"12".to\_i #=> 12

"abc12".to\_i #=> 0

"12".to\_i.class #=> Fixnum

"abc12".to\_i.class #=> Fixnum

12.to\_s.class #=> String

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

#!/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

## 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 the 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 –

## 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 the 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

## Ruby Pseudo-Variables

They are special variables that have the appearance of local variables but behave like constants. You cannot assign any value to these variables.

* **self** − The receiver object of the current method.
* **true** − Value representing true.
* **false** − Value representing false.
* **nil** − Value representing undefined.
* **\_\_FILE\_\_** − The name of the current source file.
* **\_\_LINE\_\_** − The current line number in the source file.

## 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 -230 to 230-1 or -262 to 262-1. Integers within 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 the 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 floating 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}";

This will produce the following result −

Multiplication Value : 86400

## Backslash Notations

Following is the list of Backslash notations supported by Ruby −

|  |  |
| --- | --- |
| **Notation** | **Character represented** |
| \n | Newline (0x0a) |
| \r | Carriage return (0x0d) |
| \f | Formfeed (0x0c) |
| \b | Backspace (0x08) |
| \a | Bell (0x07) |
| \e | Escape (0x1b) |
| \s | Space (0x20) |
| \nnn | Octal notation (n being 0-7) |
| \xnn | Hexadecimal notation (n being 0-9, a-f, or A-F) |
| \cx, \C-x | Control-x |
| \M-x | Meta-x (c | 0x80) |
| \M-\C-x | Meta-Control-x |
| \x | Character x |

For more detail on Ruby Strings, go through [Ruby Strings](https://www.tutorialspoint.com/ruby/ruby_strings.htm).

## Ruby Arrays

Literals of Ruby Array are created by placing a comma-separated series of object references between the square brackets. A trailing comma is ignored.

### Example

#!/usr/bin/ruby

ary = [ "fred", 10, 3.14, "This is a string", "last element", ]

ary.each do |i|

puts i

end

This will produce the following result −

fred

10

3.14

This is a string

last element

For more detail on Ruby Arrays, go through [Ruby Arrays](https://www.tutorialspoint.com/ruby/ruby_arrays.htm).

## Ruby Hashes

A literal Ruby Hash is created by placing a list of key/value pairs between braces, with either a comma or the sequence => between the key and the value. A trailing comma is ignored.

### Example

#!/usr/bin/ruby

hsh = colors = { "red" => 0xf00, "green" => 0x0f0, "blue" => 0x00f }

hsh.each do |key, value|

print key, " is ", value, "\n"

end

This will produce the following result −

red is 3840

green is 240

blue is 15

For more detail on Ruby Hashes, go through [Ruby Hashes](https://www.tutorialspoint.com/ruby/ruby_hashes.htm).

## Ruby Ranges

A Range represents an interval which is a set of values with a start and an end. Ranges may be constructed using the s..e and s...e literals, or with Range.new.

Ranges constructed using .. run from the start to the end inclusively. Those created using ... exclude the end value. When used as an iterator, ranges return each value in the sequence.

A range (1..5) means it includes 1, 2, 3, 4, 5 values and a range (1...5) means it includes 1, 2, 3, 4 values.

### Example

#!/usr/bin/ruby

(10..15).each do |n|

print n, ' '

end

This will produce the following result −

10 11 12 13 14 15

## Ruby Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition − Adds values on either side of the operator. | a + b will give 30 |
| − | Subtraction − Subtracts right hand operand from left hand operand. | a - b will give -10 |
| \* | Multiplication − Multiplies values on either side of the operator. | a \* b will give 200 |
| / | Division − Divides left hand operand by right hand operand. | b / a will give 2 |
| % | Modulus − Divides left hand operand by right hand operand and returns remainder. | b % a will give 0 |
| \*\* | Exponent − Performs exponential (power) calculation on operators. | a\*\*b will give 10 to the power 20 |

## Ruby Comparison Operators

Assume variable a holds 10 and variable b holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the value of two operands are equal or not, if yes then condition becomes true. | (a == b) is not true. |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |
| <=> | Combined comparison operator. Returns 0 if first operand equals second, 1 if first operand is greater than the second and -1 if first operand is less than the second. | (a <=> b) returns -1. |
| === | Used to test equality within a when clause of a *case* statement. | (1...10) === 5 returns true. |
| .eql? | True if the receiver and argument have both the same type and equal values. | 1 == 1.0 returns true, but 1.eql?(1.0) is false. |
| equal? | True if the receiver and argument have the same object id. | if aObj is duplicate of bObj then aObj == bObj is true, a.equal?bObj is false but a.equal?aObj is true. |

## Ruby Assignment Operators

Assume variable a holds 10 and variable b holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, assigns values from right side operands to left side operand. | c = a + b will assign the value of a + b into c |
| += | Add AND assignment operator, adds right operand to the left operand and assign the result to left operand. | c += a is equivalent to c = c + a |
| -= | Subtract AND assignment operator, subtracts right operand from the left operand and assign the result to left operand. | c -= a is equivalent to c = c - a |
| \*= | Multiply AND assignment operator, multiplies right operand with the left operand and assign the result to left operand. | c \*= a is equivalent to c = c \* a |
| /= | Divide AND assignment operator, divides left operand with the right operand and assign the result to left operand. | c /= a is equivalent to c = c / a |
| %= | Modulus AND assignment operator, takes modulus using two operands and assign the result to left operand. | c %= a is equivalent to c = c % a |
| \*\*= | Exponent AND assignment operator, performs exponential (power) calculation on operators and assign value to the left operand. | c \*\*= a is equivalent to c = c \*\* a |

## Ruby Parallel Assignment

Ruby also supports the parallel assignment of variables. This enables multiple variables to be initialized with a single line of Ruby code. For example −

a = 10

b = 20

c = 30

This may be more quickly declared using parallel assignment −

a, b, c = 10, 20, 30

Parallel assignment is also useful for swapping the values held in two variables −

a, b = b, c

## Ruby Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation.

Assume if a = 60; and b = 13; now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

------------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following Bitwise operators are supported by Ruby language.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (a & b) will give 12, which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (a | b) will give 61, which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (a ^ b) will give 49, which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~a ) will give -61, which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | a << 2 will give 240, which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | a >> 2 will give 15, which is 0000 1111 |

## Ruby Logical Operators

The following logical operators are supported by Ruby language

Assume variable *a* holds 10 and variable *b* holds 20, then −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | Called Logical AND operator. If both the operands are true, then the condition becomes true. | (a and b) is true. |
| or | Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true. | (a or b) is true. |
| && | Called Logical AND operator. If both the operands are non zero, then the condition becomes true. | (a && b) is true. |
| || | Called Logical OR Operator. If any of the two operands are non zero, then the condition becomes true. | (a || b) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false. | !(a && b) is false. |
| not | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false. | not(a && b) is false. |

## Ruby Ternary Operator

There is one more operator called Ternary Operator. It first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation. The conditional operator has this syntax −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ? : | Conditional Expression | If Condition is true ? Then value X : Otherwise value Y |

## Ruby Range Operators

Sequence ranges in Ruby are used to create a range of successive values - consisting of a start value, an end value, and a range of values in between.

In Ruby, these sequences are created using the ".." and "..." range operators. The two-dot form creates an inclusive range, while the three-dot form creates a range that excludes the specified high value.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| .. | Creates a range from start point to end point inclusive. | 1..10 Creates a range from 1 to 10 inclusive. |
| ... | Creates a range from start point to end point exclusive. | 1...10 Creates a range from 1 to 9. |

## Ruby defined? Operators

defined? is a special operator that takes the form of a method call to determine whether or not the passed expression is defined. It returns a description string of the expression, or *nil* if the expression isn't defined.

There are various usage of defined? Operator

### Usage 1

defined? variable # True if variable is initialized

**For Example**

foo = 42

defined? foo # => "local-variable"

defined? $\_ # => "global-variable"

defined? bar # => nil (undefined)

### Usage 2

defined? method\_call # True if a method is defined

**For Example**

defined? puts # => "method"

defined? puts(bar) # => nil (bar is not defined here)

defined? unpack # => nil (not defined here)

### Usage 3

# True if a method exists that can be called with super user

defined? super

**For Example**

defined? super # => "super" (if it can be called)

defined? super # => nil (if it cannot be)

### Usage 4

defined? yield # True if a code block has been passed

**For Example**

defined? yield # => "yield" (if there is a block passed)

defined? yield # => nil (if there is no block)

## Ruby Dot "." and Double Colon "::" Operators

You call a module method by preceding its name with the module's name and a period, and you reference a constant using the module name and two colons.

The **::** is a unary operator that allows: constants, instance methods and class methods defined within a class or module, to be accessed from anywhere outside the class or module.

**Remember** in Ruby, classes and methods may be considered constants too.

You need to just prefix the **::** Const\_name with an expression that returns the appropriate class or module object.

If no prefix expression is used, the main Object class is used by default.

Here are two examples −

MR\_COUNT = 0 # constant defined on main Object class

module Foo

MR\_COUNT = 0

::MR\_COUNT = 1 # set global count to 1

MR\_COUNT = 2 # set local count to 2

end

puts MR\_COUNT # this is the global constant

puts Foo::MR\_COUNT # this is the local "Foo" constant

**Second Example**

CONST = ' out there'

class Inside\_one

CONST = proc {' in there'}

def where\_is\_my\_CONST

::CONST + ' inside one'

end

end

class Inside\_two

CONST = ' inside two'

def where\_is\_my\_CONST

CONST

end

end

puts Inside\_one.new.where\_is\_my\_CONST

puts Inside\_two.new.where\_is\_my\_CONST

puts Object::CONST + Inside\_two::CONST

puts Inside\_two::CONST + CONST

puts Inside\_one::CONST

puts Inside\_one::CONST.call + Inside\_two::CONST

## Ruby Operators Precedence

The following table lists all operators from highest precedence to lowest.

|  |  |  |
| --- | --- | --- |
| **Method** | **Operator** | **Description** |
| Yes | :: | Constant resolution operator |
| Yes | [ ] [ ]= | Element reference, element set |
| Yes | \*\* | Exponentiation (raise to the power) |
| Yes | ! ~ + - | Not, complement, unary plus and minus (method names for the last two are +@ and -@) |
| Yes | \* / % | Multiply, divide, and modulo |
| Yes | + - | Addition and subtraction |
| Yes | >> << | Right and left bitwise shift |
| Yes | & | Bitwise 'AND' |
| Yes | ^ | | Bitwise exclusive `OR' and regular `OR' |
| Yes | <= < > >= | Comparison operators |
| Yes | <=> == === != =~ !~ | Equality and pattern match operators (!= and !~ may not be defined as methods) |
|  | && | Logical 'AND' |
|  | || | Logical 'OR' |
|  | .. ... | Range (inclusive and exclusive) |
|  | ? : | Ternary if-then-else |
|  | = %= { /= -= += |= &= >>= <<= \*= &&= ||= \*\*= | Assignment |
|  | defined? | Check if specified symbol defined |
|  | not | Logical negation |
|  | or and | Logical composition |

# Ruby - if...else, case, unless

Ruby offers conditional structures that are pretty common to modern languages. Here, we will explain all the conditional statements and modifiers available in Ruby.

## Ruby if...else Statement

### Syntax

if conditional [then]

code...

[elsif conditional [then]

code...]...

[else

code...]

end

*if* expressions are used for conditional execution. The values *false* and *nil* are false, and everything else are true. Notice Ruby uses elsif, not else if nor elif.

Executes *code* if the *conditional* is true. If the *conditional* is not true, *code*specified in the else clause is executed.

An if expression's *conditional* is separated from code by the reserved word *then*, a newline, or a semicolon.

### Example

[Live Demo](http://tpcg.io/9aOJkD)

#!/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

x is 1

## Ruby if modifier

### Syntax

code if condition

Executes *code* if the *conditional* is true.

### Example

[Live Demo](http://tpcg.io/B9LLo0)

#!/usr/bin/ruby

$debug = 1

print "debug\n" if $debug

This will produce the following result −

debug

## Ruby unless Statement

### Syntax

unless conditional [then]

code

[else

code ]

end

Executes *code* if *conditional* is false. If the *conditional* is true, code specified in the else clause is executed.

### Example

[Live Demo](http://tpcg.io/SORgCW)

#!/usr/bin/ruby

x = 1

unless x>=2

puts "x is less than 2"

else

puts "x is greater than 2"

end

This will produce the following result −

x is less than 2

## Ruby unless modifier

### Syntax

code unless conditional

Executes *code* if *conditional* is false.

### Example

[Live Demo](http://tpcg.io/USjo33)

#!/usr/bin/ruby

$var = 1

print "1 -- Value is set\n" if $var

print "2 -- Value is set\n" unless $var

$var = false

print "3 -- Value is set\n" unless $var

This will produce the following result −

1 -- Value is set

3 -- Value is set

## Ruby case Statement

### Syntax

case expression

[when expression [, expression ...] [then]

code ]...

[else

code ]

end

Compares the *expression* specified by case and that specified by when using the === operator and executes the *code* of the when clause that matches.

The *expression* specified by the when clause is evaluated as the left operand. If no when clauses match, *case* executes the code of the *else* clause.

A *when* statement's expression is separated from code by the reserved word then, a newline, or a semicolon. Thus −

case expr0

when expr1, expr2

stmt1

when expr3, expr4

stmt2

else

stmt3

end

is basically similar to the following −

\_tmp = expr0

if expr1 === \_tmp || expr2 === \_tmp

stmt1

elsif expr3 === \_tmp || expr4 === \_tmp

stmt2

else

stmt3

end

### Example

[Live Demo](http://tpcg.io/prdGMS)

#!/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

This will produce the following result −

little child

# Ruby - Loops

Loops in Ruby are used to execute the same block of code a specified number of times. This chapter details all the loop statements supported by Ruby.

## Ruby while Statement

### Syntax

while conditional [do]

code

end

Executes *code* while *conditional* is true. A *while* loop's *conditional* is separated from *code* by the reserved word do, a newline, backslash \, or a semicolon ;.

### Example

[Live Demo](http://tpcg.io/T8XjWF)

#!/usr/bin/ruby

$i = 0

$num = 5

while $i < $num do

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

$i +=1

end

This will produce the following result −

Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4

## Ruby while modifier

### Syntax

code while condition

OR

begin

code

end while conditional

Executes *code* while *conditional* is true.

If a *while* modifier follows a *begin* statement with no *rescue* or ensure clauses, *code* is executed once before conditional is evaluated.

### Example

[Live Demo](http://tpcg.io/weGWXV)

#!/usr/bin/ruby

$i = 0

$num = 5

begin

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

$i +=1

end while $i < $num

This will produce the following result −

Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4

## Ruby until Statement

until conditional [do]

code

end

Executes *code* while *conditional* is false. An *until* statement's conditional is separated from *code* by the reserved word *do*, a newline, or a semicolon.

### Example

[Live Demo](http://tpcg.io/Gzut2W)

#!/usr/bin/ruby

$i = 0

$num = 5

until $i > $num do

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

$i +=1;

end

This will produce the following result −

Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4

Inside the loop i = 5

## Ruby until modifier

### Syntax

code until conditional

OR

begin

code

end until conditional

Executes *code* while *conditional* is false.

If an *until* modifier follows a *begin* statement with no *rescue* or ensure clauses, *code* is executed once before *conditional* is evaluated.

### Example

[Live Demo](http://tpcg.io/WVpBez)

#!/usr/bin/ruby

$i = 0

$num = 5

begin

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

$i +=1;

end until $i > $num

This will produce the following result −

Inside the loop i = 0

Inside the loop i = 1

Inside the loop i = 2

Inside the loop i = 3

Inside the loop i = 4

Inside the loop i = 5

## Ruby for Statement

### Syntax

for variable [, variable ...] in expression [do]

code

end

Executes *code* once for each element in *expression*.

### Example

[Live Demo](http://tpcg.io/rZqBys)

#!/usr/bin/ruby

for i in 0..5

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

end

Here, we have defined the range 0..5. The statement for *i* in 0..5 will allow *i* to take values in the range from 0 to 5 (including 5). This will produce the following result −

Value of local variable is 0

Value of local variable is 1

Value of local variable is 2

Value of local variable is 3

Value of local variable is 4

Value of local variable is 5

A *for...in* loop is almost exactly equivalent to the following −

(expression).each do |variable[, variable...]| code end

except that a *for* loop doesn't create a new scope for local variables. A *for*loop's *expression* is separated from *code* by the reserved word do, a newline, or a semicolon.

### Example

[Live Demo](http://tpcg.io/sAFa0I)

#!/usr/bin/ruby

(0..5).each do |i|

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

end

This will produce the following result −

Value of local variable is 0

Value of local variable is 1

Value of local variable is 2

Value of local variable is 3

Value of local variable is 4

Value of local variable is 5

## Ruby break Statement

### Syntax

break

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

### Example

[Live Demo](http://tpcg.io/H3CfLc)

#!/usr/bin/ruby

for i in 0..5

if i > 2 then

break

end

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

end

This will produce the following result −

Value of local variable is 0

Value of local variable is 1

Value of local variable is 2

## Ruby next Statement

### Syntax

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).

### Example

[Live Demo](http://tpcg.io/L2RUjl)

#!/usr/bin/ruby

for i in 0..5

if i < 2 then

next

end

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

end

This will produce the following result −

Value of local variable is 2

Value of local variable is 3

Value of local variable is 4

Value of local variable is 5

## Ruby redo Statement

### Syntax

redo

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

### Example

#!/usr/bin/ruby

for i in 0..5

if i < 2 then

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

redo

end

end

This will produce the following result and will go in an infinite loop −

Value of local variable is 0

Value of local variable is 0

............................

## 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

If retry appears in the iterator, the block, or the body of the *for* expression, restarts the invocation of the iterator call. Arguments to the iterator is re-evaluated.

for i in 1..5

retry if some\_condition # restart from i == 1

end

### Example

#!/usr/bin/ruby

for i in 0..5

if i > 2

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

end

This will produce the following result and will go in an infinite loop −

Value of local variable is 1

Value of local variable is 2

Value of local variable is 1

Value of local variable is 2

Value of local variable is 1

Value of local variable is 2

............................