Course Overview and Intro to Ruby

INFO 2310:
Topics in Web Design and Programming

INFO 2310

We'll be covering the web "framework" *Ruby on Rails*.

Ruby is a programming language (like Java, or PHP, but stranger).

A "framework" is a library of files that makes it easy for you to create web applications (at least if you follow its system).

Why Rails?

- A popular framework
- Has 'thought leadership'
- ISSA/Dean Strelau were enthusiastic
- Dave Patterson is teaching it at Berkeley

Course prerequisites

INFO 2300 (230) or equivalent knowledge:

- HTML/CSS/PHP
- Database backends for webpages

Check with me if you're not sure.

Helpful if you have done some object-oriented programming elsewhere (but not necessary)

Course structure

We'll meet for 10 weeks, starting today, ending October 31.

No class on September 19.

Course outline

We'll be following a course outline created by Dean Strelau; see http://code.strelau.net/curails. The course outline is:

1. Intro to Ruby (today)
2. Getting Started with Rails
3. Rails Basics
4. Associations
5. Authentication/Authorization
6. Testing
7. Plugins
8. JavaScript/Ajax
9. Photo Gallery
10. Deployment and Scaling

Through the course of these lectures, we'll gradually be building a blog website. In Lecture 9, we'll redo Project 3 from 230 and create a Photo Gallery from scratch in one lecture.

Course caveats

I am not a Ruby or Rails expert. In some sense, we will all be learning this stuff together.

So why take a course on Rails from me instead of one of the gajillion tutorials online?

- I will try to make it as least as good as any of the tutorials.
- Phil Adams is our expert TA.
- It's more fun to help each other by learning in a group than by learning solo.

Course requirements

Given the caveats, the requirements are light: attendance + the two websites (blog, photo gallery).

E.g. an A will be given if you show up to 8 of the 10 lectures and complete (with some attention to style) the two projects.

Course books/materials

No books required. The standard reference is:

"Agile Web Development with Rails" by Thomas and Heinemeier Hansson.

Slightly out of date (since it references Rails 1.2). New third edition (covering Rails 2) coming out in October.

I'm also looking at

"Beginning Rails: From Novice to Professional" by Hardy and Carneiro, Apress

(but this is also Rails 1.2)

Course books/materials

While the machines in this cluster have everything we need to do Rails, you will need someplace to store your own files (starting next time).

Two options:

1. Bring a flash drive (I'll bring some extras for those who don't have one).
2. Install Rails on a laptop and bring it along (see www.rubyonrails.org for instructions, but you'll need to install a DB as well – SQLite or MySQL).

Course resources

Course webpage at www.orie.cornell.edu/~dpw/info2310.

Course discussion group at forums.cit.cornell.edu (INFO 2310)

Why not take a minute to register?

Ruby

Ruby is a dynamic, object-oriented, programming language.

We'll need to be comfortable with the basics of it in order to use Rails.

Fire it up

Ruby is 'interpreted', not 'compiled', so we can work with it right away.

Get a command line, and type 'irb'.
(Start -> Run… "cmd")

Your first Ruby program

```
puts "Hello World!"
```

… prints "Hello World!" and returns a value (nil).

Remember Objects

Objects are collections of data (attributes) and associated functions (methods).

In Ruby, *everything* is an object. And most objects have methods.

Numbers are objects:
```
2.next
```
Strings are objects:
```
"Hello world!".length
```
Nil is an object:
```
nil.nil?
```

To get a list of all the methods of an object, use *object*.methods; e.g.
```
3.methods
"Hi".methods
```

Conditionals

Yep, Ruby has 'em too.

```
count = 5
if count > 10
   count += 5
  puts "You win!"
else
  puts "Loser."
end
```

Notes:
- No ( ) around condition
- No { } around code blocks; instead, defined by positions of "else" and "end"
- Indentations are not significant (though good style; usually two spaces)
- Line breaks are significant!

Other options

```
unless count > 10
  puts "Your count is too low."
end

puts "You win!" if count > 10
puts "You win!" unless count <= 10
```

In the latter two, 'if'/'unless' are *statement modifiers.*

Printing

As in PHP, single vs. double quotes are significant.

```
puts “Your current count is
  #{count}.”
puts ‘Your current count is
  #{count}.’
```

Within double quotes, the #{*variable*} expression prints out the value of *variable.*

True, false, and nil

Anything except the objects ‘nil’ and ‘false’ are true (even 0).

```
0.nil?
nil.to_i
false.to_s
“”.nil?
```

Arrays

Ruby has your standard indexed arrays

```
veggies = [‘spinach’, ‘carrot’]
myveggie = veggies[0]
things = [231, “wet dog”, 4.2]
things.push(“z”)
things << myveggie
puts things
```

Hashes

Ruby also has *hashes*, which we called *associative arrays* in PHP.

```
fruit = {‘apple’ => ‘red’, ‘banana’ =>
  ‘yellow’}
fruit[‘apple’]
oddhash = {3 => ‘three’, ‘three’ => 4}
newfruit = {:kiwi => “green”}
newfruit[:kiwi]
```

:kiwi is a *symbol*; something like a string, but immutable. They're frequently used as the keys for hashes.

Where things start getting really strange

Blocks and iteration

Ruby has ‘for’ loops like other languages

```
myarray = [10, 23, 17, 39]
double = []
for i in 0…myarray.length
  double[i] = 2*myarray[i]
end
puts double
```

No ‘for’

But it’s considered bad form to use them.

Instead

```
  double = myarray.map {|i| 2*i}
```

or equivalently

```
  double = myarray.map do |i|
     2*i
  end
```

What is going on here?

```
{|i| 2*i}
```

and

```
do |i|
     2*i
end
```

are *blocks*. Can think of these as functions with no names, where |i| is the argument. Usually use the former for single lines blocks, latter for multiline blocks.

The array method “map” takes a block as an input, returns the array that is the result of applying the block to each element of the array.

```
[10, 23, 17, 39].map {|i| 2*i }
```

Get used to it

This style of code is used all over in Ruby.

```
[“this”, “that”, “those”].each
  {|word| print word, “ “}
3.upto(6) {|x| puts “#{x}”}
5.times { print “*” }
```

Examples

Let’s add up the even numbers from 1 to 100.

```
sum = 0
(1..100).each {|x| sum += x if (x
  % 2 == 0)}

sum = 0
(1..100).select {|x| x % 2 ==
  0}.each {|i| sum += i}
```

```
(1..100).select {|x| x % 2 ==
  0}.inject(0) {|sum, i| sum + i}
```

array.inject(x) sets first argument initially to x, afterwards to result of block, second argument to each array element.

Methods

Since everything is an object, we define methods, not functions…

```
def hello
  puts “Hello”
end

hello
```

Methods can have arguments.

```
def hello(name)
  puts “Hello, #{name}!”
end

hello(“Baxter”)
```

Useful to have the last argument be a hash; allows for a list of named arguments.

```
def hello(name, params)
  if params[:language]
    if params[:language] == :Spanish
      puts “Hola, #{name}!”
    elsif params[:language] == :French
      puts “Bonjour, #{name}!”
    end
  else
    puts “Hello, #{name}!”
  end
end

hello(“Mariano”, {:language => :Spanish})
hello(“Mariano”, :language => :Spanish)
hello “Mariano”, :language => :Spanish
```

Don’t need the {} for the hash, or even the () for the method call – good Ruby style, but sometimes omitted in Rails.

Sample Rails call like this

```
has_one :parent, :class_name => "Thread",
  :foreign_key => "parent_thread_id"
```

Other method facts worth knowing:

- return value is last value in method
- methods that alter their arguments end in ‘!’ (by convention)
- methods that return true/false end in ‘?’; e.g. not `isChunky` or `is_Chunky` but `chunky`?

Classes

We define objects using ‘class’.

```
class Post
  def title=(value)
    @title = value
  end

  def title
    @title
  end

  def body=(value)
    @body = value
  end

  def body
    @body
  end

  def display_post
    puts "Title:", @title
    print "Body:", @body
  end
end

mypost = Post.new
mypost.title = "My first post!"
mypost.body = "This is my first post!"
mypost.display_post
mypost.title
```

Getters/Setters

@title, @post are *instance variables*. Each instance of the class 'Post' has a copy of them.

The methods 'title' and 'post' are called 'getters'; they 'get' the value of the attributes '@title' and '@post'.

The methods 'title=' and 'post=' are called 'setters'; they 'set' the value of the attributes '@title' and '@post'.

Defaults

Default getters/setters can be generated by using `attr_accessor`.

```
class Post
  attr_accessor :title, :post

  def display_post
    puts "Title:",  @title
    print "Body:", @body
  end
end
```

initialize

Can have an initialize method that gets called when a new instance of the object is created.

```
class Post
  attr_accessor :title, :body

  def initialize(args = {})
    @title = args[:title] if args[:title]
    @body = args[:body] if args[:body]
  end

  def display_post
    puts "Title:", @title
    puts "Body:", @body
  end
end

post2 = Post.new(:title => "Another post!")
post2.title
post2.body
```

Object inheritance

PHP has this, but we didn't discuss it in 230.

```
class Person
  attr_accessor :name
  def initialize(args = {})
    raise "Every Person must have a name" unless
   args[:name]
    self.name = args[:name]
  end
end

class Cornellian < Person
  attr_accessor :net_id
  attr_accessor :email
  def initialize(args = {})
    super(args)
    raise "Every Cornellian must have a NetID" unless
   args[:net_id]
    self.net_id = args[:net_id]
  end
  def email
    @email || "#{net_id}@cornell.edu"
  end
end
```

Some new things…

```
class Cornellian < Person
```

A ‘Cornellian’ is a kind of ‘Person’; has all the same attributes/methods as ‘Person’, plus whatever we add.

We say that

- ‘Cornellian’s *inherits from* ‘Person’s
- ‘Cornellian’ *extends* ‘Person’
- ‘Cornellian’ is a *child class* of ‘Person’ (or ‘Person’ is the *parent class*).
- ‘Cornellian’ is a *subclass* of ‘Person’ (or ‘Person’ is the *superclass*).

super

```
def initialize(args = {})
    super(args)
```

‘super’ calls the same method in the parent class.

self

```
self.name = args[:name]
```

`self` like `this` from PHP/Java. Refers to the current instance of the object.

raise

```
raise "Every Person must have a name"
  unless args[:name]
```

‘raise’ raises an error.

```
irb(main):021:0> dean = Person.new
RuntimeError: Every Person must have a
  name
       from (irb):4:in `initialize'
       from (irb):21:in `new'
       from (irb):21
```

```
def email
    @email || “#{net_id}@cornell.edu"
end
```

If @email is not nil, @email is returned, otherwise the Netid address is returned.

Note that this overwrites the default getter for @email.

Try it…

Create an instance of Cornellian with name “Dean” and netID “tds28”. What happens if you omit one or the other?

What’s Dean’s email? Set it to strelau@cornell.edu. Now what is it?

Can you access Dean’s name? Why?

Learning More

- Ruby language reference is at www.rubybrain.com.
- An early edition of the standard Ruby book is online at http://www.rubycentral.com/book/.
- An intro to programming using Ruby: http://pine.fm/LearnToProgram/?Chapter=00.
- A really whacked out intro to programming using Ruby: http://poignantguide.net/ruby/.

A haxx0r challenge

Write a RomanNum class such that you can set the value of an instance either by giving a Roman numeral (e.g. 'XXIV') or a number (e.g. 24), and can get the value in either form.

Reminders…

Sign up for the class forum

Either bring a flash drive or a laptop with Rails installed.