Learning Ruby and Rails

Good morning!

Outline of the Day

- 10:00-11:30 **Introduction to Ruby**
- 11:30-Noon **Ruby Koans**
- Noon-1:00 **Lunch**
- 1:00-2:45 **Rails**
- 2:45-3:00 **Break**
- 3:00-4:30 **Rails part 2**
- 4:30 **What Next?**

What We'll Cover

- Basic concepts of programming
- Syntax and simple operations of Ruby
  - Objects, classes, and methods
- An introduction to Rails
  - How to use Rails machinery to manipulate application data
  - Data validation in Rails
  - Displaying data in Rails
  - Using scaffolding to start a simple application

What Is Programming?

- Programming is the act of writing programs
- A program is a precise, text-based, description of how to perform a **computation** on **data**
- Data
  - Numbers
  - Strings: pieces of text
  - Lists of things
  - Booleans: true and false
- Computations
  - Add 3 and 4
  - Find the largest number in a list of numbers
  - Contact Trimet's network and find the most convenient bus route

What Is a Programming Language?

- A human readable format for describing computations
- Any language can describe the same computations
- Different languages are really just a different vocabulary

Why Ruby?

- It’s easy to learn
- Portland has a strong Ruby community
- Rails is a popular web framework built on Ruby
- Rails gives Ruby a good, simple, vocabulary for web development

Starting Ruby

Open a terminal window. Type “irb” and hit enter.

Try entering a few things:

```
1
4 + 3
"hat"
[1, 2, 3, "happy"].each { |i| puts i }
```

>> is your prompt to enter text and => will appear at the start of each response.

Use the up and down arrow keys to navigate back and forth through the previous commands you’ve entered.

Basic Ruby Data

Numbers `1, 2, 3`
Strings `"chicken"`
Booleans `true, false`
Symbols `:chicken`
Arrays `[1, 7, "cake"]`
Hashes `{:chicken => "chicken", :sillyexample => :chicken}`

Type these at the command line! What happens?

That's the data, where are the computations?

Basic Ruby Computations

```
2 + 2
3 * 3
"chicken".reverse
"chicken".length
2 + 2 == 4
2 + 2 == 5
[1 , 2, 3].include?(2)
```

Basic Ruby Computations

```
2 + 2                    4
3 * 3
"chicken".reverse
"chicken".length
2 + 2 == 4
2 + 2 == 5
[1 , 2, 3].include?(2)
```

Basic Ruby Computations

```
2 + 2                    4
3 * 3                    9
"chicken".reverse
"chicken".length
2 + 2 == 4
2 + 2 == 5
[1 , 2, 3].include?(2)
```

Basic Ruby Computations

```
2 + 2                    4
3 * 3                    9
"chicken".reverse        "nekcihc"
"chicken".length
2 + 2 == 4
2 + 2 == 5
[1 , 2, 3].include?(2)
```

Basic Ruby Computations

```
2 + 2                   4
3 * 3                   9
"chicken".reverse       "nekcihc"
"chicken".length        7
2 + 2 == 4
2 + 2 == 5
[1 , 2, 3].include?(2)
```

Basic Ruby Computations

```
2 + 2                   4
3 * 3                   9
"chicken".reverse       "nekcihc"
"chicken".length        7
2 + 2 == 4              true
2 + 2 == 5
[1, 2, 3].include?(2)
```

Basic Ruby Computations

```
2 + 2                   4
3 * 3                   9
"chicken".reverse       "nekcihc"
"chicken".length        7
2 + 2 == 4              true
2 + 2 == 5              false
[1, 2, 3].include?(2)
```

Basic Ruby Computations

```
2 + 2                   4
3 * 3                   9
"chicken".reverse       "nekcihc"
"chicken".length        7
2 + 2 == 4              true
2 + 2 == 5              false
[1, 2, 3].include?(2)           true
```

What do you think `"chicken".reverse.length` does? What about `"puppy".include?('p')`?

That's Nice But…

- How do we combine steps or use the result of an action?
- Variables name particular values
- The same variable can be set to different values at different times

```
var1 = 4
var1
var2 = "chicken"
var2
var2 = var1
var2
```

Interlude: String Interpolation

Something you might want to do is print out a string that has a value contained within it.

For example, try typing

```
"Our array is #{[1,2,3]}"
```

Most useful when dealing with variables!

```
var = [1,2,3,4,5,6]
"Our array is #{var}"
```

Now back to variables!

Variables

Try typing the following, noting what happens at each step

```
thing = "chicken"
thing
thing.reverse
thing
thing = thing.reverse
```

Variables

- `thing.reverse` didn't permanently reverse the string!
- We had to **set** the value with another assignment statement
- Some functions do change state

Try

```
awesomelist = [5,2,1,8]
awesomelist.sort!
awesomelist
```

How did that happen? Actions that end with a `!` **change** the data! This is a Ruby convention, but a good one to follow.

Hashes

Let's consider a book rating system:

- We use numeric values 0-5 to represent a book that you've read
- :not_rated will represent a book that you haven't rated
- We'll store all this data in a hash

Fill in a hash with some books, e.g.

```
books = { "Left Hand of Darkness"        => 5,
          "The Word for World Is Forest" => 5,
          "Nevermind the Pollacks"       => 0,
          "Only Revolutions"             => :not_rated }
```

Hashes

We can retrieve the rating for a particular book like so

```
books["Left Hand of Darkness"]
```

We can also **set** values like so

```
books["Only Revolutions"] = 3
```

How can we add a rating for a new book? Any guesses?

Hashes

We can retrieve the rating for a particular book like so

```
books["Left Hand of Darkness"]
```

We can also **set** values like so

```
books["Only Revolutions"] = 3
```

How can we add a rating for a new book? Any guesses?

We set the value of the book just like before!

```
books["White Teeth"] = 4
```

Now type `books` to see the whole hash.

Let’s Try Something

```
40.reverse
```

What happens?

Let’s Try Something

```
40.reverse
```

What happens?

Ruby just reported `NoMethodError: undefined method `reverse' for 40:Fixnum`

That means that `reverse` is not something you can do to the number 40.

Methods

Most computations in Ruby are performed by **methods**. Type `40.methods` to see which methods are available for basic numbers.

Notice `+`, `-`, et al. in the list?

What about `"chicken".methods` ?

Objects

An object is data along with the methods, computations, that can operate on it.

Everything in Ruby is an object: numbers, strings, hashes, etc.

How do you know what kind of object something is? Ask!

```
40.class
```

What **is** a class?

Classes

- Classes are templates for making objects
- Classes create different types of objects
  - All numbers can be added
  - All strings can be reversed
- They define kinds of data and methods on that data

Classes & Methods

Let's learn the syntax by example:

```
class Counter
end
```

What can we do with `Counter`?

```
c = Counter.new
```

Classes & Methods

Let's learn the syntax by example:

```
class Counter
attr_accessor :value

end
```

Now try

```
c = Counter.new
c.value
c.value = 10
c.value
```

What happened? What does `attr_accessor :value` do?

Classes & Methods

Let's learn the syntax by example:

```
class Counter
  attr_accessor :value

  def initialize
    @value = 0
  end

end
```

Now, again, try

```
c = Counter.new
c.value
```

`initialize` gives `new` instructions on how to create an object

Classes & Methods

Let's learn the syntax by example:

```
class Counter
  attr_accessor :value

  def initialize
    @value = 0
  end

  def increment
    @value = @value + 1
  end

end
```

Classes

Let's use our Counter class:

```
count = Counter.new
count.increment
count.increment
count.value

count.class
count.methods
```

Try `count.respond_to?("increment")`. What did you see?

Class Exercise

Let's add a `Counter.increment_by` method that takes an **argument** for how much to increment.

Don't need to start all over – we **open** the class instead

```
class Counter

def increment_by(n)
  # fill in here
end

end
```

Test your code as follows:

```
c = Counter.new
c.increment_by(5)
c.value
```

Classes

You can add methods to existing classes as well:

```
class String
  def chicken?
    self == "chicken"
  end
end

"chicken".chicken?
"puppy".chicken?
```

`self` is a way of referring to the object that the method is being called on.

In `"puppy".chicken?`, `self` is `"puppy"`.

Classes Exercise

Add a method to String that will test to see if a string is a palindrome.

A palindrome is any string that is the same read forwards or backwards.

To get you started type the following at the command line:

```
class String
  def palindrome?
```

and finish the rest! Test it on `"abba".palindrome?` and `"puppy".palindrome?`

Learn more about common Ruby classes

http://rubydoc.info/stdlib/core/

Good places to start: Array, Enumerable, Fixnum, Float, Hash, NilClass, String, Time

Blocks

Some methods take blocks.

`list.each {|p| code}` runs `code` on every element of `list`

```
list = [1,2,3,4]
list.each { |n| puts (n + 1) }
```

- A **block** is a computation, but **not** a method
- A block is of the form `{ |arg1, arg2, ..| code }`
- `{ |v| v.palindrome? }`
- `{ |x, y| x * y }`

Blocks

A more complicated example of using each:

```
reviews = Hash.new(0)
books.values.each { |rate| reviews[rate] = reviews[rate] + 1 }
reviews
```

`reviews` is a count of how many reviews you gave with a particular rating

`reviews[5]`

Blocks

There's another way to write blocks. This is commonly used for multi-line expressions.

```
reviews = Hash.new(0)
books.values.each do |rate|
  reviews[rate] = reviews[rate] + 1
  # more code can go here...
end
```

Control Structures

Ruby provides control structures for writing more complicated code.

If statements are a switch on whether the argument is true or false.

```
if true
  1
else
  2
end

if false
  1
else
  2
end
```

Control Structures

Case statements evaluate a statement and execute the code in the corresponding branch:

```
case favorite_color
when "green"
  puts "Grass is green"
when "blue"
  puts "Skies are blue"
when "red"
  puts "Tomatoes are red"
else
  puts "Are you sure that's a color?"
end
```

Control Structures

For-in statements allow iteration over a structure such as a list

```
list = [1,2,3,4,5]
sum = 0
for n in list
  sum = sum + n
end
sum
```

Koans

Ruby Koans

Intro To Rails

- Rails is a framework for building and deploying web applications
- It's structured using an architecture called MVC: Model-View-Controller
- MVC separates different kinds of application logic into sections based on what it's used for. In Rails, these live in separate folders.
- Rails values "convention over configuration"—the platform will make certain assumptions about how you're going to build your application. This makes tasks that suit Rails' conventions very easy, but actions that don't follow these conventions will take more work.

An Example Rails App

To get started:

1. Go to the microrant directory
2. Type `bundle install`
3. Then type `rails server`
4. Open http://localhost:3000 in your web browser

Microrant

- Twitter? Never heard of it.
- 10 characters to express your anger
- Basic activities: create, read, update, delete (CRUD)
- Go to http://localhost:3000/users/new to create a new user with your name
- Then go to /rants/new and create a new rant
  - Rants can also be edited and deleted—give it a try!
- What's behind the curtain?
  - Rails provides classes that allow us to retrieve and manipulate data easily
  - Logic for the application is written in normal Ruby, and a module called ActiveRecord does all of the database connection work for you

Behind the Scenes: the database

At The Console

- Close down the Rails server with Ctrl-C
- Run `rails console`
- You can modify the data using `Rant` and `User` classes
- The `User` class has `id` and `name` attributes
- The `Rant` class has `id`, `message`, and `user_id` attributes

At The Console

How can we look up a rant after it's been created?

```
r = Rant.find(1)
r.message
r.user_id
```

What about modifying a rant?

```
r.message = "RANTING!"
r.save
```

We need to `save` the changes when we're done editing.

Where's My Stuff

Rails models have built-in finder methods that help you retrieve records from the database.

Try:

- `Rant.all`
- `Rant.order("created_at DESC")`
- `Rant.first`
- `Rant.where("user_id = 1")`
- `Rant.where(:user_id => 1)`

At the Console

Let's try creating a new Rant, for `user_id` 1

```
user = User.find(1)
rant = user.rants.build
rant.message = "WHAT IS TH"
rant.save
```

Note that we didn't need to set the `id` field of the message! That was automatically set by `new`.

We `save` the record when we're done, in order to create it, just like when we were editing a rant before.

At The Console

Creating a rant that way was verbose. Is there a better way?

```
User.find(1).rants.create(:message => "E POINT!?")
```

Notice that you don't need to save when you use `create`—it combines `new` and `save` into a single action.

Looking at models

Open up the user.rb and rant.rb model files.

```
class Rant < ActiveRecord::Base

  belongs_to :user

  validates_presence_of :user, :message
  validates_length_of :message, :maximum => 10

end

class User < ActiveRecord::Base

  has_many :rants

  validates_presence_of :name

end
```

Exercises

Some people are writing in lowercase! This won't do!

Let's write a method that can convert all lowercase rants to uppercase. First, go find the rant.rb file in your Rails models directory.

```
class Rant < ActiveRecord::Base
before_save :convert_message_to_uppercase
def convert_message_to_uppercase
  ...
```

Rails provides many extensions to Ruby classes to make common tasks easier. Look in http://api.rubyonrails.org/classes/ActiveSupport/Multibyte/Chars.html to find the helper you need for this exercise.

Once you've added the code, try creating a lowercase rant and see what happens.

Exercises

Sometimes you want to work with your data in a different format. Go back to the Rails console and write a program that creates a hash that contains rant messages paired with user names.

How do you create a new Hash again?

- Earlier we saw `Hash.new`, but `{}` is a simpler way.
- Try `new_hash = {}`

The result should look like:

```
=> {"RANTING!"=>"caylee", "WTF"=>"audrey"}
```

Can you use an iterator to print this out as a series of formatted strings?

Callbacks

The `before_save` code we wrote earlier uses a **callback**. Rails defines several callbacks that allow you to change the input to a model before or after actions like save, create, update, and delete.

```
before_save :geocode_address
after_create :notify_admins
before_validate :check_over_some_other_details
```

Quality Control

Validations provide an easy wrapper for checking that a particular attribute is present, in the right format, or other requirements before the application will save the data. The microblog application already contains some of these. Look at the Rant and User models to see the ones we've included.

Can you think of other validations you'd like to add? Try changing the length of a valid Rant message.

Displaying the data

Rails will automatically create views with content when you use the `scaffold` command. Go to /app/views and have a look around.

These files are making use of a templating system called ERB that converts Ruby statements in the view into static HTML to display. A typical ERB statement might look like:

Documentation on ERB syntax: http://ruby-doc.org/stdlib/libdoc/erb/rdoc/classes/ERB.html

Exercises

The scaffolding system is quick to use, but not always pretty. Our list of rants lists the `user_id` for the person who said it, and not their name. Let's change that.

Go to /app/views/rants/ and open the index.html file. By default, the scaffolder makes the index view show a list of all records of the related model's type. Find the line that displays the user id and change it to show the user's name instead.

Exercises

On the users index page, let's add a column to the table to show how many rants each person has written. ActiveRecord makes this easy by providing a method called `count` that you can add to a collection of records to get the database count instead of the full records. For example, `User.count` or `Rants.count`.

These also work through has_many associations, so given a particular user record, you could try `user.kittens.count` or `user.books.count` (if those associations existed) or …

Routes

How does Rails know what page to go to when you visit http://localhost:3000/rants/new ?

Open /config/routes.rb to see how this works.

What about the C in MVC?

Now that we've seen and edited models and views, let's take a quick look at the controller files in our project.

What does a controller do?

- It connects the view and the model, calling methods on the models to get the data needed for the view, and allowing you to access input parameters, such as those from a form.
- It responds to the HTTP request from the browser, and renders the requested type of output (in our scaffolds, this is html or xml, but other things like json can easily be added—instant API!)
- The controller is also where you would add access control, like only allowing the user who wrote the rant to delete it.

Making Your Own

- Exit `microrant` directory
- Run `rails new betterrant`
- Enter the betterrant directory and run:

```
bundle install
rails generate scaffold User name:string
rails generate scaffold Rant user_id:integer message:string
rake db:migrate
```

At this point you can start the server and take a look around.

Summary

- We've reviewed the basics of the Ruby language
  - For a refresher, Try Ruby is a browser based overview of Ruby
  - Ruby has a number of simple types of data such as strings, numbers, arrays, and hashes
  - Most computations in Ruby are handled by methods.
  - A class is a template for creating objects that bundles data and methods
  - Classes can be extended as new needs arise

Summary

- We've covered a small Rails application: microrants
  - A good next tutorial is Rails For Zombies
  - More detail can be found in the Ruby Guides
  - ActiveRecord allows us to manipulate our persistent data like normal objects
  - .erb files allow us to intergrate Ruby code with html
  - Scaffolding gives a way of automating the start of a Rails project

Next Steps

- Try to make betterrant match or improve upon microrant
- Get involved in the local Ruby user group
- Start coming to Code-n-Splode

Stepping outside