# Ruby (programming language)

From Wikipedia, the free encyclopedia

Not to be confused with Ruby on Rails.

Ruby

Paradigm(s) multi-paradigm: object-oriented, imperative, functional, reflective

Appeared in 1995

Designed by Yukihiro Matsumoto

Developer Yukihiro Matsumoto, et al.

Stable release 2.0.0-p0 (February 24, 2013)

Typing discipline duck, dynamic

Scope lexical, sometimes dynamic

Major implementations Ruby MRI, YARV, Rubinius, MagLev, JRuby, MacRuby, RubyMotion, HotRuby, IronRuby, mruby

Influenced by Ada,[1] C++,[1] CLU,[2] Dylan,[2] Eiffel,[1] Lisp,[2] Perl,[2] Python,[2] Smalltalk[2]

Influenced Elixir, Falcon, Fancy,[3] Groovy, Ioke,[4] Mirah, Nu,[5] Reia

OS Cross-platform

License Ruby License or BSD License[6][7]

Usual filename extensions .rb, .rbw

www.ruby-lang.org Ruby Programming at Wikibooks

Ruby is a dynamic, reflective, general-purpose object-oriented programming language that combines syntax inspired by Perl with Smalltalk-like features. It was also influenced by Eiffel and Lisp.[8] Ruby was first designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan.

Ruby supports multiple programming paradigms, including functional, object oriented and imperative. It also has a dynamic type system and automatic memory management; it is therefore similar in varying respects to Smalltalk, Python, Perl, Lisp, Dylan, and CLU.

The standard 1.8.7 implementation is written in C, as a single-pass interpreted language. Starting with the 1.9 branch, and continuing with the current 2.0 branch, YARV has been used, and will eventually supersede the slower Ruby MRI. The language specifications for Ruby were developed by the Open Standards Promotion Center of the Information-Technology Promotion Agency (a Japanese government agency) for submission to the Japanese Industrial Standards Committee and then to the International Organization for Standardization. It was accepted as a Japanese Industrial Standard (JIS X 3017) in 2011[9] and an international standard (ISO/IEC 30170) in 2012.[10] As of 2010, there are a number of complete or upcoming alternative implementations of Ruby, including YARV, JRuby, Rubinius, IronRuby, MacRuby (and its iOS counterpart, RubyMotion), mruby, HotRuby, Topaz and Opal. Each takes a different approach, with IronRuby, JRuby, MacRuby and Rubinius providing just-in-time compilation and MacRuby and mruby also providing ahead-of-time compilation.

History [edit]

Ruby was conceived on February 24, 1993 by Yukihiro Matsumoto who wished to create a new language that balanced functional programming with imperative programming.[11] Matsumoto has said, "I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That's why I decided to design my own language."[12]

At a Google Tech Talk in 2008 Matsumoto further stated, "I hope to see Ruby help every programmer in the world to be productive, and to enjoy programming, and to be happy. That is the primary purpose of Ruby language."[13]

Choice of the name "Ruby" [edit]

The name "Ruby" originated during an online chat session between Matsumoto and Keiju Ishitsuka on February 24, 1993, before any code had been written for the language.[14] Initially two names were proposed: "Coral" and "Ruby". Matsumoto chose the latter in a later email to Ishitsuka.[15] Matsumoto later noted a factor in choosing the name "Ruby" - it was the birthstone of one of his colleagues.[16][17]

First publication [edit]

The first public release of Ruby 0.95 was announced on Japanese domestic newsgroups on December 21, 1995.[18][19] Subsequently three more versions of Ruby were released in two days.[14] The release coincided with the launch of the Japanese-language ruby-list mailing list, which was the first mailing list for the new language.

Already present at this stage of development were many of the features familiar in later releases of Ruby, including object-oriented design, classes with inheritance, mixins, iterators, closures, exception handling and garbage collection.[20]

Ruby 1.0 [edit]

Ruby reached version 1.0 on December 25, 1996.[14]

Following the release of Ruby 1.3 in 1999 the first English language mailing list ruby-talk began,[12] which signalled a growing interest in the language outside of Japan. In September 2000, the first English language book Programming Ruby was printed, which was later freely released to the public, further widening the adoption of Ruby amongst English speakers.

This section requires expansion with: history and new features for pre-1.9 versions. (May 2012)

Ruby 1.2 [edit]

Ruby 1.2 was initially released in December 1998.

Ruby 1.4 [edit]

Ruby 1.4 was initially released in August 1999.

Ruby 1.6 [edit]

Ruby 1.6 was initially released in September 2000.

Ruby 1.8 [edit]

Ruby 1.8 was initially released in August 2003, and was stable for a long time. Although deprecated, there is still code based on it. Ruby 1.8 is incompatible with Ruby 1.9.

Ruby on Rails [edit]

Around 2005, interest in the Ruby language surged in tandem with Ruby on Rails, a popular web application framework written in Ruby. Rails is frequently credited with making Ruby "famous".[21]

Ruby 1.9 [edit]

Ruby 1.9 was released in December 2007. Effective with Ruby 1.9.3, released October 31, 2011,[22] Ruby switched from being dual-licensed under the Ruby License and the GPL to being dual-licensed under the Ruby License and the two-clause BSD license.[23] Adoption of 1.9 was slowed by changes from 1.8 which required many popular third party gems to be rewritten.

Ruby 1.9 introduces many significant changes over the 1.8 series.[24] Examples:

Block local variables (variables that are local to the block in which they are declared)

An additional lambda syntax (f = ->(a,b) { puts a + b })

Per-string character encodings are supported

New socket API (IPv6 support)

require\_relative import security

Ruby 2.0 [edit]

Ruby 2.0 added several new features, including:

keyword arguments,functional

a new method – Module#prepend – for extending a class,

a new literal for creating an array of symbols,

new API for the lazy evaluation of Enumerables, and

a new convention of using #to\_h to convert objects to Hashes.[25]

Features [edit]

Thoroughly object-oriented with inheritance, mixins and metaclasses[30]

Dynamic typing and duck typing

Everything is an expression (even statements) and everything is executed imperatively (even declarations)

Succinct and flexible syntax[31] that minimizes syntactic noise and serves as a foundation for domain-specific languages[32]

Dynamic reflection and alteration of objects to facilitate metaprogramming[33]

Lexical closures, iterators and generators, with a unique block syntax[34]

Literal notation for arrays, hashes, regular expressions and symbols

Embedding code in strings (interpolation)

Default arguments

Four levels of variable scope (global, class, instance, and local) denoted by sigils or the lack thereof

Garbage collection

First-class continuations

Strict boolean coercion rules (everything is true except false and nil)

Exception handling

Operator overloading

Built-in support for rational numbers, complex numbers and arbitrary-precision arithmetic

Custom dispatch behavior (through method\_missing and const\_missing)

Native threads and cooperative fibers (fibers are 1.9/YARV feature)

Initial support for Unicode and multiple character encodings (no ICU support)[35]

Native plug-in API in C

Interactive Ruby Shell (a REPL)

Centralized package management through RubyGems

Implemented on all major platforms

Large standard library

Semantics [edit]

Ruby is object-oriented: every value is an object, including classes and instances of types that many other languages designate as primitives (such as integers, booleans, and "null"). Variables always hold references to objects. Every function is a method and methods are always called on an object. Methods defined at the top level scope become members of the Object class. Since this class is an ancestor of every other class, such methods can be called on any object. They are also visible in all scopes, effectively serving as "global" procedures. Ruby supports inheritance with dynamic dispatch, mixins and singleton methods (belonging to, and defined for, a single instance rather than being defined on the class). Though Ruby does not support multiple inheritance, classes can import modules as mixins.

Ruby has been described as a multi-paradigm programming language: it allows procedural programming (defining functions/variables outside classes makes them part of the root, 'self' Object), with object orientation (everything is an object) or functional programming (it has anonymous functions, closures, and continuations; statements all have values, and functions return the last evaluation). It has support for introspection, reflection and metaprogramming, as well as support for interpreter-based[36] threads. Ruby features dynamic typing, and supports parametric polymorphism.

According to the Ruby FAQ,[37] "If you like Perl, you will like Ruby and be right at home with its syntax. If you like Smalltalk, you will like Ruby and be right at home with its semantics. If you like Python, you may or may not be put off by the huge difference in design philosophy between Python and Ruby/Perl."

Syntax [edit]

The syntax of Ruby is broadly similar to that of Perl and Python. Class and method definitions are signaled by keywords. In contrast to Perl, variables are not obligatorily prefixed with a sigil. When used, the sigil changes the semantics of scope of the variable. One difference from C and Perl is that keywords are typically used to define logical code blocks, without braces (i.e., pair of { and }). For practical purposes there is no distinction between expressions and statements.[38] Line breaks are significant and taken as the end of a statement; a semicolon may be equivalently used. Unlike Python, indentation is not significant.

One of the differences of Ruby compared to Python and Perl is that Ruby keeps all of its instance variables completely private to the class and only exposes them through accessor methods (attr\_writer, attr\_reader, etc.). Unlike the "getter" and "setter" methods of other languages like C++ or Java, accessor methods in Ruby are created with a single line of code via metaprogramming. As invocation of these methods does not require the use of parentheses, it is trivial to change an instance variable into a full function, without modifying a single line of code or having to do any refactoring achieving similar functionality to C# and VB.NET property members.

Python's property descriptors are similar, but come with a tradeoff in the development process. If one begins in Python by using a publicly exposed instance variable, and later changes the implementation to use a private instance variable exposed through a property descriptor, code internal to the class may need to be adjusted to use the private variable rather than the public property. Ruby’s design forces all instance variables to be private, but also provides a simple way to declare set and get methods. This is in keeping with the idea that in Ruby, one never directly accesses the internal members of a class from outside of it; rather, one passes a message to the class and receives a response.

See the examples section for samples of code demonstrating Ruby syntax.

Deviations from behavior elsewhere [edit]

Some features which differ notably from languages such as C or Perl:

The language syntax is sensitive to the capitalization of identifiers, in all cases treating capitalized variables as constants. Class and Module names are constants and refer to objects derived from Class and Module.

The sigils $ and @ do not indicate variable data type as in Perl, but rather function as scope resolution operators.

To denote a floating point without a decimal component, one must follow with a zero digit (99.0) or an explicit conversion (99.to\_f). It is insufficient to append a dot (99.) since numbers are susceptible to method syntax.

Boolean evaluation of non-boolean data is strict: 0, "" and [] are all evaluated to true. In C, the expression 0 ? 1 : 0 evaluates to 0 (i.e. false). In Ruby, however, it yields 1, as all numbers evaluate to true; only nil and false evaluate to false. A corollary to this rule is that Ruby methods by convention — for example, regular-expression searches — return numbers, strings, lists, or other non-false values on success, but nil on failure. This convention is also used in Smalltalk, where only the special objects true and false can be used in a boolean expression.

Versions prior to 1.9 use plain integers to represent single characters, much like C. This may cause surprises when slicing strings: "abc"[0] yields 97 (the ASCII code of the first character in the string); to obtain "a" use "abc"[0,1] (a substring of length 1) or "abc"[0].chr.

The notation statement until expression, like Perl but unlike other languages' equivalent statements (e.g. do { statement } while (!(expression)); in C/C++/...), actually never runs the statement if the expression is already true. This is because statement until expression is actually syntactic sugar over until expression; statement; end, the equivalent of which in C/C++ is while (!(expression)) { statement; }, just as statement if expression is equivalent to if (expression) { statement; }. However, the notation begin statement end until expression in Ruby will in fact run the statement once even if the expression is already true, acting similar to the "do-while" of other languages. (Matz has expressed a desire to remove the special behavior of begin statement end until expression,[39] but it still exists as of Ruby 1.9.)

Because constants are references to objects, changing what a constant refers to generates a warning, but modifying the object itself does not. For example, Greeting << " world!" if Greeting == "Hello" does not generate an error or warning. This is similar to final variables in Java or a const pointer to a non-const object in C++, but Ruby provides the functionality to "freeze" an object, unlike Java.

Some features which differ notably from other languages:

The usual operators for conditional expressions, and and or, do not follow the normal rules of precedence: and does not bind tighter than or. Ruby also has expression operators || and && which work as expected.

A list of so-called gotchas may be found in Hal Fulton's book The Ruby Way, 2nd ed (ISBN 0-672-32884-4), Section 1.5. A similar list in the 1st edition pertained to an older version of Ruby (version 1.6), some problems of which have been fixed in the meantime. retry, for example, now works with while, until, and for, as well as iterators.

Interaction [edit]

See also: Interactive Ruby Shell

The Ruby official distribution also includes "irb", an interactive command-line interpreter which can be used to test code quickly. The following code fragment represents a sample session using irb:

$ irb

irb(main):001:0> puts "Hello, World"

Hello, World

=> nil

irb(main):002:0> 1+2

=> 3

Examples [edit]

The following examples can be run in a Ruby shell such as Interactive Ruby Shell or saved in a file and run from the command line by typing ruby <filename>.

Classic Hello world example:

puts "Hello World!"

Some basic Ruby code:

# Everything, including a literal, is an object, so this works:

-199.abs # 199

"ice is nice".length # 11

"ruby is cool.".index("u") # 1

"Nice Day Isn't It?".downcase.split("").uniq.sort.join # " '?acdeinsty"

Conversions:

puts "What's your lucky number?"

number = gets.chomp

puts number.to\_i

output\_number = number.to\_i + 1

puts output\_number.to\_s + ' is a bigger and better favorite number.'

Strings [edit]

There are a variety of ways to define strings in Ruby.

The following assignments are equivalent and support Variable interpolation:

a = "\nThis is a double-quoted string\n"

a = %Q{\nThis is a double-quoted string\n}

a = %{\nThis is a double-quoted string\n}

a = %/\nThis is a double-quoted string\n/

a = <<-BLOCK

This is a double-quoted string

BLOCK

The following assignments are equivalent and produce raw strings:

a = 'This is a single-quoted string'

a = %q{This is a single-quoted string}

Collections [edit]

Constructing and using an array:

a = [1, 'hi', 3.14, 1, 2, [4, 5]]

puts a[2] # 3.14

puts a.[](2) # 3.14

puts a.reverse # [[4, 5], 2, 1, 3.14, 'hi', 1]

puts a.flatten.uniq # [1, 'hi', 3.14, 2, 4, 5]

Constructing and using an associative array (called hashes in Ruby):

hash = Hash.new #equivalent to hash = {}

hash = { :water => 'wet', :fire => 'hot' } # makes the previous line redundant as we are now assigning hash to a new, separate hash object

puts hash[:fire] # Prints: hot

hash.each\_pair do |key, value| # Or: hash.each do |key, value|

puts "#{key} is #{value}"

end

# Prints: water is wet

# fire is hot

hash.delete :water # Deletes water: 'wet'

hash.delete\_if {|key,value| value=='hot'} # Deletes :fire => 'hot'

Blocks and iterators [edit]

The two syntaxes for creating a code block:

{ puts "Hello, World!" } # Note the { braces }

#or

do

puts "Hello, World!"

end

When a code block is created it is always attached to a method as an optional block argument.

Parameter-passing a block to be a closure:

# In an object instance variable (denoted with '@'), remember a block.

def remember(&a\_block)

@block = a\_block

end

# Invoke the above method, giving it a block which takes a name.

remember {|name| puts "Hello, #{name}!"}

# When the time is right (for the object) -- call the closure!

@block.call("Jon")

# => "Hello, Jon!"

Creating an anonymous function:

proc {|arg| print arg}

Proc.new {|arg| print arg}

lambda {|arg| print arg}

# introduced in Ruby 1.9

->(arg) {print arg}

Returning closures from a method:

def create\_set\_and\_get(initial\_value=0) # Note the default value of 0

closure\_value = initial\_value

return Proc.new {|x| closure\_value = x}, Proc.new { closure\_value }

end

setter, getter = create\_set\_and\_get # ie. returns two values

setter.call(21)

getter.call # => 21

#You can also use a parameter variable as a binding for the closure.

#So the above can be rewritten as...

def create\_set\_and\_get(closure\_value=0)

return proc {|x| closure\_value = x } , proc { closure\_value }

end

Yielding the flow of program control to a block which was provided at calling time:

def use\_hello

yield "hello"

end

# Invoke the above method, passing it a block.

use\_hello {|string| puts string} # => 'hello'

Iterating over enumerations and arrays using blocks:

array = [1, 'hi', 3.14]

array.each {|item| puts item }

# => 1

# => 'hi'

# => 3.14

array.each\_index {|index| puts "#{index}: #{array[index]}" }

# => 0: 1

# => 1: 'hi'

# => 2: 3.14

# The following uses a Range

(3..6).each {|num| puts num }

# => 3

# => 4

# => 5

# => 6

A method such as inject() can accept both a parameter and a block. Inject iterates over each member of a list, performing some function on it while retaining an aggregate. This is analogous to the foldl function in functional programming languages. For example:

[1,3,5].inject(10) {|sum, element| sum + element} # => 19

On the first pass, the block receives 10 (the argument to inject) as sum, and 1 (the first element of the array) as element; this returns 11. 11 then becomes sum on the next pass, which is added to 3 to get 14. 14 is then added to 5, to finally return 19.

Blocks work with many built-in methods:

File.open('file.txt', 'w') do |file| # 'w' denotes "write mode".

file.puts 'Wrote some text.'

end # File is automatically closed here

File.readlines('file.txt').each do |line|

puts line

end

# => Wrote some text.

Using an enumeration and a block to square the numbers 1 to 10 (using a range):

(1..10).collect {|x| x\*x} # => [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Or invoke a method on each item

(1..5).collect(&:to\_f) # => [1.0, 2.0, 3.0, 4.0, 5.0]

Classes [edit]

The following code defines a class named Person. In addition to 'initialize', the usual constructor to create new objects, it has two methods: one to override the <=> comparison operator (so Array#sort can sort by age) and the other to override the to\_s method (so Kernel#puts can format its output). Here, "attr\_reader" is an example of metaprogramming in Ruby: "attr\_accessor" defines getter and setter methods of instance variables, "attr\_reader" only getter methods. Also, the last evaluated statement in a method is its return value, allowing the omission of an explicit 'return'.

class Person

attr\_reader :name, :age

def initialize(name, age)

@name, @age = name, age

end

def <=>(person) # Comparison operator for sorting

age <=> person.age

end

def to\_s

"#{name} (#{age})"

end

end

group = [

Person.new("Bob", 33),

Person.new("Chris", 16),

Person.new("Ash", 23)

]

puts group.sort.reverse

The above prints three names in reverse age order:

Bob (33)

Ash (23)

Chris (16)

Person is a constant and is a reference to a Class object.

Open classes [edit]

In Ruby, classes are never closed: you can always add methods to an existing class. This applies to the classes you write as well as the standard, built-in classes. All you have to do is open up a class definition for an existing class, and the new contents you specify will be added to whatever's there. A simple example of adding a new method to the standard library's Time class:

# re-open Ruby's Time class

class Time

def yesterday

self - 86400

end

end

today = Time.now # => Thu Aug 14 16:51:50 +1200 2008

yesterday = today.yesterday # => Wed Aug 13 16:51:50 +1200 2008

Adding methods to previously defined classes is often called monkey-patching. This practice, however, can lead to possible collisions of behavior and subsequent unexpected results, and is a concern for code scalability if performed recklessly.

Exceptions [edit]

An exception is raised with a raise call:

raise

An optional message can be added to the exception:

raise "This is a message"

You can also specify which type of exception you want to raise:

raise ArgumentError, "Illegal arguments!"

Alternatively, you can pass an exception instance to the raise method:

raise ArgumentError.new("Illegal arguments!")

This last construct is useful when you need to raise a custom exception class featuring a constructor which takes more than one argument:

class ParseError < Exception

def initialize input, line, pos

super "Could not parse '#{input}' at line #{line}, position #{pos}"

end

end

raise ParseError.new("Foo", 3, 9)

Exceptions are handled by the rescue clause. Such a clause can catch exceptions which inherit from StandardError. Also supported for use with exceptions are else and ensure

begin

# Do something

rescue

# Handle exception

else

# Do this if no exception was raised

ensure

# Do this whether or not an exception was raised

end

It is a common mistake to attempt to catch all exceptions with a simple rescue clause. To catch all exceptions one must write:

begin

# Do something

rescue Exception

# don't write just rescue -- that only catches StandardError, a subclass of Exception

# Handle exception

end

Or catch particular exceptions:

begin

# ...

rescue RuntimeError

# handling

end

It is also possible to specify that the exception object be made available to the handler clause:

begin

# ...

rescue RuntimeError => e

# handling, possibly involving e, such as "print e.to\_s"

end

Alternatively, the most recent exception is stored in the magic global $!.

You can also catch several exceptions:

begin

# ...

rescue RuntimeError, Timeout::Error => e

# handling, possibly involving e

end

Metaprogramming [edit]

Ruby code can programmatically modify, at runtime, aspects of its own structure that would be fixed in more rigid languages, such as class and method definitions. This sort of metaprogramming can be used to write more concise code and effectively extend the language.

For example, the following Ruby code generates new methods for the built-in String class, based on a list of colors. The methods wrap the contents of the string with an HTML tag styled with the respective color.

COLORS = { black: "000",

red: "f00",

green: "0f0",

yellow: "ff0",

blue: "00f",

magenta: "f0f",

cyan: "0ff",

white: "fff" }

class String

COLORS.each do |color,code|

define\_method "in\_#{color}" do

"<span style=\"color: ##{code}\">#{self}</span>"

end

end

end

The generated methods could then be used like so:

"Hello, World!".in\_blue

==> "<span style=\"color: #00f\">Hello, World!</span>"

To implement the equivalent in many other languages, the programmer would have to write each method (in\_black, in\_red, in\_green, etc.) by hand.

Some other possible uses for Ruby metaprogramming include:

intercepting and modifying method calls

implementing new inheritance models

dynamically generating classes from parameters

automatic object serialization

interactive help and debugging

More examples [edit]

More sample Ruby code is available as algorithms in the following articles:

Exponentiating by squaring

Trabb Pardo-Knuth algorithm

Implementations [edit]

See also: Ruby MRI#Operating systems

Ruby 1.9 has multiple implementations:

The official Ruby interpreter often referred to as the Matz's Ruby Interpreter or MRI. This implementation is written in C and uses its own Ruby-specific virtual machine,

JRuby, a Java implementation that runs on the Java virtual machine,

Rubinius, C++ bytecode VM that uses LLVM to compile to machine code at runtime. The bytecode compiler and most core classes are written in pure Ruby.

Other Ruby implementations:

MagLev (software), a Smalltalk implementation on VMware’s GemStone VM.

MacRuby, an OS X implementation on the Objective-C runtime,

Cardinal, an implementation for the Parrot virtual machine,

IronRuby, an implementation on the .NET Framework.

The maturity of Ruby implementations tends to be measured by their ability to run the Ruby on Rails (Rails) framework, because it is a complex framework to implement, and it uses many Ruby-specific features. The point when a particular implementation achieves this goal is called The Rails singularity. The reference implementation (MRI), JRuby, and Rubinius[40] are all able to run Rails unmodified in a production environment. IronRuby[41][42] is starting to be able to run Rails test cases, but is still far from being production-ready.

Ruby is available on many operating systems, such as Linux, Mac OS X, Microsoft Windows, Windows Phone,[43] Windows CE and most flavors of Unix.

Ruby 1.9 has recently been ported onto Symbian OS 9.x.[44]

Repositories and libraries [edit]

RubyGems is Ruby's package manager. A Ruby package is called a 'gem' and can easily be installed via the command line. There are over 50,000 Ruby gems hosted on RubyGems.org.

Many new and existing Ruby libraries are hosted on GitHub, a service that offers version control repository hosting for Git.

See also [edit]

Free software portal

Computer programming portal

Comparison of programming languages

Ruby MRI, the reference C implementation

JRubyRuby 2.0 is intended to be fully bac