Ruby

Ruby is a pure object-oriented programming language. It was created in 1993 by Yukihiro Matsumoto of Japan.

Features of Ruby

- Ruby is an open-source and is freely available on the Web, but it is subject to a license.
- Ruby is a general-purpose, interpreted programming language.
- Ruby is a true object-oriented programming language.
- Ruby is a server-side scripting language similar to Python and PERL.
- Ruby can be used to write Common Gateway Interface (CGI) scripts.
- Ruby can be embedded into Hypertext Markup Language (HTML).
- Ruby has a clean and easy syntax that allows a new developer to learn very quickly and easily.
- Ruby has similar syntax to that of many programming languages such as C++ and Perl.
- Ruby is very much scalable and big programs written in Ruby are easily maintainable.
- Ruby can be used for developing Internet and intranet applications.
- Ruby can be installed in Windows and POSIX environments.
- Ruby support many GUI tools such as Tcl/Tk, GTK, and OpenGL.
- Ruby can easily be connected to DB2, MySQL, Oracle, and Sybase.
- Ruby has a rich set of built-in functions, which can be used directly into Ruby scripts.

```
puts "Hello, Ruby!";
```

Ruby Identifiers

Identifiers are names of variables, constants, and methods. Ruby identifiers are case sensitive.

These reserved words may not be used as constant or variable names. They can, however, be used as method names.

BEGIN	do	next	then
END	else	nil	true
alias	elsif	not	undef
and	end	or	unless
begin	ensure	redo	until
break	false	rescue	when
case	for	retry	while
class	if	return	while
def	in	self	__FILE__
defined?	module	super	__LINE__

Comments are lines of annotation within Ruby code that are ignored at runtime. A single line comment starts with # character and they extend from # to the end of the line as follows

```
# This is a single line comment.

puts "Hello, Ruby!"
```

Ruby Multiline Comments

```
puts "Hello, Ruby!"

=begin
This is a multiline comment and con spwan as many lines as you
like. But =begin and =end should come in the first line only.
=end
```

Ruby - Operators

For each operator (+ - * / % ** & | ^ << >> && ||), there is a corresponding form of abbreviated assignment operator (+= -= etc.).

+	Addition − Adds values on either side of the operator.
−	Subtraction − Subtracts right hand operand from left hand operand.
*	Multiplication − Multiplies values on either side of the operator.
/	Division − Divides left hand operand by right hand operand.
%	Modulus − Divides left hand operand by right hand operand and returns remainder.
**	Exponent − Performs exponential (power) calculation on operators.

Ruby Comparison Operators

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

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.

```
a = 10
b = 20
c = 30

example: a, b, c = 10, 20, 30
```

Ruby Bitwise Operators

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

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 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.

```
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
```

Ruby if...else Statement

```
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 if modifier

Syntax

```
code if condition
```

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

Example

```
$debug = 1
print "Hai\n" if $debug
```

This will produce the following result −

```
Hai
```

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.

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

```
x is less than 2
```

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
```

```
$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
```

```
little child
```

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

```
$i = 0
$num = 5

while $i < $num  do
   puts("Inside the loop i = #$i" )
   $i +=1
end
```

```
nside 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

```
$i = 0
$num = 5

until $i > $num  do
   puts("Inside the loop i = #$i" )
   $i +=1;
end
```

```
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

```
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

```
(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).

```
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

```
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

```
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 - Built-in Functions

Functions for Numbers

```
num = 12.40
puts num.floor      # 12
puts num + 10       # 22.40
puts num.integer?   # false  as num is a float.
```

Sr.No.	Methods & Description
1	**n + num** **n - num** **n * num** **n / num** Performs arithmetic operations: addition, subtraction, multiplication, and division.
2	**n % num** Returns the modulus of n.
3	**n ** num** Exponentiation.
4	**n.abs** Returns the absolute value of n.
5	**n.ceil**

	Returns the smallest integer greater than or equal to n.
6	**n.coerce(num)** Returns an array containing num and n both possibly converted to a type that allows them to be operated on mutually. Used in automatic type conversion in numeric operators.
7	**n.divmod(num)** Returns an array containing the quotient and modulus from dividing n by num.
8	**n.floor** Returns the largest integer less than or equal to n.
9	**n.integer?** Returns true if n is an integer.
10	**n.modulo(num)** Returns the modulus obtained by dividing n by num and rounding the quotient with *floor*
11	**n.nonzero?** Returns n if it isn't zero, otherwise nil.
12	**n.remainder(num)** Returns the remainder obtained by dividing **n** by **num** and removing decimals from the quotient. The **result** and **n** always have same sign.
13	**n.round** Returns n rounded to the nearest integer.
14	**n.truncate** Returns n as an integer with decimals removed.

15	**n.zero?** Returns zero if n is 0.
16	**n & num** **n | num** **n ^ num** Bitwise operations: AND, OR, XOR, and inversion.
17	**n << num** **n >> num** Bitwise left shift and right shift.
18	**n[num]** Returns the value of the **num**th bit from the least significant bit, which is n[0].
19	**n.chr** Returns a string containing the character for the character code **n**.
20	**n.next** **n.succ** Returns the next integer following n. Equivalent to n + 1.
21	**n.size** Returns the number of bytes in the machine representation of **n**.
22	**n.step(upto, step) {|n| ...}** Iterates the block from **n** to **upto**, incrementing by **step** each time.
23	**n.times {|n| ...}** Iterates the block **n** times.

24	**n.to_f** Converts **n** into a floating point number. Float conversion may lose precision information.
25	**n.to_int** Returns **n** after converting into interger number.

Functions for Math

Sr.No.	Methods & Description
1	**atan2(x, y)** Calculates the arc tangent.
2	**cos(x)** Calculates the cosine of x.
3	**exp(x)** Calculates an exponential function (e raised to the power of x).
4	**frexp(x)** Returns a two-element array containing the nominalized fraction and exponent of x.
5	**ldexp(x, exp)** Returns the value of x times 2 to the power of exp.
6	**log(x)** Calculates the natural logarithm of x.
7	**log10(x)**

	Calculates the base 10 logarithm of x.
8	**sin(x)** Calculates the sine of x.
9	**sqrt(x)** Returns the square root of x. x must be positive.
10	**tan(x)** Calculates the tangent of x.

```
puts "Value of arc cosine"
puts Math::acos(0)

puts "Value of PI"
puts Math::PI
```

exp

This function is used to calculate the value of e_a, Return type of this function is float.

Code:

```
puts "Exponential value"

puts Math::exp(3)


puts "Value of square root"
puts Math.sqrt(9)

puts "Value of natural logarithm"
puts Math.log(5)
```