Topic 6 Introduction to Ruby
CITS3403 Web & Internet Technologies

References: Sebesta, Chapter 14; Ruby et al, Appendix A
What is Ruby?

• Ruby is a **dynamic, reflective, general purpose object-oriented programming language that combines syntax inspired by Perl with Smalltalk-like features** - Wikipedia

• Developed by Yukihiro Matsumoto (a.k.a. “Matz”) in Japan in mid-1990s
  – started as replacement for Perl and Python which Matz found inadequate

• Notable characteristics
  – Purely interpreted
  – An interactive Ruby interpreter! `irb`
  – Regular expressions based on Perl
  – Dynamic classes based on JavaScript
  – Like Smalltalk, **all** data types are objects
Motivation for Ruby

–Back in 1993, I was talking with a colleague about scripting languages. I was pretty impressed by their power and their possibilities. I felt scripting was the way to go.

As a long time object-oriented programming fan, it seemed to me that OO programming was very suitable for scripting too... I found that Perl 5, which had not released yet, was going to implement OO features, but it was not really what I wanted...

Then I came across Python. It was an interpretive, object-oriented language. But I didn't feel like it was a "scripting" language. In addition, it was a hybrid language of procedural programming and object-oriented programming.

I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That's why I decided to design my own language.
Motivation for Ruby

–Stewart: Did you have a guiding philosophy when designing Ruby?
–Matz: Yes, it's called the "principle of least surprise." I believe people want to express themselves when they program. They don't want to fight with the language. Programming languages must feel natural to programmers. I tried to make people enjoy programming and concentrate on the fun and creative part of programming when they use Ruby.

Source: An Interview with the Creator of Ruby,
Motivation for Ruby

–Stewart: I gather you had worked with both Perl and Python before creating Ruby. What bits of Perl did you incorporate in Ruby?
–Matz: A lot. Ruby’s class library is an object-oriented reorganization of Perl functionality--plus some Smalltalk and Lisp stuff. I used too much I guess. I shouldn't have inherited $_, $&, and the other, ugly style variables.
–Stewart: How about Python? What aspects of that language did you try to reuse in Ruby?
–Matz: Far less than Perl. But I stole a few things, like exception names. Plus I learned a lot from its code.
–Stewart: Why should someone already familiar with Perl or Python switch to Ruby?
–Matz: Why should you switch to Ruby? If you are happy with Perl or Python, you don't have to. But if you do feel there must be a better language, Ruby may be your language of choice. Learning a new language is harmless. It gives you new ideas and insights. You don't have to switch, just learn and try it. You may find yourself comfortable enough with Ruby to decide to switch to it.

Running Ruby

• Interactively

$ which irb
/usr/local/bin/irb
$ irb
irb(main):001:0> 3 * 4
=> 12
irb(main):002:0> puts ("Hello")
Hello
=> nil

• Program in file

$ ruby sayhello.rb
Hello

• irb options:
  – -v, -- simple-prompt
Scalar Types and Their Operations

• Three categories of data
  – Scalars
  – Arrays
  – Hashes

• All values are objects, including numeric values

Diagram:
- Array
  - Numeric
    - Float
    - FixNum
  - String
    - Single Quote
    - Double Quote
- Scalar
  - Integer
- Hash
  - BigNum
Numeric Data and String Literals

• Numeric data are descendants of class Numeric
  – Float and Integer
  – Integers are either Fixnum or Bignum

• String Literals
  – Single or double quoted
  – Double quoted has escape characters and expression interpolation
    • Expression is specified by #{}...
  – Single quoted have no escape characters or expression interpolation

• Alternative forms (like Perl), eg: q$...$ for a single quoted string
Variables

• Variables are not formally declared and do not have a fixed type

• Variables are all references to objects

• Variable names indicate category of variables
  – An identifier begins with a lower case letter or underscore followed by letters and/or digits and/or underscore
  – Named constants are named like variables but have names that begin with a capital letter
  – There are some implicit variables such as $_
    • $_ is the last string read by the Kernel methods gets and readline
Numeric Operators

• Standard operators, with:
  – ** for exponentiation
  – no ++ or --

• Precedence and associativity

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Right</td>
</tr>
<tr>
<td>+, –</td>
<td>Right</td>
</tr>
<tr>
<td>*, /, %</td>
<td>Left</td>
</tr>
<tr>
<td>binary +, –</td>
<td>Left</td>
</tr>
</tbody>
</table>

The operators listed first have the highest precedence.

Math module: cos, sin, log, sqrt, ...
String Methods

• Concatenation is indicated by +

• \(<\) appends its right operand to the left
  – \(+=\) is a synonym

• \texttt{capitalize}, \texttt{chop}, \texttt{chomp}, \texttt{upcase}, ...
  – These create new strings

• Mutator versions \texttt{capialize!}, \texttt{chop!}, \texttt{chomp!}, \texttt{upcase!} modify the string
String Comparison

• Three “equals”
  - == compares strings for equality based on content
  - equal? compares for the same object
  - eql? compares for type and value

• <=>
  compares two strings returning -1 if the first is smaller, 0 if they are equal, +1 if the second is smaller

• * is used to repeat a string: “x” * 3 is “xxx”
Screen Output

• `puts` displays a string on standard output
  – A new line is added to the output
• `print` displays a string without the new line
Keyboard Input

- `gets` gets a line of input from the console (keyboard)
- `gets.chomp` returns the next line of input without the terminating newline
- `gets.to_i` returns the next line of input converted to an integer
- `gets.to_f` returns the next line of input converted to a float
**Relational Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Is equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Is not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Is less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Is greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Is less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Is greater than or equal to</td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td>Compare, returning −1, 0, or +1</td>
</tr>
<tr>
<td>eql?</td>
<td>True if the receiver object and the parameter both have the same type and equal values</td>
</tr>
<tr>
<td>equal?</td>
<td>True if the receiver object and the parameter have the same object ID</td>
</tr>
</tbody>
</table>

- Any value but nil is considered **true**
### Operator precedence and associativity

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>**</code></td>
<td>Right</td>
</tr>
<tr>
<td>!, unary + and -</td>
<td>Right</td>
</tr>
<tr>
<td><code>*, /, %</code></td>
<td>Left</td>
</tr>
<tr>
<td><code>+,-</code></td>
<td>Left</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>Left</td>
</tr>
<tr>
<td><code>+,-</code></td>
<td>Left</td>
</tr>
<tr>
<td><code>&gt; , &lt;= , &gt;=</code></td>
<td>Nonassociative</td>
</tr>
<tr>
<td><code>==, !=, &lt;=&gt;</code></td>
<td>Nonassociative</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>Left</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td>`=, +=, -=, *=, **=, /=, %=, &amp;=, &amp;&amp;=,</td>
<td></td>
</tr>
<tr>
<td><code>not</code></td>
<td>Right</td>
</tr>
<tr>
<td><code>or, and</code></td>
<td>Left</td>
</tr>
</tbody>
</table>

Highest-precedence operators are listed first. The method names for unary minus and plus are `-e` and `+e`, respectively.
Control Expressions

• Assignments can be used as control expressions
  – eg:

    ```
    while (next = gets) { ... }
    ```

• true as long a there is more input
• becomes false when input is exhausted
Selection and Loop Statements

• If statement

```plaintext
if control-expression
  statements
elsif control-expression
  statements
else
  statements
end
```

• Note: `no parentheses () around control expressions or brackets {} around statement blocks`
Unless

- Reverses if, no else or elsif

  unless control-expression

  statements

  end
Two kinds of case constructs

case expression
    when value then
        - statement sequence
    when value then
        - statement sequence
    else
        - statement sequence
end
14.4 Case Semantics

- Value is matched to when’s using `===`
  - Defined for all built-in classes
  - If the when value is a class, the comparison is true if the case value is an object of the class or one of its superclasses
  - If the value is a range, the case value matches if it is in that range
  - If the when value is a regular expression, the match is based on pattern matching
- When’s do not cascade, so no break statements are needed
Two kinds of case constructs

case

  when Boolean expression then expression

  ...

  when Boolean expression then expression

  else expression

end

The first true Boolean expression causes the matching expression to be evaluated as the value of the case

The else expression is used if none of the Booleans are true
Loops

- **while** loop repeats while control expression is true
- **until** loop repeats until control expression is true
- **loop** introduces an infinite loop (exit using **break**)
- No **for** loop in traditional C/Java etc sense
  - iterator methods instead (better!)
Fundamentals of Arrays

- *dynamic* in size
- can store *different types* of data in different elements
- Creating an array
  - `Array.new(size)`
  - `Array.new(size, value)`
    - A literal list such as `[2, 4, 6]`
- Element access through subscript `[sub]`
- Element assignment through `[sub]=`
  - `mylist[2]=6`
The for-in Statement

• Syntax

```plaintext
for variable in list
    statements
end
```

• The variable takes on each value in the list
  – This is not a reference but a *value copy*
Built-in Methods for Arrays and Lists

- `push`, `pop`, `shift`, `unshift`
- Arrays catenated with `+`
- Method `reverse` returns a reversed copy
- Method `reverse!` reverses the array
- `include?` searches an array
- `sort` sorts an array, returns a new array
14.5 An Example

- The process_names.rb example illustrates using arrays
Hashes

- Hashes are like arrays but use string indexes
  - indexes are *keys* and elements are *values*
  - elements are not ordered

- Creating a Hash
  - `my_hash = Hash.new`
    - Literal hash as in `my_hash = {"a"=>1, "b"=>2}`

- Element access using string ‘subscripts’
  - `myhash["c"] = 3`

- Elements are removed using `delete`, `clear`

- `has_key?` checks if a string is a key in a hash

*keys* and return arrays of the respective components.
Methods

• Functions or subprograms in Ruby are all *methods*

• Methods can however be defined outside of a class
  – in this case they belong to the default object *self*
  – called without an object reference (implicit)
Fundamentals of Methods

• Method syntax

    def name [ ( formal-parameters ) ]

    statements

    end

    – Parentheses can be omitted if there are no formal parameters

• Return statement ends execution of the method

    – With a value, it returns the value as the value of the method
    – If no return ends a method, the last expression is the value of the method (not recommended)
Local Variables

• Variables used in a method are local to the method
  – Global variables with the same name are hidden
• Local variable names must begin with a lowercase letter or an underscore
Parameters

- **Scalar** parameters are **passed by value**
  - formal parameters receive a *copy* of the object
  - e.g.
    ```python
def swap(x,y)  # erroneous attempt to swap values
t=x
x=y
y=t
end
```

```
a=1
b=2
swap(a,b)
print("a = ", a, "\n")  # a = 1
print("b = ", b, "\n")  # b = 2
```
Parameters

- **Arrays and hashes** are effectively *passed by reference*
  - changes to formal parameters change objects
  - eg

```ruby
def swapArray(list)  # swap values
  list.reverse!
  return list
end

alist = ['a', 'b']
swapArray(alist)
for char in alist
  print(char+' ')  # b a
end
```
Parameters

• The number of actual parameters must match the number of formal parameters
  – except if the last formal parameter is preceded by an asterisk, it receives all actual parameters not matched to earlier formal parameters

• Formal parameters can be assigned default values, so the corresponding actual parameter may be omitted

  def printlist (list, owner = 'Jo')
“Keyword” Parameters

- Some languages, such as Ada and Fortran 95, support *keyword parameters*
  - actual parameter specifies name of associated formal parameter
  - eg: `printlist(list => myRecords, owner => 'Jaye')`
  - advantage - eliminate risk of assigning to wrong parameter, particularly when there are many
- Ruby can achieve this using hashes...
  - eg: `find(age, {‘first’=>’Davy’, ‘last’=>’Jones’})`
  - if a hash literal is passed as an actual parameter and it follows all normal scalar parameters (and precedes array and block parameters), the braces can be omitted
  - or better still...
Symbols

• Ruby includes category of objects not found in other common languages - *symbols*
  
  – unquoted string preceded by a colon :  
  – instances of `Symbol` class
• Can be used to specify the keys of elements of hash literals when used as parameters
  
  – eg: `find(age, :first=>’Davy’, :last=>’Jones’)`
• Convention of Ruby and Rails to use symbols rather than literal strings for keys in hash literals used as parameters
Basics of Classes

• Syntax

    class class-name

    ...

    end

    – Class names begin with an UPPERCASE letter
    – Instance variables have instances for each object
      • names begin with @

• Constructor is named initialize

• Example Stack_class.rb implements a stack class

• Arguments to new are passed to initialize

  – eg. mystack = Stack.new(50)
Classes are Dynamic

- Class members can be dynamically added or removed at any time
  - added by including additional class definition
  - removed using the method `remove_method`
- Trade-off between readability (& reliability) versus flexibility
Access Control

- Different access for data and methods
  - unlike other languages!
- Instance data is *private* - this cannot be changed
  - to allow access must include “getter” and “setter” methods
  - Ruby provides short cuts for defining getter and setter methods
    - `attr_reader, attr_writer, attr_accessor`
Attributes

• Method `attr_reader` takes symbol arguments and creates variables and defines getter methods named after the symbols
  
  ```ruby
  • attr_reader :val
    – Defines a method named `val` that returns the value of `@val`
  ```

• Method `attr_writer` takes symbol arguments and defines an assignment method
  
  ```ruby
  • attr_writer :val
    – Defines a method named `val=` that allows assignment to the attribute as in
      ```ruby
      • obj.val = 17;
      ```
  ```

• Method `attr_accessor` creates getter and setter
Access Control

• Methods have three levels of access control
  – Public: every method has access
  – Private: only methods of the object have access
  – Protected: only methods of the class or subclasses have access

• Access control is dynamic
  – only checked during execution
  – can be changed by public, private and protected methods
Inheritance and Modules

- **Inheritance** syntax:

  ```
  class My_Subclass < Base_class
  ```

  - Access level of a method can be changed in a subclass

- **Modules** define namespaces that are used to hold sets of methods

  - included with `include` statement, eg
    ```
    include Math
    ```
  - inherit functions in module ("proxy superclass")
  - called a **mixin**
  - kind of “multiple inheritance”
Code Blocks and Iterators

- A **code block** is a sequence of statements delimited by braces or by do/end
  - can be passed to methods (following formal parameters) to create very useful constructs, in particular iterators
  - simple example

```ruby
> 4.times {puts("Hey!")}
Hey!
Hey!
Hey!
Hey!
```
Code Blocks and Iterators

- A code block can have parameters, listed at the beginning of the block, surrounded by vertical bars: \[ a, b \]

- The parameters can be set, for example in an iterator, each time the block is executed
  - eg. apply block to values in a list...

```ruby
> list = [2,4,6,8]
> list.each { |value| puts(value)}
2
4
6
8
```
Built-in Iterators

• `each` is a method of arrays and takes a block that has one parameter
  – block is executed with parameter taking each element of the list in turn
• `upto` takes an integer parameter and a block with one parameter
  – block is executed for each value in the range from the target integer to the actual parameter, inclusive

```ruby
> 5.upto(8) { |value| print(value)}
5 6 7 8
```
• `collect` creates an array from the results of applying a block to each element in an array

• eg.

```ruby
> list = [2, 4, 6, 8]
> list.collect { |value| value = value ** 2}
=> [4, 16, 36, 64]
```
User-defined Iterators

- Ruby allows you to *develop your own iterators* using code blocks
  - a call to `yield` executes the block
  - actual parameters of the call are matched with the block parameters
- Very useful and powerful feature!
User-defined Iterators

```ruby
> def pretty_print (list)
>   puts "Here comes my listing..."
>   for item in list
>     yield(item)
>   end
> end

> pretty_print(["a","b","c"]) {|thing| puts("---"+thing+"---")}
Here comes my listing...
---a---
---b---
---c---

> pretty_print(["a","b","c"]) {|thing| puts("-> "+thing+" <-")}
Here comes my listing...
-> a <-
-> b <-
-> c <-
```
Flow Control

Calling code

```ruby
pretty_print(["a","b","c"])
do |thing|
  puts("---"+thing+"---")
end
```

Iterator method

```ruby
def pretty_print(list)
  puts "My listing..."
  for item in list
    yield item
  end
end
```
Pattern Matching

• Pattern matching in Ruby is based on pattern matching in Perl
  – Patterns are delimited by slashes
  – =~ is the pattern matching operator
  – The `split` method is used to split a string with separators given by a regular expression
• The example `word_table.rb` uses the `split` method to divide up input lines into words
  – A hash is used to count word frequencies

• For example:
  – 'This is some test' =~ /(s\s*)/ will return 3
Remembering Matches

- Parts of a regular expression pattern can be enclosed in parentheses.
- If there is a match, the variables $1, $2 ... hold the strings matched by the parenthesized parts.
- If there is a match, $\`, $& and $' hold the part of the target before the match, the part matched and the part after the match.
- For example
  - $1$ in the previous example is “s ”
  - $\`$ in the previous example is “Thi”
  - $'$ in the previous example is “is some test”
  - $`$ in the previous example is “s ”
Substitutions

- Method `sub` substitutes its second parameter for the first match found of the first parameter

- Method `gsub` is similar but substitutes for all matches
  - Both methods create new strings

- `sub!` and `gsub!` change the target string

- The `i` modifier on a pattern causes the match to ignore letter case

- For example:
  - `'This is some test'.sub(/is/, 'at')` produces a new string `'That is some test'`