An Introduction to Ruby

CS342 - Fall 2016

Again, Why Ruby?

- Other strong OO languages aren’t as dynamic
  - C++, Java, Python
- It is dynamically typed
- It has closures - treats functions as variables
- It has mixins - generic methods shared between classes
- It is designed for design patterns
- Everyone’s doing it

Ruby with a Firehose

- This is not a ‘Learn Ruby’ course
  - If you are proficient in a C based language: Java, C++, Python, etc., you can pick up Ruby quickly
- If you have a Mac, you already have Ruby
  - just type irb in terminal
  - We will be using Ruby 2.0 or greater
  - Installation instructions for other Systems
    - *https://www.ruby-lang.org/en/documentation/installation/*

‘Hello World’

- script
  - uses .rb extension : hello.rb
  - running a script : > ruby hello.rb
- puts()
  - prints to console with a newline
  - puts(‘Hello World’)
- The parenthesis are optional in ruby for all methods
  - puts ‘Hello World’

Single or Double Quoted Strings

- Single quotes produce uninterpolated (wysiwyg) strings
    - string: ‘This is \n a string’
    - output: This is \n a string
- Double quotes produce interpolated (preprocessed) strings
    - string: “This is \n a string”

      output: This is

       a string

Variables

- case Sensitive
  - follows same restrictions as Java, C, C++
- Variable convention:
  - use underscores for spaces, i.e. snake case
    - *my_variable*
- comments
  - # - single line
  - multi-line comments =begin

```
comment
=end
```

String Features

- Strings are flexible and easy to work with
- batman = “Bruce” + “ “ + “Wayne”
- Reassignment
    - robin = “Dick Greyson”

      robin = ‘Jason Todd’

      robin = “Tim Drake”

User Input

- We use the ‘gets()’ method to get the user input
  - input comes in as a string
- gets() grabs everything you input up to the newline
  - including a new-line character because you pressed Enter, which is a character
- If you need to remove the newline, use ‘chomp()’
  - chomp() removes newlines from the end of a string
    - *http://ruby-doc.org/core-2.2.0/String.html#method-i-chomp*

The string and all

- Program:

  -
    ```
    puts(“What is your name?”)
    name = gets()
    puts(“Hello ” + name + “, how are you?”)
    ```
- Ouput

  -
    ```
    what is your name?
    steven
    Hello steven
    , how are you?
    ```

The string and nothing but the string

- Program:

  -
    ```
    puts(“What is your name?”)
    #removes the newline at the end of the string
    name = gets().chomp()
    puts(“Hello ” + name + “, how are you?”)
    ```
- Ouput

  -
    ```
    what is your name?
    steven
    Hello steven, how are you?
    ```

Dynamic Typing

- Variables do not need to be declared
- Variables take on the types of their values when assigned

    -
      ```
      #typed as int
      superteam = 4
      #retyped as string
      superteam = “Fantastic Four”
      #retyped as array
      superteam = array(“Mr. Fantastic”, “Invisible Woman”, “Human
      Torch”, “The Thing”)
      ```

Conventions

- Ruby uses convention over configuration
  - Just agree on a way to do things rather than configuring an environment
- Starting a variable with uppercase denotes a constant
  - use all uppercase for a constant : PI_VALUE = 3.14
  - Constants are dynamic, i.e. overwritable
- Arithmetic with only integers produces integers
  - value = 7 / 2 # 3
- Arithmetic with at least one floats produces a float
  - value = 7/ 2.0 # 3.5

Everything is Objects

- Because everything is an object, everything has built in methods
  - 7.class # Gives you the class Fixnum
  - 3.14159.class # Gives you the class Float
- Because everything is an object, there are no primitives
  - Ruby has no ‘built-In’ types
- This means all variables are references
  - ruby does not have assignment

nil, true, and false

- *nil*: similar to C null, except it too is an object
    - nil.class returns NilClass
- *true*: yup, it’s an object
    - an instance of TrueClass
- *false*: surprise, also an object
    - an instance of FalseClass

Boolean Review

- not('z' > 'a')
- not('a' > 'z')
- if(0)

Our First Design Pattern

- There is only one instance each of ‘nil’, ‘true’, and ‘false’
  - You cannot create a NilClass, TrueClass, or FalseClass object
- A single instance of a class in which another instance is forbidden is a Singleton
  - The constructor always returns a reference to the one true object
    - *We will come back to this pattern much later in the semester*

Conditional Statements

If statement

```
if (<condition>)
    ...
elsif (<condition>)
    ...
else(<condition>)
    ...
end
```

While statement

```
while (<condition>)
    ...
end
```

each statement

```
array.each do |i|
    ...
end
```

Arrays

- Two syntaxes to create arrays
  - array = [ ]
  - array = Array.new
- Arrays are 0 indexed
  - array[0] gives you first element, array[2] the third element
- Get an array length array.length
- Strings are indexed just like arrays

Dynamically sized and typed arrays

- You do not need to size or resize arrays
- array = []
  array[3] = 1
  - automatically adds the first 3 elements and initializes them to nil
  - [nil, nil, nil, 1]
- arrays are not limited to single types
  - [“hello”, 123, my_object] is valid

Hashes

- built in data type defined with { }
  - {“first”=> “Hello”, “second”=>”World”}
- Index can be any valid object
  - prefer symbols for indices
  - symbols are like constant strings defined with :
    - *{:first => “Hello”, :second =>”World”}*
  - You can use symbols just like variables, except they are immutable (do not change)
    - *essentially they are constant strings, more on them later in the semester*

Classes

- Must use uppercase to start a class name
  - Should use camelcase
- @ define instance variables or data members
  - You do not need to predefine them

```
class BankAccount
  def initialize( account_owner )
    @owner = account_owner
    @balance = 0
  end
  def deposit( amount )
    @balance = @balance + amount
end
  def withdraw( amount )
    @balance = @balance - amount
end end
```

Classes

- The constructor is called ‘initialize’
  - all methods are preceded with ‘def’ to let the compiler know it is a method
- Question? What’s the difference between a method and a function
  - Does ruby have functions?

```
class BankAccount
  def initialize( account_owner )
    @owner = account_owner
    @balance = 0
  end
  def deposit( amount )
    @balance = @balance + amount
end
  def withdraw( amount )
    @balance = @balance - amount
end end
```

Creating and Using Objects

- Initializing an Object:
  - my_account = BankAccount.new(‘Joe’);
- All instance variables are private
  - We must have getters and setters for every instance variable
    - *We can write a setter with the same names as the instance variable*

      -
        ```
        def inst_var=(val)
            @inst_var = val
        end
        ```

      -
        ```
        def inst_var()
            inst_var
        end
        ```

Attributes Accessors and Readers

- Fortunately, Ruby will do this for you
  - Add the following to your class
    - *attr_accessor :var, :var2, :var3*
      - notice they are symbols, not variable names
  - Now the getter and setter methods have been automatically generated
- For read only
  - Only the getter methods will be available
    - *attr_reader :var, :var2, :var3*

Self-Reference

- To reference an object within the object methods, use the keyword ‘self’
    - works exactly like ‘this’ in other languages
- Example:

    -
      ```
      def myMethod()
          puts(“I am ” + string(self))
      end
      ```

Inheritance

- Ruby only supports single inheritance
  - All classes inherit from the superclass Object (eventually)
    - *Like Java*
- Specify a superclass with < **SuperClass**
  - example: class ChildClass < ParentClass
- Call superclass methods with **super**
  - example: def initialize(val)

```
    @var = val
    super()
end
```

More about Methods

- Default Parameters
  - def myMethod(param = nil)
    - *default parameters must come last*
- Methods automatically return the last value
  - Do not need a return statement (though good style)
- Automatic instance variable creation
  - Anything you want to be a instance variable (persistent for the life of the object), just precede with @
    - *@inst_var*

Modules

- Modules are mixins
  - snippets of code that can be reused in classes
  - use ‘include’ to include them in your classes
- Example:

  -
    ```
    module Hello
        def hello()
            puts(“Hello”)
        end
    end
    ```

```
class Greetings
    include Hello
end
```

Begin / Rescue

- Ruby try/catch block works the same as other languages
    - begin

      …

      rescue

      … #catch the exception

      end
- ensure keyword
    - make sure something is done, such as closing files, not matter what

      happens

Exceptions

- Ruby exceptions use begin/rescue
  - You can catch any error (bad style) or define specific errors (good style)
- Example:

  -
    ```
    begin
    …
    rescue ZeroDivisionError
        puts(“You tried to break the universe. Stop it.”)
    end
    ```

Raising Exceptions

- Raise your own exception with ‘raise’
  - Use exceptions to define interfaces and abstract classes
    - *raise NoMethodError*
- Raise specific exceptions
  - raise ZeroDivisionError
- Raise generic exceptions with a string message
  - raise “You messed up”

Raise...Unless

- raise an exception, unless a condition is met
    - be proactive in catching errors

```
def inverse(x)
    raise ArgumentError('Argument is not numeric')
        unless x.is_a? Numeric
    1.0 / x
end
```

Exception Rules

- Always use exceptions rather than let your program break
- Never ‘Swallow’ exceptions without solving them or displaying a message and quitting
- Always rescue specific exceptions first, then generic
- Raise specific exceptions when possible, or give a specific message

Separate Source files

- Each class should go into its own file
  - any associated functionality or data should also go into that file
- include separate source files with ‘require_relative’
  - do not need to add .rb, to include the file ‘MyFile.rb
    - *just add ‘require_relative MyFile’*
  - require_relative automatically ensures you include a file only once
    - *you may also us ‘require’ but must provide an absolute path*

main

- You should always have a main.rb for the main procedure (driver) of the program and one off functions
    - Your ‘top-most’ program logic goes here
- The driver code should be inside a main function, not a global script, that gets called globally

    -
      ```
      def main()
          ...
      end
      main()
      ```
        - *The main() call should be your ONLY global code*

Classwork 3