What is Ruby (is programing language)

* Ruby is open source and platform independent language
* Ruby is interpreter based language (line by line execution)

RoR – Ruby on Rails (is framework for web Apps development)

Ruby is a dynamic, open source, object oriented and reflective programming language. Ruby is considered similar to Perl and Smalltalk programming languages. It runs on all types of platforms like Windows, Mac OS and all versions of UNIX.

It is fully object oriented programming language. Everything is an object in Ruby. Each and every code has their properties and actions. Here properties refer to variables and actions refer to methods.

Ruby is considered to follow the principle of POLA (principle of least astonishment). It means that the language behaves in such a way to minimize the confusion for experienced users.

## Step 1 - Install RVM (Ruby Version Manager)

RVM (Ruby Version Manager) is a command-line tool based on Bash and Ruby to manage the ruby installation. RVM allows you to install and configure multiple ruby versions on one system.

The first step we will do is to install the rvm packages using the installer script.

Add the rvm key to the server.

*gpg --keyserver hkp://keys.gnupg.net --recv-keys 409B6B1796C275462A1703113804BB82D39DC0E3 \  
7D2BAF1CF37B13E2069D6956105BD0E739499BDB*

Install the rvm stable version by running the command below.

*curl -sSL https://get.rvm.io | bash -s stable --ruby*

The command will automatically install packages required, and install the latest stable rvm version.

After the installation is complete, run the following command.

*source /usr/local/rvm/scripts/rvm*

[](https://www.howtoforge.com/images/how_to_install_ruby_on_rails_on_ubuntu_1804_lts/big/1.png)

Now you can use the rvm command to manage the ruby version.

*rvm version*

## Step 2 - Setup Ruby Latest Version

The latest version of ruby at this day is Ruby 2.5.1, and it will be automatically installed during the rvm installation, when there is no ruby package on the system.

In this step, we will set up the default ruby version on the Ubuntu system.

Update the rvm to the latest stable version.

*rvm get stable --autolibs=enable  
usermod -a -G rvm root*

Now check all available ruby versions.

*rvm list known*

And you will get a lot of available versions of ruby - install the latest stable version Ruby 2.5.1 using the rvm command as shown below.

*rvm install ruby-2.5.1*

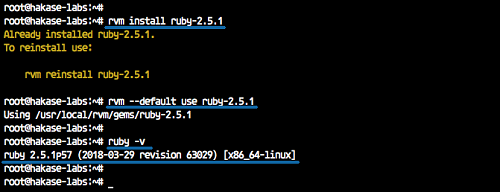
After all installation is complete, make the ruby 2.5.1 as a default version on the Ubuntu system.

*rvm --default use ruby-2.5.1*

Check the Ruby version.

*ruby -v*

Now you will see ruby 2.5.1 is default ruby version on the Ubuntu 18.04 system.

[](https://www.howtoforge.com/images/how_to_install_ruby_on_rails_on_ubuntu_1804_lts/big/2.png)

To call ruby file use : ruby your\_program.rb

To execute your ruby file as script:

1. start your program with #!/usr/bin/env ruby
2. run that script using ./your\_program.rb param
3. If you are not able to execute this script check permissions for file.

Open your terminal and open folder where file is saved.  
Ex /home/User1/program/test.rb

1. Open terminal
2. cd /home/User1/program
3. ruby test.rb

class : is wrapper of body (commands, function /modules/reciepe)

puts : show result

gets : take input (read input)

# : comment line

Def : define or create block (function\_

New : to initialize the object of class

‘’’

‘’’

format or test.rb

class Test

def initialize

puts "I love India"

end

end

# initialize object

Test.new

Let us write a simple program in ruby. All ruby files will have extension **.rb**. So, put the following source code in a test.rb file.

#!/usr/bin/ruby -w

puts "Hello, Ruby!";

## Reserved Words

The following list shows the reserved words in Ruby. These reserved words may not be used as constant or variable names. They can, however, be used as method names.

|  |  |  |  |
| --- | --- | --- | --- |
| BEGIN : start block | do | Next: | then |
| END : closing block | else | Nil | true |
| Alias | elsif | not | undef |
| And | end | or | unless |
| Begin | ensure | redo | until |
| Break | false | rescue | when |
| Case | for | retry | while |
| Class | if | return | while |
| Def : create function body  Example:  Def test{ …} | in | self | \_\_FILE\_\_ |
| defined? | module | super | \_\_LINE\_\_ |

Comments

#!/usr/bin/ruby -w

# This is a single line comment.

If condition

#!/usr/bin/ruby

x = 1

if x > 2

puts "x is greater than 2"

elsif x <= 2 and x!=0

puts "x is 1"

else

puts "I can't guess the number"

end

Case when

#!/usr/bin/ruby

$age = 5

case $age

when 0 .. 2

puts "baby"

when 3 .. 6

puts "little child"

when 7 .. 12

puts "child"

when 13 .. 18

puts "youth"

else

puts "adult"

end

Loop

#!/usr/bin/ruby

$i = 0

$num = 5

while $i < $num do

puts("Inside the loop i = #$i" )

$i +=1

end

for

#!/usr/bin/ruby

for i in 0..5

puts "Value of local variable is #{i}"

end

Break

Terminates the most internal loop. Terminates a method with an associated block if called within the block (with the method returning nil).

Next

Jumps to the next iteration of the most internal loop. Terminates execution of a block if called within a block (with *yield* or call returning nil).

Redo

Restarts this iteration of the most internal loop, without checking loop condition. Restarts *yield* or *call* if called within a block

## Ruby retry Statement

### Syntax

retry

If *retry* appears in rescue clause of begin expression, restart from the beginning of the begin body.

begin

do\_something # exception raised

rescue

# handles error

retry # restart from beginning

end

Method

#!/usr/bin/ruby

def test(a1 = "Ruby", a2 = "Perl")

puts "The programming language is #{a1}"

puts "The programming language is #{a2}"

end

test "C", "C++"

test

## Class Methods

When a method is defined outside of the class definition, the method is marked as *private* by default. On the other hand, the methods defined in the class definition are marked as public by default. The default visibility and the *private* mark of the methods can be changed by *public* or *private* of the Module.

Whenever you want to access a method of a class, you first need to instantiate the class. Then, using the object, you can access any member of the class.

Ruby gives you a way to access a method without instantiating a class. Let us see how a class method is declared and accessed −

class Accounts

def reading\_charge

end

def Accounts.return\_date

end

end

See how the method return\_date is declared. It is declared with the class name followed by a period, which is followed by the name of the method. You can access this class method directly as follows −

Accounts.return\_date

## Ruby opening a file

A Ruby file can be created using different methods for reading, writing or both.

There are two methods to open a file in Ruby:

* **File.new method :** Using this method a new file can be created for reading, writing or both.
* **File.open method :** Using this method a new file object is created. That file object is assigned to a file.

Difference between both the methods is that File.open method can be associated with a block while File.new method can't.

**Syntax:**

1. f = File.new("fileName.rb")

Or,

1. File.open("fileName.rb", "mode") **do** |f|

**Example to create a file**

Let's create a file in Ruby using File.open method to read or write data from files.

**Step 1)** In file hello.rb, write the code to create a new file as shown below.

1. #!/usr/bin/ruby
2. File.open('about', 'w') **do** |f|
3. f.puts "This is JavaTpoint"
4. f.write "You are reading Ruby tutorial.\n"
5. f << "Please visit our website.\n"
6. **end**

**Step 2)** Type the following two commands in the console to view the created file.

1. ruby hello.rb
2. cat about

## Ruby reading a file

There are three different methods to read a file.

To return a single line, following syntax is used.

**Syntax:**

1. f.gets
2. code...

To return the whole file after the current position, following syntax is used.

**Syntax:**

1. f.read
2. code...

To return file as an array of lines, following syntax is used.

**Syntax:**

1. f.readlines
2. [code...]

## Example to read a file

Let's create a file in Ruby using File.open method to read or write data from files.

**Step 1)** In file hello.rb, write the code to read an already existing file as shown below.

1. #!/usr/bin/ruby
2. **while** line = gets
3. puts line
4. **end**

**Step 2)** Type the following command in the console to read the file.

1. ruby hello.rb about

### The sysread Method

The sysread method is also used to read the content of a file. With the help of this method you can open a file in any mode.

**Example:**

In file hello.rb, write the code to read an already existing file as shown below.

1. #!/usr/bin/ruby
3. aFile = File.new("about.txt", "r")
4. **if** aFile
5. content = aFile.sysread(40)
6. puts content
7. **else**
8. puts "Unable to open file!"
9. **end**

## Ruby writing a file

With the help of syswrite method, you can write content into a file. File needs to be opened in write mode for this method.

The new content will over ride the old content in an already existing file.

**Example:**

1. #!/usr/bin/ruby
3. aFile = File.new("about.txt", "r+")
4. **if** aFile
5. aFile.syswrite("New content is written in this file.\n")
6. **end**

## Ruby renaming and deleting a file

Ruby files are renamed using rename method and deleted using delete mehtod.

To **rename** a file, following syntax is used.

**Syntax:**

1. File.rename("olderName.txt", "newName.txt")

**Example:**

1. #!/usr/bin/ruby
3. File.rename("about.txt", "new.txt")

In the above output, about.txt file no longer exist as its name has been changed to new.txt file.

To **delete** a file, following syntax is used.

**Syntax:**

1. File.delete("filename.txt")

**Example:**

1. #!/usr/bin/ruby
3. File.delete("new.txt")

Create directory

Dir.mkdir 'foo/bar'

directory\_name = "name"

Dir.mkdir(directory\_name) unless File.exists?(directory\_name)

#create file

out\_file = File.new("out.txt", "w")

#...

out\_file.puts("write your stuff here")

#...

out\_file.close

# How to reference a method in another Ruby code file?

When writing your first few Ruby programs, you tend to place all of your code in a single file. But as you grow as a Ruby programmer, your Ruby programs will also grow, and at some point you will realize that having a single file containing all of your code just won't do. It is easier to break your code up into logical groupings and place each group in a separate file or files. When you begin using multiple files, you have a need for the Ruby's **require** and **load**methods (both are global functions defined in **Object**, but are used like language keywords) that help you include other files in your program.

The **load** method includes the named Ruby source file *every time* the method is executed:

1. load 'filename.rb'

The more commonly used **require** method loads any given file *only once*:

1. require 'filename'

**require** gives you access to the many extensions and programming libraries bundled with the Ruby programming language-as well as an even larger number of extensions and libraries written independently by other programmers and made available for use with Ruby.

Note that you say **require 'filename'**, not **require 'filename.rb'**. Aside from looking nicer, this bareword way of referring to the extension is necessary because not all extensions use files ending in .rb. Specifically, extensions written in C are stored in files ending with .so or .dll. To keep the process transparent-that is, to save you the trouble of knowing whether the extension you want uses a .rb file or not-Ruby accepts a bareword and then does some automatic file-searching and trying out of possible filenames until it finds the file corresponding to the extension you have requested.

**require(string) => true or false**  
  
Ruby tries to load the library named string, returning true if successful. If the filename does not resolve to an absolute path, it will be searched for in the directories listed in **$:**. If the file has the extension ".rb", it is loaded as a source file; if the extension is ".so", ".o", or ".dll", or whatever the default shared library extension is on the current platform, Ruby loads the shared library as a Ruby extension. Otherwise, Ruby tries adding ".rb", ".so", and so on to the name. The name of the loaded feature is added to the array in **$:**.

**IN RAILS**: Rails uses **load** in preference to **require**, for example, in development mode - which means that if you're trying your application in a browser and making changes to the code at the same time, your changes are reloaded, overriding any caching behavior on the part of the Web server. Multiple **require**calls in the same place don't have the same effect if the application has already read the file in once.

Now, let's look at an example of another class - **p030motorcycle.rb**

1. **class** MotorCycle
2. **def** initialize(make, color)
3. # Instance variables
4. @make = make
5. @color = color
6. **end**
7. **def** start\_engine
8. **if** @engine\_state
9. **puts** 'Engine is already Running'
10. **else**
11. @engine\_state = **true**
12. **puts** 'Engine Idle'
13. **end**
14. **end**
15. **end**

We write another program **p031motorcycletest.rb** to test out the above class.

1. # p031motorcycletest.rb
2. require\_relative 'p030motorcycle'
3. m = MotorCycle.**new**('Yamaha', 'red')
4. m.start\_engine

We use **require\_relative** for this because the location of the file we're loading is relative to the file we're loading it from - they're both in the same directory

If you are using Ruby 1.9 or later, this is the simplest way to do it:

require\_relative 'somelogic'

If you want your code to work in 1.9 and older versions of Ruby, you should do this instead:

require File.join File.dirname(\_\_FILE\_\_), 'somelogic'

Whichever line you choose, you should put it at the top of your ruby file. Then any classes, modules, or global variables defined in somelogic.rb will be available to your program.

Here is the scenario:

/home/user/code/somelogic.rb

class MyMath

def self.sin(number)

...

end

end

You want to use the methods sin in your other file mylogic.rb.

Depending on the version of ruby, do one the following:

Ruby 1.8.x

require "somelogic"

class OtherThings

def some\_method

MyMath.sin(42)

end

end

The use pattern is for all ruby versions the same, but the require statement may be different.

Ruby 1.9.x

require\_relative "somelogic"

or variation

Ruby 1.9.x

require "./somelogic"

The first variation works all the time, the second one only if you call ruby mylogic.rb in the directory where mylogic.rb and somelogic.rb are located.

If you want to load files from that directory from a starting point located in another directory, you should use:

Ruby 1.8.x and Ruby 1.9.x

$: << File.dirname(\_\_FILE\_\_)

Recipe Install Tomcat

## Resources

### tomcat\_install

tomcat\_install installs an instance of the tomcat binary direct from Apache's mirror site. As distro packages are not used we can easily deploy per-instance installations and any version available on the Apache archive site can be installed.

#### properties

* version: The version to install. Default: 8.0.47
* install\_path: Full path to the install directory. Default: /opt/tomcat\_INSTANCENAME\_VERSION
* tarball\_base\_uri: The base uri to the apache mirror containing the tarballs. Default: 'http://archive.apache.org/dist/tomcat/'
* checksum\_base\_uri: The base uri to the apache mirror containing the md5 or sha512 file. Default: 'http://archive.apache.org/dist/tomcat/'
* verify\_checksum: Whether the checksum should be verified against checksum\_base\_uri. Default: true.
* dir\_mode: Directory permissions of the install\_path. Default: '0750'.
* tarball\_uri: The complete uri to the tarball. If specified would override (tarball\_base\_uri and checksum\_base\_uri). checksum will be loaded from "#{tarball\_uri}.{md5,sha512}". This attribute is useful, if you are hosting tomcat tarballs from artifact repositories such as nexus. sha512 sums are used for version constraints: ~> 7.0.84, ~> 8.0.48, ~> 8.5.24, ~> 9.0.10.
* tarball\_path: Local path on disk to the tarball. If the file does not exist, or the checksum does not match, it will be downloaded from tarball\_uri.
* tarball\_validate\_ssl: Validate the SSL certificate, if tarball\_uri is using HTTPS. Default true.
* exclude\_docs: Exclude ./webapps/docs from installation. Default true.
* exclude\_examples: Exclude ./webapps/examples from installation. Default true.
* exclude\_manager: Exclude ./webapps/manager from installation. Default: false.
* exclude\_hostmanager: Exclude ./webapps/host-manager from installation. Default: false.
* tomcat\_user: User to run tomcat as. Default: tomcat\_INSTANCENAME
* tomcat\_group: Group of the tomcat user. Default: tomcat\_INSTANCENAME
* tomcat\_user\_shell: Shell of the tomcat user. Default: /bin/false

#### example

Install an Tomcat 8.0.36 instance named 'helloworld' to /opt/tomcat\_helloworld\_8\_0\_36/ with a symlink at /opt/tomcat\_helloworld/

tomcat\_install 'helloworld' do

version '8.0.36'

end

Install an Tomcat instance named 'helloworld' from a local tarball to /opt/tomcat\_helloworld\_8\_0\_36/ with a symlink at /opt/tomcat\_helloworld/

tomcat\_install 'helloworld' do

version '8.0.36'

verify\_checksum false

tarball\_path '/tmp/apache-tomcat-8.0.36.tar.gz'

end

### tomcat\_service

tomcat\_service sets up the installed tomcat instance to run using the appropriate init system (sys-v, upstart, or systemd)

#### properties

* install\_path: Full path to the install directory. Default: /opt/tomcat\_INSTANCENAME
* env\_vars: An array of hashes containing the environmental variables for Tomcat's setenv.sh script. Note: If CATALINA\_BASE is not passed it will automatically be added as the first item in the array. Default: [ {'CATALINA\_BASE' => '/opt/INSTANCE\_NAME/'}, {'CATALINA\_PID' => '$CATALINA\_BASE/bin/tomcat.pid'} ]
* service\_vars: An array of hashes containing additional systemd directives when setting up a service under systemd.
* sensitive: Excludes diffs that may expose ENV values from the chef-client logs. Default: false
* tomcat\_user: The user the service runs under
* tomcat\_group: The group the service runs under

#### actions

* start
* stop
* enable
* disable
* restart

#### example

tomcat\_service 'helloworld' do

action :start

env\_vars [{ 'CATALINA\_PID' => '/my/special/path/tomcat.pid' }]

end