A Little Rails, A Lot Of Ruby

A Little Rails, A Lot Of Ruby

- In the discussion of today's lecture we will cover a little bit about Rails and adding some CSS to our sample site.
- Then I will give you a crash course in Ruby
  - The language upon which Rails is based.
  - Your first assignment will require you to build a small portfolio of Ruby programs
    - Worth 15% of the course

A Little Rails, A Lot Of Ruby

- Now if you remember from your application last week you had a fairly longish title in your application layout.
  - Would be much handier to store in an application wide variable and use the shortened form else where if needed
  - Basically advanced prediction of the DRY rule

Application Helper

- All application wide variables are stored in the application helper
  - found in app/helpers/application_helper.rb
  - Standard module in all RoR apps
  - Add in a bit of intelligent code in the form of a function here to define a title that can be used in all pages
  - means we can set the title in the controller instead of the page

Application Helper

- def title
  - base_title = "Ruby on Rails Tutorial Sample App"
  - if @title.nil?
    - base_title
  - else
    - "#{base_title} | #{@title}"
  - end
- end

Application Helper

- What this allows us to do is modify our application layout and instead of this
  - <title>Ruby on Rails Tutorial Sample App | <%= @title %><title>
- We use
  - <title><%= title %></title>

Cascading Style Sheets

- This form of static content is quite useful to have in any web project
  - useful for making things look nice
  - we will download and install the blueprint CSS files for our project and modify our pages to use it.
  - there is a standard place in a RoR project to store these files.
    - Depends on your rails version

Cascading Style Sheets

- The book states that it should go into public/stylesheets.
  - However in RoR that I'm using (3.2.11) they are to be stored in app/assets/stylesheets
- Download the Blueprint CSS sheets for this project. You can add your own later if necessary
  - http://www.blueprintcss.org/
  - make sure to download the zip file

Cascading Style Sheets

- After you open the zip file you take a copy of the blueprint folder in blueprint-master-css/ and copy it to the appropriate stylesheets directory in your RoR project
- Thus we will now show how to include CSS files in an ERB
  - we will apply CSS site wide so we will modify our application layout file

Cascading Style Sheets

- In our application file we will add a couple of lines to the head section
  - <%= stylesheet_link_tag 'blueprint/screen' :media => 'screen' %>
  - <%= stylesheet_link_tag 'blueprint/print' :media => 'print' %>
- These lines ask RoR to create stylesheet import tags to include the given CSS files.
  - Rerun your Rails server and you will see that the formatting of the text has changed.

Ruby Basics

- Up to this point you have seen lots of Ruby code but may not have fully understood it.
  - You will now get a crash course in Ruby to help you understand what you are writing and reading.
  - I'll speed through this as I will assume that you have a good working knowledge of OO languages
    - C++, Java, Python etc
  - Should be easy to pick up
  - Will relate it back to the code you have written thus far.
  - We will use ruby directly for this

Comments

- First and foremost is how to comment code
  - ALWAYS, ALWAYS comment your code.
  - Cannot be emphasised enough
  - To make a comment in code use the # symbol.
    - Everything following this symbol to the end of the line will be ignored
- Trying to read uncommented code is a nightmare
  - particularly if algorithms are non trivial
  - also helps if some one else helps to debug your code (Hint, Hint)

Strings

- For our purposes strings are very important
  - Web applications require lots of strings
  - Thus knowing how to manipulate and use them will be advantageous
- Strings can use double quotes like C++/Java etc
  - "foo", "bar", "" # all valid ruby strings
- They can be single quoted too (will explain differences later)
  - 'foo', 'bar', '' # also valid ruby

Strings

- String concatination is similar too
  - "foo" + "bar" gives you "foobar"
- However this will be new to non Ruby people, String interpolation.
  - Using variable names in strings with some extra syntax will do string substitution for you e.g.
  - first_name = "John"
  - "#{first_name} Doe"
    - gives you the string "John Doe"

Strings

- Interpolation can be done on one or more variables e.g.
  - last_name = "Doe"
  - "#{first_name} #{last_name}"
  - also gives "John Doe"
- To print out a string to console use the puts function e.g.
  - puts "John Doe"
  - puts first_name
  - puts will print out a new line after your string

Strings

- If you do not want a new line after your string use the print function. e.g.
  - print "John Doe"
- Single quoted strings for most purposes can be used where double quoted strings can be used.
  - However String interpolation does not work in single quoted strings
  - you don't have to escape special characters like \ in single quoted strings.

Objects and Message Passing

- Everything in Ruby is an object.
  - Even the value nil is an object itself
  - Objects respond to messages (AKA functions, methods) e.g.
  - "foobar".length will return 6
  - "foobar".empty? will return false
    - ? at the end of a message is a ruby convention. it indicates that the function will return a boolean
    - booleans can be combined with the AND (&&), OR (||) and NOT (!) operators

Objects and Message Passing

- The special object nil also responds to messages too e.g.
  - nil.to_s gives back ""
    - to_s is the to string method
    - converts your object into a string representation
- Every object also has a method to check to see if it is nil e.g.
  - "foobar".nil? # returns false
  - nil.nil? # returns true

Variables

- You can define both local and instance variables in Ruby
  - local variables are variables with out and @ character. these must be given a value immediatly
  - instance variables have the @ character. You man define these but they do not need a starting value
- Ruby will complain if you define a local variable with no value but will not complain if you define an instance variable with no value

Variables

- If you remember from an earlier ERB we had the line
- Ruby on Rails Tutorial Sample App | <%= @title %>
  - RoR did not complain because we had an instance variable. If the value was not set it took it to be nil.
  - thus the result would be "Ruby on Rails Sample App | "

Control Flow

- A simple if elsif ladder
  - if x > 0
    - puts "X is positive"
  - elsif x < 0
    - puts "X is negative"
  - else
    - puts "X is neutral"
  - end

Control Flow

- There is a corresponding unless keyword that can be used to evaluate a single statement
  - unlike an if statement however the text for an unless statement goes at the end.
  - Thus these are equivilant
  - if !string.empty? then puts string end
  - puts string unless string.empty?
- Note that the if statement has the then keyword in there.
  - this is necessary when writing a one line if statement.

Truth Values in Ruby

- Most of the languages that you have dealt with up to now have specified that anything that is non zero is true
  - and zero is false
- In Ruby this is not the case
  - everything including the number zero is true bar two exceptions
    - the boolean object set false
    - the nil object itself

Methods

- methods in Ruby are defined with the def keyword and end with the end keyword
- you do not need to specify a return type or the types of your parameters
- here is an example function
  - def string_message(string)
    - if string.empty?
      - "string is empty"
    - else
      - "string is not empty"
    - end
  - end

Methods

- Note that there is no return keyword used
  - it exists in ruby but is only really used when returning early from a function
  - Everything in Ruby is an expression which returns a value
  - Therefore the last expression in a function to be evaluated with provide the return value of the function.
- At this point you will be able to understand what is going on inside the Application Helper that you wrote for the sample app

Arrays

- Of course to generate useful strings for web apps you need a lot of other useful datastructures
  - particularly if your application is in any way complex.
- Arrays are one such basic structure
  - Arrays in Ruby are objects too
  - meaning they have methods that they respond to.
- Some built in objects have methods to convert themselves into arrays

Arrays

- For example the split method on strings will split on white space and give you an array of tokens
  - test_array = "foo bar baz".split # ['foo', 'bar', 'baz']
  - puts test_array[1] # prints bar
- Arrays can also be initialised in the normal way
  - To declare an empty array you do something similar to this
  - array = []

Arrays

- However you may initialise an array with some starting values.
  - works similar to C++/Java
  - array = [42, 8, 17]
  - element access is the same as C++/Java
    - see previous slide
- However Ruby also allows for negative indicies
  - -1 starts at the end of the array and finishes at -n
  - where n is the number of elements in the array

Arrays

- For example given the array a = [42, 8, 17]
- these are the postions for positive and negative indicies
  - a[0] a[1] a[2]
  - a[-3] a[-2] a[-1]
- Arrays are also objects so they have functions
  - first and last find the elements at the end of the array
  - length will find the length of the array
  - sort, reverse and shuffle are useful too

Arrays

- To append elements to an array you use the "push" operator
  - e.g. a << 20 # gives [42, 8, 17, 20]

Ranges

- Ranges define a set of numbers or characters in an ordered sequence.
  - Saves you from having to generate a sequence yourself
  - e.g. 0..9 is the range 0, 1, 2, ..., 9
  - these can be converted to arrays using the to_a method
  - e.g. a = (0..9).to_a # gives [0, 1, 2, ..., 9]
    - note the parentheseis around the range.
    - if you don't use them to_a will only give you an array with one element [9]

Ranges

- for example to pull out the first two elements of our array we can do this
    - b = a[0..1] # b is [42, 8]
- Also works with characters
    - ('a'..'e').to_a # gives ["a", "b", "c", "d", "e"]

Ranges

- Arrays and ranges can respond to methods that accept blocks
  - very confusing the first time you see it but can be very useful
  - (1..5).each { |i| print i*2, "\n" }
    - {} represents a block
    - |i| states that each number that is passed to the block will be represented by the variable called i
- Blocks do not have to be represented by braces
  - you can use the do end syntax as well.
  - advisable for longer blocks

Ranges

- example of do end syntax
  - (1..5) do |i|
    - print i *2
    - print "\n"
  - end

Hashes

- Hashes can be thought of as associative arrays
  - in each key value pair they take a hash of the key and based on that value they store the value in the array
  - to look up the value we must know the key
- Some examples
  - user = {} # creating an empty hash
  - user["first_name"] = "John"
    - first_name is the key
    - John is the value
  - user ["last_name"] = "Doe"

Hashes

- if you are not convinced of the associations then print out the hash directly
  - print user

Symbols

- While people use strings to represent keys for hashes it is quite common to use symbols instead
  - symbols are like strings without the baggage
  - can be compared in one shot
  - unlike strings which is character by character
- Examples of symbols
  - :name, :first_name, :last_name
  - all symbols start with a colon (:)

CSS revisit

- We can now understand these lines in our CSS inclusion
  - stylesheet_link_tag 'blueprint/screen', :media => 'screen'
    - this is a function call
    - parenthesis are optional in a function call
    - if a hash is the last argument in a function call then curly braces are optional
  - stylesheet_link_tag('blueprint/screen', { :media => 'screen' })
    - same line with optional parts inserted

Classes

- Classes work in a similar way to classes in other languages so simple example will clarify most things
- In a few lectures time we will need to make a User class so we can use that as an example

Classes

- example user class
  - class User
    - attr_accessor :name, :email
    - def initialize(attributes = {})
      - @name = attributes[:name]
      - @email = attributes[:email]
    - end
    - def formatted_email
      - "#{@name} <#{@email}>"
    - end
  - end

Classes

- As you can see classes are defined in a similar way to C++/Java but with a few minor differences
  - attr_accessor :name, :email
    - this tells ruby to write getter and setter methods for the name and email instance variables
  - def initialize
    - This is the constructor for the class
    - all classes will have this method
    - or a set of overridden versions

Classes

- attributes = {}
  - this is a constructor parameter.
  - however, if no value is provided for attributes the = {} states that the default value will be an empty hash
- @name, @email
  - instance variables
  - think of these as the fields of the User class
  - accessible in all methods of the class
- formatted_email
  - a method specific to the User class

Classes

- To create a new User object do the following
  - attributes = {}
  - attributes[:name] = "John Doe"
  - attributes[:email] = "john.doe@example.com"
  - Test = User.new attributes
    - this calls the appropriate initialize method

Inheritance

- Inheritance in ruby is easy
- Say we want a superuser class that is based off the user class
  - class SuperUser < User
  - end
- Ruby only allows single inheritance
  - extra behaviours can be introduced with mixins
  - I doubt we will need mixins but if we do I'll introduce them