### Ruby

## Basics:

* Extension .rb is used for ruby files.
* Puts “ \_ ” is used to print something on console.
* Print “\_” can be used instead to puts to print something without a new line.
* “ # ” is called pound character and is used to add comment in ruby file.
* =begin , =end is used for multiline comments.
* Simple mathematical functions can be carried out within puts statement using the symbol #{ expression } ex: he has #{25+30/6} Rs for shopping.

## Variables:

Variables in ruby are dynamic. i.e. we don’t need to mention it’s type & ruby will know variable type automatically based on the value (Integer, Float or String ) assigned to the variable.

# Variables in Ruby class:

* Local variables: Variables that are defined in a method are called local variables. They are available locally for that method and not outside of it. Local variables begin with a lower case letter or \_. ( \_varname )
* Instance variables: Instance variables are available across all methods of a class for a particular instance or object. Value of instance variable changes from object to object. Instance variable begins with @ sign followed by variable name. (@varname)
* Class variables: Class variables are available across different objects. They are preceded by @@ sign followed by variable name. (@@varname)
* Global variables: To have a variable to be available across classes we need to have global variable. Global variables are preceded by a ($). ($varname)

## Data types:

Data types represents a type of data such as text, string, numbers, etc.

There are different data types in Ruby:

* Numbers
* Strings
* Symbols
* Hashes
* Arrays
* Booleans

# Numbers:

Integers, floating point, complex, rational numbers comes under this data type.

According to their size there are two types of integers Bignum and Fixnum.

Fixnum They are normal numbers 1

Bignum They are big numbers 111111111111

Float Decimal numbers 3.0

Complex Imaginary numbers 4 + 3i

Rational They are fractional numbers 9/4

BigDecimal Precision decimal numbers 6.0

# Strings:

A string is a group of letters that represent a sentence or a word. Strings are defined by enclosing a text within single (') or double (") quote.

* Two strings can be concatenated using **+** sign in between them.

“ Hello ” + “ World! ” = “Hello World!”

* Multiplying a number string with a number will repeat the string as many times. [ex :- “1” + “2” + “3” = “123” , “3” \* 2 = “33”]

# Symbols:

* Symbols are like strings, symbols are preceded by “ : ”
* Symbols doesn’t have spaced, symbols with multiple words are written with the help of \_ to seprate words.
* Ex:- :abcd , :Hello\_world

# Hashes:

A hash assign values to it’s keys. Value to a key is assigned by => sign. All key value pairs are separated by , between them and all pairs are enclosed within curly braces.

Ex: {"Akash" => "Physics", "Ankit" => "Chemistry", "Aman" => "Maths"}

puts data [“Ankit”]

output – Chemistry

# Arrays:

An array stores data or list of data. It can contain all types of data. Data in an array are separated by comma in between them and are enclosed by square bracket. For example,

["Akash", "Ankit", "Aman"]

Elements from an array are retrieved by their position. The position of elements in an array starts with 0.

# Boolean:

A Boolean data type is a value that can only be either true or false or in binary it can be 0 or 1 respectively.

A true Boolean value might indicate that the object is valid (e.g., an email address has been typed correctly). A false Boolean value indicates that the object is invalid and has not been done correctly (e.g., you’ve forgotten to fill out a required field).

## Operators:

Ruby has built in modern set of operators. Operators are symbols assigned to perform a specific operation.

Ex- +, -, \*, /, % etc.

* Unary operator
* Arithmetic operator
* Bitwise operator
* Logical operator
* Ternary operator
* Assignment operator
* Comparison operator
* Range operator

# Unary operators:

! Boolean not operator Inverse the value ( !true – false, !4 - -4 )

~ Bitwise complement Invert the bits of operand

+ unary plus

# Arithmetic operators:

+ Adds both operands

* Subtracts left operand by right operand

/ Divides left side operand with right side

% Divide left operand with right side operand & return the remainder

\* Multiply both operands

\*\* Right operand becomes exponent of left operand

# Bitwise operators:

Bitwise operators works on bits operands

& AND operator

| OR operator

<< Left shift operator

>> Right shift operator

^ XOR operator

~ Complement operator

# Logical Operators:

Logical operators work on bits operands.

&& AND operator

|| OR operator

# Ternary Operators:

First check whether given condition is true or false then execute command on the basis of that.

?: conditional expression

Conditional ? True : False

# Assignment Operators:

Assignment operator assign a value to the operands.

= Simple assignment operator

+= Add assignment operator

-= subtract assignment operator

\*= Multiply assignment operator

/= Divide assignment operator

%= Modulus assignment operator

\*\*= Exponential assignment operator

# Comparison Operators:

Compares two operands

== Equal operator

!= Not equal operator

> left operand is greater than right operand

< Right operand is greater than left operand

>= Left operand is greater than or equal to right operand

<= Right operand is greater than or equal to left operand

<=> Combined comparison operator

.eql? Checks for equality and type of the operands

equal? Checks for the object ID

# Range operators:

Range operators create a range of successive values consisting of a start, end and range of values in between.

The (..) creates a range including the last term and (...) creates a range excluding the last term.

For example, for the range of 1..5, output will range from 1 to 5.

and for the range of 1...5, output will range from 1 to 4.

## Control Statements:

1. IF - else
2. Case
3. For
4. While do-while
5. Until
6. Break next
7. Redo retry
8. Comments

## IF – Else:

* IF statement:

If statement tests the condition and the block statements are executed if the conditional is true.

If (condition)

#Code to be executed

end

OR

( Code ) if ( Condition )

# If else:

Ruby if else statement tests the condition. The if block statement is executed if condition is true otherwise else block statement is executed.

Syntax:

if(condition)

//code if condition is true

else

//code if condition is false

end

* If-else-If (elsif):

Ruby if else if statement tests the condition. The if block statement is executed if condition is true otherwise else block statement is executed.

Syntax:

if(condition1)

//code to be executed if condition1is true

elsif (condition2)

//code to be executed if condition2 is true

else

//code to be executed if condition3 is true

end

* Ternary (Shortened if statement)

In Ruby ternary statement, the if statement is shortened. First it evaluats an expression for true or false value then execute one of the statements.

Syntax:

test-expression ? if-true-expression : if-false-expression

Example:

var = gets.chomp.to\_i;

a = (var > 3 ? true : false);

puts a

## Case Statements:

The case statement matches with one of the when statements the code corresponding to that when statement will run.

Case expression

When expression1

Code…

When expression2

Code…

When expression3

Code…

Else

Code…

End

## For loop:

Ruby for loop iterates over a specific range of numbers. Hence, for loop is used if a program has fixed number of iterations.

Ruby for loop will execute once for each element in expression.

For variable in (Range / Expression) do

Puts i

End

Examples:

|  |  |
| --- | --- |
| a = gets.chomp.to\_i  for i in range 1..a do  puts i  end | x = [“green”, “blue, “white”, “black”]  for i in range x do  puts i  end |
| Input=5, output= 1 2 3 4 5 | Output: Green blue white black |

## While loop:

Ruby while loop is used to iterate a program several times. While loop is used when the number of iterations is not known for a program.

Ruby while loop executes a condition while a condition is true, once the condition becomes false loop stops executing.

# Syntax:

While condition [do]

Code…

End

# Modifier:

Code while condition

# Ruby do while:

The Ruby do while loop iterates a part of program several times. It is quite similar to a while loop with the only difference that loop will execute at least once. It is due to the fact that in do while loop, condition is written at the end of the code.

Syntax: loop do

code to be executed

break if condition

end

Ex: loop do

puts "Checking for answer"

answer = gets.chomp

if answer != '5'

break

end

end

## Untill loop:

The Ruby until loop runs until the given condition evaluates to true. It exits the loop when condition becomes true. It is just opposite of the while loop which runs until the given condition evaluates to false.

# Syntax:

until conditional

code

end

Example:

i = 1

until i == 10

print i\*10, "\n"

i += 1

end

# Untill modifier:

code until condition

## Break and Next statement:

Break: The Ruby break statement is used to terminate a loop. It is mostly used in

while loop where value is printed till the condition is true, then break statement terminates the loop.

The break statement is called from inside the loop.

Next: The Ruby next statement is used to skip loop's next iteration. Once the next statement is executed, no further iteration will be performed.

The next statement in Ruby is equivalent to continue statement in other languages.

# Break example:

i = 1

while true

if i\*5 >= 25

**break**

end

puts i\*5

i += 1

end

loop will run untill if statement returns true.

# Next example:

for i in 5...11

if i == 7 then

next

end

puts i

end

loop will print values of i from 5 to 11 except when i ==7.

## Redo and Retry:

Redo: Ruby redo statement is used to repeat the current iteration of the loop. The redo statement is executed without evaluating the loop's condition.

Retry: Ruby retry statement is used to repeat the whole loop iteration from the start.

The retry statement is used inside a loop.

## OOPs:

# Ruby classes and objects:

Object – Object is a physical as well as a logical entity whereas a class is only a logical entity.

All ruby objects has default root **Object**.Ruby objects inherit from **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 a predefined method in ruby.

Syntax – objectName = className.new

Class – Each ruby class is an instance of class **class**. All classes are inherted from BasicObject class. Classes in ruby are first class objects.

Classes are defined as first it starts with keyword class followed by ClassName and is finished with the keyword end.

Syntax –

Class ClassName

Codes..

End

# Concepts –

Ruby is true object oriented language. Everything in ruby is an object.

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

Basic object oriented programming concepts –

1. Encapsulation
2. Polymorphism
3. Inheritance
4. Abstraction

* Inheritance:

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

The class whose attributes and features is inherited into another class is called base class and the class inheriting those from base class is called the derived class.

Syntax:

Class base

#data and methords

End

Class derived < base

#data and members

End

< symbol is used for inheriting all the data and 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 car < vehicle

Def description

Puts “ this is a car ”

End

End

Class bus<vehicle

Def display\_this

Puts “ this is a bus ”

end

object1 = car.new(‘nissan’, ‘red’)

object2 = bus.new(‘volvo’, ‘white’)

object1.description

object2.description

object2.display\_this

Output –

This is a car

This is a vehicle

This is a bus

In above example we have one base class “vehicle” and two derived classes “car” and “bus”. Car and bus inherits methods from vehicle but we have a common name description in both base class vehicle and derived class car and derived class car however their functionality is different.

Car has two methord called description. One is car.description and vehicle.description but for object of car only car.description methord executes I,e – car description overrides vehicle.description

But if we use super keyword in the derived class method. The control gets back to base class and the method of base class gets executed.

Derived class attributes –

Now suppose we want our derived class to have attributes of their own. We would still have to pass the variables for the base class as well as the derived class. then we use super to call the constructor of the base class and then we initialize the derived class attributes.

class Vehicle

    attr\_accessor :vehicle\_name

    attr\_accessor :vehicle\_color

    def initialize(*vehicle\_name*, *vehicle\_color*)

        @vehicle\_name = vehicle\_name

        @vehicle\_color = vehicle\_color

    end

end

class Car *< Vehicle*

     attr\_accessor :car\_model

  def initialize(*vehicle\_name*, *vehicle\_color*, *car\_model*)

        # Using super keyword

        super(vehicle\_name, vehicle\_color)

        @car\_model = car\_model

    end

end

# creating object

object = *Car*.new('Nissan', 'white', 'xyz')

# calling object

puts object.vehicle\_name

puts object.vehicle\_color

puts object.car\_model

Output:

Nissan

white

xyz

We introduced an attribute car\_model in the derived class Car while creating the object we passed values for the base as well as derived class (Car) as the derived class has the attributes of the base class (Vehicle) as well.

From the below statement, object = Car.new('Nissan', 'white', 'xyz') The control goes to the initialize method of Car, from there we use super to pass the Vehicle attributes to the initialize method of Vehicle i.e. super(vehicle\_name, vehicle\_color). The control then passed into the initialize method of vehicle and then returns to the place it was called. then the derived attributes are initialized i.e. @car\_model = car\_model

# Multiple inheritance:

We need multiple inheritance when want to have

Class A to inherit class B

Class A to inherit class C

i.e for a class A we want to inherit attributes and methods from both class B and C simultaneously.

Hence class A to have more than one parent class

In ruby this is not possible without mixins

Mixin – mixin is a methord of using modules and include them in class to get multiple inheritance.

Module A

Def a1

Code..

End

Def a2

Code..

End

Module B

Def b1

Code..

End

Def b2

Code..

End

Class sample

Include A

Include B

end

in above ex. Class sample inherits code from both module A and B hence multiple inheritance is achived.

# Modules:

Modules are like blocks of code that contains methods and even constants while coding, we might have a lot of tools we wish to use however this might clutter up the whole program. So we put these in modules and only use the modules when we wish to use the methods and constants inside they are similar to classes with the exception that we cannot create objects from modules.

Module syntax :

module ModuleName

#methods and constants

End

Example:

module ConstantsAndMethods

CONST\_ONE = 10

CONST\_TWO = 20

def method1

puts ‘This belongs to ConstantsAndMethods’

end

end

Modules are written with CapitalizedCamelCase, which is just capitalizing the first letter of every word in the module name with no spaces.

Constants should be written in caps and underscores between words and we have a method called method1. to use the data and methods within a module we use the keyword require.