Web Programming

Lecture 9 – Introduction to Ruby

Origins of Ruby

- Ruby was designed by Yukihiro Matsumoto ("Matz") and released in 1996.
- It was designed to replace Perl and Python, which Matz considered inadequate.
- It grew quickly in Japan and then spread around the world.
- Its expansion was a result of the increasing popularity of Rails, a Web development framework that was written in Ruby and that uses Ruby.

Uses of Ruby

- Because Ruby is implemented by pure interpretation, it's easy to use.
- Example

```
irb(main):001:0> puts "hello, world"
hello, world
=> nil
```

- Ruby uses regular expressions and implicit variables like Perl, objects like JavaScript but is quite different from these languages.

Scalar Types in Ruby

- Ruby has three categories of data types:
  - Scalars – either numerics or character strings.
  - Arrays – that uses dynamic sizing
  - Hashes – associative arrays, similar to PHP.
- Everything in Ruby is an object

Numeric and String Literals

- All numeric data types are derived from the base class **`Numeric`**, has two derived classes **`Float`** and **`Integer`**.

`Integer` Literals

- **`Integer`** has two derived classes:
  - **`FixNum`** - fits the range of a machine word (usually 32 bits).
  - **`BigNum`** – numbers outside the range of **`FixNum`**. (if an operation on **`BigNum`** produces a smaller value, it will be coerced into **`FixNum`**).
- Ruby ignores underscores in integer literals so they can be more readable.
  - **`1_234_567_890`** is more readable than **`1234567890`**

`Float` Literals

- A numeric literal with either an embedded decimal point or an exponent following it is a `Float` object.
- `Float` objects are stored as double-precision floating point numbers.
- Decimal points must have a digit on both sides of it.

`String` Literals

- All string literals are `String` objects, which are sequences of bytes that represent characters.
- `String` objects are either single-quoted or double-quoted.

Single-Quoted `String` Literals

- Single quoted strings cannot have escape sequences.
- Examples
  - `'I\'ll meet you at O\'Malleys'`
    the inner apostrophes are included correctly.
  - `'Some apples are red, \n some are green.'`
    contains a backslash followed by n (not a newline).

Delimiting Single-Quoted `String` Literals

- You can use a different delimiter by beginning the string with a q followed by another character. It will even match up braces, brackets or parentheses.
- Examples

```
q$Don't you think she's pretty$
q<don't you think she's pretty>
```

Double-Quoted `String` Literals

- Double-quoted strings can contain the special characters specified by escape sequences. And the values of variable names can be substituted into the string.
- Example
  - `"Runs \t Hits \t Errors"` will include the expected tabs
- For a different delimited for double-quotes strings, begin the string with Q:
  - `Q@"Why not learn Ruby", he asked.@`

Naming Local Variables

- A local variable is not a class nor an instance variable. It belong to the block, method definition, etc. in which it is located.
- Local variable names begin with a lowercase letter or an underscore, followed by letters, digits or underscores. While variable names are case-sensitive, the convention is not to use uppercase letters.

Using Variables In Strings

- The value associated with a local variable can be inserted in a double-quoted string:
- `"Tuesday's high temperature was #{tue_high} "`

  is printed as

  `"Tuesday's high temperature was 83"`
- Everything in Ruby is an object, so we are really working with their references, which are typeless. As a result, all variables are implicitly declared (how we use them determines their type).

Constants in Ruby

- Constants in Ruby begin with an uppercase letter.
- A constant is created by assigning it a value, which can be any constant expression.
  - Constants in Ruby can be assigned new values, but there will be a warning message.

Predefined Variables

- Ruby has predefined variables (like Perl), which consist of $ followed by a special character.
  - Examples - `$_`, `$^`, `$\`

Numerical Operators

Operator	Associativity
**	Right
Unary +, -	Right
*, /, %	Left
Binary +, -	Left

Assignment Statements

- Assignment statements are like those in C-based languages.
- Ruby includes the Math module, which has basic trigonometric and transcendental functions, including

  `Math.cos` (cosine) **`Math.sin`** (sine)

  `Math.log`(logarithm) **`Math.sqrt`** (square root)

  all of these return a **`Float`** value.

Interactive Ruby (**`irb`**)

```
irb(main):001:0> 17*3
=> 51
irb(main):002:0> conf.prompt_i = ">>"
=> ">>"
>>
```

`String` Methods

- Ruby's **`String`** class has over 75 methods, many of which can be used as if they were operators.
- These include:
  - **`+`** - concatenation
  - **`<<`** append
- Example

```
>>"Happy" + " " + "Holidays!"
=> "Happy Holidays!"
>>
```

Assigning `String` Values

- **`<<`** appends a string to the right of another string.

```
irb(main):001:0> mystr = "G'day, "
=> "G'day, "
irb(main):002:0> mystr << "mate"
=> "G'day, mate"
irb(main):003:0>
```

- This created the string literal and assigned its reference to **`mystr`**.

Assigning `String` Values (continued)

```
irb(main):003:0> mystr = "Wow!"
=> "Wow!"
irb(main):004:0> yourstr = mystr
=> "Wow!"
irb(main):005:0> yourstr
=> "Wow!"
irb(main):006:0>
```

- Ruby assigned **`yourstr`** a copy of the same reference that **`mystr`** held.

Assigning `String` Values (continued)

```
irb(main):001:0> mystr = "Wow!"
=> "Wow!"
irb(main):002:0> yourstr = mystr
=> "Wow!"
irb(main):003:0> mystr = "What?"
=> "What?"
irb(main):004:0> yourstr
=> "Wow!"
irb(main):005:0>
```

- After the assignment, **`yourstr`** has the same reference as **`mystr`**. But when **`mystr`** is assigned a different string literal, Ruby sets aside another memory location for the new literal and that is the reference that **`mystr`** now holds.

Assigning `String` Values (continued)

- If you want to change the value in the location that **`mystr`** references but have **`mystr`** reference the same location in memory, use the **`replace`** method:

```
irb(main):001:0> mystr = "Wow!"
=> "Wow!"
irb(main):002:0> yourstr = mystr
=> "Wow!"
irb(main):003:0> mystr.replace("Golly!")
=> "Golly!"
irb(main):004:0> mystr
=> "Golly!"
irb(main):005:0> yourstr
=> "Golly!"
irb(main):006:0>
```

Assigning `String` Values (continued)

- You can also use **`+=`** to perform the append operation.

```
irb(main):001:0> mystr = "check"
=> "check"
irb(main):002:0> mystr += "mate"
=> "checkmate"
irb(main):003:0>
```

Commonly Used String Methods

Method	Action
`capitalize`	Converts the first letter to uppercase and the rest of the letters to lowercase
`chop`	Removes the last character
`chomp`	Removes a newline from the right end if there is one
`upcase`	Converts all of the lowercase letters in the object to uppercase
`downcase`	Converts all of the uppercase letters in the objects to lowercase
`strip`	Removes the spaces on both ends
`lstrip`	Removes the spaces on the left end
`rstrip`	Removes the spaces on the right end
`reverse`	Reverses the characters of the string
`swapcase`	Converts all uppercase letters to lowercase and all lowercase letters to uppercase

Commonly Used String Methods

- The methods mentioned before produce new string and do NOT modify the given string in place.
- If you wish to modify the string instead of producing a new string, place a ! at the end of the method name. Such methods are called ***bang methods*** or ***mutator methods***.

Mutator Methods – An Example

```
irb(main):001:0> str = "Frank"
=> "Frank"
irb(main):002:0> str.upcase
=> "FRANK"
irb(main):003:0> str
=> "Frank"
irb(main):004:0> str.upcase!
=> "FRANK"
irb(main):005:0> str
=> "FRANK"
irb(main):006:0>
```

Ruby Strings as Arrays

- Ruby strings can be indexed, in a manner similar to arrays, with indices starting at `0`.
- The brackets serve as an accessor for a single character, returned as an ASCII value. If you wish the character, use the **`chr`** method.
- More recent implementations of Ruby may return the character instead of ASCII value for the [] operator.

Ruby Strings as Arrays – An Example

```
irb(main):006:0> str = "Shelley"
=> "Shelley"
irb(main):007:0> str[1]
=> "h"
irb(main):008:0> str[1].chr
=> "h"
```

Ruby Strings and Substring

- A multicharacter substring can be accessed by specifying the starting character and number of characters in the substring:

```
irb(main):009:0> str = "Shelley"
=> "Shelley"
irb(main):010:0> str[2,4]
=> "elle"
irb(main):011:0>
```

Changing a String With a Substring

- The **[]=** operator can be used to specify characters of a substring and to what they are be changed:

```
irb(main):013:0> str = "Donald"
=> "Donald"
irb(main):014:0> str[3,3] = "nie"
=> "nie"
irb(main):015:0> str
=> "Donnie"
irb(main):016:0>
```

Comparing Strings for Equality

- **==** is used to see if two string have the same content.
- **equal?** tests to see if both are the same object
- Example

```
irb(main):016:0> "snowstorm" == "snowstorm"
=> true
irb(main):017:0> "snowie" == "snowy"
=> false
irb(main):018:0> "snowstorm".equal?("snowstorm")
=> false
irb(main):019:0>
```

Comparing Numeric Values

- The **==** operator determines if the values are equivalent regardless of type.
- The **eql?** operator returns true if the types and values match.

```
irb(main):023:0> 7 == 7.0
=> true
irb(main):024:0> 7.eql?(7.0)
=> false
irb(main):025:0>
```

<=>

- The **<=>** operator compares two different values and returns -1 if the second operator is greater than the first, 0 if they are equal and 1 if the first is greater than the second.

<=> - Examples

```
irb(main):025:0> 7 <=> 5
=> 1
irb(main):026:0> "grape" <=> "grape"
=> 0
irb(main):027:0> "grape" <=> "apple"
=> 1
irb(main):030:0> "apple" <=> "prune"
=> -1
irb(main):031:0>
```

Repetition Operator (*)

- The repetition operator (*) takes a string as its left operand a numeric expression as its right operand and replicates the left operand as many times as indicated by the right operand.
- Example

```
irb(main):031:0> "More!" * 3
=> "More!More!More!"
irb(main):032:0>
```

Screen Output

- Output is directed to the screen using the puts method (or operator).
- The operand for puts is a string literal with a newline implicitly appended to the end.
- A variable's value can be included in the string by writing **#{***variableName***}**
- **print** works in the same way except with the included newline.
- **sprintf** works as it does in C, allowing for formatted output.

Screen Output – An Example

```
irb(main):032:0> name = "Pudgy"
=> "Pudgy"
irb(main):033:0> puts "My name is #{name}"
My name is Pudgy
=> nil
irb(main):034:0> print "My name is #{name}"
My name is Pudgy=> nil
irb(main):035:0> total = 10
=> 10
irb(main):036:0> str = sprintf("%5.2f", total)
=> "10.00"
irb(main):037:0>
```

Keyboard Input

- The gets method gets a line of input from the keyboard. The retrieved line includes the newline character. You can get rid of it with chomp:

```
irb(main):037:0> name = gets
apple
=> "apple\n"
irb(main):038:0> name = name.chomp
=> "apple"
irb(main):039:0> name = gets.chomp
apple
=> "apple"
irb(main):040:0>
```

Keyboard Input (continued)

- Since the input is taken to be a string, it needs to be converted if its numeric:

```
irb(main):042:0> age = gets.to_i
29
=> 29
irb(main):043:0> age = gets.to_f
28.9
=> 28.9
irb(main):044:0>
```

quadeval.rb

```
#quadeval.rb – A simple Ruby program
# Input:    Four numbers, representing the values of
#           a, b, c, and x
# output:   The value of the expression
#           a*x**2 _ b*x + c
# Get input
puts  "please input the value of a"
a = gets.to_i
puts  "please input the value of b"
b = gets.to_i
puts  "please input the value of c"
c = gets.to_i
```

```
# compute and display the result
result = a * x ** 2  +  b * x  +  c
puts "The value of the expression is #{result}"
```

Running `quadeval.rb`

```
C:\>ruby quadeval.rb
please input the value of a
1
please input the value of b
2
please input the value of c
1
 Please input the value of x
5
The value of the expression is 36

C:\>
```

Relational Operators

Operator	Operation
`==`	Is equal to
`!=`	Is not equal to
`<`	Is less than
`>`	Is greater than
`<=`	Is less than or equal to
`>=`	Is greater than or equal to
`<=>`	Compare, returning -1, 0 or +1
`eql?`	True if the receiver object and the parameter have the same type and equal values
`equal?`	True if the receiver object and the parameter have the same object ID

Operator Precedence

Operator	Associativity
`**`	Right
`!`, unary + and –	Right
*, /, `%`	Left
`+.–`	Left
`&`	Left
`+, –`	Left
`>, <, >=, <=`	Nonassociative
`==, !=, <=>`	Nonassociative
`&&`	Left
`||`	Left
`=, +=. –=, *=, **=, /=, %=, &=, &&=, ||=`	Right
`not`	Right
`or, and`	Left

`if` Statement in Ruby

- **`if`** statements in Ruby do not require parentheses around the control expression, but they do require **`end`**:

```
irb(main):045:0> if a > 10
irb(main):046:1> b = a * 2
irb(main):047:1> end
=> nil
irb(main):048:0>
```

if..elsif..else

```
if snowrate < 1
  puts "Light snow"
elsif snowrate < 2
  puts "Moderate snow"
else
  puts "Heavy snow"
end
```

`unless` Statement

The **`unless`** statement is the opposite of the **`if`** statement

```
unless sum > 100
  puts "We are not finished yet!"
end
```

`case` Statements

```
case Expression
when value then
  Statement
…
when value then
  Statement
[else
    Statement]
end
```

```
case BooleanExpression
  then Expression
…
case BooleanExpression
  then Expression
else
    Expression
end
```

`case` – An Example

```
case in_val
when -1 then
  neg_count += 1
when 0 then
  zero_count += 1
when 1 then
  pos_count += 1
else
  puts "Error – in_val is out of range"
end
```

`case` – An Example

```
leap = case
       when year % 400 then true
       when year % 100 then false
       else year %4 == 0
       end
```

`while` Statement

- The syntax for a **`while`** statement:

  `while` *ControlExpression*
  Statement(s)
  `end`

- Example

```
i = 0
while i < 5 do
   puts i
   i += 1
end
```

`until` Statement

- The syntax for a **`until`** statement:

```
until ControlExpression
   Statement(s)
end
```

- Example

```
i = 4
until i >= 0 do
   puts i
   i -= 1
end
```

`loop` Statement

- **`loop`** statement are infinite loops – there is no built-in mechanism to limit its iterations.
- **`loop`** statements can be controlled using:
  - the **`break`** statement – which goes to the first statement after the loop
  - the **`next`** statement – which goes to the first statement within the loop

loop Statement - Examples

```
sum = 0
loop do
  dat = gets.to_i
  if dat < 0 break
  sum += dat
end
```

```
sum = 0
loop do
  dat = gets.to_i
  if dat < 0 next
  sum += dat
end
```

Arrays in Ruby

- In Ruby, array size is dynamic, growing and shrinking as necessary
- Arrays in Ruby can store different types of data in the same array.
- Arrays can be created by:
  - Using the predefined Array class.
  - Assign a list literal to a variable.

Initializing Arrays - Examples

```
irb(main):001:0> list1 = Array.new(5)
=> [nil, nil, nil, nil, nil]
irb(main):002:0> list2 = [2, 4, 3.14159, "Fred", [] ]
=> [2, 4, 3.14159, "Fred", []]
irb(main):003:0> list3 = Array.new(5, "Ho")
=> ["Ho", "Ho", "Ho", "Ho", "Ho"]
irb(main):004:0>
```

Working With Arrays - Examples

```
irb(main):004:0> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
irb(main):005:0> second = list[1]
=> 4
irb(main):006:0> list[3] = 9
=> 9
irb(main):007:0> list
=> [2, 4, 6, 9]
irb(main):009:0> list[2.99999] # indices are
truncated
=> 6
irb(main):010:0> len = list.length
=> 4
irb(main):011:0>
```

`for-in` Statement

- The **`for-in`** statement is used to process elements of an array.
- The scalar variable takes on the values in the array one at a time.
- The scalar variable gets the ***value***, ***not*** a reference to a value. Therefore, operations on the scalar variable do not affect the array.

`for-in` Statement – An Example

```
irb(main):001:0> sum = 0
=> 0
irb(main):002:0> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
irb(main):003:0> for value in list
irb(main):004:1>   sum += value
irb(main):005:1> end
=> [2, 4, 6, 8]
irb(main):006:0> sum
=> 20
irb(main):007:0>
```

for-in Statement – Another Example

```
irb(main):001:0> list = [1, 3, 5, 7]
=> [1, 3, 5, 7]
irb(main):002:0> for value in list
irb(main):003:1>   value += 2
irb(main):004:1> end
=> [1, 3, 5, 7]
irb(main):005:0> list
=> [1, 3, 5, 7]
irb(main):006:0>
```

for-in Statement – Another Example

```
irb(main):001:0> list = [2, 4, 6]
=> [2, 4, 6]
irb(main):002:0> for index in [0, 1, 2]
irb(main):003:1> puts "For index = #{index}, the
value is #{list[index]}"
irb(main):004:1> end
For index = 0, the value is 2
For index = 1, the value is 4
For index = 2, the value is 6
=> [0, 1, 2]
irb(main):005:0>
```

Built-in Methods for Arrays and Lists

- There are many built-in methods that are a part of Ruby. They include:
  - **`shift`** – removes and returns the first element of the list
  - **`pop`** – removes and return the last element of the list
  - **`unshift`** – takes a scalar or an array literal and appends it to the beginning of the array.
  - **`push`** - takes a scalar or an array literal and appends it to the end of the array.

Built-in Methods for Arrays and Lists

- There are many built-in methods that are a part of Ruby. They include:
  - **`+`** - catenates two arrays
  - **`reverse`** – returns an array with the order of elements of the array reversed
  - **`include?`** – returns true if the specific object is in the array.
  - **`sort`** – sorts elements as long as Ruby has a way to compare them.

shift – An Example

```
irb(main):001:0> list = [3, 7, 13, 17]
=> [3, 7, 13, 17]
irb(main):002:0> first = list.shift
=> 3
irb(main):003:0> list
=> [7, 13, 17]
irb(main):004:0>
```

pop – An Example

```
irb(main):004:0> list = [2, 4, 6]
=> [2, 4, 6]
irb(main):005:0> last = list.pop
=> 6
irb(main):006:0> list
=> [2, 4]
irb(main):007:0>
```

unshift – An Example

- **irb(main):009:0>** *list = [2, 4, 6]*
- **=> [2, 4, 6]**
- **irb(main):010:0>** *list.unshift(8, 10)*
- **=> [8, 10, 2, 4, 6]**
- **irb(main):011:0>**

push – An Example

- **irb(main):007:0>** *list = [2, 4, 6*]
- **=> [2, 4, 6]**
- **irb(main):008:0>** *list.push(8, 10*)
- **=> [2, 4, 6, 8, 10]**
- **irb(main):009:0>**

`concat` - An Example

```
irb(main):011:0> list1 = [1, 3, 5, 7]
=> [1, 3, 5, 7]
irb(main):012:0> list2 = [2, 4, 6, 8]
=> [2, 4, 6, 8]
irb(main):013:0> list1.concat(list2)
=> [1, 3, 5, 7, 2, 4, 6, 8]
irb(main):014:0>
```

`+` - An Example

```
irb(main):014:0> list1 = [1, 3, 5, 7]
=> [1, 3, 5, 7]
irb(main):015:0> list2 = [2, 4, 6, 8]
=> [2, 4, 6, 8]
irb(main):016:0> list3 = list1 + list2
=> [1, 3, 5, 7, 2, 4, 6, 8]
irb(main):017:0>
```

`reverse` – An Example

```
irb(main):018:0> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
irb(main):019:0> list.reverse
=> [8, 6, 4, 2]
irb(main):020:0> list
=> [2, 4, 6, 8]
irb(main):021:0> list.reverse!
=> [8, 6, 4, 2]
irb(main):022:0> list
=> [8, 6, 4, 2]
irb(main):023:0>
```

`include?` – An Example

```
irb(main):023:0> list = [2, 4, 6, 8]
=> [2, 4, 6, 8]
irb(main):024:0> list.include?(4)
=> true
irb(main):025:0> list.include?(10)
=> false
irb(main):026:0>
```

`sort` – An Example

```
irb(main):028:0> list = [16, 8, 2, 4]
=> [16, 8, 2, 4]
irb(main):029:0> list.sort
=> [2, 4, 8, 16]
irb(main):030:0> list2 = ["jo", "fred", "mike",
"larry"]
=> ["jo", "fred", "mike", "larry"]
irb(main):031:0> list2.sort
=> ["fred", "jo", "larry", "mike"]
irb(main):032:0>
```

```
irb(main):032:0> list = [2, "jo", 8, "fred"]
=> [2, "jo", 8, "fred"]
irb(main):033:0> list.sort
ArgumentError: comparison of Fixnum with String
failed
        from (irb):33:in `sort'
        from (irb):33
        from C:/Ruby193/bin/irb:12:in `<main>'
irb(main):034:0>
```