Ruby Language Overview

- Dynamically typed
- Interpreted
- Can be modified at runtime
- Object oriented
- Blocks & lambdas
- Nice support for Regular Expressions

Getting Started with Ruby (irb)

IRB = Interactive RuBy console

in your terminal, type:

```
$ irb
>> 4 + 4
=> 8
```

General Syntax Notes

Ruby aims to be elegant and readable so punctuation and boilerplate are minimal. No semicolons are needed to end lines, but may be used to separate statements on the same line. Indenting is not significant, unlike python.

# this is a comment

A backslash (\) at the end of a line is used for continuation

Parentheses required only as needed to specify precedence.

```
>> "Hello".gsub ‘H’, ‘h’
=> "hello"

>> "Hello".gsub("H", "h").reverse
=> "olleh"
```

Variables

Variable names are composed of letters, numbers and underscores

There is no need to declare variables (like C or Java)

In Ruby, everything is an object.

There are no special literals like you may have seen in other languages, even numbers and strings are objects.

Try this in irb:

```
>> "test".upcase
>> "test".class
>> "test".methods
```

Notice that everything evaluates to some value.

```
>> 2 + 2
>> (2+2).zero?
```

Methods

- Methods are Messages
- Operators are methods

Method names may end with "?", "!", or "=".

- Methods ending with a "?" imply a boolean return value. These kinds of methods are called "predicates." Common examples are empty?, nil?, and instance_of?
- Methods ending with "!" imply something changes to the object itself, like strings being modified in place (eg, "upcase!"), or in Rails this convention is used in methods that throw exceptions on failure.
- Methods ending with "=" are setters.

An Introduction to Classes and Methods

We define a new type of object by writing a "class." The behavior of the object is defined in blocks of code we call "methods."

```
class Calculator
  def describe
    "I am a really neat calculator!"
  end
  def add(a, b)
    a + b
  end
end
```

Return Values

Notice that the value returned from the method is whatever was last evaluated.

Ruby has a return keyword, but people use it rarely. It will let you return from the middle of a method, which is rarely a good idea.

To load a class (or any ruby file), start irb in the same directory, load the calculator, then you can use the Ruby class defined in the file.

```
>> load "calculator.rb"
>> c = Calculator.new
>> c.describe
>> c.add(1,2)
>> c.add(56, 45)
```

Instance Variables

When we want to associate data with an object, we define an instance variable by naming the variable with an @ at the beginning of its name.

```
class Calculator
  def initialize
    @num_calculations = 0
  end

  def how_many
    @num_calculations
  end

  def add(a, b)
    @num_calculations = @num_calculations + 1
    a + b
  end
end
```

More about Classes

Class names are constants, and must start with a capital letter.

Classes can inherit from other classes. The derived class, “subclass” can access behavior and data from the *super* class.

Here, **`SpecialThing`** is a subclass of **`Thing`**.

```
class Thing
  def do_something(a,b)
    a + b
  end
end
class SpecialThing < Thing
  # special code would go here
end
```

Attributes

To make an instance variable accessible outside of the context of a class, methods need to be declared to access it...

```
class Thing
  def name=(n)
    @name = n
  end
  def name
    @name
  end
end
```

Ruby provides short-cuts that allow you to access instance variables without declaring methods

```
class Thing
  attr_accessor :name
  attr_reader :created_at
  attr_writer :something
```

Special Kinds of Variables

Class variables belong to the innermost enclosing class or module. Class variables used at the top level are defined in Object, and behave like global variables.

```
var   # a local variable
@var  # instance variable
@@var # class variable

VAR   # constant
$var  # global variable
```

Constants

Constants defined within a class or module may be accessed unadorned anywhere *within the class or module*. Class variables belong to the innermost enclosing class or module. Class variables used at the top level are defined in Object, and behave like global variables.

Outside the class or module, they may be accessed using the scope operator, ``::” prefixed by an expression that returns the appropriate class or module object. Constants defined outside any class or module may be accessed unadorned or by using the scope operator ``::” with no prefix.

Constants may not be defined in methods.

Class variables are available throughout a class or module body. Class variables must be initialized before use. A class variable is shared among all instances of a class and is available within the class itself.

Example:

```
TAX = 0.085

class Payment
   TIP = 0.15

   def calculate(amount)
      amount += amount * (TIP + TAX)
   end
end
```

```
>> TAX
=> 0.085
>> TIP
NameError: uninitialized constant TIP
   from (irb):11
>> Payment::TIP
=> 0.15
>> p = Payment.new.calculate(100)
=> 123.5
```

Class Variables

A class variable will store data that is shared by all instances of the class.

Example:

```
class Song
   @@total_songs = 0

   def initialize
     @@total_songs += 1
   end

   def Song.total
    @@total_songs
  end
end
```

```
>> load ‘song.rb’
=> true
>> s = Song.new
>> Song.total
=> 1
>> s2 = Song.new
>> Song.total
=> 2
```

Modules as Mixins

Ruby does not support multiple inheritance directly but Ruby modules pretty much eliminate the need for multiple inheritance, providing a facility called a **`mixin`**.

```
module Speech
  def speak
    puts @noise
  end
end

class Dog
  include Speech
  def initialize
    @noise = "woof"
  end
end

class Bird
  include Speech
  def initialize
    @noise = "tweet"
  end
end
```

```
>> Dog.new.speak
woof
 => nil
>>> Bird.new.speak
tweet
 => nil
```

You may include multiple modules. Note that name clashes are not flagged -- the last one wins, following Ruby's policy of re-definition.

Conditionals

if expressions are used for conditional execution. The values false and nil are false, and everything else are true.

```
  if age >= 12 then
    print "adult fee\n"
  else
    print "child fee\n"
  end
```

```
  if age >= 18 then
    print "can vote\n"
  elsif age >= 13 then
    print "can facebook\n"
  else
    print "child\n"
  end
```

if may act as modifiers

```
print "debug\n" if $debug
```

The case expressions are also for conditional execution.
Case comparisons are done by the "threequal" operator ===.

```
case $age
   when 0 .. 2
   "baby"
   when 3 .. 6
   "little child"
when 7 .. 12
   "child"
when 12 .. 18
    # Note: 12 already
matched
    "youth"
else
    "adult"
end
```

conditionals brought to you by:
http://web.njit.edu/all_topics/Prog_Lang_Docs/html/ruby/syntax.html#control

Truth

Checking for false:

```
if !(name == “superman”) …
if not (name == “superman”) …
```

“unless” provides us with another way of checking if a condition is false:

```
unless human
  status = “superhero”
end
```

Everything evaluates to true except for:

false

nil

Therefore:

0 is true

"" is true

Boolean Operators

Evaluates left hand side, then if the result is true, evaluates right hand side.

```
test && set

test and set
```

Evaluates left hand side, then if the result is false, evaluates right hand side. The value returned is the right-hand side.

```
demo || die

demo or die
```

A common idiom is to use ll= to initialize an instance variable if it has not yet been set.

```
@x ||= "something"
```

String Interpolation

Ruby provides a concise syntax for inserting the result on an expression into a string.

```
>> a = "world"
>> puts "hello #{a}"
hello world

>> a = 2
>> puts "hello #{a}"
hello 2

>> a = nil
>> puts "hello #{a}"
hello
"string #{ruby code} string"
```

Common String Operations

```
"Ru" + "by"
=> "Ruby"

"I" << "love" << "Ruby"
=> "IloveRuby"

myString = "Welcome to JavaScript!"

myString["JavaScript"]= "Ruby"

puts myString
=> "Welcome to Ruby!"

myString[10]= " the land of "

puts myString
=> "Welcome to the land of Ruby!"

s[8..18] = "friends"
=> "friends"

puts myString
=> "Welcome friends of Ruby!"
```

To change case:

`capitalize` - first character to uppercase
`downcase` - all to lower case
`swapcase` - changes the case of all letters
`upcase` - all to upper case

To rejustify:

`center` - add white space to center the string
`ljust` - pads string, left justified
`rjust` - pads string, right justified

To trim:

`chop` - remove last character
`chomp` - remove trailing line separators
`squeeze` - reduces successive equal chars
`strip` - deletes leading and trailing white space

To examine:

`count` - return a count of matches
`empty?` - returns true if empty
`include?` - is a specified target string present?
`index` - return the position of one string in another
`length` or `size` - return the length of a string
`rindex` - returns the last position of one string in another
`slice` - returns a partial string

To alter:

`replace` - replace one string with another
`reverse` - turns the string around
`slice!` - deletes a partial string, returns deleted part
`split` - returns an array of partial strings exploded at separator
`tr` - to map all specified char(s) to other char(s)
`tr_s` - as tr, then squeeze out resultant duplicates
`unpack` - extract string into an array using a template

To iterate:

`each_line` - process each line in a string
`each_byte` - process each byte in turn

To format:

`sprintf(format, ...)` - returns a string formatted according to a format like the printf in C

```
"cat dog hat".split(" ").join(",")
=> "cat,dog,hat"

"abc".each_byte{|c| printf "<%c>", c}; print "\n"
<a><b><c>
=> nil

"hello".gsub(/[aeiou]/, '*')
=> "h*ll*"

65.chr => "A"
```

gsub returns a modified string, leaving the original string unchanged.
gsub! directly modify the string object on which the method was called.

For more information, see: http://www.ruby-doc.org/core/classes/String.html

Collections

Arrays are sized dynamically and can be of mixed types.

```
>> a = [1, 2, 3]
>> a.push "four"
=> [1, 2, 3, "four"]

>> a.pop
=> "four"

>> a[0]
=> 1
```

%w shortcut

Since many arrays are composed of a series of strings, Ruby provides a shortcut for this kind of array creation.

```
>> animals = %w{cat, dog, frog}
=> ["cat,", "dog,", "frog"]
```

Hashes an unordered list of key-value pairs.

In other languages they may be called a dictionary, a map or an associative array

=> is called a "hash rocket"

```
>> states = {"MA" => "Massachusetts",
        "CA" => "California"}

>> states["MA"]
=> Massachusetts
>> my_hash = {:a_symbol => 3,
         "a_string"=> 4}

>> my_hash[:a_symbol]
=> 3
```

Iteration

```
>> 4.times do
?>  puts "hello"
?> end

hello
hello
hello
hello

>> 2.times {puts "foo"}
foo
foo


my_array = ["cat", "dog", "world"]
my_array.each do |item|
  puts "hello " + item
end


my_hash = { :type => "cat",
            :name => "Beckett",
            :breed => "alley cat" }

my_hash.each do |key, value|
  puts "My " + key.to_s + " is " + value
end
```

Enumerables

Ruby's Enumerable module has methods for all kinds of tasks that operate on a collection. Most likely if you can imagine a use for the #each method other than simply iterating, there is a good change a method exists to do what you had in mind. Arrays and hashes are Enumerable.

- Collection objects (like Array, Hash, etc.) "mixin" the Enumerable module
- The Enumerable module gives objects of collection classes additional collection-specific behaviors.
- The class requiring the Enumerable module must have an #each method because the additional collection-specific behaviors given by Enumerable are defined in terms of #each

```
>> talk {puts "whatever"}
hello
whatever
goodbye
=> nil
```

```
>> talk_much {puts "hey"}
hello
hey
hey
hey
bye
=> nil
```

Creating an Enumerable Collection

All you need to do to make your own class respond to all of the Enumerable methods is to define an #each method, since every Enumerable method can be implemented with each.

How do you know if something is Enumerable?

To see if you can call these methods, you can check if an instance responds to a particular method (preferred in code) or you can test the instance or class:

```
>> a = [1,2,3]
  => [1, 2, 3]

>> a.respond_to? :any?
  => true

>> a.is_a? Enumerable
  => true

>> Array < Enumerable
  => true
```

Try this to see all of the classes that mixin Enumerable:

```
>> ObjectSpace.each_object(Class) { |cl| puts cl if cl < Enumerable}
```

Sources

This section is based on this excellent post:
http://vision-media.ca/resources/ruby/ruby-enumerable-method-examples

Please also see reference docs: http://ruby-doc.org/core/classes/Enumerable.html

Our test subject for the following examples will be the following array:

```
     vehicles = %w[ car truck boat plane helicopter bike ]
```

Standard Iteration With Each

Standard iteration is performed using the each method. This is typical in many languages. For instance in PHP this would be for each however in Ruby this is not built-in, rather this is a method call on the vehicles array object. The sample code below simply outputs a list of our vehicles.

```
vehicles.each do |vehicle|
  puts vehicle
end
```

Modifying Every Member with Map

The map method allows us to modify each member in the same way and return a new collection with new members that were produced by the code inside our block.

```
>> vehicles.map { |v| v.upcase }
  => ["CAR", "TRUCK", "BOAT", "PLANE", "HELICOPTER", "BIKE"]
```

Searching Members With Grep

The grep method allows us to 'search' for members using a regular expression. Our first example below returns any member which contains an 'a'. The grep method also accepts a block, which is passed each matching value, 'collecting' the results returned and returning those as shown in the second example.

```
vehicles.grep /a/
  # => ["car", "boat", "plane"]

vehicles.grep(/a/) { |v| v.upcase }
  # => ["CAR", "BOAT", "PLANE"]
```

Evaluating All Members With all?

The all? method accepts a block and simply returns either true or false based on the evaluation within the block.

```
vehicles.all? { |v| v.length >= 3 }
  # => true
```

```
vehicles.all? { |v| v.length < 2 }
  # => false
```

Checking Evaluation For A Single Using Any?
The any? method compliments all? in the fact that when the block evaluates to true at any time, then true is returned.

```
vehicles.any? { |v| v.length == 3 }
  # => true

vehicles.any? { |v| v.length > 10 }
  # => false
```

Enumerable Methods with Complex Data

For our next examples we will be working with vehicles as well, however more complex data structures using Hashes.

```
irb
>> load ‘vehicles.rb’
>> $vehicles
```

Collecting a List
The collect method is meant for this very task. Collect accepts a block whose values are collected into an array. This is commonly used in conjunction with the join method to create strings from complex data.

```
$vehicles.collect { |v| v[:name] }
  # => ["Car", "Truck", "Boat", "Plane", "Helicopter", "Bike", "Sea
  Plane"]


$vehicles.collect { |v| v[:name] }.join ', '
  # => "Car, Truck, Boat, Plane, Helicopter, Bike, Sea Plane"
```

Note: this is a synonym for map

Finding Members Using The Find Method
The find and find_all methods are the same although different in the obvious fact that one halts iteration after it finds a member, the other continues and finds the rest.

Consider the following examples, we are simply trying to find members that match names, have many wheels, or are ground or air based. The collect method is used to collect arrays of the names for demonstration display purposes, instead of displaying data from the #inspect method.

```
$vehicles.find { |v| v[:name] =~ /Plane/ }[:name]
  # => "Plane"

$vehicles.find_all { |v| v[:name] =~
/Plane/ }.collect { |v| v[:name] }
  # => ["Plane", "Sea Plane"]

$vehicles.find_all { |v| v[:wheels] > 0
}.collect { |v| v[:name] }
  # => ["Car", "Truck", "Bike"]
```

```
class MyCollection

   include Enumerable
   #lots of code

   def each
      #more code
   end
end
```

```
$vehicles.find_all { |v| v[:classes].include? :ground }.collect { |v|
v[:name] }
  # => ["Car", "Truck", "Bike", "Sea Plane"]


$vehicles.find_all { |v| v[:classes].include? :air }.collect { |v|
v[:name] }
  # => ["Plane", "Helicopter", "Sea Plane"]
```

Iterating With Storage Using Inject
When you are looking to collect values during iteration the inject method is the perfect one for the job. This method accepts a initialization parameter which is 0 and [] in the case below, this is then passed

```
$vehicles.inject(0) { |total_wheels, v| total_wheels += v[:wheels] }
  # => 10

$vehicles.inject([]) { |classes, v| classes + v[:classes] }.uniq
  # => [:ground, :water, :air]
```

Regular Expressions

Regular expressions allow matching and manipulation of textual data. They are abbreviated as regex or regexp, or alternatively, just patterns

Using Regular Expressions in Ruby

- Scan a string for multiple occurrences of a pattern.
- Replace part of a string with another string.
- Split a string based on a matching separator.

Regular expressions appear between two forward slashes: /match_me/

Match Method

Match is a method on both String and Regexp classes.

```
>> category = "power tools"
  => "power tools"

>> puts "on Sale" if category.match(/power tools/)
  on Sale

>> puts "on Sale" if /power tools/.match(category)
  on Sale
```

Match Operator =~

The match operator is like the match method, but returns the index of the match or nil. It is also defined for both String and Regexp classes.

```
>> category = "shoes"
  => "shoes"

>> puts "15 % off" if category =~ /shoes/
  15 % off

>> puts "15 % off" if /shoes/ =~ category
  15 % off

>> /pants/ =~ category
  => nil

>> /shoes/ =~ category
```

```
  => 0

>> category = "women's shoes”
>> /shoes/ =~ category
  => 8
```

scan to find multiple matches:

```
>> numbers = "one two three"
  => "one two three"

>> numbers.scan(/\w+/)
  => ["one", "two", "three”]
```

split on a regular expression:

```
>> "one two\tthree".split(/\s/)
  => ["one", ”two", "three"]
```

gsub to replace the matched pattern

```
”fred,mary,john".gsub(/fred/, “XXX”)
  => “XXX,mary,john”
```

may also be called with a block

```
"one two\tthree".gsub(/(\w+)/) do |w|
   puts w
end

one
two
three
```

Title Case Example

Capitalize All Words of a Sentence:

```
>> full_name.gsub(/\b\w/){|s| s.upcase}
  => "Yukihiro Matsumoto"
```

Blocks

Blocks are nameless functions. Blocks were designed for loop abstraction. The most basic usage of blocks is to let you define your own way for iterating over the items in a collection.

```
def talk
  puts "hello"
  yield
  puts "goodbye"
end
```

You can call yield multiple times:

```
def talk_much
  puts “hello”
  yield
  yield
  yield
  puts “bye”
end
```

You may also explicitly declare the block as an argument and then use the “call” method to invoke the block. The following example is functionally identical to the first:

```
def talk(&block)
  puts "hello"
  block.call
  puts "goodbye"
end
```

If you want to turn a snippet of code into a block, you can use the lambda function:

```
def contrived_example
    do_something = lambda { puts "foo" }
    do_something.call
end
```

For more information about blocks, lambdas and their close cousin Proc, read this good article: http://eli.thegreenplace.net/2006/04/18/understanding-ruby-blocks-procs-and-methods/

method_missing

When you send a message to a Ruby object, Ruby looks for a method to invoke with the same name as the message you sent.

There are a bunch of different ways to send the message, but the most common one is:

```
obj.method_name
```

But you can make the fact that you are sending a message explicit with:

```
obj.send(:method_name)
```

Characters that need to be escaped

. | ( ) [ ] { } + \ ^ $ * ?

with a backward slash (\).

First it looks in the current self object's own scope: in local variable, then instance methods. Then it looks in the list of instance methods that all objects of that class share, and then in each of the included modules of that class, in reverse order of inclusion. Then it looks in that class's superclass, and then in the superclass's included modules, all the way up until it reaches the class Object. If it still can't find a method, the very last place it looks is in the Kernel module, included in the class Object. And there, if it comes up short, it calls method_missing.

You can override method_missing anywhere along that method lookup path, and tell Ruby what to do when it can't find a method.

http://www.thirdbit.net/articles/2007/08/01/10-things-you-should-know-about-method_missing/

```
class Thing
  def method_missing(m, *args, &block)
    puts "There's no method called #{m} here -- please try again."
    puts "parameters = #{args.inspect}"
  end
end
```

```
>> t = Thing.new
>> t.anything("ddd",3)
  There's no method called anything here -- please try again.parameters =
  ["ddd", 3]=> nil
```