Ruby

Basics :-

- Extention for ruby files is .rb
- puts "_" is used to print something on console.
- # is called the pound character & is used to add comment
- =begin , =end are used for multiline comments.
- Simple mathematical functions can be carried out within puts statements by using #{} in ruby.
  Ex :- puts "Alok has #{25+30/6} Rs in his Pocket."
  Output :- Alok has 30 Rs in his pocket.
- Print"_" can be used instead of "puts" to print without a newline

Variables

Variables in ruby are dynamic, i.e we don't need to mention its type & ruby will know its type automatically.

Ex :-
```
cars = 100
drivers = 30
puts "There are #{cars} cars and #{drivers} drivers."
```
Output There are 100 cars and 30 drivers.

Getting input

- gets.chomp is used to take input from user.
- gets.chomp.to_i is used to get Integer input from user.
- gets.chomp.to_f is used to get float (decimal) input from user.

```
print "give a number"
number = gets.chomp.to_i
puts "you just entered #{number}
```

Methods :-

Methods are collection of statements that perform some specific task & return the result.

Define & call the method :-

Methods are defined with the help of def Keyword followed by method-name and they ends with end Keyword.

Method can be called by simply writing its name after defining it

Ex :-
```
def Jit                    # define method with name Jit
puts "Hello world!"  # pri
end

Jit
```

Output *Hello world!*

Parameters passing :-

parameters passing is similar to other languages simply write the parameters in () brackets seperated by a , comma.

```
def method_name (var1, var2, var3)
# Statement 1
# Statement 2
end
```

To take variable number of arguments we use (* var_name)
It is useful when we dont know number of parameters to be passed while defining a method.

Ex: def method_name (* variable name)
Statement 1
Statement 2
:
end

Return Statement

It is used to return one or more values.
by default a method always return last statement. that was evaluated by the body of the method.

Ex→ def num
a=10
b=39
Sum = a+b
- return Sum
end

puts "The result is : #{num}"

Classes & Objects

Ruby is an ideal object-oriented programming language. The features of an object-oriented programing are :- data encapsulation, polymorphism, inheritance, data abstraction, operator overloading. etc.

Class :-
A class is a blueprint from which objects are created.

Object :-
Object is an instance of class.

To create a class :-
Write class Keyword followed by the name of class. first letter of class name should be in capital.

Syntax :-
```
class Class_name
  ⋮
end
```

A class is terminated by end Keyword all data members stays in between class defination and end Keyword.

Create Objects using "new" method :-

We can create a number of objects from a single class. In Ruby objects are created by new method.

```
Object_name = Class_name.new
```

for a specific method

Specific for a Object can be used by. One or more

Same for all objects of class.

available for diff. classes & methods

Variables in ruby class

- Local variables :-

for a specific method

Local variables are the variables that are defined in a method. They are not available outside the method. Local variable begin with a lower case letters or _. (_var_name)

- Instance variables :-

for specific for a Object can be used for overwritten

Instance variable are available across all methods for any particular instance or object. That means Instance variable changes from object to object. Instance variables begins with @ sign followed by variable name. (@var_name)

- Class variable :-

same for all objects of class.

Class variables are available across different objects. They are preceded by @@ sign followed by var_name. (@@ var_name)

- Global variables :-

available for diff. classes & methods

Class variables are not available across classes. If we want to have a single variable, to be available across classes, we need to have a global variable & global variable are preceded by a dollar sign ($). ($var_name)

Constructors:-

A Constructor is a special method of the class which gets automatically invoked whenever an instance of the class is created.

- Constructors are used to initialize the instance variables.
- In Ruby, the constructors has a different name
- A constructor is defined using the initialize and def keyword
- Constructors can be overloaded in ruby.
- Constructors can't be inherited.

Whenever an object of the class is created using new method, internally it calls the initialize method on the new object. all the arguments passed to new will automatically be passed to method initialize.

```
Ex: Class Customer
      @@ no_of_customers = 0
      def initialize (id, name, addr)
         @cust_id = id
         @cust_name = name
         @cust_addr = addr.
      end
    End.

Cust1 = Customer.new ("1", "John", "Wisdom apt, Indri")
Cust2 = Customer.new ("2", "paul", "New Empire road")
```

Ruby constants.

Constants begin with an uppercase letter. Constants defined within a class or module can be accessed from within that class or module. Those defined outside can be accessed anywhere.

Fixnum & Bignum

Integers from range -2^{30} to 2^{30-1} are Fixnum. Integers outside this range are Bignum.

Arrays:- adding comma separated elements in square braces.

ary = ["name", "age", "string", "last element"].

(ex. → (0..10).to_a = [0, 1, ... 10])

(expression) .to_a => to_a converts results into an array.

Hashes:-

Adding key value pairs in between curly braces. With the sequence => between key and the value.

hsh = colors = {"red" => 0xf00, "green" => 0x0f0}

Ranges:-

Ranges represents ~~values~~ a set of values with a start & end. Ranges can be constructed using s..e & s...e

Ranges with .. run with start to end inclusively. & those with ... excluded the end value.

If statement:-

```
if conditional
    code..
elsif conditional
    code..
else
    code..
end.
```

if Statements Executes code if conditional is true. if the Conditional is not true code specified in else clause is executed.

Conditional is seprated from code by a reserved word then, a newline, or a semicolon.

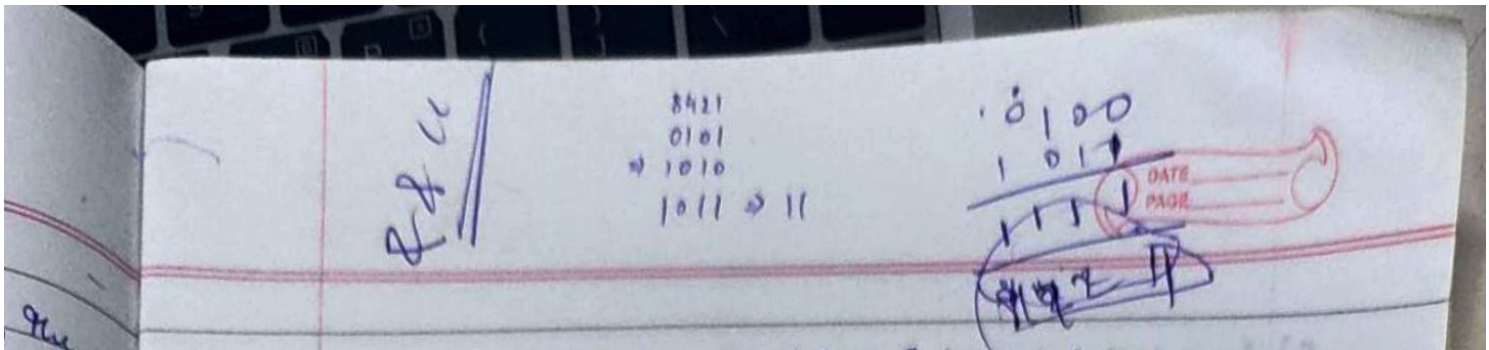
Ex:- x=1
```
if x>2
  puts "x is greater than 2"
elsif x<=2 and x!=0
  puts "x is 1"
else
  puts "I cant guess the number"
end.
```
IF modifier.

⇒ (code) if (condition)

Ruby ternary operator (?:)

↳ It includes a conditional statement & two possible outcomes
↳ it is a way to write a compact if/else expression in one line.

⇒ Ex:-
```
if apple-stock > 1
   eat_apple
else
   buy_apple
end
```
⇒ apple-stock > 1 ? eatapple : buyapple



Ternary operator ⇒ Condition ? true : false

Operators:-

Ruby has built-in modern set of operators.

Types of operators.

1. Unary operator ⇒ for single operand ⇒ !, ~ (inverts all bits), + (plus)
2. Arithmatic operator.
3. bitwise operator.
4. Logical operator.
5. Ternary operator.
6. Assignment operator.
7. Comparison operator.
8. Range operator.

Brackets, Unary, Arithmatic, shift, ~~Logical~~, Relational, Bitwise, logical, Ternary, Assignment, comma

1) Unary operator :-

Operators which works on single operands.

a. ! → Boolean not
b. ~ → Bitwise Complement
c. + → Unary plus

2) Airthmatic operators:-

a. + → Addition of both operands
b. - → Substraction
c. / → division of left side operand with right side
d. * → multiplication
e. ** → right side operand becomes exponent of left side
f. % → divide left side operand with right side operand & return remainder.

3) Logical operator.

Logical operators works on bits operands

a && → AND operator
b. || → OR operator.

4) Ternary operator

First check whether given conditions are true or false then execute condition.

Operator	description
?:	conditional expression

conditional ? true : false.

5) Assignment operator.

Assigns a value to the operands.

a	=	Simple assignment operator
b	+=	Add assignment operator. (adds value of right to left & assigns result to left)
c	-=	Substract assignment operator.
d	*=	Multiply " "
e	/=	divide " "
f	%=	Modulus " "
g	**=	Exponential " "

6) Comparision operator.
Compares two operands.

a. ~~Equal~~ == equal
b. != not equal
c. > Left operand is greater than right operand
d. < right operand is greater than left operand.
e. >= Left operand greater than or equal to right
f. <= right operand is greater than or equal to left.
* g. <=> combined comparision
* h. .eql? checks equality & type of operands.
* i. equal? Checks for object ID.

7) Range operator.
Creates range of successive values.

The (..) creates a range including end term (...) creates range excluding last term.
1..5 → 1, 2, 3, 4, 5
1...5 → 1, 2, 3, 4

⇒ Control Statements.

1) if-else:-
types ↳
- if statement
- if-else statement
- if-else-if (elsif) statement
- ternary (shortened if statement)

Ruby Case Statement :-

The case statement matches one statement (expression) with multiple conditions. if case expression is same as any when expression its code run.

```
Case expression
When expression 1
   code
When expression 2
   code
When expression 3
   code
~~When exp~~
else
   Code
end.
```

For Loop

for loop itirates over a specific range of numbers used if program has fixed number of itirations. loop will execute for each element in expression

```
for variable in (range/expression) do
    puts i
end.
```

```
a = gets.chomp.to_i
for i in 1..a do
   puts i
end
```

```
x = ["Blue", "Red", "Green", "whit"]
for i in x do
   puts i
end
```

Multiple inheritance

In multiple inheritance we want Class A to Inherit class B Class A to Inherit class C both but it is not possible in ruby. for a class to have more than one parent class

Mixin is a method of using modules & include them in class to get multiple Inheritance.

Mixins

```
module A
  def a1
    code...
  end
module B
  def a2
    code..
  end
module B
  def b1
    code...
  end
  def b2
    code...
  end

class sample
  include A
  include B

  def c1
    code..
  end
end
```

here class sample inherits methods from both module A & Module B, Thus class can access all four methods Thus we can say Sample shows multiple inheritance.

Each do ⇒ update score for every user mentioned in array

⇒ def scoring(array)
  array.each do |a|
    a.update_score
  end
end.

method is defined with array as arg
each do IIrate over each element & assign that value to var "a" then for each value of a, update_score method runs.

Unless
update score for users in array except for Admin.

def scoring(array)
  array.each do |a|
    unless a.is_admin?
      a.update_score
    end
  end
end.

// is admin method check is done is admin.

Infinite loops

loop do
  Coder.practice
  break if Coder.oh-one?
end.

OR ⇒ Loop do
  Coder.practice
  if Coder.oh-one
    break
  end
end.

(run this method)
(until Coder.oh,one is) break the loop

Until (while not)

run the method coder.practice until coder.oh_one?

~~until~~ becomes true.

```
Loop do
  coder.practice
  break until coder.oh_one?
end.
```

Case Statement

given an object as input. check if that object belongs to any predefined class.

```
def identify_class(obj)
  case obj
  when Hacker
    puts "it's a hacker"
  when Submission
    puts "it's a Submission"
  when TestCase
    puts "it's a testcase"
  when Course
    puts "it's a course"
  else
    puts "It's an unknown model"
  end
end.
```

Map method allows us to run an operation on each of array's object and return them all in same place.

to increment every array element by 1

```
[1,2,3].map{|x| x+1}
⇒ [2, 3, 4]
```

Exception handling:

any code which can possibly give an error or exception can be added in begin ~~rescue~~ block & the alternate code to run when begin block throws error is added in rescue block.

Ex
```
begin
  num = 10/0
rescue
  puts "division by zero error"
end
```

⇒ Specifying errors.

```
Lucky_nums = [4, 8, 15, 16, 23, 42]

begin
  Lucky_nums["dogs"]        (This give TypeError)
  num = 10/0                (This gives ZeroDivisionError)
rescue ZeroDivisionError
  puts "Division by zero error"
rescue TypeError
  puts "Wrong type"
end
```

Storing an error:-

```
rescue ZeroDivisionError => var_name
  puts var_name          { will print the error}
```

Naming routes

get 'exit', to: 'session#destroy', as: :logout. (to have rails routes as logout_path/url)

Routing constrains

: constraints is used to match routes to a specific format of path using regexp.

get 'photos/:id', to: "photos#show", constraints: { id: /[0-9]{3}/ }

: Constraint doesn't lets anchors ^ & $ to match at the beginning & end because it's default.

Redirection:-

We can redirect any path to another by using redirect helper.

get '/stories', to: redirect('/articles')

blocks are set of Ruby commands

```
do
  code
end
```

for a single line code

```
do { code }
```

Object oriented

- Ruby class and objects.

Object is a physical as well as a logical entity. Whereas class is a logical entity only.

All ruby objects has default root Object. Ruby objects inherit BasicObject (parent class of all classes in ruby) which allows creation of alternate object hierarchies.

Creating Objects.

Objects are created by calling "new" method for a class. It is predefined method in ruby.
Objects are instances of a class.

Syntax → `ObjectName = ClassName.new`

Ruby Class

~~Each ruby class is an instance of class Class~~

Each ruby class is an instance of class Class.
classes in ruby are first class objects

classes are defined as first It starts with Keyword class followed by the class name. Class is finished with end Keyword.

```
class ClassName
  Codes...
end
```

OOPs concepts.

Ruby is true object oriented language, everything in ruby is an object.

~~following are so~~

OOPs is a programming concept that uses object and their interactions to design applications and computer programms.

following are some basic concepts in OOPs.

- Encapsulation
- polymorphism
- Inheritance
- Abstraction.

Inheritance :

Inheritance is one of the solid fundamental characteristics of object oriented programming. Sometimes we need to certain features of a class to be replicated into another class. instead of creating it again we can inherit that atribute from the other class.

The class that is being inherited from called base class & the class that is Inheriting from base class is called the derived class.

Syntax :-

```
class base
  # data & methods.
end
class derived < base
  # data & methods.
end.
```

< symbol is used for inheriting all the data & methods of the base class to the derived class.

```
class vehicle
  def initialize (vehicle_name, vehicle_color)
      @ vehicle_name = vehicle_name
      @ vehicle_color = vehicle_color.
  end
  def description
      puts 'this is a vehicle'
  end
end.
```

~~class Vehicle~~
Class Car < Vehicle
  def description
    puts 'This is a car'
  end
end

if we use super keyword after this before end

Class Bus < Vehicle
  def display_this
    puts "This is a bus"
  end.

Object 1 = Car.new ('nissan', 'red')
Object 2 = bus.new ('volvo', 'white')

Object 1. description
Object 2. description
Object 2. display_this

⇒ ~~Object~~ output ↓

This is a car
This is a vehicle
This is a bus.

In above ex. we have one base class - "vehicle" and two derived classes - "car" and "bus", car & bus inherit methods from vehicle. but we have a common name description in both base class vehicle & derived class car however their functionality is different.

Encapsulation is hiding pieces of functionality and making it unavailable to the rest of the code base. it is a form of data protection, so that the data cannot be manipulated or changed without obvious intention. It is what defines the boundries in your application and allows your code to achieve new levels of complexity. Ruby like many other OO languages accomplishes this task by creating objects. and exposing interfaces (i.e methods) to interact with those objects.

14-06 ⇒ Encapsulation, polymorphism, Include & Extend.

Include & Extend

Include is used to import module code to the class. as a subclass hence we cannot access module code directly for the class but for its objects or instances.

Extend is used to import module code to the class directly as class method & hence we can access modeule code directly for the class.

for instance methods we use include & for class methods we use extend.

DATE
PAGE

Data Abstraction in ruby

The process of hiding details of functionality & showing only significant details is called data abstraction. Through data abstraction the Interface & Implementation are isolated.

data abstraction in modules:-

Modules are defined as a set of methods, classes and constants together.

data abstraction in classes:-

We can use classes to perform data abstraction in ruby. The class allows us to group information and methods using access specifiers (private, protected & public). to determine which information should be visible & which not.

data abstraction using access control:-

There are three types of access control in ruby (public, private & protected)

- Members that are declared public can be accessed from anywhere in the program
- members declared to be private can only be accessed within that class. They are not allowed to be accessed by any part of code outside it.

Resource & resources

Resources creates seperate page for

RESTful representation

Application to manage library.

1. List of all books (GET)
2. GET detailed information about particular book.
3. GET a form to "add a new book"
4. POST the values from the form to create a new book
5. GET the form with existing book's data prepopulated to edit information
6. UPDATE the system with the edited data that was just submitted.
7. DELETE the book.

These 7 actions collectively are known as RESTful representation of book.

resources : books

Verb	URI pattern	Controller # Action	Used for
GET	/books	books # index	List books
POST	/books	books # Create	Create a book
GET	/books/new	books # new	form for new book
~~POST~~	~~/~~		
GET	/books/:id/edit	books # ~~show~~ edit	form for editing book
GET	/books/:id	books # show	Show info about book
PATCH	/books/:id	books # update	Update info about book
PUT	/books/:id	books # update	Update info about book
DELETE	/books/:id	books # destroy	Delete info about book

Lambdas

Can be defined using the method lambda or can be defined as stuby lambda

```
lamb = lambda { |n| puts "I am a lambda" }
lamb = -> (n) { puts "I am a stuby lambda" }
```

Procs and Lambda

- procs don't care about the correct number of arguments. While lambda will raise an exception.
- Return & break behaves differently in procs and lambdas. A block returns from context while lamba return from itself.
- There is no dedicated lambda class. Lambda is just a special Proc object.

Class Method and Instance method

We can call a class method on the class itself but we cannot call a class method directly on objects.

```
class Example
  def self.class_method
    puts "class method"
  end
  def instance_method
    puts "instance method"
  end
end
```

Example.class_method ⇒ ✓
Example.instance_method ⇒ ✗ Undefined

Example.new.instance_method ⇒ ✓
Example.new.class_method ⇒ ✗ undefined.

Active records basics.

Active record is the M in MVC

↳ It gives ability to:
- Represent models & their data
- Represent associations between these models
- Represent inheritance hierarchies through related models.
- Validate models before they get persisted to the database
- perform database operations in an object oriented manner.

→ When we generate a model with name Book we get a Database table books.
Naming conventions should be like:-
~~ModelClass~~ Model/Class → Singular with first letter of each word capitalized (e.g. BookClub)

Database/Tables → plural with underscores seperating words ~~(books)~~ (e.g 'books_clubs')

Naming conventions for the columns in database tables.
- Foreign keys : should be named following the pattern.
  ↳ Singularized_table_name_id . fields that ActiveRecord will look on when we create association between your models.
- primary key - by default Active record use an integer column named id as tables primary key

⇒ Create Active Record Models : to create we can use generator or simply subclass the Application record class with model name

Validation

There are two types of active record objects: those that correspond to a row inside our database & those that do not.
Ex: using a new method the objects we create those does not belong to the database before we call save upon them.

Creating & saving a new record will send an SQL insert operation to the database. Updating an existing record will send an sql Update operation instead. Validations runs before these operations are sent to the database. if any validation fails. these operations won't be executed on db

- Create, Create!, Save, Save!, Update, Update!
  - These methods runs validations
  - bang versions (Create!, Save!, Update!) raise an exception if record is invalid, non bang versions return false, and Create returns the object.

- There are methods which skip validations to create records. Ex → Insert, Insert!, Update_all, increment etc..
- To skip validations for save method we can pass validate: false as an argument → save(validate: false)

→ To run validations on our own we can use valid? & invalid? these returns true & false for valid & invalid objects
→ valid? ⇒ true if object is valid & false if it is not | Obj.valid?
→ Invalid? ⇒ false if object is valid & true if it is not | Obj.invalid?

Validation helpers

to use directly into class validations. rails offers many pre-defined validations rules. every time validation fails, an error is added to the objects errors collection

So, with a single line of code we can add same kind of validation to several attributes.

all of them accept :on and :message options. which tells when validations should be run and what message should be added to the object's errors collection.

1. acceptance

This method validates that a checkbox on the UI was checked when a form was submitted.

- Validates : terms_of_service, acceptance : true
We can write custom message other than default "must be accepted"
= Validates : terms_of_service, acceptance : { message : 'must be abided' }

2. Validates_associated :

This helper is used when our model has associations with other models. and they needs to be validated. when we try to save our object. valid? will be called upon each

has_many : books
Validates_associated : books.

2.6 Format

This helper validates the attributes values by testing whether they match a given regular expression, which is specified using the :with option

Class Product < ApplicationRecord

Validates : Legacy_code, format : { with: /\A[a-z]+\z/, message : "only allow letters" }

Alternatively we can require that specified attributes does not match regular expression by using : Without option.

for app Validates : title, format : { with: /[a-zA-Z0-9]+\z/, message : "only allow letters" }

Regex is a sequence of special characters which holds an expression, used to match a pattern in a string. in ruby a pattern is written between forward slashes.

---

3. Confirmation

you should use this helper when a two text field which should remove exactly the same content.

⇒ Validates : email, confirmation : true

↳ In views template we can use

<%= text_field : person, :email %>

<%= text_field : person, :email_confirmation %>

4. Comparison

Validates : start_date, comparison : (greater_than : : end_date)

↳ options :- greater_than, greater_than_or_equal_to, equal_to, less_than, less_than_or_equal_to, other_than.

5 inclusion and exclusion

→ Validates : size, inclusion : {in : %w (small medium large), message : "%{value} is not a valid size"}

exclusion

inclusion has an option :in that recieves the set of values that will be accepted :in option has an alias called within. that we can use for same purpose

7) length :

This helps validates the length of the attributes value. It has many options so we can specify length constraints in different ways.

⇒ Validates : name, length : {minimum : 2}
Validates : bio, length : {maximum : 500}
Validates : password, length {in : 6..20}
Validates : registration_number, length : {is : 6}

default error messages depends on the type of length validation being performed. we can customize it using the, :wrong_length, :too_long and :too_short options and %{count}

→ Validates : bio, length : {maximum : 1000, too_long : "%{count}"%{max}}

8. numericality

This helps validates that our attribute have only numeric values. by default, it will match an optional sign followed by an integer or floating point number.

To specify that only integer numbers are allowed, we set :only_integer to be true. Then it will use

≠ /\A[+-]?\d+\z/ regular expression to validate.

⇒ Validates :points, numericality: true
Validates :games_played, numericality: {only_integer: true}

9) presence and blank?

To check if value of attribute are not empty. It uses the blank method to check if the value is either nil or a blank string.

⇒ validates :name, :login, :email, :presence:true.

10) Uniqueness

It checks an existing record in the model's table, searching for an existing record with same value in that attributes.

CanCanCan

In the controllers where all actions has to be authorized.

Class ---Controller < application_controller.
  load_and_authorize_resource

Create ability.rb file

Class Ability
  include CanCan::Ability

  def initialize(user, session)
    if session[:role] == " "
      can :(action), :(Model)
      cannot :(action), :(Model)
    end

→ To use session in ability.rb / In the controller where user data is processed.

∴ session[:role] = params[:user][:role_title]
= {"string that is passed"}

And in Application controller

  def current_ability
    @current_ability ||= Ability.new(current_user, session)
  end.

Then this can be accessed in initialize method of ability.rb file.

unpublish specific version
gem yank <name> -v <version>

DATE
PAGE

To create a new gem.
= bundle gem <gem-name>
./git repo for hosting code.
./bin files for command line execution.
./lib Actual source code of gem. → version file present to change.
.gemspec basic info for gem.
Gemfile other gems on which our gem is dependent.
Rakefile setup for test suite rake utilities.
readme info to install gem.

To build the gem
→ gem build <gem-name>.gemspec. / rake install = gem build + gem install.

Test the gem.
→ gem install <name-of-gem>
↳ irb - require <name of gem>.

To publish
Sign in to host platform
1 ⇒ gem signin
to push the gem repo.
2 ⇒ gem push <name of gem>.gemfile

OR

rake release (both cmd + tag)
↙

↳ Increase version on each publish.

↳ name of the gem unique globally on that platform.
↳ gem name.gem binary file should not be committed.

Gem version → X.X.X
↙ Major release
↓ Minor release
→ patch release (bug fixes) x0m.