Irb used to quickly check what happens instantly while pry is mostly for debugging.

Hmwk

**Q1 - In byebug, what is the difference between the next and step commands?**

**step:** Step execution into the next line or method. Takes an optional numeric argument to step multiple times.

**next:** Step over to the next line within the same frame. Also takes an optional numeric argument to step multiple lines

Class answer - Step goes to the next line (while running that function), but if it encounters a method it goes there, where as the next just goes to the next line whether it finds a method or not.

**Q2 - What is the difference between the select and detect methods in ruby (you can call them on arrays and other "Enumerables")?**

detect / find stops iterating after the condition returns true for the first time. detect is a method of Enumerable.

on the other hand, select / find\_all will iterate until the end of the input list is reached and returns all of the items in an array for which the block returned true.

Class - Both of them will look for something according to conditions but select will find multiple things whereas detect will find the first thing only. Example 🡪 [1,2,3,4].select{ do |x| x> 2 }

Will find all numbers bigger than 2 🡪 3 & 4 where as detect will only spit out 3.

Q3 -What is the block that is passed into select used for?

Q4 - What's an Enumerable?

The Enumerable mixin provides collection classes with several traversal and searching methods, and with the ability to sort. The class must provide a method each, which yields successive members of the collection. IfEnumerable#max, #min, or #sort is used, the objects in the collection must also implement a meaningful <=> operator, as these methods rely on an ordering between members of the collection.

Class answer -

Q5 - Name two types of objects in Ruby that you think are enumerable. *Bonus:* name a third type.

Each, select, detect, map

Lecture

**Topics**

* Take up homework as a class (cherry pick questions)
* Debugging with pry
* Enumerables and their commonly used methods
* Using require to share code between files
* Regex groups
* Blocks
  + yielding
  + Implementing our own select method
* Lambdas & Procs and how they compare to blocks (with enough time)

Enumerables and their commonly used methods –

* A = [1,2,3,4,5,6,7,8,9]
* X = A.each do |i|
  + P i
* End
* P x
* Each takes a block –(a block is when there are do |x| and end or the curly braces) – itll run whatever code is b.w the do and end and run it for every index in the array.
* \*\*When using def some\_method(number) …. End – the (number) is the first parameter well be passing in. and it is the same thing in the pipeline in do and ends.
* Very similar method called.map
* Y = A.map do |i| p i … end .. p y --– this will do the same thing as each. So right now x and y will be the same the each and map will do something and return the result to x,y.
* \*\*\* puts --- stands for put string – whatever your giving it turns it into a string.. where as p – gives the actual true value.. as in if the answer is an integer it will give an integer. If you p an array it will print the array WITH the square brackets. 🡪 so if you puts an empty array 🡪 it will return nil where as p [] will return []
* so in an example –
* A = [1,2,3,4,5,6,7,8,9]
* X = A.each do |i|
  + P I \*2
* End
* P x
* y = A.each do |i|
  + P I \*2
* End
* P y
* each returns all the numbers in a vertical list then returns the original array – [1,2,3,4,5,6,7,8,9] where as map returns the same numbers in a vertical list then returns the new array [2,4,6,8,10,12,14,16,18].. **so maps creates a new array – so its useful to save it somewhere.**
* map will always give you an array of the same length. Your just transforming..
* Where as we use each just to see the side effects.
* A = [1,2,3,4,5,6,7,8,9]
* Sum = 0
* X = A.each do |i|

Sum += I \* 2

* End
* Will add the sum of each index of the array 🡪 this is an example of where each can be used because it doesn’t change the original array but only uses it.
* .
* Now lets do the select (for same array)
* Z = a.select do |i| I % 2 == 0
* End
* P z
* This returns [2,4,6,8]. We use this to filter your arrays. It keeps the things if your condition is true.
* The opposite of this is .reject Where it REMOVES the things if your condition is true. So in the reject case the return would be [1,3,5,7,9]
* Now lets do detect
* Detect will always return one of the values in the array (google)./
* Writing our own method
* Def sum\_and\_multiply(arr, m)
  + - Sum = 0 -- \*\*
    - Arr.each do |x|
    - S += x \*m

End

Return s

* End
* P sum\_and\_multiply([1,2,3,4], 2) do |x|
* Puts “Asashash”
* End
* (first part) So this sum\_and\_multiply is going to take the given array and m and notice that sum starts at 0 then it creates an s variable so it has some place to put the data – then it multiplies x (each index in array) and m then add them all together and put the result in s then it asks to return s
* Now lets say the first and second part are together and you want to put the result of the second part INTO the first part meaning instead of doing x \* m in s you want to multiply x \* the result of the second part. So that’s why ruby has a built in method called yield. This would be
* Def sum\_and\_multiply(arr, m)
  + - Sum = 0 -- \*\*
    - Arr.each do |x|
    - S += x \* yield()

End

Return s

* End
* Result = sum\_and\_multiply([1,2,3,4]) do
  + 2
* End
* so lets say the (even though its kind of silly) the result (second part) is always 2, and we want to incorporate it in the first part we are using yield. The yield gives the result of the block in the second part.
* A better example for this would be
* Def sum\_and\_multiply(arr)
  + - Sum = 0
    - Arr.each do |x|
    - S += x \* yield(x)

End

Return s

* End
* Result = sum\_and\_multiply([1,2,3,4]) do |i|
  + If I % 2 == 0; 10; else 1 end
* End
* P result
* So some\_and\_multiply(arr) takes an array and when you scan the function you see yield, so the we create a variable called s and we take x and multiply It by the result of passing in x to the second part (yield). Then it will add it to s .
* The yield is telling you to take the result block and run it with x and multiply it with x then add all together for each index in the array
* Def select(arr
* Result = []
* Arr.each do |i|
* If yield(i)
* Result << i
* End
* End
* Return result
* End

#using rubys select

* [12345].select do |x|
* x % 2 == 0
* end

#using our own select

* select([1,2,3,4,5,6]) do |x|
* x% 2 == 0
* end
* so from the top – we added an if statement we only want to add things into the yield if the result comes to true –

Requires

* require “pry” 🡪 takes a different gem and implements it in your current file.
* **require is only used for gems**
* require\_relative is used for files – so taking a previous ruby file and putting it in your current file
* you also shouldn’t have to specify .rb

REGEX CAPTURE GROUPS

* Say you had a string and you want to extract the postal code and analyze both parts of it
* “My postal code is A1A 2B2.”
* Lets try to write a regex to find the postal code
* There are 2 different ways for reg ex
  + 1. Entire string matches regex
  + 2. String contains regex
* a regex for #1. 🡪 /.\*[A-Z]\d[A-Z]\*space\*?\d[A-Z]\d.\*/
* the .\* at the beginning and the end because the postal code (A1A 2B2) could be anywhere.
* [A-Z] for all letters, \d for digits, [a-z] for letters maybe a space or not then \d for #, [a-z] for letter \d for #.
* A regex for #2. 🡪 /[A-Z]\d[A-Z]\*space\*?\d[A-Z]\d/ 🡪 just removing the .\*
* .
* when asked to blacnk out some numbers of a social insurance number 🡪 we have to use the method -- .gsub
* examplre
* “MY POSTAL CODE IS A1A 2B2.” .gsub(/[A-Z]\d[A-Z]\*space\*?\d[A-Z]\d/, “XXXXXX”)
* this will return “my postal code is XXXXXX”
* if we do “MY POSTAL CODE IS A1A 2B2.” .gsub(/[A-Z]\d[A-Z]\*space\*?\d[A-Z]\d/, ‘\1’)
* this iwill return “my postal code is “A1A”
* if we do “MY POSTAL CODE IS A1A 2B2.” .gsub(/[A-Z]\d[A-Z]\*space\*?\d[A-Z]\d/, ‘\1\2’)
* this iwill return “my postal code is “A1A 2b2”
* if we do “MY POSTAL CODE IS A1A 2B2.” .gsub(/[A-Z]\d[A-Z]\*space\*?\d[A-Z]\d/, ‘\1XXX’)
* this will return “my postal code is “A1AXXX”
* if we do “MY POSTAL CODE IS A1A 2B2.” .gsub(/[A-Z]\d[A-Z]\*space\*?\d[A-Z]\d/, ‘\2\2\1’)
* THIS WILL RETURN “MY POSTAL CODE IS 2B22B2A1A.”

Restul = /.\*(a)/.match(“hello raf”)

Restul[0]

🡪 “hello ra”

Restul[1]

🡪”a”

Breakout

Lets say

People = []

People << {name: “Faisal”}

People << {name: “sara”}

People 🡪 returns [{:name => “Faisal}, {:name => “Sara”}] 🡪 this is an array of hashes.

Hobbies = [“soccer”, “squash”] – creating another array

People[0][:name] 🡪 returns faisal

People[0][:name] = “Mike” 🡪 returns “Mike” – changing values.

Say you want to insert your hobbies array into your hash ---

People[0][:hobbies] = hobbies

Replacing the key in the first index of the people hash to hobbies.

Now if you do people 🡪 it returns [{:name=> “Mike”, :hobbies =>[“Soccer”, “Squash]}, {:name=>”Sara”}]

So now if we want to access “Soccer”. We have to look in the first hash for people and the hobbies key and the first value in that ---

People[0][:hobbies][0] 🡪 returns “Soccer”

\*\*\*\*\* LOOK INTO .SELECT & .FILTER \*\*\*\*\*