Class: Array (Ruby 2.1.2)

Repetition — With a String argument, equivalent to `ary.join(str)`.

Otherwise, returns a new array built by concatenating the `int` copies of `self`.

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

ary + other_ary → new_ary
Concatenation — Returns a new array built by concatenating the two arrays together to produce a third array.

```
[ 1, 2, 3 ] + [ 4, 5 ]    #=> [ 1, 2, 3, 4, 5 ]
a = [ "a", "b", "c" ]
c = a + [ "d", "e", "f" ]
c                         #=> [ "a", "b", "c", "d", "e", "f" ]
a                         #=> [ "a", "b", "c" ]
```

See also #concat.

Array Difference

Returns a new array that is a copy of the original array, removing any items that also appear in `other_ary`. The order is preserved from the original array.

It compares elements using their hash and eql? methods for efficiency.

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

If you need set-like behavior, see the library class Set.

ary << obj → ary
Append—Pushes the given object on to the end of this array. This expression returns the array itself, so several appends may be chained together.

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

ary <=> other_ary → -1, 0, +1 or nil
Comparison — Returns an integer (`-1`, `0`, or `+1`) if this array is less than, equal to, or greater than `other_ary`.

`nil` is returned if the two values are incomparable.

Each object in each array is compared (using the <=> operator).

Arrays are compared in an “element-wise” manner; the first two elements that are not equal will determine the return value for the whole comparison.

If all the values are equal, then the return is based on a comparison of the array lengths. Thus, two arrays are “equal” according to Array#<=> if, and only if, they have the same length and the value of each element is equal to the value of the corresponding element in the other array.

```
[ "a", "a", "c" ]    <=> [ "a", "b", "c" ]   #=> -1
[ 1, 2, 3, 4, 5, 6 ] <=> [ 1, 2 ]            #=> +1
```

ary == other_ary → bool
Equality — Two arrays are equal if they contain the same number of elements and if each element is equal to (according to Object#==) the corresponding element in `other_ary`.

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

ary[index] → obj or nil
ary[start, length] → new_ary or nil
ary[range] → new_ary or nil
slice(index) → obj or nil
slice(start, length) → new_ary or nil
slice(range) → new_ary or nil
Element Reference — Returns the element at `index`, or returns a subarray starting at the `start` index and continuing for `length` elements, or returns a subarray specified by `range` of indices.

Negative indices count backward from the end of the array (-1 is the last element). For `start` and `range` cases the starting index is just before an element. Additionally, an empty array is returned when the starting index for an element range is at the end of the array.

Returns `nil` if the index (or starting index) are out of range.

```
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]                #=> []
```

ary[index] = obj → obj
ary[start, length] = obj or other_ary or nil → obj or other_ary or nil
ary[range] = obj or other_ary or nil → obj or other_ary or nil
Element Assignment — Sets the element at `index`, or replaces a subarray from the `start` index for `length` elements, or replaces a subarray specified by the `range` of indices.

If indices are greater than the current capacity of the array, the array grows automatically. Elements are inserted into the array at `start` if `length` is zero.

Negative indices will count backward from the end of the array. For `start` and `range` cases the starting index is just before an element.

An IndexError is raised if a negative index points past the beginning of the array.

See also #push, and #unshift.

```
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"]
```

assoc(obj) → new_ary or nil
Searches through an array whose elements are also arrays comparing `obj` with the first element of each contained array using `obj.==`.

Returns the first contained array that matches (that is, the first associated array), or `nil` if no match is found.

See also #rassoc

```
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
```

at(index) → obj or nil
Returns the element at `index`. A negative index counts from the end of `self`. Returns `nil` if the index is out of range. See also #[].

```
a = [ "a", "b", "c", "d", "e" ]
a.at(0)     #=> "a"
a.at(-1)    #=> "e"
```

bsearch {|x| block } → elem
By using binary search, finds a value from this array which meets the given condition in O(log n) where n is the size of the array.

You can use this method in two use cases: a find-minimum mode and a find-any mode. In either case, the elements of the array must be monotone (or sorted) with respect to the block.

In find-minimum mode (this is a good choice for typical use case), the block must return true or false, and there must be an index i (0 <= i <= ary.size) so that:

- the block returns false for any element whose index is less than i, and
- the block returns true for any element whose index is greater than or equal to i.

This method returns the i-th element. If i is equal to ary.size, it returns nil.

```
ary = [0, 4, 7, 10, 12]
ary.bsearch {|x| x >=   4 } #=> 4
ary.bsearch {|x| x >=   6 } #=> 7
ary.bsearch {|x| x >=  -1 } #=> 0
ary.bsearch {|x| x >= 100 } #=> nil
```

In find-any mode (this behaves like libc's bsearch(3)), the block must return a number, and there must be two indices i and j (0 <= i <= j <= ary.size) so that:

- the block returns a positive number for ary if 0 <= k < i,
- the block returns zero for ary if i <= k < j, and
- the block returns a negative number for ary if j <= k < ary.size.

Under this condition, this method returns any element whose index is within i…j. If i is equal to j (i.e., there is no element that satisfies the block), this method returns nil.

```
ary = [0, 4, 7, 10, 12]
# try to find v such that 4 <= v < 8
ary.bsearch {|x| 1 - x / 4 } #=> 4 or 7
# try to find v such that 8 <= v < 10
ary.bsearch {|x| 4 - x / 2 } #=> nil
```

You must not mix the two modes at a time; the block must always return either true/false, or always return a number. It is undefined which value is actually picked up at each iteration.

clear → ary
Removes all elements from `self`.

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

collect { |item| block } → new_ary
collect → Enumerator
Invokes the given block once for each element of `self`.

Creates a new array containing the values returned by the block.

See also Enumerable#collect.

If no block is given, an Enumerator is returned instead.

```
a = [ "a", "b", "c", "d" ]
a.collect { |x| x + "!" }        #=> ["a!", "b!", "c!", "d!"]
a.map.with_index{ |x, i| x * i } #=> ["", "b", "cc", "ddd"]
a                                #=> ["a", "b", "c", "d"]
```

collect! {|item| block } → ary
collect! → Enumerator
Invokes the given block once for each element of `self`, replacing the element with the value returned by the block.

See also Enumerable#collect.

If no block is given, an Enumerator is returned instead.

```
a = [ "a", "b", "c", "d" ]
a.map! {|x| x + "!" }
a #=>  [ "a!", "b!", "c!", "d!" ]
a.collect!.with_index {|x, i| x[0...i] }
a #=>  ["", "b", "c!", "d!"]
```

combination(n) { |c| block } → ary
combination(n) → Enumerator
When invoked with a block, yields all combinations of length `n` of elements from the array and then returns the array itself.

The implementation makes no guarantees about the order in which the combinations are yielded.

If no block is given, an Enumerator is returned instead.

Examples:

```
a = [1, 2, 3, 4]
a.combination(1).to_a  #=> [[1],[2],[3],[4]]
a.combination(2).to_a  #=> [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
a.combination(3).to_a  #=> [[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
a.combination(4).to_a  #=> [[1,2,3,4]]
a.combination(0).to_a  #=> [[]] # one combination of length 0
a.combination(5).to_a  #=> []   # no combinations of length 5
```

compact → new_ary
Returns a copy of `self` with all `nil` elements removed.

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

compact! → ary or nil
Removes `nil` elements from the array.

Returns `nil` if no changes were made, otherwise returns the array.

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

concat(other_ary) → ary
Appends the elements of `other_ary` to `self`.

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

See also Array#+.

Returns the number of elements.

If an argument is given, counts the number of elements which equal `obj` using `==`.

If a block is given, counts the number of elements for which the block returns a true value.

```
ary = [1, 2, 4, 2]
ary.count                  #=> 4
ary.count(2)               #=> 2
ary.count { |x| x%2 == 0 } #=> 3
```

cycle(n=nil) { |obj| block } → nil
cycle(n=nil) → Enumerator
Calls the given block for each element `n` times or forever if `nil` is given.

Does nothing if a non-positive number is given or the array is empty.

Returns `nil` if the loop has finished without getting interrupted.

If no block is given, an Enumerator is returned instead.

```
a = ["a", "b", "c"]
a.cycle { |x| puts x }     # print, a, b, c, a, b, c,.. forever.
a.cycle(2) { |x| puts x }  # print, a, b, c, a, b, c.
```

delete(obj) → item or nil
delete(obj) { block } → item or result of block
Deletes all items from `self` that are equal to `obj`.

Returns the last deleted item, or `nil` if no matching item is found.

If the optional code block is given, the result of the block is returned if the item is not found. (To remove `nil` elements and get an informative return value, use #compact!)

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

delete_at(index) → obj or nil
Deletes the element at the specified `index`, returning that element, or `nil` if the `index` is out of range.

See also #slice!

```
a = ["ant", "bat", "cat", "dog"]
```

```
a.delete_at(2)    #=> "cat"
a                 #=> ["ant", "bat", "dog"]
a.delete_at(99)   #=> nil
```

delete_if { |item| block } → ary
delete_if → Enumerator
Deletes every element of `self` for which block evaluates to `true`.

The array is changed instantly every time the block is called, not after the iteration is over.

See also #reject!

If no block is given, an Enumerator is returned instead.

```
scores = [ 97, 42, 75 ]
scores.delete_if {|score| score < 80 }   #=> [97]
```

drop(n) → new_ary
Drops first `n` elements from `ary` and returns the rest of the elements in an array.

If a negative number is given, raises an ArgumentError.

See also #take

```
a = [1, 2, 3, 4, 5, 0]
a.drop(3)             #=> [4, 5, 0]
```

drop_while { |arr| block } → new_ary
drop_while → Enumerator
Drops elements up to, but not including, the first element for which the block returns `nil` or `false` and returns an array containing the remaining elements.

If no block is given, an Enumerator is returned instead.

See also #take_while

```
a = [1, 2, 3, 4, 5, 0]
a.drop_while {|i| i < 3 }   #=> [3, 4, 5, 0]
```

each { |item| block } → ary
each → Enumerator
Calls the given block once for each element in `self`, passing that element as a parameter.

An Enumerator is returned if no block is given.

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

produces:

```
a -- b -- c --
```

Returns `true` if `self` contains no elements.

```
[].empty?   #=> true
```

eql?(other) → true or false
Returns `true` if `self` and `other` are the same object, or are both arrays with the same content (according to Object#eql?).

fetch(index) → obj
fetch(index, default) → obj
fetch(index) { |index| block } → obj
Tries to return the element at position `index`, but throws an IndexError exception if the referenced `index` lies outside of the array bounds. This error can be prevented by supplying a second argument, which will act as a `default` value.

Alternatively, if a block is given it will only be executed when an invalid `index` is referenced. Negative values of `index` count from the end of the array.

```
a = [ 11, 22, 33, 44 ]
a.fetch(1)               #=> 22
a.fetch(-1)              #=> 44
a.fetch(4, 'cat')        #=> "cat"
a.fetch(100) { |i| puts "#{i} is out of bounds" }
                         #=> "100 is out of bounds"
```

fill(obj) → ary
fill(obj, start [, length]) → ary
fill(obj, range) → ary
fill { |index| block } → ary
fill(start [, length]) { |index| block } → ary
fill(range) { |index| block } → ary
The first three forms set the selected elements of `self` (which may be the entire array) to `obj`.

A `start` of `nil` is equivalent to zero.

A `length` of `nil` is equivalent to the length of the array.

The last three forms fill the array with the value of the given block, which is passed the absolute index of each element to be filled.

Negative values of `start` count from the end of the array, where `-1` is the last element.

```
a = [ "a", "b", "c", "d" ]
a.fill("x")              #=> ["x", "x", "x", "x"]
a.fill("z", 2, 2)        #=> ["x", "x", "z", "z"]
a.fill("y", 0..1)        #=> ["y", "y", "z", "z"]
a.fill { |i| i*i }       #=> [0, 1, 4, 9]
a.fill(-2) { |i| i*i*i } #=> [0, 1, 8, 27]
```

find_index(obj) → int or nil
find_index { |item| block } → int or nil
find_index → Enumerator
Returns the *index* of the first object in `ary` such that the object is `==` to `obj`.

If a block is given instead of an argument, returns the *index* of the first object for which the block returns `true`. Returns `nil` if no match is found.

See also #rindex.

An Enumerator is returned if neither a block nor argument is given.

```
a = [ "a", "b", "c" ]
a.index("b")              #=> 1
a.index("z")              #=> nil
a.index { |x| x == "b" }  #=> 1
```

first → obj or nil
first(n) → new_ary
Returns the first element, or the first `n` elements, of the array. If the array is empty, the first form returns `nil`, and the second form returns an empty array. See also #last for the opposite effect.

```
a = [ "q", "r", "s", "t" ]
a.first     #=> "q"
a.first(2)  #=> ["q", "r"]
```

flatten → new_ary
flatten(level) → new_ary
Returns a new array that is a one-dimensional flattening of `self` (recursively).

That is, for every element that is an array, extract its elements into the new array.

The optional `level` argument determines the level of recursion to flatten.

```
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]]
```

flatten! → ary or nil
flatten!(level) → ary or nil
Flattens `self` in place.

Returns `nil` if no modifications were made (i.e., the array contains no subarrays.)

The optional `level` argument determines the level of recursion to flatten.

```
a = [ 1, 2, [3, [4, 5] ] ]
a.flatten!   #=> [1, 2, 3, 4, 5]
a.flatten!   #=> nil
a            #=> [1, 2, 3, 4, 5]
a = [ 1, 2, [3, [4, 5] ] ]
a.flatten!(1) #=> [1, 2, 3, [4, 5]]
```

frozen? → true or false
Return `true` if this array is frozen (or temporarily frozen while being sorted). See also Object#frozen?

hash → fixnum
Compute a hash-code for this array.

Two arrays with the same content will have the same hash code (and will compare using eql?).

include?(object) → true or false
Returns `true` if the given `object` is present in `self` (that is, if any element `==` `object`), otherwise returns `false`.

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

index(obj) → int or nil
index { |item| block } → int or nil
index → Enumerator
Returns the *index* of the first object in `ary` such that the object is `==` to `obj`.

If a block is given instead of an argument, returns the *index* of the first object for which the block returns `true`. Returns `nil` if no match is found.

See also #rindex.

An Enumerator is returned if neither a block nor argument is given.

```
a = [ "a", "b", "c" ]
a.index("b")              #=> 1
a.index("z")              #=> nil
a.index { |x| x == "b" }  #=> 1
```

initialize_copy(other_ary) → ary
Replaces the contents of `self` with the contents of `other_ary`, truncating or expanding if necessary.

```
a = [ "a", "b", "c", "d", "e" ]
a.replace([ "x", "y", "z" ])   #=> ["x", "y", "z"]
a                              #=> ["x", "y", "z"]
```

insert(index, obj...) → ary
Inserts the given values before the element with the given `index`.

Negative indices count backwards from the end of the array, where `-1` is the last element.

```
a = %w{ a b c d }
a.insert(2, 99)         #=> ["a", "b", 99, "c", "d"]
a.insert(-2, 1, 2, 3)   #=> ["a", "b", 99, "c", 1, 2, 3, "d"]
```

inspect → string
to_s → string
Creates a string representation of `self`.

```
[ "a", "b", "c" ].to_s     #=> "[\"a\", \"b\", \"c\"]"
```

Also aliased as: to_s
join(separator=$,) → str
Returns a string created by converting each element of the array to a string, separated by the given `separator`. If the `separator` is `nil`, it uses current $,. If both the `separator` and $, are nil, it uses empty string.

```
[ "a", "b", "c" ].join        #=> "abc"
[ "a", "b", "c" ].join("-")   #=> "a-b-c"
```

keep_if { |item| block } → ary
keep_if → Enumerator
Deletes every element of `self` for which the given block evaluates to `false`.

Invokes the given block once for each element of `self`.

Creates a new array containing the values returned by the block.

See also Enumerable#collect.

If no block is given, an Enumerator is returned instead.

```
a = [ "a", "b", "c", "d" ]
a.collect { |x| x + "!" }        #=> ["a!", "b!", "c!", "d!"]
a.map.with_index{ |x, i| x * i } #=> ["", "b", "cc", "ddd"]
a                                #=> ["a", "b", "c", "d"]
```

map! {|item| block } → ary
map! → Enumerator
Invokes the given block once for each element of `self`, replacing the element with the value returned by the block.

See also Enumerable#collect.

If no block is given, an Enumerator is returned instead.

```
a = [ "a", "b", "c", "d" ]
a.map! {|x| x + "!" }
a #=>  [ "a!", "b!", "c!", "d!" ]
a.collect!.with_index {|x, i| x[0...i] }
a #=>  ["", "b", "c!", "d!"]
```

pack (aTemplateString) → aBinaryString
Packs the contents of *arr* into a binary sequence according to the directives in *aTemplateString* (see the table below) Directives “A,” “a,” and “Z” may be followed by a count, which gives the width of the resulting field. The remaining directives also may take a count, indicating the number of array elements to convert. If the count is an asterisk (“`*`”), all remaining array elements will be converted. Any of the directives “`sSiIlL`” may be followed by an underscore (“`_`”) or exclamation mark (“`!`”) to use the underlying platform’s native size for the specified type; otherwise, they use a platform-independent size. Spaces are ignored in the template string. See also `String#unpack`.

```
a = [ "a", "b", "c" ]
n = [ 65, 66, 67 ]
a.pack("A3A3A3")   #=> "a  b  c  "
```

```
a.pack("a3a3a3")   #=> "a\000\000b\000\000c\000\000"
n.pack("ccc")      #=> "ABC"
```

Directives for `pack`.

```
Integer      | Array   |
Directive    | Element | Meaning
---------------------------------------------------------------------------
   C         | Integer | 8-bit unsigned (unsigned char)
   S         | Integer | 16-bit unsigned, native endian (uint16_t)
   L         | Integer | 32-bit unsigned, native endian (uint32_t)
   Q         | Integer | 64-bit unsigned, native endian (uint64_t)
             |         |
   c         | Integer | 8-bit signed (signed char)
   s         | Integer | 16-bit signed, native endian (int16_t)
   l         | Integer | 32-bit signed, native endian (int32_t)
   q         | Integer | 64-bit signed, native endian (int64_t)
             |         |
   S_, S!    | Integer | unsigned short, native endian
   I, I_, I! | Integer | unsigned int, native endian
   L_, L!    | Integer | unsigned long, native endian
   Q_, Q!    | Integer | unsigned long long, native endian (ArgumentError
             |         | if the platform has no long long type.)
             |         | (Q_ and Q! is available since Ruby 2.1.)
             |         |
   s_, s!    | Integer | signed short, native endian
   i, i_, i! | Integer | signed int, native endian
   l_, l!    | Integer | signed long, native endian
   q_, q!    | Integer | signed long long, native endian (ArgumentError
             |         | if the platform has no long long type.)
             |         | (q_ and q! is available since Ruby 2.1.)
             |         |
   S> L> Q>  | Integer | same as the directives without ">" except
   s> l> q>  |         | big endian
   S!> I!>   |         | (available since Ruby 1.9.3)
   L!> Q!>   |         | "S>" is same as "n"
   s!> i!>   |         | "L>" is same as "N"
   l!> q!>   |         |
             |         |
   S< L< Q<  | Integer | same as the directives without "<" except
   s< l< q<  |         | little endian
   S!< I!<   |         | (available since Ruby 1.9.3)
   L!< Q!<   |         | "S<" is same as "v"
   s!< i!<   |         | "L<" is same as "V"
```

```
   l!< q!<   |         |
             |         |
   n         | Integer | 16-bit unsigned, network (big-endian) byte order
   N         | Integer | 32-bit unsigned, network (big-endian) byte order
   v         | Integer | 16-bit unsigned, VAX (little-endian) byte order
   V         | Integer | 32-bit unsigned, VAX (little-endian) byte order
             |         |
   U         | Integer | UTF-8 character
   w         | Integer | BER-compressed integer

Float        |         |
Directive    |         | Meaning
-------------------------------------------------------------------------
   D, d      | Float   | double-precision, native format
   F, f      | Float   | single-precision, native format
   E         | Float   | double-precision, little-endian byte order
   e         | Float   | single-precision, little-endian byte order
   G         | Float   | double-precision, network (big-endian) byte order
   g         | Float   | single-precision, network (big-endian) byte order

String       |         |
Directive    |         | Meaning
-------------------------------------------------------------------------
   A         | String  | arbitrary binary string (space padded, count is width)
   a         | String  | arbitrary binary string (null padded, count is width)
   Z         | String  | same as ``a'', except that null is added with *
   B         | String  | bit string (MSB first)
   b         | String  | bit string (LSB first)
   H         | String  | hex string (high nibble first)
   h         | String  | hex string (low nibble first)
   u         | String  | UU-encoded string
   M         | String  | quoted printable, MIME encoding (see RFC2045)
   m         | String  | base64 encoded string (see RFC 2045, count is width)
             |         | (if count is 0, no line feed are added, see RFC 4648)
   P         | String  | pointer to a structure (fixed-length string)
   p         | String  | pointer to a null-terminated string

Misc.        |         |
Directive    |         | Meaning
-------------------------------------------------------------------------
   @         | ---     | moves to absolute position
   X         | ---     | back up a byte
   x         | ---     | null byte
```

permutation { |p| block } → ary
permutation → Enumerator
permutation(n) { |p| block } → ary
permutation(n) → Enumerator
When invoked with a block, yield all permutations of length `n` of the elements of the array, then return the array itself.

If `n` is not specified, yield all permutations of all elements.

The implementation makes no guarantees about the order in which the permutations are yielded.

If no block is given, an Enumerator is returned instead.

Examples:

```
a = [1, 2, 3]
a.permutation.to_a    #=> [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
a.permutation(1).to_a #=> [[1],[2],[3]]
a.permutation(2).to_a #=> [[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]]
a.permutation(3).to_a #=> [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
a.permutation(0).to_a #=> [[]] # one permutation of length 0
a.permutation(4).to_a #=> []   # no permutations of length 4
```

pop → obj or nil
pop(n) → new_ary
Removes the last element from `self` and returns it, or `nil` if the array is empty.

If a number `n` is given, returns an array of the last `n` elements (or less) just like `array.slice!(-n, n)` does.
See also #push for the opposite effect.

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

product(other_ary, ...) → new_ary
product(other_ary, ...) { |p| block } → ary
Returns an array of all combinations of elements from all arrays.

The length of the returned array is the product of the length of `self` and the argument arrays.

If given a block, product will yield all combinations and return `self` instead.

```
[1,2,3].product([4,5])     #=> [[1,4],[1,5],[2,4],[2,5],[3,4],[3,5]]
[1,2].product([1,2])       #=> [[1,1],[1,2],[2,1],[2,2]]
```

```
[1,2].product([3,4],[5,6]) #=> [[1,3,5],[1,3,6],[1,4,5],[1,4,6],
                           #     [2,3,5],[2,3,6],[2,4,5],[2,4,6]]
[1,2].product()            #=> [[1],[2]]
[1,2].product([])          #=> []
```

push(obj, ...) → ary
Append — Pushes the given object(s) on to the end of this array. This expression returns the array itself, so several appends may be chained together. See also #pop for the opposite effect.

```
a = [ "a", "b", "c" ]
a.push("d", "e", "f")
        #=> ["a", "b", "c", "d", "e", "f"]
[1, 2, 3,].push(4).push(5)
        #=> [1, 2, 3, 4, 5]
```

rassoc(obj) → new_ary or nil
Searches through the array whose elements are also arrays.

Compares `obj` with the second element of each contained array using `obj.==`.

Returns the first contained array that matches `obj`.

See also #assoc.

```
a = [ [ 1, "one"], [2, "two"], [3, "three"], ["ii", "two"] ]
a.rassoc("two")    #=> [2, "two"]
a.rassoc("four")   #=> nil
```

reject {|item| block } → new_ary
reject → Enumerator
Returns a new array containing the items in `self` for which the given block is not `true`.

See also #delete_if

If no block is given, an Enumerator is returned instead.

reject! { |item| block } → ary or nil
reject! → Enumerator
Equivalent to #delete_if, deleting elements from `self` for which the block evaluates to `true`, but returns `nil` if no changes were made.

The array is changed instantly every time the block is called, not after the iteration is over.

See also Enumerable#reject and #delete_if.

If no block is given, an Enumerator is returned instead.

repeated_combination(n) { |c| block } → ary
repeated_combination(n) → Enumerator
When invoked with a block, yields all repeated combinations of length `n` of elements from the array and then returns the array itself.

The implementation makes no guarantees about the order in which the repeated combinations are yielded.

If no block is given, an Enumerator is returned instead.

Examples:

```
a = [1, 2, 3]
a.repeated_combination(1).to_a  #=> [[1], [2], [3]]
a.repeated_combination(2).to_a  #=> [[1,1],[1,2],[1,3],[2,2],[2,3],[3,3]]
a.repeated_combination(3).to_a  #=> [[1,1,1],[1,1,2],[1,1,3],[1,2,2],[1,2,3],
                                #    [1,3,3],[2,2,2],[2,2,3],[2,3,3],[3,3,3]]
a.repeated_combination(4).to_a  #=> [[1,1,1,1],[1,1,1,2],[1,1,1,3],[1,1,2,2],[1,1,2,3],
                                #    [1,1,3,3],[1,2,2,2],[1,2,2,3],[1,2,3,3],[1,3,3,3],
                                #    [2,2,2,2],[2,2,2,3],[2,2,3,3],[2,3,3,3],[3,3,3,3]]
a.repeated_combination(0).to_a  #=> [[]] # one combination of length 0
```

repeated_permutation(n) { |p| block } → ary
repeated_permutation(n) → Enumerator
When invoked with a block, yield all repeated permutations of length `n` of the elements of the array, then return the array itself.

The implementation makes no guarantees about the order in which the repeated permutations are yielded.

If no block is given, an Enumerator is returned instead.

Examples:

```
a = [1, 2]
a.repeated_permutation(1).to_a  #=> [[1], [2]]
a.repeated_permutation(2).to_a  #=> [[1,1],[1,2],[2,1],[2,2]]
a.repeated_permutation(3).to_a  #=> [[1,1,1],[1,1,2],[1,2,1],[1,2,2],
                                #    [2,1,1],[2,1,2],[2,2,1],[2,2,2]]
a.repeated_permutation(0).to_a  #=> [[]] # one permutation of length 0
```

replace(other_ary) → ary
Replaces the contents of `self` with the contents of `other_ary`, truncating or expanding if necessary.

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

```
a.replace([ "x", "y", "z" ])   #=> ["x", "y", "z"]
a                              #=> ["x", "y", "z"]
```

reverse → new_ary
Returns a new array containing `self`'s elements in reverse order.

```
[ "a", "b", "c" ].reverse   #=> ["c", "b", "a"]
[ 1 ].reverse               #=> [1]
```

reverse! → ary
reverse_each { |item| block } → ary
reverse_each → Enumerator
Same as #each, but traverses `self` in reverse order.

```
a = [ "a", "b", "c" ]
a.reverse_each {|x| print x, " " }
```

produces:

```
c b a
```

rindex(obj) → int or nil
rindex { |item| block } → int or nil
rindex → Enumerator
Returns the *index* of the last object in `self` `==` to `obj`.

If a block is given instead of an argument, returns the *index* of the first object for which the block returns `true`, starting from the last object.

Returns `nil` if no match is found.

See also #index.

If neither block nor argument is given, an Enumerator is returned instead.

```
a = [ "a", "b", "b", "b", "c" ]
a.rindex("b")             #=> 3
a.rindex("z")             #=> nil
a.rindex { |x| x == "b" } #=> 3
```

rotate(count=1) → new_ary
Returns a new array by rotating `self` so that the element at `count` is the first element of the new array.

If `count` is negative then it rotates in the opposite direction, starting from the end of `self` where `-1` is the last

element.

```
a = [ "a", "b", "c", "d" ]
a.rotate         #=> ["b", "c", "d", "a"]
a                #=> ["a", "b", "c", "d"]
a.rotate(2)      #=> ["c", "d", "a", "b"]
a.rotate(-3)     #=> ["b", "c", "d", "a"]
```

rotate!(count=1) → ary
Rotates `self` in place so that the element at `count` comes first, and returns `self`.

If `count` is negative then it rotates in the opposite direction, starting from the end of the array where `-1` is the last element.

```
a = [ "a", "b", "c", "d" ]
a.rotate!        #=> ["b", "c", "d", "a"]
a                #=> ["b", "c", "d", "a"]
a.rotate!(2)     #=> ["d", "a", "b", "c"]
a.rotate!(-3)    #=> ["a", "b", "c", "d"]
```

sample → obj
sample(random: rng) → obj
sample(n) → new_ary
sample(n, random: rng) → new_ary
Choose a random element or `n` random elements from the array.

The elements are chosen by using random and unique indices into the array in order to ensure that an element doesn't repeat itself unless the array already contained duplicate elements.

If the array is empty the first form returns `nil` and the second form returns an empty array.

The optional `rng` argument will be used as the random number generator.

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

select { |item| block } → new_ary
select → Enumerator
Returns a new array containing all elements of `ary` for which the given `block` returns a true value.

If no block is given, an Enumerator is returned instead.

```
[1,2,3,4,5].select { |num|  num.even?  }   #=> [2, 4]
```

```
a = %w{ a b c d e f }
a.select { |v| v =~ /[aeiou]/ }  #=> ["a", "e"]
```

See also Enumerable#select.

select! {|item| block } → ary or nil
select! → Enumerator
Invokes the given block passing in successive elements from `self`, deleting elements for which the block returns a `false` value.

If changes were made, it will return `self`, otherwise it returns `nil`.

See also #keep_if

If no block is given, an Enumerator is returned instead.

shift → obj or nil
shift(n) → new_ary
Removes the first element of `self` and returns it (shifting all other elements down by one). Returns `nil` if the array is empty.

If a number `n` is given, returns an array of the first `n` elements (or less) just like `array.slice!(0, n)` does. With `ary` containing only the remainder elements, not including what was shifted to `new_ary`. See also #unshift for the opposite effect.

```
args = [ "-m", "-q", "filename" ]
args.shift     #=> "-m"
args           #=> ["-q", "filename"]

args = [ "-m", "-q", "filename" ]
args.shift(2)  #=> ["-m", "-q"]
args           #=> ["filename"]
```

shuffle → new_ary
shuffle(random: rng) → new_ary
Returns a new array with elements of `self` shuffled.

```
a = [ 1, 2, 3 ]           #=> [1, 2, 3]
a.shuffle                 #=> [2, 3, 1]
```

The optional `rng` argument will be used as the random number generator.

```
a.shuffle(random: Random.new(1))  #=> [1, 3, 2]
```

shuffle! → ary
shuffle!(random: rng) → ary
Shuffles elements in `self` in place.

The optional `rng` argument will be used as the random number generator.

size()
Alias for: length
slice(index) → obj or nil
slice(start, length) → new_ary or nil
slice(range) → new_ary or nil
Element Reference — Returns the element at `index`, or returns a subarray starting at the `start` index and continuing for `length` elements, or returns a subarray specified by `range` of indices.

Negative indices count backward from the end of the array (-1 is the last element). For `start` and `range` cases the starting index is just before an element. Additionally, an empty array is returned when the starting index for an element range is at the end of the array.

Returns `nil` if the index (or starting index) are out of range.

```
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]               #=> []
```

slice!(index) → obj or nil
slice!(start, length) → new_ary or nil
slice!(range) → new_ary or nil
Deletes the element(s) given by an `index` (optionally up to `length` elements) or by a `range`.

Returns the deleted object (or objects), or `nil` if the `index` is out of range.

```
a = [ "a", "b", "c" ]
a.slice!(1)     #=> "b"
a               #=> ["a", "c"]
```

```
a.slice!(-1)    #=> "c"
a               #=> ["a"]
a.slice!(100)   #=> nil
a               #=> ["a"]
```

sort → new_ary
sort { |a, b| block } → new_ary
Returns a new array created by sorting `self`.

Comparisons for the sort will be done using the `<=>` operator or using an optional code block.

The block must implement a comparison between `a` and `b`, and return `-1`, when `a` follows `b`, `0` when `a` and `b` are equivalent, or +`1` if `b` follows `a`.

See also Enumerable#sort_by.

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

sort! → ary
sort! { |a, b| block } → ary
Sorts `self` in place.

Comparisons for the sort will be done using the `<=>` operator or using an optional code block.

The block must implement a comparison between `a` and `b`, and return `-1`, when `a` follows `b`, `0` when `a` and `b` are equivalent, or +`1` if `b` follows `a`.

See also Enumerable#sort_by.

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

sort_by! { |obj| block } → ary
sort_by! → Enumerator
Sorts `self` in place using a set of keys generated by mapping the values in `self` through the given block.

If no block is given, an Enumerator is returned instead.

take(n) → new_ary
Returns first `n` elements from the array.

If a negative number is given, raises an ArgumentError.

See also #drop

```
a = [1, 2, 3, 4, 5, 0]
a.take(3)             #=> [1, 2, 3]
```

take_while { |arr| block } → new_ary
take_while → Enumerator
Passes elements to the block until the block returns `nil` or `false`, then stops iterating and returns an array of all prior elements.

If no block is given, an Enumerator is returned instead.

See also #drop_while

```
a = [1, 2, 3, 4, 5, 0]
a.take_while { |i| i < 3 }  #=> [1, 2]
```

to_a → ary