**LYNDA.COM RUBY TUTORIAL NOTES**

Ruby – not a compiled language; don’t have to run through another language to run our application; object oriented, reads like English

-unsurprising syntax, naming, and behavior

-whitespace independent – doesn’t matter about spacing

-no semi-colons to deal with

-syntactic “sugar” – write in a much simpler way

Ruby on Rails – web framework written in Ruby; Ruby not just for web

Ruby –e puts 123 with a line return

Ruby –e print 123 no line return

cd.. to go back; make sure correct route followed

irb - interactive ruby shell – interactive code in real life, like a caluculator, way to test your code quickly

simpler prompt in irb: irb—simple-prompt

quit – gets you out of irb

nil = not zero; nothing; nul

i.e., the return value of puts is nil

“Hello”.reverse => “olleH”

in terminal Ruby – ri.upcase or ri.Object#.inspect = to look up commands or functions you want to know what they do, you can use ri

q – to get out of it

Ruby classes – think of a classroom that has objects like desks, students, etc. and can move them around objects; objects can also be abstract like their conversation

Variables – NOT objects; may seem like they are objects but they are not; undefined until you define them; always use lower case w/ an underscore between words; give them good common sense names

Variable Scope Indicators: access to variables within classes

Global $variable

Class @@variable

Instance @variable

Local variable

Block variable

Integers: 2 types 1. Integers – numbers

2. Floats – floating point numbers , decimals, or numbers w/precision

1. Integer example: x = 4

x += 2 or x = x + 2 is the same or 6

Subclasses of integers: Fixnum or Bignum (Ruby switches back and forth and all done internally depending on the size of number

-200.abs (absolute value = 200)

200.next (next integer = 201)

2. Floats example: x = 10

x.class => Fixnum

y = 10.0

y.class => Float

With floats, tell Ruby precision is important as in 10/3 = 3, Ruby erases the precision after the decimal point; if you want the precision 10.0/3 = 3.33333 will produce what you want.

Other functions: 12345.6789.round => 12346

12345.6789.to\_i => 12345

12345.6789.floor => 12345

12345.6789.ceil => 12346

Strings

“ “ or ‘ ‘ for strings; you can add strings together like numbers or multiply as in “Hello”\*5 => “HelloHelloHelloHelloHello”

‘I\’m escaped.’ => “I’m escaped.”

\ will keep the actual quotes; \\ if you needed the \ in there

Example:

puts “\ta\tb\nc\nd” would produce:

a b

c

d

#{variable} – only works with double-quoted strings; can drop in variables in expressions

object.reverse.upcase.length – you can daisy chain methods w/strings or objects

Arrays

Like a file folder and can be empty; if you want a specific order

What can go in an array: strings, numbers, other arrays, mixed types

Array = [1, “2”, 3.0, “Dog”, [“a”,”b”]]

Array << “ “ – adds or appends to the end of the array

Array.to\_s or Array.join would do the same thing

Array.uniq – no repeat values

Array! – permanently changes the array

Array.delete – can delete @ the position in the array or you can tell it exactly what to delete, i.e.,:

Array.delete\_at(4) would delete whatever is at index 4 or array.delete(4) would delete an actual 4

Hashes

Unordered collection of objects; instead of indexed by position it’s indexed by key-value pairs; like a hanging file folder where we want to find a label; so use an array when order really matters; use a hash when a label matters

Hash {} use the curly brackets

Person = {‘first\_name’ => ‘Dolly’, ‘last\_name’ => ‘Throckmorton}

Person [‘last\_name’]

Person.key(‘Throckmorton’)

Key is to the left of the =>

Value is to the right of the =>

Hash.to\_a => will return all the items in the hash as an array of key-value pairs

Symbol – NOT a variable; label to identify a piece of data; stored in memory one time while as a string is stored each time in memory

:test.object\_id - will return the same object id every time

“test”.object\_id - will return a different object id every time

hash – string as a key, it’s close or good enough; however, if the sequence of characters is important, a string is best; but if a label is enough, then a hash is used

Operators

X == 1 Comparison

X = 1 Assignment

! NOT operator

!= not equal to

&& and

|| or

then of course <,>, <=, and >=

2.between?(3,4)

false – to see if something appears between something else

Range

Range of numbers; could be an array but 2 types:

1. Inclusive Range 1..10 would include 10
2. Exclusive Range 1…10 would not include 10

Applying a method like .class to a range you must use (1..10).class

Example: x = 1..10

Z = [\*x] splat operator

Z = 1,2,3,4,5,6,7,8,9,10

You can also do this with LETTERS!!

Constant

Not used often; similar to variables; not true objects, point to objects

A constant is constant; use a capital letter in a constant

In Ruby, it will change your constant but will give you a warning.

For example:

TEST = 1

=> 1

TEST = 2

* 2 but will give you a warning

Conditionals:

Control structure to provide actions on your Ruby code

If Boolean

elsif

if

feature – puts conditionals in same line of code if you prefer, i.e.,

puts “This is Dolly” if name == “Dolly”

Convenience Conditionals:

1. Unless: if! Boolean or if not true, do this
2. Case: when Boolean or when true, do this else this, end; when you have a lot of parallel assignments then this is helpful: case test\_value
3. Ternery operator: shortened else and elsif; the : acts like else or elsif

Boolean? code 1: code 2 – and this code is short or quick code not involved

1. or/or-equals – using the goal posts or || as in x=y||z

Example: unless x (if value is x, keep)

x=y (if not then set to y)

end shorter version: x||=y

Loops:

loop do

code block between do and end

end

Controlling loops:

1. Break = terminate whole loop
2. Next = jump to next loop
3. Redo = redo the loop
4. Retry = starts the whole loop over fro the beginning

X = 0

loop do

x+=2

break if x >=20

puts x

end output is 2 4 6 8 10 12 14 16 18

While Boolean

Code block

End

Until Boolean so, while this is not true, do this code block

Code block

End

X=0

While x < 20

X+=2 matters what x is at start of loop

Puts x

End

Can also put in one line of code if simple statement:

X = 3245

Puts x/=2 until x<=1

Iterators

Like loops but traverse over a fixed set of data; so, you’ll do a code block with each set of data

5.times do

puts “Hello”

end

or 1.upto(5) {puts “Hello”}

or 5.downto(1) {puts “Hello”} these are all shortened syntax

or (1..5).each {puts “Hello”}

do, end - code block

{ } also code block

1.upto(5) do |i|

puts “Hello” +i.to\_s

end

output is: Hello 1 Hello 2 Hello 3 Hello 4 Hello 5

fruits = [‘banana’, ‘apple’, ‘pear’]

fruits.each do |fruit|

puts fruit.capitalize

end

or

for fruit in fruits

puts fruit..capitalize

end

output is the same: Banana Apple Pear

break, next, redo and retry also work in iterators; redo and retry work better with interators

Iterators can be more useful in loops:

Integers/floats: times, upto, downto, step

Range: each, step

String: each, each\_line,each\_byte

Array: each, each\_index, each\_with\_index

Hash: each, each\_key, each\_value, each\_pair

Code Blocks:

Do

…. Multi-line code block

End

Array.each{|num| puts num \* 20}

|num| - ruby doesn’t keep this code block or num after if finished the block and it’s only accessible within those goal posts.

Local and block variables have different scopes

So, if you do: num = 1

Array = [ 1,2,3,4,5]

Array.each {|num| puts num \* 20 + x}

Output is 21,41,61,81,101

Puts num => 1; so ruby holds on to the original value

Give local variables more long, descriptive names while block variable names should be short

Common Methods of using code blocks:

1. Find/Detect, find\_all/select, any?, all?, delete\_if

Find/detect will yield an object or nil, find\_all/select will yield an array, any? And all? Are Booleans, delete\_if will delete if it agrees with the code block

Examples:

(1..10).detect{|i| I % 3 == 0} =>1

(1..10).detect{|i| (1..10).include?(1\*3)} =>1

[\*1..10].delete\_if {|i| i%3 == 0} => 1,2,4,5,7,8,10

1. Merge – use for hashes only

Example: h1 = {“a” => 111, “b” => 222}

H2 = {“b” => 333, “c” => 444}

H1.merge(h2)

If conflicting keys, h2 wins out, so “b” => 333; if reversed h2.merge(h1), “b” =>222 wins out

H1.merge(h2) {|key,old\_value,new\_value| new}

New is Whatever we want to be returned or the value we want to win out if a conflict occurs

You can make permanent with !

Making code easier:

H1.merge(hw) do |key,old,new|

If old < new

Old

Else

New

End

End

Can turn in to this 1 line of code:

H1.merge(h2) {|k,o,n| o<n? o:n}

1. Collect/map – can use with arrays, hashes, or ranges

Array = [1,2,3,4,5]

Array.collect{|i} i \* 40} => 40,80,120,160,200

Will do this to everything in the array even if a string:

Fruits = [‘apple’,’banana’,’orange’]

Fruits.map {|fruit| fruit.capitalize if fruit == ‘banana’}

Remember: the number of the items that go in to collect will equal the number of items that come out

Hash = {“a” => 111, “b” => 222}

Hash.map {|k,v| k} => [“a”, “b”] or hash.map{|k,v| “#{k}: #{v\*20}”} =>

[“a:2220”,”b:4440”]

Remember: you want to return a useful value or you’ll get nil (puts of course gives nil)

1. Sort = comparison

Spaceship operator: <=>

Value1 ⬄ value 2

-1 less than or moves “left”

0 equal to or stays the same

1 more than or moves “right”

Example:

Array = [3,1,2,4]

Array.sort {|v1,v2| va ⬄ v2} – you cn do array.sort, but if you have

complexity within your hash, then do this

can do this with strings as well  
 fruits.sort\_by{|fruit| fruit.length} – so if you had fruits defined it would sort by the

length of word

! makes permanent

hash.to\_a sorting a hash into an array

hash = {“c” = 222, “a” = 555} hash.sort {|item1, item2| item1[0] ⬄ item2[0]} => [“a”,555],[“c”,222]] – hashes get converted into arrays so you need to talk to the key or the value when sorting and make sure you are referring to the result you actually want

1. Inject – accumulator; accumulating our values; memo

(1..10).inject { |memo,n| memo + 1}

memo = 1 memo = memo +2 memo = memo +3 memo = memo + 4 memo =

memo +5

example: array = [\*1..10]

sum = array.inject{|memo,n| memo+n} => 55

sum = array.inject(100) {|memo,n| memo+ n} =>155

make sure that this is really what you want to store in memo; if not, a valid returned value it will not continue the code block and returns nil

You can also use with strings as in:

Fruits = [‘apple’, ‘banana’,’pear’]

Longest\_word = fruits.inject do |memo,fruit|

If memo.length > fruit.length

Memo

Else

Fruit

End

End => ‘banana’

Define Ruby methods:

Stand-alone functions or in ruby they are called methods

Stand-alone method – just call the method after def – which is very nice if you have complication in your code

Require the file in irb and run your methods if you want to interact with it or run it from your terminal prompt if not

Variable scope in methods:

Local variable – local to the structure they are in; once it leaves that structure, it’s discarded and can’t be used outside the method

Make sure your method name and your local variable names are not confusing and if they are the same, be clear which is which

Local or block variables are not available outside the method

Arguments or args:

Come in comma separated list of values passed into methods; args to work needa local variable to capture

When calling args at the bottom of code, don’t have to use () at the time you call your method, you can add in your args then which gives power and flexibility to your code

Argument Default Values:

Set a default value as in: so if nothing passed in to your method, it will not error but default

Def welcome (name=”World”)

Puts “Hello #{name}”

End

If you have required args, always put first before unrequired args.

Argument list – think what should be your default

Returned Value:

The last value returned from the whole function or method

Return – exits from the method as it returns the final value

It works well with conditionals like if or can be used after an end as well

Example: def add\_and\_subtract(n1=0, n2)

Add = n1 + n2

Sub = na-n2

Return [add,sub] - brackets are optional and how your return

End multiple objects as return brings one

Result = add\_and\_subtract(2,2)

Puts result [0]

Puts result [1]

Add,sub = add\_and\_subtract(8,3)

Array even without the []

Recap methods:

1. Methods have a default return value: the last operations’ return value

2. Return will both return a value and exit the method

3. Returning a value and using puts outside a method can provide power and

flexibility

4. Return is useful w/conditional statements

5. Methods can return only one object; use arrays to return more

Operators:

Also methods like + , - , etc.

They have syntactic sugar vs. syntactic vinegar

Syntactic sugar Syntactic vinegar (what ruby does)

8 + 2 8.+(2)

8-2 8.-(2)

array<<4 array.<<(4)

array[2] = ‘x’ array.[] = (2, ‘x’)

a variable is not an object