Author: Edward Ocampo-Gooding
Date: July 27th, 2006
Title: Notes on Ruby Session 1: Variables, Arrays, and Hashes, oh my! (Also: blocks)

Recap of 27-July-2006's Ruby session

Major topics we discussed yesterday while reading through the accompanying code listing for an address-book program (which really does work, but I'd say that keeping a regular text file to handle that sort of thing is probably a better solution) include:

- Variables
  - What are they? What's with this dynamic typing business, and why you don't need that pseudo Hungarian-notation that Perl uses?
  - How do I declare them, and what's with the funky initialization?
- Arrays
  - What are they?
  - What can I do with them?
  - What are these "block"-things, and how do I use them with arrays?
- Hashes
  - What are they?
  - What can I do with them?
  - How do blocks play with hashes?
- Functions (which are really methods, but we'll talk about that later)
  - What do they look like? (i.e. what's with the dots? (Thanks Elizabeth))
  - Syntax sugar

Curious about a weird function method, or just hungry for a quick reference? From xhost02, xhost03, or lego (the recently deployed web-development server), use the `ri` command to get quick documentation. Here are some examples:

```
{xhost03}~> ri Array
{xhost03}~> ri Array.index
{xhost03}~> ri Array#index
```

Variables

- Variables
  - What are they? What's with this dynamic typing business, and why you don't need that pseudo Hungarian-notation that Perl uses?
  - How do I declare them, and what's with the funky initialization?

In Ruby, variables are dynamically typed, which is to say that to the interpreter, a variable is a reference to *something*. That something could be a `String`, an `Array`, a `Hash`, or as Mark Mayo pointed out, a `Cake`.

Unlike Perl, Ruby has no “basic types” like scalars, hashes, arrays, and globs. There’s no need to prefix your variables with a special character (what’s the prefix for a Cake type, right?). (Thanks for asking this, Dan.)

Variables are declared (and usually initialized) at the same time, e.g.

```
# The long way of using a String
palindrome = String.new()
palindrome = "Damn! I, Agassi, miss again! Mad!"

# A shorter way
palindrome = String.new("Damn! I, Agassi, miss again! Mad!")

# The conventional Ruby way
palindrome = "Damn! I, Agassi, miss again! Mad!"

# Using a String without having to declare it
if "Damn! I, Agassi, miss again! Mad!" == "Damn! I, Agassi, miss again! Mad!".reverse
    puts "This is a palindrome!"
end

# Even more concise, but using the prior initialized variable
puts "This is a palindrome!" if palindrome == palindrome.reverse
```

The reason why this works will be revealed when we talk about object oriented programming (or if you just come talk to me on the 5th floor).

Arrays

- Arrays
  - What are they?
  - What can I do with them?
  - What are these “block”-things, and how do I use them with arrays?

Arrays in Ruby are lists of references, similar to Perl’s.

```
arr = ["me", "hearties", 42]

# Slice them
arr[0..1]               # => ["me", "hearties"]

# Dice them
arr.indexes(2, 0, 1)    # => [42, "me", "hearties"]
```

Here’s an example of using a block:

```
# Make julienne fries
arr.each do |array_element|
  puts array_element
end
```

Outputs:

```
# me
# hearties
# 42
```

The idea of the block is that the `Array#each` instance method (err… function of `Array` that’s called using a type that you’ve initialized, or “filled in”) is that it gives you this iterator, or a “thing that will carefully walk through your array, one element at a time”, to which you give a block of code.

Here, our block starts with “do”, and ends with “end”. What you fill in between the pipes is the block variable that your iterator will fill in with every step through the array (in this case, it’s `|array_element|`). After the iterator makes a single step, it fills in the element that it’s currently at, and runs your block of code.

Can you have more than one line in the block? Sure, you can write an entire program there, and it’ll get run every time the iterator hands you your block variable.

Hashes

- Hashes
  - What are they?
  - What can I do with them?
  - How do blocks play with hashes?

Hashes in Ruby and Perl work similarly:

```
h = { "The final answer" => 42,
      4 => "Fingers on a Simpson hand" }

# Note that the `[]` operator is used with both Hashes and Arrays,
# unlike Perl which uses the `{}` operator
h["The final answer"]   # => 42
h[4]                    # => "Fingers on a Simpson hand"
```

In the address-book example, we use the `Hash#select` method to tell the hash to give us back [key, value] pairs whenever the key or value matches a query string that we’re looking for. Here’s an excerpt, altered for clarity:

```
found_entries_array_of_key_value_pairs = addresses_hash.select do |key, value|
    key.include?(query_string) || value.include?(query_string)
end
```

The `addresses_hash.select` method call gives out an iterator to which you can attach a block. By declaring our block variables in the form `|key, value|`, the iterator is smart enough to know that if it the element that it just pulled out is an array (which in our case, it is, because it’s the `[key, value]` pair), then it should put the first element of that key-value pair (a string) into the first block variable (which is also called `key`), and the second element of the key-value pair (a string) into the second block variable (which is also called `value`).

In the block, we’re asking each string (the block variables `key` and `value`) if they include the `query_string`, and we’re binary OR-ing them for a result: either true or false.

If the result for that run of the block is true, then the key-value array-pair that came from the iterator (something like `["Oscar", "Sesame Street"]`) is added to `found_entries_array_of_key_value_pairs` (which could look something like `[ ["Big Bird", "Sesame Street"], ["Oscar", "Sesame Street"] ]`).

Functions (which are really methods)

- Functions (which are really methods, but we’ll talk about that later)
  - What do they look like?
  - How do they work?

A Perl function for finding the length of a string would look something like

```
length($str)
```

In Ruby, consider the function to be a part of the variable, and call it a “method” instead of a “function”. Here’s an example:

```
the_old_lie = "Dulce et Decorum est \nPro patria mori.".length()
```

N.B. that Ruby is full of syntactical sugar; if you think a method call appears more clear in its intent without parentheses, or that it clutters things up unnecessarily, then don’t use them.

```
the_old_lie = "Dulce et Decorum est \nPro patria mori.".length
```

References

Ruby syntax guide

Expressions reference