# Ruby Tutorial

**Overview**

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

You can find the name Yukihiro Matsumoto on the Ruby mailing list at www.ruby-lang.org. Matsumoto is also known as Matz in the Ruby community.

**Ruby is "A Programmer's Best Friend".**

Ruby has features that are similar to those of Smalltalk, Perl, and Python. Perl, Python, and Smalltalk are scripting languages. Smalltalk is a true object-oriented language. Ruby, like Smalltalk, is a perfect object-oriented language. Using Ruby syntax is much easier than using Smalltalk syntax.

# 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 embeded into Hypertext Markup Language (HTML).
- Ruby has a clean and easy syntax that allows a new developer to learn Ruby 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.

**Environments**

we can run ruby in two ways.

1. we can create a file with the extension .rb and run the file in the terminal like below
   ruby application_name.rb

2. Interactive Ruby (IRb):

Interactive Ruby (IRb) provides a shell for experimentation. Within the IRb shell, you can immediately view expression results, line by line.

This tool comes along with Ruby installation so you have nothing to do extra to have IRb working.

Just type **irb** at your command prompt and an Interactive Ruby Session will start as given below:

```
$irb
irb 0.6.1(99/09/16)
irb(main):001:0> def hello
irb(main):002:1> out = "Hello World"
irb(main):003:1> puts out
irb(main):004:1> end
nil
irb(main):005:0> hello
Hello World
nil
irb(main):006:0>
```

## Comments

A comment hides a line, part of a line, or several lines from the Ruby interpreter. You can use the hash character (#) at the beginning of a line:

```
# I am a comment. Just ignore me.
```

Or, a comment may be on the same line after a statement or expression:

**Reserved Words:**

The following list shows the reserved words in Ruby. 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 | nill | 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__ |

Ruby Classes
Ruby is a perfect Object Oriented Programming Language. The features of the object-oriented programming language include:

- Data Encapsulation:
- Data Abstraction:
- Polymorphism:
- Inheritance:

These features have been discussed in Object Oriented Ruby.

An object-oriented program involves classes and objects. A class is the blueprint from which individual objects are created. In object-oriented terms, we say that your *bicycle* is an instance of the *class of objects* known as bicycles.

Take the example of any vehicle. It comprises wheels, horsepower, and fuel or gas tank capacity. These characteristics form the data members of the class Vehicle. You can differentiate one vehicle from the other with the help of these characteristics.

A vehicle can also have certain functions, such as halting, driving, and speeding. Even these functions form the data members of the class Vehicle. You can, therefore, define a class as a combination of characteristics and functions.

A class Vehicle can be defined as:

```
Class Vehicle
{
   Number no_of_wheels
   Number horsepower
   Characters type_of_tank
   Number Capacity
   Function speeding
   {
   }
   Function driving
   {
   }
   Function halting
   {
   }
}
```

By assigning different values to these data members, you can form several instances of the class Vehicle. For example, an airplane has three wheels, horsepower of 1,000, fuel as the type of tank, and a capacity of 100 liters. In the same way, a car has four wheels, horsepower of 200, gas as the type of tank, and a capacity of 25 litres.

## Defining a Class in Ruby:

To implement object-oriented programming by using Ruby, you need to first learn how to create objects and classes in Ruby.

A class in Ruby always starts with the keyword *class* followed by the name of the class. The name should always be in initial capitals. The class *Customer* can be displayed as:

```
class Customer
end
```

You terminate a class by using the keyword *end*. All the data members in the *class* are between the class definition and the *end* keyword.

## Variables in a Ruby Class:

Ruby provides four types of variables:

- **Local Variables:** Local variables are the variables that are defined in a method. Local variables are not available outside the method. You will see more detail about method in subsequent chapter. Local variables begin with a lowercase letter or _.
- **Instance Variables:** Instance variables are available across methods for any particular instance or object. That means that instance variables change from object to object. Instance

variables are preceded by the at sign (@) followed by the variable name.

- **Class Variables:** Class variables are available across different objects. A class variable belongs to the class and is a characteristic of a class. They are preceded by the sign @@ and are followed by the variable name.
- **Global Variables:** Class variables are not available across classes. If you want to have a single variable, which is available across classes, you need to define a global variable. The global variables are always preceded by the dollar sign ($).

**Example:**

Using the class variable @@no_of_customers, you can determine the number of objects that are being created. This enables in deriving the number of customers.

```
class Customer
   @@no_of_customers=0
end
```

## Creating Objects in Ruby using *new* Method:

Objects are instances of the class. You will now learn how to create objects of a class in Ruby. You can create objects in Ruby by using the method *new* of the class.

The method *new* is a unique type of method, which is predefined in the Ruby library. The new method belongs to the *class* methods.

Here is the example to create two objects cust1 and cust2 of the class Customer:

```
cust1 = Customer. new
cust2 = Customer. new
```

Here, cust1 and cust2 are the names of two objects. You write the object name followed by the equal to sign (=) after which the class name will follow. Then, the dot operator and the keyword *new* will follow.

## Custom Method to create Ruby Objects :

You can pass parameters to method *new* and those parameters can be used to initialize class variables.

When you plan to declare the *new* method with parameters, you need to declare the method *initialize* at the time of the class creation.

The *initialize* method is a special type of method, which will be executed when the *new* method of the class is called with parameters.

Here is the example to create initialize method:

```
class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
end
```

In this example, you declare the *initialize* method with **id, name**, and **addr** as local variables. Here *def* and *end* are used to define a Ruby method *initialize*. You will learn more about methods in

subsequent chapters.

In the *initialize* method, you pass on the values of these local variables to the instance variables @cust_id, @cust_name, and @cust_addr. Here local variables hold the values that are passed along with the new method.

Now you can create objects as follows:

```
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
```

## Member Functions in Ruby Class:

In Ruby, functions are called methods. Each method in a *class* starts with the keyword *def* followed by the method name.

The method name always preferred in **lowercase letters**. You end a method in Ruby by using the keyword *end*.

Here is the example to define a Ruby method:

```
class Sample
   def function
      statement 1
      statement 2
   end
end
```

Here *statement 1* and *statement 2* are part of the body of the method *function* inside the class Sample. These statments could be any valid Ruby statement. For example we can put a method *puts* to print *Hello Ruby* as follows:

```
class Sample
   def hello
      puts "Hello Ruby!"
   end
end
```

Now in the following example create one object of Sample class and call *hello* method and see the result:

```
#!/usr/bin/ruby

class Sample
   def hello
      puts "Hello Ruby!"
   end
end

# Now using above class to create objects
object = Sample. new
object.hello
```

This will produce following result:

```
Hello Ruby!
```

Variables are the memory locations which holds any data to be used by any program.

There are five types of variables supported by Ruby. You already have gone through a small description of these variables in previous chapter as well. These five types of variables are

explained in this chapter.

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

Here is an example showing usage of global variable.

```
#!/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
```

Here $global_variable is a global variable. This will produce following result:

**NOTE:** In Ruby you CAN access value of any variable or constant by putting a hash (#) character just before that variable or constant.

```
Global variable in Class1 is 10
Global variable in Class2 is 10
```

### 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 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 following result:

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
```

# 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 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 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
```

Here VAR1 and VAR2 are constant. This will produce following result:

```
Value of first Constant is 100
Value of second Constant is 200
```

### Ruby Pseudo-Variables:

They are special variables that have the appearance of local variables but behave like constants. You can not 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 $-2^{30}$ to $2^{30-1}$ or $-2^{62}$ to $2^{62-1}$. Integers with-in 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 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 integer 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 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 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.

## Ruby Arrays:

Literals of Ruby Array are created by placing a comma-separated series of object references between 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 following result:

```
fred
10
3.14
This is a string
last element
```

## 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 following result:

```
green is 240
red is 3840
blue is 15
```

## Ruby Ranges:

A Range represents an interval.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 following result:

```
10 11 12 13 14 15
```

## Operators:

1. Arthamatic Operators(+,-,*,/,%,**)
2. Comparision Operators(==,!=,>,<,>=,<=,<=>,.eql?,equal?)
3. Assignment operators(=,+=,-=,*=,/=,%=,**=)
4. parallel Assignment(=(a,b,c = 10,20,30))
5. Bitwise Operators(&,|,^,~,<<,>>)
6. Logical Operators(and,or,&&,||,!,not)
7. Ternary Operator(?:)
8. Range Operator(..,...)

**Conditional Statements:**

1. if(condition)
   statements;
   end

   if condition
   statements;
   end

2. if condition
   statements
   else
   statements
   end
3. unless condition
   statements
   end
4. condition? True statement : False statement
   a == 10? puts “a value is 10” : puts “a value is not 10”

**Looping Statements:**

1. while condition
   statements
   end

2. statements while condition

3. until condition
   statements
   end

4. for i in 0..5
   statements
   end

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

**Ruby Methods**

Methods are indicate with def followed by method name.

```
def method_name
    statements
end

def method_name(var1,var2)
    statements
end
```

```
def  method_name(a=val1,b=val2)
        statements
end
```

**Ruby Arrays:**

Arrays are ordered, integer-indexed collections of any object.

Array indexing starts at 0, as in C or Java. A negative index is assumed to be relative to the end of the array---that is, an index of -1 indicates the last element of the array, -2 is the next to last element in the array, and so on.

Creating of arrays:

A new array can be created by using the literal constructor `[]`. Arrays can contain different types of objects. For example, the array below contains an Integer, a String and a Float:

```
ary = [1, "two", 3.0] #=> [1, "two", 3.0]
ary = Array.new    #=> []

Array.new(3)       #=> [nil, nil, nil]

Array.new(3, true) #=> [true, true, true]

Array.new(4) { Hash.new } #=> [{}, {}, {}, {}]
empty_table = Array.new(3) { Array.new(3) } #=> [[nil, nil, nil], [nil, nil, nil], [nil, nil, nil]]

Array({:a => "a", :b => "b"}) #=> [[:a, "a"], [:b, "b"]]

arr = [1, 2, 3, 4, 5, 6]



arr[2]    #=> 3



arr[100]  #=> nil



arr[-3]   #=> 4



arr[2, 3] #=> [3, 4, 5]



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

```
arr.at(0) #=> 1

arr = ['a', 'b', 'c', 'd', 'e', 'f']

arr.fetch(100) #=> IndexError: index 100 outside of array bounds: -6...6

arr.fetch(100, "oops") #=> "oops"

arr.first #=> 1

arr.last  #=> 6

arr.take(3) #=> [1, 2, 3]

arr.drop(3) #=> [4, 5, 6]

browsers = ['Chrome', 'Firefox', 'Safari', 'Opera', 'IE']

browsers.length #=> 5

browsers.count #=> 5

browsers.empty? #=> false
```

```
browsers.include?('Konqueror') #=> false

arr = [1, 2, 3, 4]

arr.push(5) #=> [1, 2, 3, 4, 5]

arr << 6    #=> [1, 2, 3, 4, 5, 6]



arr.unshift(0) #=> [0, 1, 2, 3, 4, 5, 6]

arr.insert(3, 'apple')  #=> [0, 1, 2, 'apple', 3, 4, 5, 6]

arr.insert(3, 'orange', 'pear', 'grapefruit')

#=> [0, 1, 2, "orange", "pear", "grapefruit", "apple", 3, 4, 5, 6]



arr =  [1, 2, 3, 4, 5, 6]

arr.pop #=> 6
```

arr #=> [1, 2, 3, 4, 5]

arr.shift #=> 1

arr #=> [2, 3, 4, 5]

arr.delete_at(2) #=> 4

arr #=> [2, 3, 5]

arr = [1, 2, 2, 3]

arr.delete(2) #=> [1, 3]

r.compact  #=> ['foo', 0, 'bar', 7, 'baz']

arr  #=> ['foo', 0, nil, 'bar', 7, 'baz', nil]

arr.compact! #=> ['foo', 0, 'bar', 7, 'baz']

arr  #=> ['foo', 0, 'bar', 7, 'baz']

arr = [2, 5, 6, 556, 6, 6, 8, 9, 0, 123, 556]

arr.uniq #=> [2, 5, 6, 556, 8, 9, 0, 123]

Array Iterations

```
arr = [1, 2, 3, 4, 5]
arr.each { |a| print a -= 10, " " }
# prints: -9 -8 -7 -6 -5
#=> [1, 2, 3, 4, 5]

words = %w[rats live on no evil star]
str = ""
words.reverse_each { |word| str += "#{word.reverse} " }
str #=> "rats live on no evil star "

arr.map { |a| 2*a }   #=> [2, 4, 6, 8, 10]
arr                   #=> [1, 2, 3, 4, 5]
arr.map! { |a| a**2 } #=> [1, 4, 9, 16, 25]
arr

arr = [1, 2, 3, 4, 5, 6]
arr.select { |a| a > 3 }     #=> [4, 5, 6]
arr.reject { |a| a < 3 }     #=> [3, 4, 5, 6]
arr.drop_while { |a| a < 4 } #=> [4, 5, 6]
arr                          #=> [1, 2, 3, 4, 5, 6]

arr.delete_if { |a| a < 4 } #=> [4, 5, 6]
arr                         #=> [4, 5, 6]
```

```
arr = [1, 2, 3, 4, 5, 6]

arr.keep_if { |a| a < 4 } #=> [1, 2, 3]

arr                       #=> [1, 2, 3]


Array.[]( 1, 'a', /^A/ ) # => [1, "a", /^A/]

Array[ 1, 'a', /^A/ ]    # => [1, "a", /^A/]

[ 1, 'a', /^A/ ]         # => [1, "a", /^A/]


first_array = ["Matz", "Guido"]


second_array = Array.new(first_array) #=> ["Matz", "Guido"]


first_array.equal? second_array       #=> false


Array.new(3){ |index| index ** 2 }

# => [0, 1, 4]


a = Array.new(2, Hash.new)

# => [{}, {}]


a[0]['cat'] = 'feline'

a # => [{"cat"=>"feline"}, {"cat"=>"feline"}]
```

```
a[1]['cat'] = 'Felix'

a # => [{"cat"=>"Felix"}, {"cat"=>"Felix"}]


a = Array.new(2) { Hash.new }

a[0]['cat'] = 'feline'

a # => [{"cat"=>"feline"}, {}]


Array.try_convert([1])   #=> [1](it will convert into array if object is can't convert into array it returns nil like below)

Array.try_convert("1")   #=> nil


[ 1, 1, 3, 5 ] & [ 1, 2, 3 ]                 #=> [ 1, 3 ]

[ 'a', 'b', 'b', 'z' ] & [ 'a', 'b', 'c' ]   #=> [ 'a', 'b' ]


(& will give the common elements)


[ 1, 2, 3 ] * 3    #=> [ 1, 2, 3, 1, 2, 3, 1, 2, 3 ]

[ 1, 2, 3 ] * ","  #=> "1,2,3"


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

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

a + [ "d", "e", "f" ]

a                          #=> [ "a", "b", "c", "d", "e", "f" ]


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

```
[ 1, 2 ] << "c" << "d" << [ 3, 4 ]

      #=>  [ 1, 2, "c", "d", [ 3, 4 ] ]
```

(<< will insert the element into array)

```
[ "a", "a", "c" ]    <=> [ "a", "b", "c" ]   #=> -1

[ 1, 2, 3, 4, 5, 6 ] <=> [ 1, 2 ]           #=> +1
```

( if both are equal then it will return 0, if left one has less number of matchings then it will return -1 or else it return +1)

```
[ "a", "c" ]    == [ "a", "c", 7 ]     #=> false

[ "a", "c", 7 ] == [ "a", "c", 7 ]     #=> true

[ "a", "c", 7 ] == [ "a", "d", "f" ]   #=> false


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

a[2] +  a[0] + a[1]    #=> "cab"

a[6]                  #=> nil

a[1, 2]               #=> [ "b", "c" ]

a[1..3]               #=> [ "b", "c", "d" ]

a[4..7]               #=> [ "e" ]

a[6..10]              #=> nil

a[-3, 3]              #=> [ "c", "d", "e" ]

# special cases

a[5]                  #=> nil
```

```
a[6, 1]              #=> nil

a[5, 1]              #=> []

a[5..10]             #=> []


a = Array.new

a[4] = "4";                 #=> [nil, nil, nil, nil, "4"]

a[0, 3] = [ 'a', 'b', 'c' ] #=> ["a", "b", "c", nil, "4"]

a[1..2] = [ 1, 2 ]          #=> ["a", 1, 2, nil, "4"]

a[0, 2] = "?"               #=> ["?", 2, nil, "4"]

a[0..2] = "A"               #=> ["A", "4"]

a[-1]   = "Z"               #=> ["A", "Z"]

a[1..-1] = nil              #=> ["A", nil]

a[1..-1] = []               #=> ["A"]

a[0, 0] = [ 1, 2 ]          #=> [1, 2, "A"]

a[3, 0] = "B"               #=> [1, 2, "A", "B"]


s1 = [ "colors", "red", "blue", "green" ]

s2 = [ "letters", "a", "b", "c" ]

s3 = "foo"

a  = [ s1, s2, s3 ]

a.assoc("letters")  #=> [ "letters", "a", "b", "c" ]

a.assoc("foo")      #=> nil


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

```
a.at(0)     #=> "a"
a.at(-1)    #=> "e"

a = [ "a", "b", "c", "d", "e" ]
a.clear    #=> [ ]

a = [ "a", "b", "c", "d" ]
a.map { |x| x + "!" }   #=> ["a!", "b!", "c!", "d!"]
a                       #=> ["a", "b", "c", "d"]

[ "a", nil, "b", nil, "c" ].compact! #=> [ "a", "b", "c" ]
[ "a", "b", "c" ].compact!           #=> nil

 [ "a", "b" ].concat( ["c", "d"] ) #=> [ "a", "b", "c", "d" ]
a = [ 1, 2, 3 ]
a.concat( [ 4, 5 ] )
a                              #=> [ 1, 2, 3, 4, 5 ]

ary = [1, 2, 4, 2]
ary.count                  #=> 4
ary.count(2)               #=> 2

a = [ "a", "b", "b", "b", "c" ]
a.delete("b")                  #=> "b"
```

```
a                           #=> ["a", "c"]
a.delete("z")               #=> nil
a.delete("z") { "not found" }   #=> "not found"

a = [ "a", "b", "c" ]
a.each {|x| print x, " -- " }   # a--b--c



a = [ "a", "b", "c" ]
a.each_index {|x| print x, " -- " } # 1 --2 -- 3

[].empty?   #=> true

s = [ 1, 2, 3 ]          #=> [1, 2, 3]
t = [ 4, 5, 6, [7, 8] ]   #=> [4, 5, 6, [7, 8]]
a = [ s, t, 9, 10 ]       #=> [[1, 2, 3], [4, 5, 6, [7, 8]], 9, 10]
a.flatten                 #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
a = [ 1, 2, [3, [4, 5] ] ]
a.flatten(1)              #=> [1, 2, 3, [4, 5]]

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

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

a.pop     #=> "d"

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

a         #=> ["a"]


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

a.sort                    #=> ["a", "b", "c", "d", "e"]

a.sort { |x,y| y <=> x }  #=> ["e", "d", "c", "b", "a"]
```