Things to get clarification on

BUBBLE SORT \*

Things to write in notebook

A string literal with single quotes DOES NOT support string interpolation 🡪

a = 1

‘ This is the number #{a} ’ wont work but “ This is the number #{a} “ will work

To search in a string we use the method .include? – remember the ? at the end returns a boolean

Ex: “ I cant believe it Rayed You suck at this “.include? “Rayed” will return true

We can also check if a string starts with something 🡪 “Ruby is beautiful”.start\_with? “Ruby” returns true

\*\*Same thing with -- .end\_with?

.upcase -- .downcase -- .swapcase – swaps all cases so🡪 ‘I am RaYeD’ becomes 🡪 ‘i AM rAyEd’

The .split method splits a string into an array of the separate words 🡪 “I am Rayed”.split becomes 🡪 [“I”, “am”, “Rayed”]

To replace words in a string we use .sub which only replaces the first occurrence. If we want to replace all occurences we use .gsub. 🡪 “I am great because I am awesome”.sub(“I”, “We”) 🡪 “We am great beecaise I am awesome”. -- “I am great because I am awesome”.gsub(“I”, “We”) 🡪 “We am great because We am awesome”

'RubyMonk'.gsub(/[aeiou]/,'1') --- can also use regex for this, this replaces all vowels with the #1

'RubyMonk Is Pretty Brilliant'.gsub(/[A-Z]/, '0')

a loop can be halted by using the break command

loop do

monk.meditate

break if monk.nirvana?

end

In Ruby, the method map is used to transform the contents of an array according to a specified set of rules defined inside the code block

So for example: [1,2,3,4,5].map { |i| i+ 1} will return [2,3,4,5,6]

To filter elements in an array we use .select 🡪 [1,2,3,4,5,6].select { |num| num % 2 ==0 } 🡪 [2,4,6]

names = ['rock', 'paper', 'scissors', 'lizard', 'spock']

names.select { |words| words.length > 5 } 🡪 [‘scissors’, ‘lizard’]

[1,2,3,4,5,6,7].delete\_if{|i| i < 4 } 🡪 [4,5,6,7]

The Array#eachmethod accepts a block to which each element of the array is passed in turn

def array\_copy(source)

destination = []

source.each do |i|

destination << i if i < 4

end

return destination

end

Hashes store key-value pairs – restaurant\_menu = { “Ramen” => 3, “Dal” => 4, “Tea” => 2} – if you want to fetch a value from the hash we can do restaurant\_menu[“Ramen”] 🡪 this will return 3

You can also create an empty hash by restaurant\_menu = {} or restaurant\_menu = Hash.new() then add values later like this – restaurant\_menu[“Ramen”] = 3 .. here in the square bracket is the key and it is equal to the value.

Now to manipulate the key/value pairs – if lets say the restaurant increases their price for everything by 10% we can do – restaurant\_menu.each do |item, price| restaurant\_menu[item] = (price \* 1.1) end

If you want to see all of the menu items we can do restaurant\_menu.keys – coz remember its restaurant.menu.each do|key,value|

An object built by a certain class is called 'an instance of that class.' Typically, calling the new method on a class results in an instance being created.

So creating a new hash is creating a new instance of the hash class

Creating a class it must have 2 features to justify its existence, a state (defines the attributes of its instances) – example a class Rectangle, the states could simply be its length and width. And it needs a behaviour (must do something meaningful) – happens by developers adding methods to the class that interact with the state to give us meaningful results.

A method, then, is simply programming jargon for *something one object can do for another*.

\*\*\*\* Be cautious when using return - calling return also exits the method at that point. No code in the method after the return statement is executed. \*\*\*\*\* Example:

def demonstrate\_early\_return

return

puts “You will never see this, because we never get here”

end

puts demonstrate\_early\_return.class

When creating an object the default value on the parameters is always 0. So

def say\_hello(name); “Hello #{name}”; end would return nil if no name is passed in.

Multi line comments in ruby start with =begin and end with =end. Instead of having to # every line.

WHILE LOOPS

Sometimes you want to repeat an action in Ruby while a certain condition is true, but you don't know how many times you'll have to repeat that action. A good example would be prompting a user for a certain type of input: if they insist on giving you the wrong thing, you may have to re-ask them several times before you get the kind of input you're looking for.

To accomplish this, we use something called a whileloop. It checks to see if a certain condition is true, andwhile it is, the loop keeps running. As soon as the condition stops being true, the loop stops!

Complement to the while loop is the until loop. Its like backwards while – it keeps going UNTIL a certain thing gets to comething.

FOR LOOPS

Sometimes you *do* know how many times you'll be looping, however, and when that's the case, you'll want to use a forloop.

Example: for i in (1..20); puts i; i = i\*1.5; end

Instead of the loops it is also possible to repeat an action using an iterator. An iterator is just a ruby method that repeatedly invokes a block of code. The code block just contains the instructions to be repeated.

Example: loop { print “hello world” } – these curly braces are generally interchangeable with the keywords do and end. We can make a better loop knowing this for example

i = 0

loop do

i += 1

print “#{i}”

break if i > 5

end

The break keyword breaks a loop as soon as its condition is met

THE ‘NEXT’ KEYWORD can be used to skip over certain steps in the loop. For example

For i in 1..5

next if i % 2 == 0

print i

end

A more useful iterator is the .each method, which can apply an expression to each element of an object, one at a time. The syntax looks like this: ---- object.each { |item| # do something } – can also use the do and end instead of the {} 🡪 object.each do |item| #do something. The variable name b/w the | | can be anything, its just a placeholder for each element of the object youre using .each on.

.TIMES

The .times method is like a super compact for loop: it can perform a task on each item in an object a specified number of times. example if you wanna write bacon 10 times 🡪 10.times do print “Bacon” end

Making hashes --- family = Hash.new() --- adding values to hashes --- var[“Fahad”] = “brother”

accessing key value pairs outside of creating the hash. ---- puts var[“Fahad”] 🡪 returns “brother”

To print out the entire hash ---- family.each { |k,v| puts #{k}: #{v} }

Combined Comparison Operator .. book\_1 = “a wrinkle in time”; book\_2 = “a brief history of time”; book\_1 <=> book\_2. Itll return 0 if the 2 are equal, 1 if first operand is greater and -1 if first operand is less

it’s hard to sort an array of strings in descending order. must use the combined comparison operator. example:

array = [“Aardvark”, “Boris, “Zelda”, “War”, “Jar”] ---- arr.sort { |word\_1, word\_2| word\_2 <=> word\_1 }

Tying it all together