactive ruby) which lets you type ruby commands and see the results instantly.

If you start irb, you have to exit before you can run any 'standard' terminal commands such as 'ls', 'cd', or 'ruby filename.rb'. To exit irb, type 'exit'.

Ruby Basics

Much like the command line, programs in Ruby are made of **statements**, which are (usually) one line long. Statements are like sentences, in that they are composed of nouns, which represent things, and verbs, which perform actions.

In ruby, we call nouns **objects**, and we call verbs **methods** (sometimes called **functions**).

Objects

The term *objects* carries quite a bit of meaning in programming (which we'll cover later). But for now, it's enough to know that objects represent *things* or *information* in our programs. Examples of objects are **numbers** and **strings** of text.

Numbers

In Ruby, like in other programming languages, we have two common types of numbers, integers and decimals.

Here are some integers

```
1
9129312312
-2
0
87
```

And here are some decimals:

```
1.1
2.000
9999.9999
15874.8493
```

Numbers work pretty much like we'd expect. For example, create a file called `numbers.rb` and make it something like this:

```
puts 1+1
puts 100 - 10
puts 8 * 8
puts 40 / 5
puts 2.2 + 3.2
```

Save the file and run it using `ruby numbers.rb`, and you should get output like this:

```
2
90
64
8
5.4
```

The `puts` command

You may be wondering what that `puts` part of each statment means. We'll talk about methods more later, but puts is a method that outputs information to the screen when you run the program. Without the `puts`, ruby would run all the calculations in the program, but we wouldn't see the results.

Strings

Strings are bits of text. To make a string, we put the text in quotes: **"This is a string of text."**

Strings can be joined together with the '+' method (also called **concatenate**). For example:

```
"Elihu " + "Yale"
=> "Elihu Yale"
```

(The `=>` indicates the result returned by the statement previous statment.)

Ruby gives us lots of built-in methods to work with strings. Here are just a few:

```
"How now, brown cow?".reverse
=> "?woc nworb ,won woH"

"How now, brown cow?".upcase
=> "HOW NOW, BROWN COW?"

"How now, brown cow?".downcase
=> "how now, brown cow?"

"How now".length
=> 7
```

Mixing Numbers and Strings

When we're working with text, it's common to mix letters and numbers, so it's important to understand how Ruby treats them in combinaton.

Here are a few examples, which work as we'd expect:

```
puts 10 + 10
=> 20

puts "10" + "10"
=> "1010"

puts "10 + 10"
=> "10 + 10"
```

But what if we try to mix them?

```
puts "10" + 10
#=> TypeError: can't convert Fixnum into String
```

In this first case, Ruby sees `"10" + ___` and expects the second object to be a string as well. When it sees a number instead, it complains.

What about the opposite order?

```
puts 10 + "10"
#=> TypeError: String can't be coerced into Fixnum
```

Similarly, in the second case, Ruby sees `10 + ___` and exects the second object to be a number as well. When it sees a string instead, again, it complains.

Note the errors are similar, but not exactly the same.

But you may be thinking, 'Aren't computers pretty smart? Shouldn't Ruby be able to see `"10" + 10` and know I want `20`? Well, Ruby can convert strings to numbers and vice versa, but the trick is the ambiguity. In this case, we might just have well expected `"10" + 10` to equal `1010`.

To fix this, we have to tell Ruby exactly what we want, but converting the numbers to strings, or the strings to numbers.

To do this, we use the methods `to_i` or `to_s` to convert objects to **integers** or **strings**, respectively.

```
puts "10" + 10.to_s
=> "1010"
```

or

```
puts 10 + "10".to_i
=> 20
```

Methods

We've already encountered a few methods, including `puts`, `reverse`, `length`, `to_i`, `to_s`, but also `+`, `-`, and the other math methods.

All methods are ways of taking action. In the above examples, Ruby came with those methods built-in, but as we'll see, we can also create our own custom methods.

Calling Methods

Just like it doesn't make sense to have a verb without a noun, all methods have to be *called* (or take action) on a specific object.

Most commonly, a method is explicitly called on an object, using the following syntax:

```
some_object.some_method
```

In this case, we can think about the method as being a part of the object. It's written specifically for that type of object. For example, the `reverse` method was designed for strings, so we can call `.reverse` on any string. But we can't call `5.reverse` because numbers don't have this method.

But what about `puts`? We don't seem to call `puts` on anything?

The truth is, if we don't specificy what to call a method on, Ruby assumes we're calling it on whatever object we're *inside*. That concept is a bit advanced, so we'll have to get to it later. But for now, know that methods like `puts` are called on something, we just don't see it.

Arguments

Methods can take *arguments*, which let us pass additional information to the method, which, for example, might to tell it how to do its work.

For example, consider the the `+` method for numbers. While we often write something like `7 + 5`,that's just a special form Ruby gives us for convenience.

In reality, we're calling the `+` method on the first number, or in other words `7.+`. But what about the second number? That's actually an *argument* to the addition method that tells it what exactly to add to its object.

So, another way to write this would be:

```
7 + 5
# is the same as
7.+ 5
```

Ruby gives us flexibility in how we indicate the arguments we're passing to a method. We can also put the arguments in `()` when we need to be more explicit:

```
7.+(5)
```

Methods can take more than one argument, such as the `insert` method for strings:

```
"abcd".insert(0, 'X')
=> "Xabcd"
```

As we can see, we need to tell this method where to insert the letter, and what letter to insert. (Ruby, like almost every other programming language, starts counting at 0, not 1).

Variables

In programming, we often want to store objects so we can use them again later. We can do just that with *variables*. When we want to store something in a variable for use later, we *assign* that object to the variable, using the `=` operator.

To keep track of things, we have to give each variable a name. Names have to start with a lowercase letter, but after that, we can include any letter, number, or '_'.

Here are some examples of storing things in variables:

```
my_favorite_dog = "Lassie"
the_7th_guest = "Alice"
number_of_hot_dogs_in_a_costco_sized_pack = 57
```

Variables can store any object, but they can't point to other variable. If we assign one variable to another, we are actually assigning the *value* of the second variable to the first.

```
a = 100
b = 200
a = b
```

At this point, both a and b are storing the number 200. But what happens if we store something else in `b`?

```
b = 999

puts a
=> 200
puts b
=> 999
```

Once we store an object in a variable, we can use that variable anywhere we might use the object it represents.

```
my_name = "Adam"
greeting = "Hello, " + my_name
puts greeting
=> "Hello, Adam"

one_hundred = 100
puts 50 + one_hundred
=> 150
```

Comparisons and Control

In programming, we often need to compare things, and make decision in our program based on the result of the comparison.

Comparisons

Ruby gives us a number of methods to compare objects. We have the common greater-than or lesser-than comparison:

```
puts 1 < 2
=> true
puts 1 > 2
=> false
```

`==` tests if two objects are equal, `!=` tests whether they are unequal:

```
puts 1 == 2
=> false
puts 1 == 1
=> true

puts 1 != 1
=> false
puts 1 != 2
=> true
```

These also work on strings:

```
puts "xbox" == "xbox"
=> true
puts "xbox" == "PS3"
```

The `<` and `>` compare strings alphabetically, as in a dictionary:

```
puts "apple" < "pear"
=> true
puts "Xena" > "Buffy"
=> true
```

Branching

The `if` statement allows us to run different code, depending on whether an expression is `true` or `false`. Here's a simple example:

```
puts "What is your name?"
name = gets.chomp
if name == "Adam"
    puts "Hello, my friend!"
else
    puts "Hello, stranger!"
end

puts "This line is run after everything above, no matter what."
```

The part immediatly after the `if` is the expression that is evaluated. If it is true, then the code in the first section is run. If it's not true, then the code in the second section is run instead.

The `end` tells Ruby when we're done runnign code depending on the results of the comparison.

Ruby doesn't care about the indentations, but they make our code much easier to read, so you should use them.

true and false

`true` and `false` are special in Ruby, they are actually each their own objects.In other words, they are not the same thing as the *strings* "true" and "false".

In Ruby, we don't have to make an explicit comparison (such as `1 < 2`) to get a true/false value.

If we put an object (string, number, etc) in a place where ruby expects a true/false value, it will convert it to true/false using a simple rule:

the `false` and `nil` objectsare both false, everything else is true

Here is a (very) incomplete list of objects that evaluate to true in Ruby:

```
true
"true"
1
99
0
"dog"
"false"
```

That last one may be confusing, but remember that the string "false" is not the same as the `false` object.

Looping

We often want to repeat the same task over an over again. Loops make this possible (and dare I say fun!)

Of course, computers are really dumb (shhh don't tell them), so we have to tell them when to stop repeating, or they'll just keep going like the Energizer Bunny.

We start a loop with the `while` command, followed by an expression. As long as the expression is `true` the loop will run another time.

Here's an example:

```
number_of_doughnuts = gets.chomp
while number_of_doughnuts < 5
    puts "Keep eating!"
    number_of_doughnuts = gets.chomp
end
```

However, we can use the `break` command to make this even simpler. `break` will exit out of the loop we're in, to the line right after the `end`.

```
while true
    number_of_doughnuts = gets.chomp
    if number_of_doughnuts < 5
        puts "Keep eating!"
    else
        break
    end
end
```