Methods

A method in Ruby is a set of expressions that returns a value. Within a method, you can organize your code into subroutines which can be easily invoked from other areas of their program.

A method name must start a letter or a character with the eight-bit set. It may contain letters, numbers, an \_ (underscore or low line) or a character with the eight-bit set. The convention is to use underscores to separate words in a multiword method name:

* A method definition starts with the 'def' keyword followed by the method name.
* Method parameters are specified between parentheses following the method name.
* The method definition ends with 'end' keyword on the bottom.

**Example:**

def print\_data(value)

puts value

end

**Method Names:**

A method name must start a letter or a character with the eight-bit set. It may contain letters, numbers, an \_ (underscore or low line) or a character with the eight-bit set. The convention is to use underscores to separate words in a multiword method name.

* Method names are US-ASCII compatible since the keys to type them exist on all keyboards.
* Method names may end with a ! (bang or exclamation mark), a ? (question mark) or = equals sign.
* The bang methods(! at the end of method name) are called and executed just like any other method.
* Methods that end with a question mark by convention return boolean. But they may not always return just true or false, often they return an object to indicate a true value.
* Methods that end with an equals sign indicate an assignment method. For assignment methods the return value is ignored, the arguments are returned instead.

**Return Values:**

By default, a method returns the last statement that was evaluated in the body of the method. You can do so by using the return keyword. In the following example, the last expression evaluated was the simple sum 100 + 200.

def add\_values

return 100 + 200

end

The statement is also used to return before the last expression is evaluated

def add\_values

return 100 + 200

end

300 + 200 # this expression is never evaluated

end

For assignment methods, the return value is always ignored. Instead the argument will be returned:

def x = (value)

return 100 + value

end

p(x = 500) # prints 500

**Arguments:**

In Ruby a method can accept arguments.

* The argument list follows the method name.
* The parentheses around the arguments are optional.
* Multiple arguments are separated by a comma.
* The arguments are positional.

See the following example :

def add\_values(num1, num2)

return num1 + num2

end

Copy

In the above example when you call the method you must provide two arguments. Within the method body, the arguments are local variables. The method will then add two arguments and return the value. If given 100 and 200 this method will return 300.

**Default Values:**

You can set default values of arguments. The default value does not need to appear first, but arguments with defaults must be grouped together.

def add\_values(num1, num2 = 15)

num1 + num2

end

Copy

**Call a method:**

You can call methods with or without parentheses. Here is the syntax :

# With parentheses

UserMethod()

# Without parentheses

UserMethod

Here is an example:

# Without parentheses

def Print\_String()

puts("Method without parentheses")

end

# With parentheses

def Add\_values(num1, num2)

return num1 + num2

end

# Default Values

def default\_values(num1, num2 = 12)

return num1 + num2

end

Print\_String()

puts(Add\_values(100, 200))

puts(default\_values(50))

Copy

Output :

Method without parentheses

300

62

**Scope:**

You can define an instance method on a specific class with the class keyword :

class Student

def admission

# ...

end

end

Copy

A method may be defined on another object. You may define a "class method" like this:

class Student

def self.admission

# ...

end

end

Copy

Here is the syntax for adding a method to an object :

language = "Ruby"

def language.tutorial

puts(self + ", Learning.")

end

language.tutorial # returns "Ruby, Learning."

Copy

Note: self is a keyword referring to the current object under consideration by the compiler.

**Overriding**

When Ruby executes the def keyword, it simply redefines it (whether the method already exists or not). This is called overriding. See the following example :

def language

puts("We are learning PHP")

end

def language

puts("We are learning Ruby")

end

# Now call the method

language

Copy

Output :

We are learning Ruby

More on Arguments

**Array Decomposition:**

You can extract values from an Array using extra parentheses in the arguments:

def add\_numbers((n1, n2))

p n1: n1, n2: n2

end

add\_numbers([10, 20]))

Copy

Output:

{:n1=>10, :n2=>20}

Extra argument in the array is ignored :

def add\_numbers((n1, n2))

p n1: n1, n2: n2

end

add\_numbers([10, 20, 30]))

Copy

Output:

{:n1=>10, :n2=>20}

Using a \* you can collect the remaining arguments. This splits an Array into a first element and the rest:

def add\_numbers((n1, \*n2))

p n1: n1, n2: n2

end

add\_numbers([10, 20, 30]))

Copy

Output:

{:n1=>10, :n2=>[20, 30]}

**Array/Hash Argument:**

Prefix an argument with \* to convert an argument to an Array. The array argument must be the last positional argument, it must appear before any keyword arguments.

def gather\_arguments(\*arguments)

p arguments

end

gather\_arguments 1, 2, 3 # prints [1, 2, 3]

Copy

The array argument will capture a Hash as the last entry if a hash was sent by the caller after all positional arguments. This only occurs if the method does not declare any keyword arguments.

gather\_arguments 1, a: 2 # prints [1, {:a=>2}]

Copy

Also, note that a bare \* can be used to ignore arguments:

def ignore\_arguments(\*)

end

**Keyword Arguments:**

Keyword arguments are similar to positional arguments with default values:

def add\_values(first: 1, second: 2)

first + second

end

Copy

Arbitrary keyword arguments will be accepted with \*\*:

def gather\_arguments(first: nil, \*\*rest)

p first, rest

end

gather\_arguments first: 1, second: 2, third: 3

# prints 1 then {:second=>2, :third=>3}

Note:   
a. When calling a method with keyword arguments the arguments may appear in any order. If an unknown keyword argument is sent by the caller an ArgumentError is raised.  
b. When mixing keyword arguments and positional arguments, all positional arguments must appear before any keyword arguments.

**Block Argument:**

The block argument is used to pass a block to another method. The block argument is indicated by & and must come last.

def each\_item(&block)

@items.each(&block)

end

**Exception Handling:**

Methods have an implied exception handling block so you do not need to use begin or end to handle exceptions.

def my\_method

begin

# code that may raise an exception

rescue

# handle exception

end

end

May be written as:

def my\_method

# code that may raise an exception

rescue

# handle exception

end

If you wish to rescue an exception for only part of your method use begin and end. For more details see the page on exception handling