Ruby

Ruby is a scripting language designed by Yukihiro Matsumoto, also known as Matz. It runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This tutorial gives a complete understanding on Ruby.

Ruby is a pure object-oriented programming language. It was created in 1993 by Yukihiro Matsumoto of Japan.

Ruby Basic Literals:

The rules Ruby uses for literals are simple and intuitive. This section explains all basic Ruby Literals.

Integer Numbers:

Ruby supports integer numbers. An integer number can range from -2^{30} to 2^{30-1} or -2^{62} to 2^{62-1}. Integers with-in this range are objects of class *Fixnum* and integers outside this range are stored in objects of class *Bignum*.

You write integers using an optional leading sign, an optional base indicator (0 for octal, 0x for hex, or 0b for binary), followed by a string of digits in the appropriate base. Underscore characters are ignored in the digit string.

You can also get the integer value corresponding to an ASCII character or escape sequence by preceding it with a question mark.

Example:

```
123                 # Fixnum decimal
1_234               # Fixnum decimal with underline
-500                # Negative Fixnum
0377                # octal
0xff                # hexadecimal
0b1011              # binary
?a                  # character code for 'a'
```

```
?\n                 # code for a newline (0x0a)

12345678901234567890 # Bignum
```

NOTE: Class and Objects are explained in a separate chapter of this tutorial.

Floating Numbers:

Ruby supports integer numbers. They are also numbers but with decimals. Floating-point numbers are objects of class *Float* and can be any of the following:

Example:

```
123.4                # floating point value

1.0e6                # scientific notation

4E20                 # dot not required

4e+20                # sign before exponential
```

String Literals:

Ruby strings are simply sequences of 8-bit bytes and they are objects of class String. Double-quoted strings allow substitution and backslash notation but single-quoted strings don't allow substitution and allow backslash notation only for \\ and \'

Example:

```
#!/usr/bin/ruby -w


puts 'escape using "\\"';

puts 'That\'s right';
```

This will produce the following result:

```
escape using "\"

That's right
```

You can substitute the value of any Ruby expression into a string using the sequence **#{ expr }**. Here, expr could be any ruby expression.

```
#!/usr/bin/ruby -w


puts "Multiplication Value : #{24*60*60}";
```

Ruby Syntax:

☐ Whitespace characters such as spaces and tabs are generally ignored in Ruby code, except when they appear in strings.

☐ Ruby interprets semicolons and newline characters as the ending of a statement. However, if Ruby encounters operators, such as +, -, or backslash at the end of a line, they indicate the continuation of a statement.

☐ Identifiers are names of variables, constants, and methods. Ruby identifiers are case sensitive. It mean Ram and RAM are two different itendifiers in Ruby.

☐ Ruby comments start with a pound/sharp (#) character and go to EOL.

Ruby Data Types:

Basic types are numbers, strings, ranges, arrays, and hashes.

Integer Numbers in Ruby:

123 # Fixnum decimal

1_6889 # Fixnum decimal with underline

-5000 # Negative Fixnum

0377 # octal

0xee # hexadecimal

0b1011011 # binary

?b # character code for 'b'

?\n # code for a newline (0x0a)

12345678901234567890 # Bignum

Ruby Arrays:

Ruby arrays are ordered, integer-indexed collections of any object. Each element in an array is associated with and referred to by an index.

Array indexing starts at 0, as in C or Java. A negative index is assumed relative to the end of the array --- that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array, and so on.

Ruby arrays can hold objects such as String, Integer, Fixnum, Hash, Symbol, even other Array objects. Ruby arrays are not as rigid as arrays in other languages. Ruby arrays grow automatically while adding elements to them.

Creating Arrays:

There are many ways to create or initialize an array. One way is with the *new*class method:

```
names = Array.new
```

You can set the size of an array at the time of creating array:

```
names = Array.new(20)
```

The array *names* now has a size or length of 20 elements. You can return the size of an array with either the size or length methods:

```
#!/usr/bin/ruby


names = Array.new(20)
puts names.size  # This returns 20
puts names.length # This also returns 20
```

This will produce the following result:

```
20
20
```

You can assign a value to each element in the array as follows:

```
#!/usr/bin/ruby

names = Array.new(4, "mac")

puts "#{names}"
```

This will produce the following result:

```
macmacmacmac
```

You can also use a block with new, populating each element with what the block evaluates to:

```
#!/usr/bin/ruby

nums = Array.new(10) { |e| e = e * 2 }

puts "#{nums}"
```

This will produce the following result:

```
024681012141618
```

There is another method of Array, []. It works like this:

```
nums = Array.[](1, 2, 3, 4,5)
```

One more form of array creation is as follows :

```
nums = Array[1, 2, 3, 4,5]
```

The *Kernel* module available in core Ruby has an Array method, which only accepts a single argument. Here, the method takes a range as an argument to create an array of digits:

```
#!/usr/bin/ruby

digits = Array(0..9)
```

```
puts "#{digits}"
```

This will produce the following result:

```
0123456789
```

Ruby Hashes:

A Hash is a collection of key-value pairs like this: "employee" => "salary". It is similar to an Array, except that indexing is done via arbitrary keys of any object type, not an integer index.

The order in which you traverse a hash by either key or value may seem arbitrary and will generally not be in the insertion order. If you attempt to access a hash with a key that does not exist, the method will return *nil*.

Creating Hashes:

As with arrays, there is a variety of ways to create hashes. You can create an empty hash with the *new* class method:

```
months = Hash.new
```

You can also use *new* to create a hash with a default value, which is otherwise just *nil*:

```
months = Hash.new( "month" )

or

months = Hash.new "month"
```

When you access any key in a hash that has a default value, if the key or value doesn't exist, accessing the hash will return the default value:

```
#!/usr/bin/ruby

months = Hash.new( "month" )

puts "#{months[0]}"
```

```
puts "#{months[72]}"
```

This will produce the following result:

```
month
month
```

```
#!/usr/bin/ruby


H = Hash["a" => 100, "b" => 200]


puts "#{H['a']}"
puts "#{H['b']}"
```

This will produce the following result:

```
100
200
```

You can use any Ruby object as a key or value, even an array, so following example is a valid one:

```
[1,"jan"] => "January"
```

Features of Ruby

- Ruby is an open-source and is freely available on the Web, but it is subject to a license.
- Ruby is a general-purpose, interpreted programming language.
- Ruby is a true object-oriented programming language.
- Ruby is a server-side scripting language similar to Python and PERL.
- Ruby can be used to write Common Gateway Interface (CGI) scripts.
- Ruby can be embedded into Hypertext Markup Language (HTML).

- Ruby has a clean and easy syntax that allows a new developer to learn Ruby very quickly and easily.
- Ruby has similar syntax to that of many programming languages such as C++ and Perl.
- Ruby is very much scalable and big programs written in Ruby are easily maintainable.
- Ruby can be used for developing Internet and intranet applications.

Ruby is a perfect Object Oriented Programming Language. The features of the object-oriented programming language include:

- Data Encapsulation:
- Data Abstraction:
- Polymorphism:
- Inheritance:

These features have been discussed in Object Oriented Ruby.

An object-oriented program involves classes and objects. A class is the blueprint from which individual objects are created. In object-oriented terms, we say that your *bicycle* is an instance of the *class of objects* known as bicycles.

Take the example of any vehicle. It comprises wheels, horsepower, and fuel or gas tank capacity. These characteristics form the data members of the class Vehicle. You can differentiate one vehicle from the other with the help of these characteristics.

A vehicle can also have certain functions, such as halting, driving, and speeding. Even these functions form the data members of the class Vehicle. You can, therefore, define a class as a combination of characteristics and functions.

A class Vehicle can be defined in Java as follows :

```
Class Vehicle
{
   Number no_of_wheels
   Number horsepower
   Characters type_of_tank
   Number Capacity
   Function speeding
   {
   }
   Function driving
   {
   }
   Function halting
   {
   }
}
```

By assigning different values to these data members, you can form several instances of the class Vehicle. For example, an airplane has three wheels, horsepower of 1,000, fuel as the type of tank, and a capacity of 100 liters. In the same way, a car has four wheels, horsepower of 200, gas as the type of tank, and a capacity of 25 litres.

Defining a Class in Ruby:

To implement object-oriented programming by using Ruby, you need to first learn how to create objects and classes in Ruby.

A class in Ruby always starts with the keyword *class* followed by the name of the class. The name should always be in initial capitals. The class *Customer* can be displayed as:

```
class Customer
end
```

You terminate a class by using the keyword *end*. All the data members in the*class* are between the class definition and the *end* keyword.

Variables in a Ruby Class:

Ruby provides four types of variables:

- **Local Variables:** Local variables are the variables that are defined in a method. Local variables are not available outside the method. You will see more details about method in subsequent chapter. Local variables begin with a lowercase letter or _.
- **Instance Variables:** Instance variables are available across methods for any particular instance or object. That means that instance variables change from object to object. Instance variables are preceded by the at sign (@) followed by the variable name.
- **Class Variables:** Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign @@ and are followed by the variable name.
- **Global Variables:** Class variables are not available across classes. If you want to have a single variable, which is available across classes, you need to define a global variable. The global variables are always preceded by the dollar sign ($).

Example:

Using the class variable @@no_of_customers, you can determine the number of objects that are being created. This enables in deriving the number of customers.

```
class Customer
   @@no_of_customers=0
end
```

Creating Objects in Ruby using *new* Method:

Objects are instances of the class. You will now learn how to create objects of a class in Ruby. You can create objects in Ruby by using the method *new* of the class.

The method *new* is a unique type of method, which is predefined in the Ruby library. The new method belongs to the *class* methods.

Here is the example to create two objects cust1 and cust2 of the class Customer:

```
cust1 = Customer. new

cust2 = Customer. new
```

Here, cust1 and cust2 are the names of two objects. You write the object name followed by the equal to sign (=) after which the class name will follow. Then, the dot operator and the keyword *new* will follow.

Custom Method to create Ruby Objects :

You can pass parameters to method *new* and those parameters can be used to initialize class variables.

When you plan to declare the *new* method with parameters, you need to declare the method *initialize* at the time of the class creation.

The *initialize* method is a special type of method, which will be executed when the *new* method of the class is called with parameters.

Here is the example to create initialize method:

```
class Customer

   @@no_of_customers=0

   def initialize(id, name, addr)

      @cust_id=id

      @cust_name=name

      @cust_addr=addr

   end

end
```

In this example, you declare the *initialize* method with **id, name**, and **addr** as local variables. Here, *def* and *end* are used to define a Ruby

method *initialize*. You will learn more about methods in subsequent chapters.

In the *initialize* method, you pass on the values of these local variables to the instance variables @cust_id, @cust_name, and @cust_addr. Here local variables hold the values that are passed along with the new method.

Now, you can create objects as follows:

```
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
```

Member Functions in Ruby Class:

In Ruby, functions are called methods. Each method in a *class* starts with the keyword *def* followed by the method name.

The method name always preferred in **lowercase letters**. You end a method in Ruby by using the keyword *end*.

Here is the example to define a Ruby method:

```
class Sample
   def function
      statement 1
      statement 2
   end
end
```

Here, *statement 1* and *statement 2* are part of the body of the method *function*inside the class Sample. These statments could be any valid Ruby statement. For example we can put a method *puts* to print *Hello Ruby* as follows:

```
class Sample
   def hello
      puts "Hello Ruby!"
   end
```

```
end
```

Now in the following example, create one object of Sample class and call *hello*method and see the result:

```
#!/usr/bin/ruby


class Sample
   def hello
      puts "Hello Ruby!"
   end
end


# Now using above class to create objects
object = Sample. new
object.hello
```

This will produce the following result:

```
Hello Ruby!
```

Ruby Global Variables:

Global variables begin with $. Uninitialized global variables have the value *nil*and produce warnings with the -w option.

Assignment to global variables alters global status. It is not recommended to use global variables. They make programs cryptic.

Here is an example showing usage of global variable.

```
#!/usr/bin/ruby


$global_variable = 10
class Class1
  def print_global
     puts "Global variable in Class1 is #$global_variable"
  end
```

```
end
class Class2
  def print_global
     puts "Global variable in Class2 is #$global_variable"
  end
end

class1obj = Class1.new
class1obj.print_global
class2obj = Class2.new
class2obj.print_global
```

Here $global_variable is a global variable. This will produce the following result:

NOTE: In Ruby you CAN access value of any variable or constant by putting a hash (#) character just before that variable or constant.

```
Global variable in Class1 is 10
Global variable in Class2 is 10
```

Ruby Instance Variables:

Instance variables begin with @. Uninitialized instance variables have the value*nil* and produce warnings with the -w option.

Here is an example showing usage of Instance Variables.

```
#!/usr/bin/ruby

class Customer
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
```

```
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
end

# Create Objects
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# Call Methods
cust1.display_details()
cust2.display_details()
```

Here, @cust_id, @cust_name and @cust_addr are instance variables. This will produce the following result:

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
```

Ruby Class Variables:

Class variables begin with @@ and must be initialized before they can be used in method definitions.

Referencing an uninitialized class variable produces an error. Class variables are shared among descendants of the class or module in which the class variables are defined.

Overriding class variables produce warnings with the -w option.

Here is an example showing usage of class variable:

```
#!/usr/bin/ruby


class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
    def total_no_of_customers()
       @@no_of_customers += 1
       puts "Total number of customers: #@@no_of_customers"
    end
end

# Create Objects
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# Call Methods
cust1.total_no_of_customers()
cust2.total_no_of_customers()
```

Here @@no_of_customers is a class variable. This will produce the following result:

```
Total number of customers: 1
Total number of customers: 2
```

Ruby Local Variables:

Local variables begin with a lowercase letter or _. The scope of a local variable ranges from class, module, def, or do to the corresponding end or from a block's opening brace to its close brace {}.

When an uninitialized local variable is referenced, it is interpreted as a call to a method that has no arguments.

Assignment to uninitialized local variables also serves as variable declaration. The variables start to exist until the end of the current scope is reached. The lifetime of local variables is determined when Ruby parses the program.

In the above example local variables are id, name and addr.

Ruby Constants:

Constants begin with an uppercase letter. Constants defined within a class or module can be accessed from within that class or module, and those defined outside a class or module can be accessed globally.

Constants may not be defined within methods. Referencing an uninitialized constant produces an error. Making an assignment to a constant that is already initialized produces a warning.

```
#!/usr/bin/ruby


class Example
   VAR1 = 100
   VAR2 = 200
   def show
       puts "Value of first Constant is #{VAR1}"
       puts "Value of second Constant is #{VAR2}"
   end
end


# Create Objects
```

```
object=Example.new()

object.show
```

Here VAR1 and VAR2 are constant. This will produce the following result:

```
Value of first Constant is 100

Value of second Constant is 200
```

Iterators

Iterators are nothing but methods supported by *collections*. Objects that store a group of data members are called collections. In Ruby, arrays and hashes can be termed collections.

Iterators return all the elements of a collection, one after the other. We will be discussing two iterators here, *each* and *collect*. Let's look at these in detail.

Ruby *each* Iterator:

The each iterator returns all the elements of an array or a hash.

Syntax:

```
collection.each do |variable|

   code

end
```

Executes *code* for each element in *collection*. Here, *collection* could be an array or a ruby hash.

Example:

```
#!/usr/bin/ruby


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

ary.each do |i|

   puts i
```

```
end
```

This will produce the following result:

```
1
2
3
4
5
```

You always associate the *each* iterator with a block. It returns each value of the array, one by one, to the block. The value is stored in the variable **i** and then displayed on the screen.

Ruby *collect* Iterator:

The *collect* iterator returns all the elements of a collection.

Syntax:

```
collection = collection.collect
```

The *collect* method need not always be associated with a block. The *collect*method returns the entire collection, regardless of whether it is an array or a hash.

Example:

```
#!/usr/bin/ruby


a = [1,2,3,4,5]
b = Array.new
b = a.collect
puts b
```

This will produce the following result:

```
1
2
```

```
3
4
5
```

NOTE: The *collect* method is not the right way to do copying between arrays. There is another method called a *clone*, which should be used to copy one array into another array.

You normally use the collect method when you want to do something with each of the values to get the new array. For example, this code produces an array b containing 10 times each value in a.

```
#!/usr/bin/ruby


a = [1,2,3,4,5]
b = a.collect{|x| 10*x}
puts b
```

This will produce the following result:

```
10
20
30
40
50
```

Pattern Matching

A `Regexp` holds a regular expression, used to match a pattern against strings. Regexps are created using the `/.../` and `%r{...}` literals, and by the`Regexp::new` constructor. Regular expressions (*regexp*s) are patterns which describe the contents of a string. They're used for testing whether a string contains a given pattern, or extracting the portions that match. They are created with the `/`*pat*`/` and`%r{`*pat*`}` literals or the `Regexp.new` constructor.
A regexp is usually delimited with forward slashes (/). For example:

```
/hay/ =~ 'haystack'    #=> 0
/y/.match('haystack') #=> #<MatchData "y">
```

If a string contains the pattern it is said to *match*. A literal string matches itself.

Here 'haystack' does not contain the pattern 'needle', so it doesn't match:

```
/needle/.match('haystack') #=> nil
```

Here 'haystack' contains the pattern 'hay', so it matches:

```
/hay/.match('haystack')     #=> #<MatchData "hay">
```

Specifically, `/st/` requires that the string contains the letter *s* followed by the letter *t*, so it matches *haystack*, also.

=~ and #match

Pattern matching may be achieved by using =~ operator or #match method.

=~ operator

=~ is Ruby's basic pattern-matching operator. When one operand is a regular expression and the other is a string then the regular expression is used as a pattern to match against the string. (This operator is equivalently defined byRegexp and String so the order of String and Regexp do not matter. Other classes may have different implementations of =~.) If a match is found, the operator returns index of first match in string, otherwise it returns `nil`.

```
/hay/ =~ 'haystack'    #=> 0
'haystack' =~ /hay/    #=> 0
/a/    =~ 'haystack'    #=> 1
/u/    =~ 'haystack'    #=> nil
```

Using =~ operator with a String and Regexp the $~ global variable is set after a successful match. $~ holds a MatchData object. ::last_match is equivalent to$~.

#match method

The match method returns a MatchData object

Ruby is a general-purpose language; it can't properly be called a *web language*at all. Even so, web applications and web tools in general are among the most common uses of Ruby.

Not only can you write your own SMTP server, FTP daemon, or Web server in Ruby, but you can also use Ruby for more usual tasks such as CGI programming or as a replacement for PHP.

Please spend few minutes with CGI Programming Tutorial for more detail on CGI Programming.

Writing CGI Scripts:

The most basic Ruby CGI script looks like this:

```
#!/usr/bin/ruby


puts "HTTP/1.0 200 OK"

puts "Content-type: text/html\n\n"

puts "<html><body>This is a test</body></html>"
```

If you call this script *test.cgi* and uploaded it to a Unix-based Web hosting provider with the right permissions, you could use it as a CGI script.

For example, if you have the Web site http://www.example.com/ hosted with a Linux Web hosting provider and you upload *test.cgi* to the main directory and give it execute permissions, then visiting http://www.example.com/test.cgi should return an HTML page saying ***This is a test***.

Here when *test.cgi* is requested from a Web browser, the Web server looks for*test.cgi* on the Web site, and then executes it using the Ruby interpreter. The Ruby script returns a basic HTTP header and then returns a basic HTML document.

Using cgi.rb:

Ruby comes with a special library called **cgi** that enables more sophisticated interactions than those with the preceding CGI script.

Let's create a basic CGI script that uses cgi:

```
#!/usr/bin/ruby


require 'cgi'


cgi = CGI.new
```

```
puts cgi.header

puts "<html><body>This is a test</body></html>"
```

Here, you created a CGI object and used it to print the header line for you.

Form Processing:

Using class CGI gives you access to HTML query parameters in two ways. Suppose we are given a URL of /cgi-bin/test.cgi?FirstName=Zara&LastName=Ali.

You can access the parameters *FirstName* and *LastName* using CGI#[] directly as follows:

```
#!/usr/bin/ruby


require 'cgi'

cgi = CGI.new

cgi['FirstName'] # =>  ["Zara"]

cgi['LastName']  # =>  ["Ali"]
```

There is another way to access these form variables. This code will give you a hash of all the key and values:

```
#!/usr/bin/ruby


require 'cgi'

cgi = CGI.new

h = cgi.params  # =>  {"FirstName"=>["Zara"],"LastName"=>["Ali"]}

h['FirstName']  # =>  ["Zara"]

h['LastName']   # =>  ["Ali"]
```

Following is the code to retrieve all the keys:

```
#!/usr/bin/ruby


require 'cgi'

cgi = CGI.new
```

```
cgi.keys         # =>  ["FirstName", "LastName"]
```

If a form contains multiple fields with the same name, the corresponding values will be returned to the script as an array. The [] accessor returns just the first of these.index the result of the params method to get them all.

In this example, assume the form has three fields called "name" and we entered three names "Zara", "Huma" and "Nuha":

```
#!/usr/bin/ruby


require 'cgi'
cgi = CGI.new
cgi['name']        # => "Zara"
cgi.params['name'] # => ["Zara", "Huma", "Nuha"]
cgi.keys           # => ["name"]
cgi.params         # => {"name"=>["Zara", "Huma", "Nuha"]}
```

Note: Ruby will take care of GET and POST methods automatically. There is no separate treatment for these two different methods.

An associated, but basic, form that could send the correct data would have HTML code like so:

```
<html>
<body>
<form method="POST" action="http://www.example.com/test.cgi">
First Name :<input type="text" name="FirstName" value="" />
<br />
Last Name :<input type="text" name="LastName" value="" />


<input type="submit" value="Submit Data" />
</form>
</body>
</html>
```