Ruby Training

1)include ?-> checks whether the string is present

2)for num in 1..10 (double dot includes the value and triple dot excludes the value)

3)gsub! -> global substitution for string and exclamation stands for on place string replacement

4)continue is achieved by “next” keyword

Next if I % 2 !=0 ---- it will skip all odd numbers

5) loop do

….

…

End

* Always use end after any loop

6).times is also used to iterate loops

For eg:10.times { ……….}

A Small example of all

puts "Text to search through: "

text = gets.chomp

puts "Word to redact: "

redact = gets.chomp

words = text.split(" ")

words.each do |word|

if word != redact

print word + " "

else

print "REDACTED "

end

end

**Hash**

my\_hash = { "name" => "Eric",

"age" => 26,

"hungry?" => true

}

puts my\_hash["name"]

puts my\_hash["age"]

puts my\_hash["hungry?"]

🡺pets = Hash.new

pets["ncj"] = "jbdshfdh"

🡺s = [["ham", "swiss"], ["turkey", "cheddar"], ["roast beef", "gruyere"]]

s.each { |sand|

sand.each { |ing|

puts ing

}

}

🡺secret\_identities = {

"The Batman" => "Bruce Wayne",

"Superman" => "Clark Kent",

"Wonder Woman" => "Diana Prince",

"Freakazoid" => "Dexter Douglas"

}

secret\_identities.each { |item , key|

puts "#{item}: #{key}"

}

🡺frequencies = Hash.new(0) (Use default value when key is not found . It will give 0)

🡺puts "Text daal"

text = gets.chomp

words = text.split(" ")

frequencies = Hash.new(0)

puts frequencies["shjdbshd"]

words.each { |word|

frequencies[word]+=1

}

🡺puts "Text daal"

text = gets.chomp

words = text.split(" ")

frequencies = Hash.new(0)

puts frequencies["shjdbshd"]

words.each { |word|

frequencies[word]+=1

}

frequencies = frequencies.sort\_by { |key ,count|

count

}.reverse!

# frequencies.reverse!

**Method**

**🡺**def array\_of\_10

puts (1..10).to\_a

end

array\_of\_10

def cubertino(n)

puts n \*\* 3

end

cubertino(8)

🡺def what\_up(greeting, \*friends)

friends.each { |friend| puts "#{greeting}, #{friend}!" }

end

what\_up("What up", "Ian", "Zoe", "Zenas", "Eleanor")

🡺def greeter(name)

return name+" greeting"

end

greeter("JIgar")

def by\_three?(number)

return number % 3 == 0

end

by\_three?(9)

🡺books = ["Charlie and the Chocolate Factory", "War and Peace", "Utopia", "A Brief History of Time", "A Wrinkle in Time"]

# To sort our books in ascending order, in-place

books.sort! { |firstBook, secondBook| firstBook <=> secondBook }

# Sort your books in descending order, in-place below

books.sort! { |firstBook, secondBook| secondBook <=> firstBook }

🡺fruits = ["orange", "apple", "banana", "pear", "grapes"]

fruits.sort! {

|one,two| two <=> one

}

🡺def alphabetize(arr , rev=false)

return arr.sort!

end

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

puts alphabetize(numbers)

🡺def alphabetize(arr , rev=false)

if rev

return arr.reverse!

else

return arr.sort!

end

end

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

puts alphabetize(numbers)

🡺puts "string".object\_id

puts "string".object\_id

puts :symbol.object\_id

puts :symbol.object\_id

output:

12320400

12319980

802268

802268

**Symbols always start with a colon (:). They must be valid Ruby variable names, so the first character after the colon has to be a letter or underscore (\_); after that, any combination of letters, numbers, and underscores is allowed.**

Symbols make good hash keys for a few reasons:

1. They’re immutable, meaning they can’t be changed once they’re created;
2. Only one copy of any symbol exists at a given time, so they save memory;
3. Symbol-as-keys are faster than strings-as-keys because of the above two reasons.

**Symbols**

🡺strings = ["HTML", "CSS", "JavaScript", "Python", "Ruby"]

# Add your code below!

symbols = []

strings.each { |item|

symbols.push(item.to\_sym)

}

puts symbols

🡺require 'benchmark'

string\_AZ = Hash[("a".."z").to\_a.zip((1..26).to\_a)]

symbol\_AZ = Hash[(:a..:z).to\_a.zip((1..26).to\_a)]

string\_time = Benchmark.realtime do

100\_000.times { string\_AZ["r"] }

end

symbol\_time = Benchmark.realtime do

100\_000.times { symbol\_AZ[:r] }

end

puts "String time: #{string\_time} seconds."

puts "Symbol time: #{symbol\_time} seconds."

🡺movie\_ratings = {

memento: 3,

primer: 3.5,

the\_matrix: 5,

truman\_show: 4,

red\_dawn: 1.5,

skyfall: 4,

alex\_cross: 2,

uhf: 1,

lion\_king: 3.5

}

# Add your code below!

good\_movies = movie\_ratings.select {

|k , v| v > 3

}

🡺my\_hash = { one: 1, two: 2, three: 3 }

my\_hash.each\_key { |k| print k, " " }

# ==> one two three

my\_hash.each\_value { |v| print v, " " }

# ==> 1 2 3

🡺movies = {

php: "djnvjx",

goal: "dnfbxjk"

}

puts "enter the movie"

choice = gets.chomp

case choice

when "add"

puts "Added!"

when "update"

puts "Updated!"

when "display"

puts "Deleted!"

when "delete"

puts "Deleted!"

else

puts "Error!"

end

🡺movies = {

php: "djnvjx",

goal: "dnfbxjk"

}

puts "enter the movie"

choice = gets.chomp

case choice

when "add"

puts "title daal"

title = gets.chomp

if movies[title.to\_sym] == nil

puts "rating ?"

rating = gets.chomp

movies[title.to\_sym] = rating.to\_i

puts "A movie has been added"

else

puts "Already there"

end

when "update"

puts "title daal"

title = gets.chomp

if movies[title.to\_sym] != nil

puts "rating ?"

rating = gets.chomp

movies[title.to\_sym] = rating.to\_i

puts "A movie has been added"

else

puts "Already there"

end

when "display"

movies.each { |key , value|

puts "#{key}: #{value}"

}

when "delete"

puts "title daal"

title = gets.chomp

if movies[title.to\_sym] != nil

movies.delete(title)

puts "A movie has been deleted"

else

puts "FO"

end

else

puts "Error!"

end

🡺# Write your code on line 2!

favorite\_language ||= "Python"

puts favorite\_language

We’ve seen that we can use the = operator to assign a value to a variable. But what if we only want to assign a variable if it hasn’t already been assigned?

For this, we can use the conditional assignment operator: ||=. It’s made up of the or (||) logical operator and the normal = assignment operator.

🡺.respond\_to? takes a symbol and returns true if an object can receive that method and falseotherwise. For example,

[1, 2, 3].respond\_to?(:push)

would return true, since you can call .push on an array object. However,

[1, 2, 3].respond\_to?(:to\_sym)

would return false, since you can’t turn an array into a symbol.

🡺alphabet = ["a", "b", "c"]

alphabet << ("d") # Update me! (push the item int*o array)*

caption = "A giraffe surrounded by "

caption << "weezards!" # Me, too!

**Yeilds**

🡺def yield\_name(name)

puts "In the method! Let's yield."

yield("Kim")

puts "In between the yields!"

yield(name)

puts "Block complete! Back in the method."

end

yield\_name("Eric") { |n| puts "My name is #{n}." }

yield\_name("Joe") {|n| puts "My name is #{n}."}

# Now call the method with your name!

Explanation:

You can also pass parameters to yield! Check out the example in the editor.

1. The yield\_name method is defined with one parameter, name.
2. On line 9, we call the yield\_name method and supply the argument "Eric" for the nameparameter. Since yield\_name has a yieldstatement, we will also need to supply a block.
3. Inside the method, on line 2, we first puts an introductory statement.
4. Then we yield to the block and pass in "Kim".
5. In the block, n is now equal to "Kim" and we puts out “My name is Kim.”
6. Back in the method, we puts out that we are in between the yields.
7. Then we yield to the block again. This time, we pass in "Eric" which we stored in the nameparameter.
8. In the block, n is now equal to "Eric" and we puts out “My name is Eric.”
9. Finally, we puts out a closing statement.

🡺def double(no)

yield(no)

end

double(8) {|n| puts n\*2}

🡺Procs are easy to define! You just call Proc.new and pass in the block you want to save. Here’s how we’d create a proc called cube that cubes a number (raises it to the third power):

cube = Proc.new { |x| x \*\* 3 }

We can then pass the proc to a method that would otherwise take a block, and we don’t have to rewrite the block over and over!

[1, 2, 3].collect!(&cube)

# ==> [1, 8, 27]

[4, 5, 6].map!(&cube)

# ==> [64, 125, 216]

(The .collect! and .map! methods do the exact same thing.)

The & is used to convert the cube proc into a block (since .collect! and .map! normally take a block). We’ll do this any time we pass a proc to a method that expects a block.

🡺floats = [1.2, 3.45, 0.91, 7.727, 11.42, 482.911]

# Write your code below this line!

round\_down = Proc.new { |n|

n.floor

}

# Write your code above this line!

ints = floats.collect(&round\_down)

print ints

🡺# Here at the amusement park, you have to be four feet tall

# or taller to ride the roller coaster. Let's use .select on

# each group to get only the ones four feet tall or taller.

group\_1 = [4.1, 5.5, 3.2, 3.3, 6.1, 3.9, 4.7]

group\_2 = [7.0, 3.8, 6.2, 6.1, 4.4, 4.9, 3.0]

group\_3 = [5.5, 5.1, 3.9, 4.3, 4.9, 3.2, 3.2]

# Complete this as a new Proc

over\_4\_feet = Proc.new { |height|

height >= 4

}

# Change these three so that they use your new over\_4\_feet Proc

can\_ride\_1 = group\_1.select(&over\_4\_feet)

can\_ride\_2 = group\_2.select(&over\_4\_feet)

can\_ride\_3 = group\_3.select(&over\_4\_feet)

puts can\_ride\_1

puts can\_ride\_2

puts can\_ride\_3

🡺map and collect returns true or false for each array element(if and only if comparisons operators are used)

🡺def greeter

yield

end

phrase = Proc.new {

puts "Hello there!"

}

greeter(&phrase)

🡺hi = Proc.new {

puts "Hello!"

}

hi.call

🡺numbers\_array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

strings\_array = numbers\_array.map(&:to\_s)

🡺strings = ["leonardo", "donatello", "raphael", "michaelangelo"]

# Write your code below this line!

symbolize = lambda {|item| item.to\_sym }

# Write your code above this line!

symbols = strings.collect(&symbolize)

print symbols

🡺def batman\_ironman\_proc

victor = Proc.new { return "Batman will win!" }

victor.call

"Iron Man will win!"

end

puts batman\_ironman\_proc

def batman\_ironman\_lambda

victor = lambda { return "Batman will win!" }

victor.call

"Iron Man will win!"

end

puts batman\_ironman\_lambda

Explanation:

First, a lambda checks the number of arguments passed to it, while a proc does not. This means that a lambda will throw an error if you pass it the wrong number of arguments, whereas a proc will ignore unexpected arguments and assign nil to any that are missing.

Second, when a lambda returns, it passes control back to the calling method; when a proc returns, it does so immediately, without going back to the calling method.

Click Run to see the return values. See how the proc says Batman will win? This is because it returns immediately, without going back to the batman\_ironman\_proc method.

Our lambda, however, goes back into the method after being called, so the method returns the last code it evaluates: "Iron Man will win!"

🡺my\_array = ["raindrops", :kettles, "whiskers", :mittens, :packages]

# Add your code below!

symbol\_filter = lambda {|item| item.is\_a? Symbol}

symbols = my\_array.select(&symbol\_filter)

puts symbols

🡺All this talk of blocks, procs, and lambdas might have your head spinning. Let’s take a minute to clarify exactly what each one is:

1. A block is just a bit of code between do..end or {}. It’s not an object on its own, but it can be passed to methods like .each or .select.
2. A proc is a saved block we can use over and over.
3. A lambda is just like a proc, only it cares about the number of arguments it gets and it returns to its calling method rather than returning immediately.

🡺odds\_n\_ends = [:weezard, 42, "Trady Blix", 3, true, 19, 12.345]

ints = odds\_n\_ends.select {|item| item.is\_a? Integer}

puts ints

🡺**What's in a @name?**

All right! Just one more step before we can create a person from our Person class: we have to make sure each person has a @name.

In Ruby, we use @ before a variable to signify that it’s an *instance variable*. This means that the variable is attached to the *instance* of the class.

class Car

def initialize(make, model)

@make = make

@model = model

end

end

kitt = Car.new("Pontiac", "Trans Am")

The code in the example above creates an instance, kitt, of the class Car. kitt has his own @make(“Pontiac”) and @model (“Trans Am”). Those variables belong to the kitt instance, which is why they’re called instance variables.

🡺Explanation

 See how some variables start with $, @, or @@? This helps mark them as global, instance, and class variables (respectively).

🡺class Computer

$manufacturer = "Mango Computer, Inc."

@@files = {hello: "Hello, world!"}

def initialize(username, password)

@username = username

@password = password

end

def current\_user

@username

end

def self.display\_files

@@files

end

end

# Make a new Computer instance:

hal = Computer.new("Dave", 12345)

puts "Current user: #{hal.current\_user}"

# @username belongs to the hal instance.

puts "Manufacturer: #{$manufacturer}"

# $manufacturer is global! We can get it directly.

puts "Files: #{Computer.display\_files}"

# @@files belongs to the Computer class.

🡺**inheritance**

🡺class ApplicationError

def display\_error

puts "Error! Error!"

end

end

class SuperBadError < ApplicationError

end

err = SuperBadError.new

err.display\_error

🡺class Creature

def initialize(name)

@name = name

end

def fight

return "Punch to the chops!"

end

end

# Add your code below!

class Dragon < Creature

puts "Instead of breathing fire..."

def fight

super //use the superclass fight method

end

end

🡺class Message

@@messages\_sent = 0

def initialize(from,to)

@from = from

@to = to

@@messages\_sent+=1

end

end

class Email < Message

def initialize(from , to)

super

end

end

my\_message = Message.new("jsdhfgh" , "shdfy")

🡺class Computer

@@users = {}

def initialize(username,password)

@username = username

@password = password

@files = {}

@@users[username] = password

end

def create(filename)

time = Time.now

@files[filename] = time

puts "#{filename} is created by #{@username} at #{time}"

end

def Computer.get\_users

@@users

end

end

my\_computer = Computer.new("hsdh" , "hdfh")

🡺**Private and public**

🡺class Person

def initialize(name, age)

@name = name

@age = age

end

public # This method can be called from outside the class.

def about\_me

puts "I'm #{@name} and I'm #{@age} years old!"

end

private # This method can't!

def bank\_account\_number

@account\_number = 12345

puts "My bank account number is #{@account\_number}."

end

end

eric = Person.new("Eric", 26)

eric.about\_me

eric.bank\_account\_number

🡺class Person

attr\_reader :name

attr\_writer :name

def initialize(name)

@name = name

end

end

Ruby does something like this for us automatically:

def name

@name

end

def name=(value)

@name = value

end

Like magic, we can read and write variables as we please! We just pass our instance variables (as symbols) to attr\_reader or attr\_writer.

(That name= might look funny, but you’re allowed to put an = sign in a method name. That’s just a Ruby convention saying, “hey, this method sets a value!”)

🡺class Person

attr\_reader :name

attr\_writer :job

def initialize(name, job)

@name = name

@job = job

end

end

🡺If we want to both read and write a particular variable, there’s an even shorter shortcut than using attr\_reader and attr\_writer. We can use attr\_accessor to make a variable readable *and*writeable in one fell swoop.

🡺module MyLibrary

FAVE\_BOOK = "dknf" //constants in ruby all uppercase

End

🡺One of the main purposes of modules is to separate methods and constants into named spaces. This is called (conveniently enough) **namespacing**, and it’s how Ruby doesn’t confuse Math::PI and Circle::PI.

See that double colon we just used? That’s called the **scope resolution operator**, which is a fancy way of saying it tells Ruby *where* you’re looking for a specific bit of code. If we say Math::PI, Ruby knows to look inside the Math module to get that PI, not any other PI (such as the one we created in Circle).

Let’s get some practice in with an existing Ruby module: Math.

🡺**Multiple inheritance**

module Action

def jump

@distance = rand(4) + 2

puts "I jumped forward #{@distance} feet!"

end

end

class Rabbit

include Action

attr\_reader :name

def initialize(name)

@name = name

end

end

class Cricket

include Action

attr\_reader :name

def initialize(name)

@name = name

end

end

peter = Rabbit.new("Peter")

jiminy = Cricket.new("Jiminy")

peter.jump

jiminy.jump

**🡺Modules**

🡺module Languages

FAVE = "Ruby" # This is what you typed before, right? :D

end

class Master

include Languages

def initialize; end

def victory

puts FAVE

end

end

total = Master.new

total.victory

🡺**Banking Application**

class Account

attr\_reader :name, :balance

def initialize(name, balance=100)

@name = name

@balance = balance

end

def display\_balance(pin\_number)

puts pin\_number == pin ? "Balance: $#{@balance}." : pin\_error

end

def withdraw(pin\_number, amount)

if pin\_number == pin

@balance -= amount

puts "Withdrew #{amount}. New balance: $#{@balance}."

else

puts pin\_error

end

end

private

def pin

@pin = 1234

end

def pin\_error

"Access denied: incorrect PIN."

end

end

my\_account = Account.new("Eric", 1\_000\_000)

my\_account.withdraw(11, 500\_000)

my\_account.display\_balance(1234)

my\_account.withdraw(1234, 500\_000)

my\_account.display\_balance(1234)

🡺class Account

attr\_reader :name

attr\_reader :balance

def initialize(name,balance=100)

@name = name

@balance = balance

end

public

def display\_balance(pin\_number)

pin\_number == pin ? "Balance: $#{@balance}." : pin\_error

end

public

def withdraw(pin\_number , amount)

if pin\_number == pin

@balance-=amount

puts "Withdrew #{amount}. New balance: $#{@balance}."

else

puts pin\_error

end

end

def deposit(pin\_number , amount)

if pin\_number == pin

@balance+=amount

puts "Deposit #{amount}. New balance: $#{@balance}."

else

puts pin\_error

end

end

private

def pin

@pin = 1234

end

private

def pin\_error

"Access denied: incorrect PIN."

end

end

checking\_account = Account.new("Jigar" , 1\_00\_000)

checking\_account.deposit(1234 , 5000)