



Ruby

A PROGRAMMER'S BEST FRIEND



By: Mohammad Tarek



/bntarek



/in/mohammadtarek

Ruby

Ruby is a pure Object-Oriented language developed by Yukihiro Matsumoto (also known as Matz in the Ruby community) in the mid 1990's in Japan.

Everything in Ruby is an object except the blocks but there are replacements too for it i.e procs and lambda.

Ruby supports mostly all the platforms like Windows, Mac, Linux.

Ruby

Ruby is based on many other languages like Perl, Lisp, Smalltalk, Eiffel and Ada.

It is an interpreted scripting language which means most of its implementations execute instructions directly and freely, without previously compiling a program into machine-language instructions.

Ruby programmers also have access to the powerful RubyGems (RubyGems provides a standard format for Ruby programs and libraries).

Ruby

Ruby is a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.

Ruby is simple in appearance, but is very complex inside, just like our human body.” —Matz, creator of the Ruby programming language

Ruby | Data Types

Data types in Ruby represents different types of data like text, string, numbers, etc. All data types are based on classes because it is a pure Object-Oriented language.

Ruby | Data Types

- Numbers
- Boolean
- Strings
- Arrays
- Hashes
- Symbols

Ruby | Duck Typing

Duck Typing is based on the well known Duck Test

“When I see a **bird** that
walks like a **duck** and
swims like a **duck** and
quacks like a **duck**,
I call that **bird** a **duck**.”

Ruby | Duck Typing

That we can translate In Ruby as following

“If an object quacks like a duck (or acts like an array), just go ahead and treat it as a duck (or an array).”

Ruby | Numbers

Generally a number is defined as a series of digits, using a dot as a decimal mark. Optionally the user can use the underscore as a separator. There are different kinds of numbers like integers and float. Ruby can handle both Integers and floating point numbers. According to their size, there are two types of integers, one is Bignum and second is Fixnum.

Ruby | Numbers | Public Instance Methods

even? → true or false

Returns true if int is an even number.

```
2.even?      #=> true
```

```
1.even?      #=> false
```

Ruby | Numbers | Public Instance Methods

odd? → true or false

Returns true if int is an odd number.

```
1.odd?           #=> true
```

```
2.odd?           #=> false
```

Ruby | Numbers | Public Instance Methods

`ceil([ndigits])` → integer or float

Returns the smallest number than or equal to int in decimal digits (default 0 digits).

```
1.2.ceil           #=> 2
1.6.ceil           #=> 2
1.2.ceil(2)        #=> 1.2
1.2.ceil(-1)       #=> 10
```

Ruby | Numbers | Public Instance Methods

floor([ndigits]) → integer or float

Returns the largest number less than or equal to int in decimal digits (default 0 digits).

```
1.2.floor           #=> 1
1.6.floor           #=> 1
1.2.floor(2)        #=> 1.2
1.2.floor(-1)       #=> 0
```

Ruby | Numbers | Public Instance Methods

`round([ndigits])` → integer or float

Rounds int to a given precision in decimal digits (default 0 digits).

```
1.2.round           #=> 1
1.6.round           #=> 2
1.2.round(2)        #=> 1.2
1.2.round(-1)       #=> 0
```

Ruby | Numbers | Public Instance Methods

abs → integer

Returns the absolute value of int.

```
-12345.abs      #=> 12345  
12345.abs       #=> 12345  
-123456789.abs  #=> 123456789
```

Ruby | Numbers | Public Instance Methods

to_s(base=10) → string

Returns a string containing the representation of int radix base (between 2 and 36).

```
12345.to_s      #=> "12345"  
123.to_s(2)     #=> "1111011"  
12345.to_s(8)   #=> "30071"  
12345.to_s(10)  #=> "12345"
```


Ruby | Boolean

Boolean data type represents only one bit of information either true or false.

```
true is True!
```

```
nil is False!
```

```
0 is True!
```

Ruby | Strings

A string is a group of letters that represent a sentence or a word. Strings are defined by enclosing a text within a single (") or double (") quotes. You can use both double quotes and single quotes to create strings.

Strings are objects of class String. Double-quoted strings allow substitution and backslash notation but single-quoted strings doesn't allow substitution and allow backslash notation only for \ and \'.

Ruby | Strings | Public Instance Methods

str + other_str → new_str

Returns a new String containing other_str concatenated to str.

```
"Hello from " + "eSpace"
```

```
#=> "Hello from eSpace"
```

Ruby | Strings | Public Instance Methods

str * integer → new_str

Returns a new String containing integer copies of the receiver. integer must be greater than or equal to 0.

```
"Ho! " * 3    #=> "Ho! Ho! Ho! "
```

```
"Ho! " * 0    #=> ""
```

Ruby | Strings | Public Instance Methods

capitalize → **new_str**

Returns a copy of str with the first character converted to uppercase and the remainder to lowercase.

```
"hello".capitalize      #=> "Hello"  
"HELLO".capitalize     #=> "Hello"  
"123ABC".capitalize    #=> "123abc"
```

Ruby | Strings | Public Instance Methods

upcase → **new_str**

Returns a copy of str with all lowercase letters replaced with their uppercase counterparts.

```
"hEllo".upcase      #=> "HELLO"
```

Ruby | Strings | Public Instance Methods

downcase → **new_str**

Returns a copy of str with all uppercase letters replaced with their lowercase counterparts.

```
"hEllo".downcase      #=> "hello"
```

Ruby | Strings | Public Instance Methods

empty? → true or false

Returns true if str has a length of zero.

```
"hello".empty?      #=> false
" ".empty?          #=> false
"".empty?           #=> true
```


Ruby | Strings | Public Instance Methods

strip → **new_str**

Returns a copy of str with leading and trailing whitespace removed.

```
"  hello  ".strip      #=> "hello"  
"\tbye\n".strip        #=> "bye"  
"\n\v\f\t".strip      #=> ""
```

Ruby | Strings | Public Instance Methods

reverse → **new_str**

Returns a new string with the characters from str in reverse order.

```
"Stressed".reverse
```

```
#=> "desserts"
```

Ruby | Strings | Public Instance Methods

length → **integer**

Returns the character length of str.

```
"hello".length
```

```
#=> 5
```

Ruby | 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 within square bracket. The position of elements in an array starts with 0. A trailing comma is ignored.

Ruby | Arrays | Common Uses

```
bookshelf = Array.new
bookshelf = [] # shorthand for Array.new
bookshelf.push("Agile Development")
bookshelf << "Ruby on Rails"
Bookshelf[1]      #=> "Ruby on Rails"
Bookshelf.first   #=> "Agile Development"
Bookshelf.last    #=> "Ruby on Rails"
```

Ruby | Arrays | Public Instance Methods

ary & other_ary → new_ary

Returns a new array containing unique elements common to the two arrays. The order is preserved from the original array.

```
[ 'a', 'b', 'b', 'z' ] & [ 'a', 'b', 'c' ]
```

```
#=> [ 'a', 'b' ]
```

Ruby | Arrays | Public Instance Methods

ary + other_ary → new_ary

Returns a new array built by concatenating the two arrays together to produce a third array.

```
[ 1, 2, 3 ] + [ 4, 5 ]      #=> [ 1, 2, 3, 4, 5 ]
```

Ruby | Arrays | Public Instance Methods

ary - other_ary → new_ary

Returns a new array that is a copy of the original array, removing any items that also appear in other_ary. The order is preserved from the original array.

```
[ 1, 1, 2, 2, 3, 3, 4, 5 ] - [ 1, 2, 4 ]
```

```
#=> [ 3, 3, 5 ]
```


Ruby | Arrays | Public Instance Methods

include?(object) → true or false

Returns true if the given object is present in self (that is, if any element == object), otherwise returns false.

```
a = [ "a", "b", "c" ]  
a.include?("b")           #=> true  
a.include?("z")           #=> false
```

Ruby | Arrays | Public Instance Methods

count(obj) → int

Returns the number of elements.

```
ary = [1, 2, 4, 2]
```

```
ary.count                      #=> 4
```

```
ary.count(2)                   #=> 2
```

Ruby | Arrays | Public Instance Methods

max(n) → obj

Returns the object in ary with the maximum value.

```
a = %w(albatross dog horse)
a.max                #=> "horse"
a.max(2)             #=> [ "horse", "dog" ]
```

Ruby | Arrays | Public Instance Methods

min(n) → array

Returns the object in ary with the minimum value.

```
a = %w(albatross dog horse)
```

```
a.min                               #=> "albatross"
```

```
a.min(2)                           #=> [ "albatross", "dog" ]
```

Ruby | Arrays | Public Instance Methods

pop(n) → new_ary

Removes the last n elements from self and returns it, or nil if the array is empty.

```
a = [ "a", "b", "c", "d" ]  
  
a.pop           #=> "d"  
  
a.pop(2)        #=> ["b", "c"]  
  
a               #=> ["a"]
```

Ruby | Arrays | Public Instance Methods

`rotate(count=1) → new_ary`

Returns a new array by rotating self so that the element at count is the first element of the new array.

Ruby | Arrays | Public Instance Methods

rotate(count=1) → new_ary

```
a = [ "a", "b", "c", "d" ]
```

```
a.rotate          #=> [ "b", "c", "d", "a" ]
```

```
a                #=> [ "a", "b", "c", "d" ]
```

```
a.rotate(2)       #=> [ "c", "d", "a", "b" ]
```

```
a.rotate(-3)      #=> [ "b", "c", "d", "a" ]
```

Ruby | Hashes

A hash assigns its values to its key. Value to a key is assigned by `=>` sign. A key pair is separated with a comma between them and all the pairs are enclosed within curly braces.

A hash in Ruby is like an object literal in JavaScript or an associative array in PHP. They're made similarly to arrays. A trailing comma is ignored.

Ruby | Hashes | Common Uses

```
hash_one = Hash.new  
hash_two = {} # shorthand for Hash.new  
hash_three = {"a" => 1, "b" => 2, "c" =>  
3}  
#=> {"a"=>1, "b"=>2, "c"=>3}  
hash_four = {:a => 1, :b => 2, :c => 3}  
#=> {:a=>1, :b=>2, :c=>3}  
hash_five = {a: 1, b: 2, c: 3}  
#=> {:a=>1, :b=>2, :c=>3}
```

Ruby | Hashes | Common Uses

```
grades = { "a" => 10, "b" => 6 }  
  
puts grades["a"]           #=> 10  
  
grades["a"] = 9  
  
puts grades["a"]           #=> 9
```

Ruby | Hashes | Public Instance Methods

keys → **array**

Returns a new array populated with the keys from this hash.

```
h = { "a" => 100, "b" => 200, "c" => 300, "d" => 400 }
```

```
h.keys    #=> ["a", "b", "c", "d"]
```

Ruby | Hashes | Public Instance Methods

values → **array**

Returns a new array populated with the values from hsh.

```
h = { "a" => 100, "b" => 200, "c" => 300 }
```

```
h.values    #=> [100, 200, 300]
```

Ruby | Hashes | Public Instance Methods

has_key?(key) → true or false

Returns true if the given key is present in hsh.

```
h = { "a" => 100, "b" => 200 }
```

```
h.has_key?("a")    #=> true
```

```
h.has_key?("z")    #=> false
```

Ruby | Hashes | Public Instance Methods

has_value?(value) → true or false

Returns true if the given value is present for some key in hsh.

```
h = { "a" => 100, "b" => 200 }
```

```
h.value?(100)    #=> true
```

```
h.value?(999)    #=> false
```

Ruby | Hashes | Public Instance Methods

to_a → array

Converts hsh to a nested array of [key, value] arrays.

```
h = { "c" => 300, "a" => 100, "d" => 400, "c" => 300 }
```

```
h.to_a    #=> [ ["c", 300], ["a", 100], ["d", 400] ]
```

Ruby | Symbols

Symbols are light-weight strings. A symbol is preceded by a colon (:). They are used instead of strings because they can take up much less memory. Symbols have better performance.

Ruby | Variable Types

- Local variables
- Instance variables
- Class variables
- Global variables
- Constants
- Pseudo Variables
- Predefined Constants

Ruby | Local Variables

A variable whose name begins with a lowercase letter (a-z) or underscore (_) is a local variable or method invocation.

A local variable is only accessible from within the block of its initialization.

```
foobar = "local variable"  
_foobar = "local variable"
```

Ruby | Instance Variables

A variable whose name begins with '@' is an instance variable of `self`.

An instance variable belongs to the object itself.

Uninitialized instance variables have a value of `nil`.

```
@foobar = "instance variable"
```

Ruby | Class Variables

A class variable is shared by all instances of a class and begins with '@@'.

A class variable is shared by all the descendants of the class.

Referencing an uninitialized class variable produces an error.

```
@@foobar = "class variable"
```

Ruby | Global Variables

A variable whose name begins with '\$' has a global scope; meaning it can be accessed from anywhere within the program during runtime.

Uninitialized global variables have a value of `nil`.

```
$foobar = "global variable"
```

Ruby | Constants

A variable whose name begins with an uppercase letter (A-Z) is a constant.

Reassigning a constant value after its initialization generates a warning.

Referencing an uninitialized constant raises the NameError exception.

```
FOOBAR = "constant"
```

Ruby | Pseudo Variables

- `self`

Execution context of the current method, which could refer to an instance, class, or module.

- `nil`

The sole-instance of the NilClass class. Expresses nothing.

- `true`

The sole-instance of the TrueClass class. Expresses true.

- `false`

The sole-instance of the FalseClass class. Expresses false.

Ruby | Predefined Constants

- `__FILE__`

The name of the current source file.

- `__LINE__`

The current line number in the source file.

Ruby | Conditional Statements

```
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
```

Ruby | Conditional Statements

```
x = 1
unless x >= 2
  puts "x is less than 2"
else
  puts "x is greater than 2"
end
```

Ruby | Conditional Modifiers

```
$var = true  
print "1 -- Value is set\n" if $var  
print "2 -- Value is set\n" unless $var  
  
$var = false  
print "3 -- Value is set\n" if $var  
print "4 -- Value is set\n" unless $var
```

Ruby | Loops

```
num = 1
while num <= 10
  puts num
  num += 1
end
```

Ruby | Loops

```
for num in 1...10  
  puts num  
end
```

Ruby | Iterators

```
[1, 2, 3, 4, 5].each do |num|  
  puts num  
end
```

Ruby | Classes and Objects

```
class Vehicle
```

```
end
```

```
vehicle = Vehicle.new
```

Ruby | Constructor Method

```
class Vehicle
  def initialize(engine_type, seats_count, max_velocity)
    @engine_type = engine_type
    @seats_count = seats_count
    @max_velocity = max_velocity
  end
end
```


Ruby | Methods

```
class Vehicle
  def initialize(engine_type, seats_count, max_velocity)
    @engine_type = engine_type
    @seats_count = seats_count
    @max_velocity = max_velocity
  end

  def engine_type
    @engine_type
  end

  def engine_type=(engine_type)
    @engine_type = engine_type
  end
end
```

Ruby | Blocks

A block is the same thing as a method, but it does not belong to an object.

Blocks are called closures in other programming languages.

Ruby | Blocks

There are some important points about Blocks in Ruby:

- Block can accept arguments and returns a value.
- Block does not have their own name.
- Block consist of chunks of code.

Ruby | Blocks

There are some important points about Blocks in Ruby:

- A block is always invoked with a function or can say passed to a method call.
- To call a block within a method with a value, yield statement is used.
- Blocks can be called just like methods from inside the method that it is passed to.

Ruby | Blocks | Common Uses

Form 1: recommended for single line blocks

```
[1, 2, 3].each { |num| puts num }
```

Form 2: recommended for multi-line blocks

```
[1, 2, 3].each do |num|
```

```
  puts num
```

```
end
```

Ruby | Blocks | Yield Keyword

```
def print_once
```

```
  Yield
```

```
end
```

```
print_once { puts "Block is being run" }
```

```
#=> Block is being run
```

Ruby | Blocks | Yield Keyword

```
def one_two_three  
  yield 1  
  yield 2  
  yield 3  
end
```

```
one_two_three { |number| puts number * 10 }
```

```
#=> 10
```

```
#=> 20
```

```
#=> 30
```

Ruby | Blocks | Explicit Blocks

```
def explicit_block(&block)
  block.call # same as yield
end

explicit_block { puts "Explicit block called" }
```


Ruby | Blocks | Implicit Blocks

```
def do_something_with_block
  return "No block given" unless block_given?
  Yield
end
```

Ruby | Procs

A Proc is a way to define a block & its parameters with some special syntax.

You can save this Proc into a variable for later use.

```
proc = Proc.new { puts "Hello World" }  
  
# The body of the Proc object gets executed when called  
proc.call  
  
#=> Hello World
```

Ruby | Lambdas

A Lambda is a way to define a block & its parameters with some special syntax.

You can save this Lambda into a variable for later use.

```
lam = lambda { puts "Hello World" }  
  
lam = -> { puts "Hello World" }  
  
lam.call  
  
#=> Hello World
```

Ruby | Proc VS Lambdas

Procs and Lambdas are both Proc objects.

```
proc = Proc.new { puts "Hello world" }  
lam = lambda { puts "Hello World" }  
  
proc.class #=> Proc  
lam.class  #=> Proc
```

Ruby | Proc VS Lambdas

However, lambdas are a different ‘flavor’ of procs. This slight difference is shown when returning the objects.

```
proc
```

```
#=> #<Proc:0x007f96b1032d30@(irb):75>
```

```
lam
```

```
#=> #<Proc:0x007f96b1b41938@(irb):76 (lambda)>
```

Ruby | Proc VS Lambdas

Procs don't care if they are passed the wrong number of arguments

```
proc = Proc.new { |x| puts x }  
# creates a proc that takes 1 argument  
  
proc.call(2)  
# prints out 2  
  
proc.call  
# returns nil  
  
proc.call(1,2,3)  
# prints out 1 and forgets about the extra arguments
```

Ruby | Proc VS Lambdas

Lambdas check the number of arguments

```
lam = lambda { |x| puts x }  
# creates a lambda that takes 1 argument  
  
lam.call(2)  
# prints out 2  
  
lam.call  
# ArgumentError: wrong number of arguments (0 for 1)  
  
lam.call(1,2,3)  
# ArgumentError: wrong number of arguments (3 for 1)
```

Ruby | attr_reader

```
class Vehicle
  attr_reader :engine_type

  def initialize(engine_type, seats_count, max_velocity)
    @engine_type = engine_type
    @seats_count = seats_count
    @max_velocity = max_velocity
  end

  def engine_type=(engine_type)
    @engine_type = engine_type
  end
end
```


Ruby | attr_writer

```
class Vehicle

  attr_reader :engine_type
  attr_writer :engine_type

  def initialize(engine_type, seats_count, max_velocity)
    @engine_type = engine_type
    @seats_count = seats_count
    @max_velocity = max_velocity
  end
end
```

Ruby | attr_accessor

```
class Vehicle
  attr_accessor :engine_type

  def initialize(engine_type, seats_count, max_velocity)
    @engine_type = engine_type
    @seats_count = seats_count
    @max_velocity = max_velocity
  end
end
```

Ruby | Class Inheritance

```
class Vehicle
  attr_accessor :engine_type
  def initialize(engine_type, seats_count, max_velocity)
    @engine_type= engine_type
    @seats_count= seats_count
    @max_velocity= max_velocity
  end
end

class ElectricCar < Vehicle
end
```

Ruby | Module

```
class Vehicle
  include Performance

  def initialize(engine_type, seats_count, max_velocity)
    @engine_type = engine_type
    @seats_count = seats_count
    @max_velocity = max_velocity
  end
end

module Performance
  def horsepower
    puts "Calculating your vehicle's horsepower.."
  end
end
```

Any Questions?

Many Thanks