Ruby Glossary

Programming reference for Ruby

Arrays

An array is a Ruby data type that holds an ordered collection of values, which can be any type of object including other arrays.

Creating arrays

Ruby arrays can be created with either literal notation or the `Array.new` constructor.

Syntax

```
# Array.new constructor
variable = Array.new([repeat], [item])
```

Example

```
empty_arr = Array.new
=> []

matzes = Array.new(3, "Matz!")
=> ["Matz!", "Matz!", "Matz!"]
```

Syntax

```
# Array.new copy constructor
variable = Array.new(some_array)
```

Example

```
more_matzes = Array.new(matzes)
=> ["Matz!", "Matz!", "Matz!"]
```

Syntax

```
# Array literal notation
variable = []
other_variable = [item1, item2, ..., itemN]
```

Example

```
bobbies = ["Bobby!", "Bobby!", "Backend Bobby!"]
=> ["Bobby!", "Bobby!", "Backend Bobby!"]
```

Read more

- http://ruby-doc.org/core-2.0/Array.html#method-c-new

Array.each

You can iterate over the elements in an array using `Array.each`, which takes a block.

Syntax

```
array.each do |arg|
  # Do something to each element, referenced as arg
end

#or

array.each { |arg|
  # Do something to each element, referenced as arg
}
```

Example

```
["Ryan", "Zach"].each do |person|
  "#{person} is such a great guy!"
end

Ryan is such a great guy!
Zach is such a great guy!
```

Read more

- http://ruby-doc.org/core-2.0/Array.html#method-i-each

Array.flatten

`.flatten` returns a one-dimensional version of a multi-dimensional array. It does not overwrite the array with the new array.

Example

```
[[1,2,3], [4,5,6], 7, [[8,9], 10]].flatten
=> [1,2,3,4,5,6,7,8,9,10]
```

Read more

- http://ruby-doc.org/core-2.0/Array.html#method-i-flatten

Array.uniq

You can remove duplicates from an array using `Array.uniq`.

Example

```
[1,1,1,2,3,4,3,3].uniq
=> [1,2,3,4]
```

Read more

- http://ruby-doc.org/core-2.0/Array.html#method-i-uniq

Blocks

A block is a chunk of code that lives inside a control statement, loop, method definition, or method call. It returns the value of its last line. In Ruby, blocks can be created two ways: with braces or with a do/end statement.

Syntax

```
# Blocks that span only one line usually use the braces form
objs.method { |obj| do_something }
```

Example

```
[1,2,3,4].each { |number| puts number }

1
2
3
4
```

Syntax

```
# Blocks that span multiple lines usually use the do/end form
objs.method do |obj|
  # do first line
  # do second line
  # ...
  # do nth line
end
```

Example

```
[1,2,3,4].each do |number|
  puts "You know what number I love?"
  puts "I love #{number}!"
end

You know what number I love?
I love 1!
You know what number I love?
I love 2!
You know what number I love?
I love 3!
You know what number I love?
I love 4!
```

Booleans

In Ruby, there are two boolean values: `true` and `false`.

Syntax

```
true
=> true

false
=> false
```

Logical Operators

Logical operators are used to compare to boolean values. Ruby has 6 operators to compare boolean values: `and`, `or`, `not`, `&&`, `||`, and `not`. `and` and `&&`, `or` and `||`, and `not` and `!` have the same functionality but the verbiage operators (`and`, `or` and `not`) have lower precedence than the symbolic operators (`&&`, `||` and `!`).

Syntax

```
// returns true if both boolean1 and boolean2 are true
boolean1 && boolean2
boolean1 and boolean2

// returns true if either boolean1 or boolean2 are true
boolean1 || boolean2
boolean1 or boolean2

// returns the opposite of boolean
!boolean
not boolean
```

Example

```
true && true
=> true
true && false
=> false
false and true
=> false
false and false
=> false

true || true
=> true
true || false
=> true
false or true
=> true
false or false
=> false

!true
=> false
not false
```

```
=> true
```

Read more

- http://phrogz.net/ProgrammingRuby/language.html#table%5F18.4

Comparison Operators

Comparison operators are used to test the relationship between two objects. The equality (==) and inequality (!=) operators can be used on almost any type of value where the other operators are used for numeric comparisons.

Syntax

```
x == y // returns true if two things are equal
x != y // returns true if two things are not equal
x <= y // returns true if x is less than or equal to y
x >= y // returns true if x is greater than or equal to y
x < y // returns true if x is less than y
x > y // returns true if x is greater than y
```

Example

```
5 == 5
=> true
5 != 5
=> false
2 <= 2
=> true
2 >= 3
=> false
1 < 2
=> true
1 > 2
=> false
```

Comments

Adding human-readable comments to your programs is a good idea to help others who read your code understand what it does. However in Ruby, it is common to not write many comments, since the language is so human-readable already. It's usually very easy to quickly understand what a good piece of Ruby code does.

Single-line

Single line comments are great for quick notes, reminders, or sharing trivial information.

Syntax

```
# comment text
```

Example

```
# This is a single line comment.
```

Multi-line

You can span a comment multiple lines, although Ruby programmers rarely use this. They are useful when making notes for documentation.

Syntax

```
=begin
comment line
comment line
=end
```

Example

```
=begin
This is a comment
that spans multiple lines.
=end
```

Hashes

Hashes are collections of key-value pairs. Like arrays, they have values associated with indices, but in the case of hashes, the indices are called "keys." Keys can be anything that's hashable, such as integers, strings, or symbols, but they must be unique for the hash they belong. The values to which keys refer can be any Ruby object.

Creating Standard Hashes

There are several ways to create hashes in Ruby. The common most two are the new constructor method and its literal notation. It is also considered a best practice to use symbols as keys. The following are valid in all versions of Ruby.

Syntax

```
# Hash.new constructor
my_hash = Hash.new([default_value])
```

Example

```
empty_hash = Hash.new
=> {}

my_hash = Hash.new("The Default")
my_hash["random_key"]
=> "The Default"
```

Syntax

```
# Hash literal notation
my_hash = {
  "key1" => value1,
```

```
  :key2  => value2,
  3 => value 3
}
```

Example

```
my_hash = {
  :a => "Artur",
  :l => "Linda",
  :r => "Ryan",
  :z => "Zach"
}
=> {:a => "Artur", :l => "Linda", :r => "Ryan", :z => "Zach"}
```

Creating Shorthand Hashes

As of Ruby 1.9, there is now a shorthand method for writing hashes that's a lot easier to write. Rather than specifying a symbol then using the hash rockets to define key value pairs, you can now just put the key followed by a colon then the value. The keys get translated into symbols.

Syntax

```
my_hash = {
  key1: value1,
  key2: value2
}
```

Example

```
my_hash = {
  name: "Artur",
  age:  21
}
=> { :name => "Artur", :age => 21 }
```

if, unless, elsif and else

if

Ruby includes an if statement that can be used to manage a program's control flow. The statement takes a boolean expression and executes certain code only if the boolean expression evaluates to true.

Syntax

```
if boolean_expression
  #do something here
end
```

Example

```
if true
  puts "I get printed!"
end
```

```
I get printed!
```

unless

This is the opposite of an if statement. The statement takes a boolean expression and executes certain code only if the boolean expression evaluates to false.

Syntax

```
unless boolean_expression
  #do something here
end
```

Example

```
unless false
  puts "I get printed!"
end

I get printed!
```

elsif

A conditional statement used to manage a program's control flow. The statement must be paired with an if or unless block and takes a boolean expression. It runs certain code only if the previous conditional statements do not run and its boolean expression evaluates to true. it is equivalent to writing an else statement that has an if statement in its block.

Syntax

```
if boolean_expression
  #do something
elsif boolean_expression_2
  #do something different
else
  #do something else
```

Example

```
x = 5
if x > 5
  print "I am big!"
elsif x == 5
  print "I am medium!"
else
  print "I am small!"
end

I am medium!
```

else

A conditional statement used to manage a program's control flow. The statement must be paired with an

if or unless block and takes no arguments. It runs certain code only if the previous conditional statements do not run.

Syntax

```
if boolean_expression
  #do something
else
  #do something else
end
```

Example

```
x = 5
if x > 5
  print "I am big!"
else
  print "I am small!"
end

I am small!
```

Loops

While Loops

Ruby includes a while loop that will execute a block of code as long as its condition is true. When the condition becomes false, the code after the end of the loop will be executed.

Syntax

```
while condition_is_true
  # do something
end
```

Example

```
i = 1
while i < 5
  puts "#{i} is less than 5!"
  i += 1
end
puts "Done!"

1 is less than 5!
2 is less than 5!
3 is less than 5!
4 is less than 5!
Done!
```

Until Loops

Ruby includes an until loop that will execute a block of code as long as its condition is false. When the condition becomes true, the code after the end of the loop will be executed.

Syntax

```
until condition_is_false
  # do something
end
```

Example

```
counter = 3
until counter <= 0
  puts counter
  counter -= 1
end
puts "Blast off!"
```

```
3
2
1
Blast off!
```

For Loops

The for loop is used to iterate an object. The Ruby .each method is preferred over the for loop because the for loop does not create a new scope for the object whereas the .each method does. The for loop is rare in Ruby.

Syntax

```
for iterator in iterable_object
  # do something
end
```

Example

```
for number in (0..5)
  puts number
end
```

```
0
1
2
3
4
5
```

Example

```
my_array = ["Matz", "chunky", "bacon"]
for item in my_array
  puts item
end
```

```
Matz
chunky
bacon
```

Math

Basic Arithmetic

Your typical addition, subtraction, multiplication, division, and exponentiation all exist in Ruby and look very similar to your typical use cases in algebra. Note that order of operations holds.

Syntax

```
x + y  #addition
x - y  #subtraction
x * y  #multiplication
x / y  #division
x ** y #exponentiation
```

Example

```
40 + 2
=> 42

100 - 17
=> 83

9 * 10
=> 90

9 / 3
=> 3

2**3
=> 8
```

Division

Division is a tricky situation. If you divide two integers, the outcome will be an integer regardless of remainder (see modulus below for more information). However, if you divide an integer with a floating point number or two floating point numbers, the outcome will account for the remainder.

Example

```
10 / 3
=> 3

10.0 / 3
=> 3.3333333333333

10 / 3.0
=> 3.3333333333333

10.0 / 3.0
=> 3.3333333333333
```

.divmod

If you want to do integer division and retrieve both the quotient and the remainder in one call, then you want to use divmod. You can use divmod on a numeric type and it will return an array with the quotient and the remainder, respectively.

Syntax

```
a.divmod(b)
```

Example

```
10.divmod(3)
=> [3, 1]
```

Modulus Division

If you would like the remainder from an integer division problem, you would use the modulus operator to retrieve the value.

Syntax

```
x % y
```

Example

```
10 % 3
=> 1
```

.floor

Returns the largest integer less than or equal to a number.

Syntax

```
expression.floor
```

Example

```
9.99.floor
=> 9

(1 + 0.5).floor
=> 1
```

.ceil

Returns the smallest integer greater than or equal to a number.

Syntax

```
expression.ceil
```

Example

```
45.4.ceil
=> 46

(4 - 1.9).ceil
=> 3
```

PI

Returns the ratio of the circumference of a circle to its diameter, approximately 3.14159 or in better terms, the value of PI (π). Note in syntax , we do not put `()` at the end of `Math::PI` because `Math::PI` is not a function but rather a constant.

Syntax

```
Math::PI
=> 3.14159265358979
```

.sqrt

Returns the square root of a number.

Syntax

```
Math.sqrt(expression)
```

Example

```
Math.sqrt(100)
=> 10.0

Math.sqrt(5+4)
=> 3.0

Math.sqrt(Math.sqrt(122+22) + Math.sqrt(16))
=> 4.0
```

Methods

A Ruby method is used to create parameterized, reusable code. Ruby methods can be created using the syntax:

Syntax

```
def method_name(arguments)
  # Code to be executed
end
```

Example

```
def sum(x,y)
  x + y
end

sum(13, 379)
```

```
=> 392
```

puts vs. print

The puts (short for "put string") and print commands are both used to display the results of evaluating Ruby code. The primary difference between them is that puts adds a newline after executing, and print does not.

Syntax

```
print some_string
puts some_string
```

Example

```
3.times { print "Hello!" }
Hello!Hello!Hello!

3.times { puts "Hello!" }
Hello!
Hello!
Hello!
```

Strings

Strings are used for storing and manipulating text in Ruby. Strings are written between quotation marks. Both single (') and double (") quotes are supported, but quotes at each end of a single string must match (no "strings' or 'strings")!

Syntax

```
single_quotes = 'some text goes here'
double_quotes = "some text goes here"
```

Example

```
my_name = "Eric"
that_computer = "Eric's Computer" #this syntax is allowed
```

Switch statement

Acts like a big if / else if / else chain. Checks a value against a list of cases, and executes the first case that is true. If it does not find a match, it attempts the default case designated by an else statement. If there is not a default case, then it exits the statement. Unlike languages like JavaScript, Ruby switch statements have no fall through and automatically break. Instead, cases can be comma delimited.

Syntax

```
case value
when expression1
  #do something
when expression2
  #do something
...
```

```
when expressionN
  #do something
else
  #do default case
end
```

Example

```
a = ["4"]
case a
when 1..4, 5
  puts "It's between 1 and 5"
when 6
  puts "It's 6"
when String
  puts "You passed a string"
else
  puts "You gave me #{a} -- I have no idea what to do with that."
end

=> You gave me 4 -- I have no idea what to do with that.
```

Symbols

In Ruby, a symbol is simply a name used in your program. One of the main uses for Ruby symbols is hash keys, especially if you would otherwise use the same string as a hash key over and over. Ruby will create an (almost) unlimited number of string instances for all your hash keys, but will only keep one copy of a symbol in memory at a time. This can really save memory for your programs in the long run.

Syntax

```
:symbol
```

Example

```
fox1 = :fox
fox2 = :fox

fox1.object_id
=> 430488
fox2.object_id
=> 430488
```

Ternary Operator

This is a shorthand statement for a simple if...else statement. It is a useful tool in situations where you have an extremely simple if...else statement where you are trying to assign a value to a variable.

Syntax

```
boolean_expression ? true_expression : false_expression
```

Example

```
grade = 88
status = grade >= 70 ? "pass" : "fail"
```

```
=> pass
```

times and each Methods

.each

.each is a built in iterator function in Ruby. It loops through each item in a list, hash, or other iterable object allowing you to perform operations on that value. The block of an .each statement creates a new scope for your variable so you don't accidentally modify the original value.

Syntax

```
iterable_obj.each do |value_of_item|
  # do something
end
```

Example

```
one_to_ten = (1..10).to_a
one_to_ten.each do |num|
  print (num**2).to_s + " "
end

1 4 9 16 25 36 49 64 81 100
```

.times

.times is a built in iterator function in Ruby. It performs an action a given number of times.

Syntax

```
num_of_times.times do
  # do something
end
```

Example

```
3.times do
  puts "I'm in the loop!"
end
puts "I'm out the loop!"
```

Variables

Variables are assigned values using the = operator, which is not to be confused with the == sign used for testing equality. A variable can hold almost any type of value including numbers, strings, arrays, and hashes.

Assignment

Syntax

```
variable_name = value
```

Example

```
name = "Artur"
=> "Artur"

name_copy = name
=> "Artur"

age = 21
=> 21
```

Changing/Reassignment

Syntax

```
variable_name = new_value
```

Example

```
name
=> "Artur"
name_copy
=> "Artur"
age
=> 21

name = "Dustin"
=> "Dustin"
name_copy
=> "Artur"
name_copy = name
=> "Dustin"
age = 22
=> 22
```

Teaching the world how to code.

Company

- About
- Stories
- We're hiring
- Blog

Resources

- Articles
- Schools

Learn To Code

- Make a Website
- Make an Interactive Website
- Learn Rails
- Ruby on Rails Authentication
- Learn AngularJS
- Learn the Command Line
- Learn SQL
- SQL: Analyzing Business Metrics
- Learn Java
- Learn Git

- HTML & CSS
- JavaScript
- jQuery
- PHP
- Python
- Ruby
- Learn APIs