Home

Pages Classes Methods

Search

Table of Contents

Show/hide navigation

Parent

Object

Included Modules

Enumerable

Methods

class Array

An Array is an ordered, integer-indexed collection of objects, called *elements*. Any object (even another array) may be an array element, and an array can contain objects of different types.

Array Indexes

Array indexing starts at 0, as in C or Java.

A positive index is an offset from the first element:

- Index 0 indicates the first element.
- Index 1 indicates the second element.
- ...

A negative index is an offset, backwards, from the end of the array:

- Index -1 indicates the last element.
- Index -2 indicates the next-to-last element.
- ...

A non-negative index is *in range* if and only if it is smaller than the size of the array. For a 3-element array:

- Indexes 0 through 2 are in range.
- Index 3 is out of range.

A negative index is *in range* if and only if its absolute value is not larger than the size of the array. For a 3-element array:

- Indexes -1 through -3 are in range.
- Index -4 is out of range.

Although the effective index into an array is always an integer, some methods (both within and outside of class Array) accept one or more non-integer arguments that are integer-convertible objects.

Creating Arrays

You can create an Array object explicitly with:

- An array literal:

```
[1, 'one', :one, [2, 'two', :two]]
```

- A array literal:

```
%w[foo bar baz] # => ["foo", "bar", "baz"]
%w[1 % *]       # => ["1", "%", "*"]
```

- A array literal:

```
%i[foo bar baz] # => [:foo, :bar, :baz]
%i[1 % *]       # => [:"1", :%, :*]
```

- Method Kernel#Array:

```
Array(["a", "b"])             # => ["a", "b"]
Array(1..5)                   # => [1, 2, 3, 4, 5]
Array(key: :value)            # => [[:key, :value]]
Array(nil)                    # => []
Array(1)                      # => [1]
Array({:a => "a", :b => "b"}) # => [[:a, "a"], [:b, "b"]]
```

- Method Array.new:

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

  Note that the last example above populates the array with references to the same object. This is recommended only in cases where that object is a natively immutable object such as a symbol, a numeric, `nil`, `true`, or `false`.

  Another way to create an array with various objects, using a block; this usage is safe for mutable objects such as hashes, strings or other arrays:

```
Array.new(4) {|i| i.to_s } # => ["0", "1", "2", "3"]
```

  Here is a way to create a multi-dimensional array:

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

A number of Ruby methods, both in the core and in the standard library, provide instance method `to_a`, which converts an object to an array.

- ARGF#to_a
- Array#to_a
- Enumerable#to_a

- Hash#to_a
- MatchData#to_a
- NilClass#to_a
- OptionParser#to_a
- Range#to_a
- Set#to_a
- Struct#to_a
- Time#to_a
- Benchmark::Tms#to_a
- CSV::Table#to_a
- Enumerator::Lazy#to_a
- Gem::List#to_a
- Gem::NameTuple#to_a
- Gem::Platform#to_a
- Gem::RequestSet::Lockfile::Tokenizer#to_a
- Gem::SourceList#to_a
- OpenSSL::X509::Extension#to_a
- OpenSSL::X509::Name#to_a
- Racc::ISet#to_a
- Rinda::RingFinger#to_a
- Ripper::Lexer::Elem#to_a
- RubyVM::InstructionSequence#to_a
- YAML::DBM#to_a

Example Usage

In addition to the methods it mixes in through the Enumerable module, the Array class has proprietary methods for accessing, searching and otherwise manipulating arrays.

Some of the more common ones are illustrated below.

Accessing Elements

Elements in an array can be retrieved using the Array#[] method. It can take a single integer argument (a numeric index), a pair of arguments (start and length) or a range. Negative indices start counting from the end, with -1 being the last element.

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

Another way to access a particular array element is by using the `at` method

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

The `slice` method works in an identical manner to `Array#[]`.

To raise an error for indices outside of the array bounds or else to provide a default value when that happens, you can use `fetch`.

```
arr = ['a', 'b', 'c', 'd', 'e', 'f']
arr.fetch(100) #=> IndexError: index 100 outside of array bounds: -6...6
arr.fetch(100, "oops") #=> "oops"
```

The special methods `first` and `last` will return the first and last elements of an array, respectively.

```
arr.first #=> 1
arr.last  #=> 6
```

To return the first `n` elements of an array, use `take`

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

`drop` does the opposite of `take`, by returning the elements after `n` elements have been dropped:

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

Obtaining Information about an Array

Arrays keep track of their own length at all times. To query an array about the number of elements it contains, use `length`, `count` or `size`.

```
browsers = ['Chrome', 'Firefox', 'Safari', 'Opera', 'IE']
browsers.length #=> 5
browsers.count #=> 5
```

To check whether an array contains any elements at all

```
browsers.empty? #=> false
```

To check whether a particular item is included in the array

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

Adding Items to Arrays

Items can be added to the end of an array by using either `push` or `<<`

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

`unshift` will add a new item to the beginning of an array.

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

With `insert` you can add a new element to an array at any position.

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

Using the `insert` method, you can also insert multiple values at once:

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

Removing Items from an Array

The method `pop` removes the last element in an array and returns it:

```
arr =  [1, 2, 3, 4, 5, 6]
arr.pop #=> 6
arr #=> [1, 2, 3, 4, 5]
```

To retrieve and at the same time remove the first item, use `shift` :

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

To delete an element at a particular index:

```
arr.delete_at(2) #=> 4
arr #=> [2, 3, 5]
```

To delete a particular element anywhere in an array, use `delete`:

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

A useful method if you need to remove `nil` values from an array is `compact`:

```
arr = ['foo', 0, nil, 'bar', 7, 'baz', nil]
arr.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']
```

Another common need is to remove duplicate elements from an array.

It has the non-destructive `uniq`, and destructive method `uniq!`

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

Iterating over Arrays

Like all classes that include the `Enumerable` module, Array has an each method, which defines what elements should be iterated over and how. In case of Array's `each`, all elements in the Array instance are yielded to the supplied block in sequence.

Note that this operation leaves the array unchanged.

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

Another sometimes useful iterator is `reverse_each` which will iterate over the elements in the array in reverse order.

```
words = %w[first second third fourth fifth sixth]
str = ""
words.reverse_each {|word| str += "#{word} "}
p str #=> "sixth fifth fourth third second first "
```

The `map` method can be used to create a new array based on the original array, but with the values modified by the supplied block:

```
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                    #=> [1, 4, 9, 16, 25]
```

Selecting Items from an Array

Elements can be selected from an array according to criteria defined in a block. The selection can happen in a destructive or a non-destructive manner. While the destructive operations will modify the array they were called on, the non-destructive methods usually return a new array with the selected elements, but leave the original array unchanged.

Non-destructive Selection

```
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]
```

Destructive Selection

`select!` and `reject!` are the corresponding destructive methods to `select` and `reject`

Similar to `select` vs. `reject`, `delete_if` and `keep_if` have the exact opposite result when supplied with the same block:

```
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]
```

What's Here

First, what's elsewhere. Class Array:

- Inherits from class Object.
- Includes module Enumerable, which provides dozens of additional methods.

Here, class Array provides methods that are useful for:

- Creating an Array
- Querying
- Comparing
- Fetching
- Assigning
- Deleting
- Combining
- Iterating
- Converting
- And more....

Methods for Creating an Array

- `::[]` : Returns a new array populated with given objects.
- `::new` : Returns a new array.
- `::try_convert` : Returns a new array created from a given object.

Methods for Querying

- `length` , `size` : Returns the count of elements.
- `include?` : Returns whether any element `==` a given object.
- `empty?` : Returns whether there are no elements.
- `all?` : Returns whether all elements meet a given criterion.
- `any?` : Returns whether any element meets a given criterion.
- `none?` : Returns whether no element `==` a given object.
- `one?` : Returns whether exactly one element `==` a given object.
- `count` : Returns the count of elements that meet a given criterion.
- `find_index` , `index` : Returns the index of the first element that meets a given criterion.
- `rindex` : Returns the index of the last element that meets a given criterion.
- `hash` : Returns the integer hash code.

Methods for Comparing

- #<=>: Returns -1, 0, or 1 * as `self` is less than, equal to, or greater than a given object.

- `==` : Returns whether each element in `self` is `==` to the corresponding element in a given object.
- `eql?` : Returns whether each element in `self` is `eql?` to the corresponding element in a given object.

Methods for Fetching

These methods do not modify `self` .

- `[]` : Returns one or more elements.
- `fetch` : Returns the element at a given offset.
- `first` : Returns one or more leading elements.
- `last` : Returns one or more trailing elements.
- `max` : Returns one or more maximum-valued elements, as determined by `<=>` or a given block.
- `min` : Returns one or more minimum-valued elements, as determined by `<=>` or a given block.
- `minmax` : Returns the minimum-valued and maximum-valued elements, as determined by `<=>` or a given block.
- `assoc` : Returns the first element that is an array whose first element `==` a given object.
- `rassoc` : Returns the first element that is an array whose second element `==` a given object.
- `at` : Returns the element at a given offset.
- `values_at` : Returns the elements at given offsets.
- `dig` : Returns the object in nested objects that is specified by a given index and additional arguments.
- `drop` : Returns trailing elements as determined by a given index.
- `take` : Returns leading elements as determined by a given index.
- `drop_while` : Returns trailing elements as determined by a given block.
- `take_while` : Returns leading elements as determined by a given block.
- `slice` : Returns consecutive elements as determined by a given argument.
- `sort` : Returns all elements in an order determined by `<=>` or a given block.
- `reverse` : Returns all elements in reverse order.
- `compact` : Returns an array containing all non-`nil` elements.
- `select` , `filter` : Returns an array containing elements selected by a given block.
- `uniq` : Returns an array containing non-duplicate elements.

- `rotate` : Returns all elements with some rotated from one end to the other.
- `bsearch` : Returns an element selected via a binary search as determined by a given block.
- `bsearch_index` : Returns the index of an element selected via a binary search as determined by a given block.
- `sample` : Returns one or more random elements.
- `shuffle` : Returns elements in a random order.

Methods for Assigning

These methods add, replace, or reorder elements in `self` .

- `[]=` : Assigns specified elements with a given object.
- `push` , `append` , `<<` : Appends trailing elements.
- `unshift` , `prepend` : Prepends leading elements.
- `insert` : Inserts given objects at a given offset; does not replace elements.
- `concat` : Appends all elements from given arrays.
- `fill` : Replaces specified elements with specified objects.
- `replace` : Replaces the content of `self` with the content of a given array.
- `reverse!` : Replaces `self` with its elements reversed.
- `rotate!` : Replaces `self` with its elements rotated.
- `shuffle!` : Replaces `self` with its elements in random order.
- `sort!` : Replaces `self` with its elements sorted, as determined by `<=>` or a given block.
- `sort_by!` : Replaces `self` with its elements sorted, as determined by a given block.

Methods for Deleting

Each of these methods removes elements from `self` :

- `pop` : Removes and returns the last element.
- `shift` : Removes and returns the first element.
- `compact!` : Removes all `nil` elements.
- `delete` : Removes elements equal to a given object.
- `delete_at` : Removes the element at a given offset.
- `delete_if` : Removes elements specified by a given block.
- `keep_if` : Removes elements not specified by a given block.

- `reject!` : Removes elements specified by a given block.
- `select!` , `filter!` : Removes elements not specified by a given block.
- `slice!` : Removes and returns a sequence of elements.
- `uniq!` : Removes duplicates.

Methods for Combining

- #&: Returns an array containing elements found both in `self` and a given array.
- `intersection` : Returns an array containing elements found both in `self` and in each given array.
- `+` : Returns an array containing all elements of `self` followed by all elements of a given array.
- `-` : Returns an array containing all elements of `self` that are not found in a given array.
- #|: Returns an array containing all elements of `self` and all elements of a given array, duplicates removed.
- `union` : Returns an array containing all elements of `self` and all elements of given arrays, duplicates removed.
- `difference` : Returns an array containing all elements of `self` that are not found in any of the given arrays..
- `product` : Returns or yields all combinations of elements from `self` and given arrays.

Methods for Iterating

- `each` : Passes each element to a given block.
- `reverse_each` : Passes each element, in reverse order, to a given block.
- `each_index` : Passes each element index to a given block.
- `cycle` : Calls a given block with each element, then does so again, for a specified number of times, or forever.
- `combination` : Calls a given block with combinations of elements of `self` ; a combination does not use the same element more than once.
- `permutation` : Calls a given block with permutations of elements of `self` ; a permutation does not use the same element more than once.
- `repeated_combination` : Calls a given block with combinations of elements of `self` ; a combination may use the same element more than once.
- `repeated_permutation` : Calls a given block with permutations of elements of `self` ; a permutation may use the same element more than once.

Methods for Converting

- `map`, `collect`: Returns an array containing the block return-value for each element.
- `map!`, `collect!`: Replaces each element with a block return-value.
- `flatten`: Returns an array that is a recursive flattening of `self`.
- `flatten!`: Replaces each nested array in `self` with the elements from that array.
- `inspect`, `to_s`: Returns a new `String` containing the elements.
- `join`: Returns a newsString containing the elements joined by the field separator.
- `to_a`: Returns `self` or a new array containing all elements.
- `to_ary`: Returns `self`.
- `to_h`: Returns a new hash formed from the elements.
- `transpose`: Transposes `self`, which must be an array of arrays.
- `zip`: Returns a new array of arrays containing `self` and given arrays; follow the link for details.

Other Methods

- `*`: Returns one of the following:
  - With integer argument `n`, a new array that is the concatenation of `n` copies of `self`.
  - With string argument `field_separator`, a new string that is equivalent to `join(field_separator)`.
- abbrev: Returns a hash of unambiguous abbreviations for elements.
- `pack`: Packs the elements into a binary sequence.
- `sum`: Returns a sum of elements according to either `+` or a given block.

Public Class Methods

`[](*args)`

Returns a new array populated with the given objects.

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

new → new_empty_array
new(array) → new_array
new(size) → new_array
new(size, default_value) → new_array
new(size) {|index| ... } → new_array

Returns a new Array.

With no block and no arguments, returns a new empty Array object.

With no block and a single Array argument `array`, returns a new Array formed from `array`:

```
a = Array.new([:foo, 'bar', 2])
a.class # => Array
a # => [:foo, "bar", 2]
```

With no block and a single Integer argument `size`, returns a new Array of the given size whose elements are all `nil`:

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

With no block and arguments `size` and `default_value`, returns an Array of the given size; each element is that same `default_value`:

```
a = Array.new(3, 'x')
a # => ['x', 'x', 'x']
```

With a block and argument `size`, returns an Array of the given size; the block is called with each successive integer `index`; the element for that `index` is the return value from the block:

```
a = Array.new(3) {|index| "Element #{index}" }
a # => ["Element 0", "Element 1", "Element 2"]
```

Raises ArgumentError if `size` is negative.

With a block and no argument, or a single argument `0`, ignores the block and returns a new empty Array.

try_convert(object) → object, new_array, or nil

If `object` is an Array object, returns `object`.

Otherwise if `object` responds to `:to_ary`, calls `object.to_ary` and returns the result.

Returns `nil` if `object` does not respond to `:to_ary`

Raises an exception unless `object.to_ary` returns an Array object.

Public Instance Methods

`array & other_array → new_array`

Returns a new Array containing each element found in both `array` and Array `other_array`; duplicates are omitted; items are compared using `eql?` (items must also implement `hash` correctly):

```
[0, 1, 2, 3] & [1, 2] # => [1, 2]
[0, 1, 0, 1] & [0, 1] # => [0, 1]
```

Preserves order from `array`:

```
[0, 1, 2] & [3, 2, 1, 0] # => [0, 1, 2]
```

Related: `Array#intersection`.

`array * n → new_array`
`array * string_separator → new_string`

When non-negative argument `Integer` `n` is given, returns a new Array built by concatenating the `n` copies of `self`:

```
a = ['x', 'y']
a * 3 # => ["x", "y", "x", "y", "x", "y"]
```

When `String` argument `string_separator` is given, equivalent to `array.join(string_separator)`:

```
[0, [0, 1], {foo: 0}] * ', ' # => "0, 0, 1, {:foo=>0}"
```

`array + other_array → new_array`

Returns a new Array containing all elements of `array` followed by all elements of `other_array`:

```
a = [0, 1] + [2, 3]
a # => [0, 1, 2, 3]
```

Related: `concat`.

array - other_array → new_array

Returns a new Array containing only those elements from `array` that are not found in Array `other_array`; items are compared using `eql?`; the order from `array` is preserved:

```
[0, 1, 1, 2, 1, 1, 3, 1, 1] - [1] # => [0, 2, 3]
[0, 1, 2, 3] - [3, 0] # => [1, 2]
[0, 1, 2] - [4] # => [0, 1, 2]
```

Related: Array#difference.

array << object → self

Appends `object` to `self`; returns `self`:

```
a = [:foo, 'bar', 2]
a << :baz # => [:foo, "bar", 2, :baz]
```

Appends `object` as one element, even if it is another Array:

```
a = [:foo, 'bar', 2]
a1 = a << [3, 4]
a1 # => [:foo, "bar", 2, [3, 4]]
```

array <=> other_array → -1, 0, or 1

Returns -1, 0, or 1 as `self` is less than, equal to, or greater than `other_array`. For each index `i` in `self`, evaluates `result = self[i] <=> other_array[i]`.

Returns -1 if any result is -1:

```
[0, 1, 2] <=> [0, 1, 3] # => -1
```

Returns 1 if any result is 1:

```
[0, 1, 2] <=> [0, 1, 1] # => 1
```

When all results are zero:

- Returns -1 if `array` is smaller than `other_array`:

  ```
  [0, 1, 2] <=> [0, 1, 2, 3] # => -1
  ```

- Returns 1 if `array` is larger than `other_array`:

```
[0, 1, 2] <=> [0, 1] # => 1
```

- Returns 0 if `array` and `other_array` are the same size:

```
[0, 1, 2] <=> [0, 1, 2] # => 0
```

`array == other_array → true or false`

Returns `true` if both `array.size == other_array.size` and for each index `i` in `array`, `array[i] == other_array[i]`:

```
a0 = [:foo, 'bar', 2]
a1 = [:foo, 'bar', 2.0]
a1 == a0 # => true
[] == [] # => true
```

Otherwise, returns `false`.

This method is different from method `Array#eql?`, which compares elements using `Object#eql?`.

`array[index] → object or nil`
`array[start, length] → object or nil`
`array[range] → object or nil`
`array[aseq] → object or nil`

Returns elements from `self`; does not modify `self`.

When a single `Integer` argument `index` is given, returns the element at offset `index`:

```
a = [:foo, 'bar', 2]
a[0] # => :foo
a[2] # => 2
a # => [:foo, "bar", 2]
```

If `index` is negative, counts relative to the end of `self`:

```
a = [:foo, 'bar', 2]
a[-1] # => 2
a[-2] # => "bar"
```

If `index` is out of range, returns `nil`.

When two `Integer` arguments `start` and `length` are given, returns a new Array of size `length` containing successive elements beginning at offset `start`:

```
a = [:foo, 'bar', 2]
a[0, 2] # => [:foo, "bar"]
a[1, 2] # => ["bar", 2]
```

If `start + length` is greater than `self.length`, returns all elements from offset `start` to the end:

```
a = [:foo, 'bar', 2]
a[0, 4] # => [:foo, "bar", 2]
a[1, 3] # => ["bar", 2]
a[2, 2] # => [2]
```

If `start == self.size` and `length >= 0`, returns a new empty Array.

If `length` is negative, returns `nil`.

When a single Range argument `range` is given, treats `range.min` as `start` above and `range.size` as `length` above:

```
a = [:foo, 'bar', 2]
a[0..1] # => [:foo, "bar"]
a[1..2] # => ["bar", 2]
```

Special case: If `range.start == a.size`, returns a new empty Array.

If `range.end` is negative, calculates the end index from the end:

```
a = [:foo, 'bar', 2]
a[0..-1] # => [:foo, "bar", 2]
a[0..-2] # => [:foo, "bar"]
a[0..-3] # => [:foo]
```

If `range.start` is negative, calculates the start index from the end:

```
a = [:foo, 'bar', 2]
a[-1..2] # => [2]
a[-2..2] # => ["bar", 2]
a[-3..2] # => [:foo, "bar", 2]
```

If `range.start` is larger than the array size, returns `nil`.

```
a = [:foo, 'bar', 2]
a[4..1] # => nil
a[4..0] # => nil
a[4..-1] # => nil
```

When a single Enumerator::ArithmeticSequence argument `aseq` is given, returns an Array of elements corresponding to the indexes produced by the sequence.

```
a = ['--', 'data1', '--', 'data2', '--', 'data3']
```

```
a[(1..).step(2)] # => ["data1", "data2", "data3"]
```

Unlike slicing with range, if the start or the end of the arithmetic sequence is larger than array size, throws `RangeError`.

```
a = ['--', 'data1', '--', 'data2', '--', 'data3']
a[(1..11).step(2)]
# RangeError (((1..11).step(2)) out of range)
a[(7..).step(2)]
# RangeError (((7..).step(2)) out of range)
```

If given a single argument, and its type is not one of the listed, tries to convert it to `Integer`, and raises if it is impossible:

```
a = [:foo, 'bar', 2]
# Raises TypeError (no implicit conversion of Symbol into Integer):
a[:foo]
```

Also aliased as: slice

`array[index] = object → object`
`array[start, length] = object → object`
`array[range] = object → object`

Assigns elements in `self`; returns the given `object`.

When `Integer` argument `index` is given, assigns `object` to an element in `self`.

If `index` is non-negative, assigns `object` the element at offset `index`:

```
a = [:foo, 'bar', 2]
a[0] = 'foo' # => "foo"
a # => ["foo", "bar", 2]
```

If `index` is greater than `self.length`, extends the array:

```
a = [:foo, 'bar', 2]
a[7] = 'foo' # => "foo"
a # => [:foo, "bar", 2, nil, nil, nil, nil, "foo"]
```

If `index` is negative, counts backwards from the end of the array:

```
a = [:foo, 'bar', 2]
a[-1] = 'two' # => "two"
a # => [:foo, "bar", "two"]
```

When `Integer` arguments `start` and `length` are given and `object` is not an Array, removes `length - 1` elements beginning at offset `start`, and assigns

`object` at offset `start`:

```
a = [:foo, 'bar', 2]
a[0, 2] = 'foo' # => "foo"
a # => ["foo", 2]
```

If `start` is negative, counts backwards from the end of the array:

```
a = [:foo, 'bar', 2]
a[-2, 2] = 'foo' # => "foo"
a # => [:foo, "foo"]
```

If `start` is non-negative and outside the array (`>= self.size`), extends the array with `nil`, assigns `object` at offset `start`, and ignores `length`:

```
a = [:foo, 'bar', 2]
a[6, 50] = 'foo' # => "foo"
a # => [:foo, "bar", 2, nil, nil, nil, "foo"]
```

If `length` is zero, shifts elements at and following offset `start` and assigns `object` at offset `start`:

```
a = [:foo, 'bar', 2]
a[1, 0] = 'foo' # => "foo"
a # => [:foo, "foo", "bar", 2]
```

If `length` is too large for the existing array, does not extend the array:

```
a = [:foo, 'bar', 2]
a[1, 5] = 'foo' # => "foo"
a # => [:foo, "foo"]
```

When Range argument `range` is given and `object` is an Array, removes `length - 1` elements beginning at offset `start`, and assigns `object` at offset `start`:

```
a = [:foo, 'bar', 2]
a[0..1] = 'foo' # => "foo"
a # => ["foo", 2]
```

if `range.begin` is negative, counts backwards from the end of the array:

```
a = [:foo, 'bar', 2]
a[-2..2] = 'foo' # => "foo"
a # => [:foo, "foo"]
```

If the array length is less than `range.begin`, assigns `object` at offset `range.begin`, and ignores `length`:

```
a = [:foo, 'bar', 2]
a[6..50] = 'foo' # => "foo"
a # => [:foo, "bar", 2, nil, nil, nil, "foo"]
```

If `range.end` is zero, shifts elements at and following offset `start` and assigns `object` at offset `start`:

```
a = [:foo, 'bar', 2]
a[1..0] = 'foo' # => "foo"
a # => [:foo, "foo", "bar", 2]
```

If `range.end` is negative, assigns `object` at offset `start`, retains `range.end.abs -1` elements past that, and removes those beyond:

```
a = [:foo, 'bar', 2]
a[1..-1] = 'foo' # => "foo"
a # => [:foo, "foo"]
a = [:foo, 'bar', 2]
a[1..-2] = 'foo' # => "foo"
a # => [:foo, "foo", 2]
a = [:foo, 'bar', 2]
a[1..-3] = 'foo' # => "foo"
a # => [:foo, "foo", "bar", 2]
a = [:foo, 'bar', 2]
```

If `range.end` is too large for the existing array, replaces array elements, but does not extend the array with `nil` values:

```
a = [:foo, 'bar', 2]
a[1..5] = 'foo' # => "foo"
a # => [:foo, "foo"]
```

`all? → true or false`
`all? {|element| ... } → true or false`
`all?(obj) → true or false`

Returns `true` if all elements of `self` meet a given criterion.

If `self` has no element, returns `true` and argument or block are not used.

With no block given and no argument, returns `true` if `self` contains only truthy elements, `false` otherwise:

```
[0, 1, :foo].all? # => true
[0, nil, 2].all? # => false
[].all? # => true
```

With a block given and no argument, calls the block with each element in `self`; returns `true` if the block returns only truthy values, `false` otherwise:

```
[0, 1, 2].all? { |element| element < 3 } # => true
[0, 1, 2].all? { |element| element < 2 } # => false
```

If argument `obj` is given, returns `true` if `obj.===` every element, `false` otherwise:

```
['food', 'fool', 'foot'].all?(/foo/) # => true
['food', 'drink'].all?(/bar/) # => false
[].all?(/foo/) # => true
[0, 0, 0].all?(0) # => true
[0, 1, 2].all?(1) # => false
```

Related: Enumerable#all?

any? → true or false
any? {|element| ... } → true or false
any?(obj) → true or false

Returns `true` if any element of `self` meets a given criterion.

If `self` has no element, returns `false` and argument or block are not used.

With no block given and no argument, returns `true` if `self` has any truthy element, `false` otherwise:

```
[nil, 0, false].any? # => true
[nil, false].any? # => false
[].any? # => false
```

With a block given and no argument, calls the block with each element in `self`; returns `true` if the block returns any truthy value, `false` otherwise:

```
[0, 1, 2].any? {|element| element > 1 } # => true
[0, 1, 2].any? {|element| element > 2 } # => false
```

If argument `obj` is given, returns `true` if `obj.===` any element, `false` otherwise:

```
['food', 'drink'].any?(/foo/) # => true
['food', 'drink'].any?(/bar/) # => false
[].any?(/foo/) # => false
[0, 1, 2].any?(1) # => true
[0, 1, 2].any?(3) # => false
```

Related: Enumerable#any?

append(*args)

Appends trailing elements.

Appends each argument in `objects` to `self`; returns `self`:

```
a = [:foo, 'bar', 2]
a.push(:baz, :bat) # => [:foo, "bar", 2, :baz, :bat]
```

Appends each argument as one element, even if it is another Array:

```
a = [:foo, 'bar', 2]
a1 = a.push([:baz, :bat], [:bam, :bad])
a1 # => [:foo, "bar", 2, [:baz, :bat], [:bam, :bad]]
```

Related: pop, shift, unshift.

Alias for: push

assoc(obj) → found_array or nil

Returns the first element in `self` that is an Array whose first element `==` `obj`:

```
a = [{foo: 0}, [2, 4], [4, 5, 6], [4, 5]]
a.assoc(4) # => [4, 5, 6]
```

Returns `nil` if no such element is found.

Related: rassoc.

at(index) → object

Returns the element at Integer offset `index`; does not modify `self`.

```
a = [:foo, 'bar', 2]
a.at(0) # => :foo
a.at(2) # => 2
```

bsearch {|element| ... } → object
bsearch → new_enumerator

Returns an element from `self` selected by a binary search.

See Binary Searching.

bsearch_index {|element| ... } → integer or nil
bsearch_index → new_enumerator

Searches `self` as described at method `bsearch`, but returns the *index* of the found element instead of the element itself.

`clear → self`

Removes all elements from `self`:

```
a = [:foo, 'bar', 2]
a.clear # => []
```

`collect -> new_enumerator`

Calls the block, if given, with each element of `self`; returns a new Array whose elements are the return values from the block:

```
a = [:foo, 'bar', 2]
a1 = a.map {|element| element.class }
a1 # => [Symbol, String, Integer]
```

Returns a new `Enumerator` if no block given:

```
a = [:foo, 'bar', 2]
a1 = a.map
a1 # => #<Enumerator: [:foo, "bar", 2]:map>
```

Also aliased as: map

`collect! -> new_enumerator`

Calls the block, if given, with each element; replaces the element with the block's return value:

```
a = [:foo, 'bar', 2]
a.map! { |element| element.class } # => [Symbol, String, Integer]
```

Returns a new `Enumerator` if no block given:

```
a = [:foo, 'bar', 2]
a1 = a.map!
a1 # => #<Enumerator: [:foo, "bar", 2]:map!>
```

Also aliased as: map!

combination(n) {|element| ... } → self
combination(n) → new_enumerator

Calls the block, if given, with combinations of elements of `self`; returns `self`. The order of combinations is indeterminate.

When a block and an in-range positive [Integer](#) argument `n` (`0 < n <= self.size`) are given, calls the block with all `n`-tuple combinations of `self`.

Example:

```
a = [0, 1, 2]
a.combination(2) {|combination| p combination }
```

Output:

```
[0, 1]
[0, 2]
[1, 2]
```

Another example:

```
a = [0, 1, 2]
a.combination(3) {|combination| p combination }
```

Output:

```
[0, 1, 2]
```

When `n` is zero, calls the block once with a new empty Array:

```
a = [0, 1, 2]
a1 = a.combination(0) {|combination| p combination }
```

Output:

```
[]
```

When `n` is out of range (negative or larger than `self.size`), does not call the block:

```
a = [0, 1, 2]
a.combination(-1) {|combination| fail 'Cannot happen' }
a.combination(4) {|combination| fail 'Cannot happen' }
```

Returns a new [Enumerator](#) if no block given:

```
a = [0, 1, 2]
```

```
a.combination(2) # => #<Enumerator: [0, 1, 2]:combination(2)>
```

compact → new_array

Returns a new Array containing all non-`nil` elements from `self`:

```
a = [nil, 0, nil, 1, nil, 2, nil]
a.compact # => [0, 1, 2]
```

compact! → self or nil

Removes all `nil` elements from `self`.

Returns `self` if any elements removed, otherwise `nil`.

concat(*other_arrays) → self

Adds to `array` all elements from each Array in `other_arrays`; returns `self`:

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

count → an_integer
count(obj) → an_integer
count {|element| ... } → an_integer

Returns a count of specified elements.

With no argument and no block, returns the count of all elements:

```
[0, 1, 2].count # => 3
[].count # => 0
```

With argument `obj`, returns the count of elements `==` to `obj`:

```
[0, 1, 2, 0.0].count(0) # => 2
[0, 1, 2].count(3) # => 0
```

With no argument and a block given, calls the block with each element; returns the count of elements for which the block returns a truthy value:

```
[0, 1, 2, 3].count {|element| element > 1} # => 2
```

With argument `obj` and a block given, issues a warning, ignores the block, and returns the count of elements `==` to `obj`.

`cycle {|element| ... } → nil`
`cycle(count) {|element| ... } → nil`
`cycle → new_enumerator`
`cycle(count) → new_enumerator`

When called with positive `Integer` argument `count` and a block, calls the block with each element, then does so again, until it has done so `count` times; returns `nil`:

```
output = []
[0, 1].cycle(2) {|element| output.push(element) } # => nil
output # => [0, 1, 0, 1]
```

If `count` is zero or negative, does not call the block:

```
[0, 1].cycle(0) {|element| fail 'Cannot happen' } # => nil
[0, 1].cycle(-1) {|element| fail 'Cannot happen' } # => nil
```

When a block is given, and argument is omitted or `nil`, cycles forever:

```
# Prints 0 and 1 forever.
[0, 1].cycle {|element| puts element }
[0, 1].cycle(nil) {|element| puts element }
```

When no block is given, returns a new Enumerator:

```
[0, 1].cycle(2) # => #<Enumerator: [0, 1]:cycle(2)>
[0, 1].cycle # => # => #<Enumerator: [0, 1]:cycle>
[0, 1].cycle.first(5) # => [0, 1, 0, 1, 0]
```

`deconstruct()`

`delete(obj) → deleted_object`
`delete(obj) {|nosuch| ... } → deleted_object or block_return`

Removes zero or more elements from `self`.

When no block is given, removes from `self` each element `ele` such that `ele == obj`; returns the last deleted element:

```
s1 = 'bar'; s2 = 'bar'
a = [:foo, s1, 2, s2]
```

```
a.delete('bar') # => "bar"
a # => [:foo, 2]
```

Returns `nil` if no elements removed.

When a block is given, removes from `self` each element `ele` such that `ele == obj`.

If any such elements are found, ignores the block and returns the last deleted element:

```
s1 = 'bar'; s2 = 'bar'
a = [:foo, s1, 2, s2]
deleted_obj = a.delete('bar') {|obj| fail 'Cannot happen' }
a # => [:foo, 2]
```

If no such elements are found, returns the block's return value:

```
a = [:foo, 'bar', 2]
a.delete(:nosuch) {|obj| "#{obj} not found" } # => "nosuch not found"
```

delete_at(index) → deleted_object or nil

Deletes an element from `self`, per the given Integer `index`.

When `index` is non-negative, deletes the element at offset `index`:

```
a = [:foo, 'bar', 2]
a.delete_at(1) # => "bar"
a # => [:foo, 2]
```

If index is too large, returns `nil`.

When `index` is negative, counts backward from the end of the array:

```
a = [:foo, 'bar', 2]
a.delete_at(-2) # => "bar"
a # => [:foo, 2]
```

If `index` is too small (far from zero), returns nil.

delete_if {|element| ... } → self
delete_if → Enumerator

Removes each element in `self` for which the block returns a truthy value; returns `self`:

```
a = [:foo, 'bar', 2, 'bat']
```

```
a.delete_if {|element| element.to_s.start_with?('b') } # => [:foo, 2]
```

Returns a new `Enumerator` if no block given:

```
a = [:foo, 'bar', 2]
a.delete_if # => #<Enumerator: [:foo, "bar", 2]:delete_if>
```

`difference(*other_arrays) → new_array`

Returns a new Array containing only those elements from `self` that are not found in any of the Arrays `other_arrays`; items are compared using `eql?`; order from `self` is preserved:

```
[0, 1, 1, 2, 1, 1, 3, 1, 1].difference([1]) # => [0, 2, 3]
[0, 1, 2, 3].difference([3, 0], [1, 3]) # => [2]
[0, 1, 2].difference([4]) # => [0, 1, 2]
```

Returns a copy of `self` if no arguments given.

Related: `Array#-`.

`dig(index, *identifiers) → object`

Finds and returns the object in nested objects that is specified by `index` and `identifiers`. The nested objects may be instances of various classes. See Dig Methods.

Examples:

```
a = [:foo, [:bar, :baz, [:bat, :bam]]]
a.dig(1) # => [:bar, :baz, [:bat, :bam]]
a.dig(1, 2) # => [:bat, :bam]
a.dig(1, 2, 0) # => :bat
a.dig(1, 2, 3) # => nil
```

`drop(n) → new_array`

Returns a new Array containing all but the first `n` element of `self`, where `n` is a non-negative `Integer`; does not modify `self`.

Examples:

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

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

drop_while {|element| ... } → new_array
drop_while → new_enumerator

Returns a new Array containing zero or more trailing elements of `self`; does not modify `self`.

With a block given, calls the block with each successive element of `self`; stops if the block returns `false` or `nil`; returns a new Array *omitting* those elements for which the block returned a truthy value:

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

With no block given, returns a new Enumerator:

```
[0, 1].drop_while # => # => #<Enumerator: [0, 1]:drop_while>
```

each {|element| ... } → self
each → Enumerator

Iterates over array elements.

When a block given, passes each successive array element to the block; returns `self`:

```
a = [:foo, 'bar', 2]
a.each {|element|  puts "#{element.class} #{element}" }
```

Output:

```
Symbol foo
String bar
Integer 2
```

Allows the array to be modified during iteration:

```
a = [:foo, 'bar', 2]
a.each {|element| puts element; a.clear if element.to_s.start_with?('b') }
```

Output:

```
foo
bar
```

When no block given, returns a new Enumerator:

```
a = [:foo, 'bar', 2]

e = a.each
e # => #<Enumerator: [:foo, "bar", 2]:each>
a1 = e.each {|element|  puts "#{element.class} #{element}" }
```

Output:

```
Symbol foo
String bar
Integer 2
```

Related: [each_index](), [reverse_each]().

each_index {|index| ... } → self
each_index → Enumerator

Iterates over array indexes.

When a block given, passes each successive array index to the block; returns `self`:

```
a = [:foo, 'bar', 2]
a.each_index {|index|  puts "#{index} #{a[index]}" }
```

Output:

```
0 foo
1 bar
2 2
```

Allows the array to be modified during iteration:

```
a = [:foo, 'bar', 2]
a.each_index {|index| puts index; a.clear if index > 0 }
```

Output:

```
0
1
```

When no block given, returns a new Enumerator:

```
a = [:foo, 'bar', 2]
e = a.each_index
e # => #<Enumerator: [:foo, "bar", 2]:each_index>
a1 = e.each {|index|  puts "#{index} #{a[index]}"}
```

Output:

```
0 foo
1 bar
2 2
```

Related: each, reverse_each.

empty? → true or false

Returns true if the count of elements in self is zero, false otherwise.

eql? other_array → true or false

Returns true if self and other_array are the same size, and if, for each index i in self, self[i].eql? other_array[i]:

```
a0 = [:foo, 'bar', 2]
a1 = [:foo, 'bar', 2]
a1.eql?(a0) # => true
```

Otherwise, returns false.

This method is different from method Array#==, which compares using method Object#==.

fetch(index) → element
fetch(index, default_value) → element
fetch(index) {|index| ... } → element

Returns the element at offset index.

With the single Integer argument index, returns the element at offset index:

```
a = [:foo, 'bar', 2]
a.fetch(1) # => "bar"
```

If index is negative, counts from the end of the array:

```
a = [:foo, 'bar', 2]
a.fetch(-1) # => 2
a.fetch(-2) # => "bar"
```

With arguments index and default_value, returns the element at offset index if index is in range, otherwise returns default_value:

```
a = [:foo, 'bar', 2]
a.fetch(1, nil) # => "bar"
```

With argument `index` and a block, returns the element at offset `index` if index is in range (and the block is not called); otherwise calls the block with index and returns its return value:

```
a = [:foo, 'bar', 2]
a.fetch(1) {|index| raise 'Cannot happen' } # => "bar"
a.fetch(50) {|index| "Value for #{index}" } # => "Value for 50"
```

fill(obj) → self
fill(obj, start) → self
fill(obj, start, length) → self
fill(obj, range) → self
fill {|index| ... } → self
fill(start) {|index| ... } → self
fill(start, length) {|index| ... } → self
fill(range) {|index| ... } → self

Replaces specified elements in `self` with specified objects; returns `self`.

With argument `obj` and no block given, replaces all elements with that one object:

```
a = ['a', 'b', 'c', 'd']
a # => ["a", "b", "c", "d"]
a.fill(:X) # => [:X, :X, :X, :X]
```

With arguments `obj` and `Integer` `start`, and no block given, replaces elements based on the given start.

If `start` is in range (`0 <= start < array.size`), replaces all elements from offset `start` through the end:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, 2) # => ["a", "b", :X, :X]
```

If `start` is too large (`start >= array.size`), does nothing:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, 4) # => ["a", "b", "c", "d"]
a = ['a', 'b', 'c', 'd']
a.fill(:X, 5) # => ["a", "b", "c", "d"]
```

If `start` is negative, counts from the end (starting index is `start + array.size`):

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, -2) # => ["a", "b", :X, :X]
```

If `start` is too small (less than and far from zero), replaces all elements:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, -6) # => [:X, :X, :X, :X]
a = ['a', 'b', 'c', 'd']
a.fill(:X, -50) # => [:X, :X, :X, :X]
```

With arguments `obj`, `Integer start`, and `Integer length`, and no block given, replaces elements based on the given `start` and `length`.

If `start` is in range, replaces `length` elements beginning at offset `start`:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, 1, 1) # => ["a", :X, "c", "d"]
```

If `start` is negative, counts from the end:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, -2, 1) # => ["a", "b", :X, "d"]
```

If `start` is large (`start >= array.size`), extends `self` with `nil`:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, 5, 0) # => ["a", "b", "c", "d", nil]
a = ['a', 'b', 'c', 'd']
a.fill(:X, 5, 2) # => ["a", "b", "c", "d", nil, :X, :X]
```

If `length` is zero or negative, replaces no elements:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, 1, 0) # => ["a", "b", "c", "d"]
a.fill(:X, 1, -1) # => ["a", "b", "c", "d"]
```

With arguments `obj` and `Range range`, and no block given, replaces elements based on the given range.

If the range is positive and ascending (`0 < range.begin <= range.end`), replaces elements from `range.begin` to `range.end`:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, (1..1)) # => ["a", :X, "c", "d"]
```

If `range.first` is negative, replaces no elements:

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

```
a.fill(:X, (-1..1)) # => ["a", "b", "c", "d"]
```

If `range.last` is negative, counts from the end:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, (0..-2)) # => [:X, :X, :X, "d"]
a = ['a', 'b', 'c', 'd']
a.fill(:X, (1..-2)) # => ["a", :X, :X, "d"]
```

If `range.last` and `range.last` are both negative, both count from the end of the array:

```
a = ['a', 'b', 'c', 'd']
a.fill(:X, (-1..-1)) # => ["a", "b", "c", :X]
a = ['a', 'b', 'c', 'd']
a.fill(:X, (-2..-2)) # => ["a", "b", :X, "d"]
```

With no arguments and a block given, calls the block with each index; replaces the corresponding element with the block's return value:

```
a = ['a', 'b', 'c', 'd']
a.fill { |index| "new_#{index}" } # => ["new_0", "new_1", "new_2", "new_3"]
```

With argument `start` and a block given, calls the block with each index from offset `start` to the end; replaces the corresponding element with the block's return value.

If start is in range (`0 <= start < array.size`), replaces from offset `start` to the end:

```
a = ['a', 'b', 'c', 'd']
a.fill(1) { |index| "new_#{index}" } # => ["a", "new_1", "new_2", "new_3"]
```

If `start` is too large(`start >= array.size`), does nothing:

```
a = ['a', 'b', 'c', 'd']
a.fill(4) { |index| fail 'Cannot happen' } # => ["a", "b", "c", "d"]
a = ['a', 'b', 'c', 'd']
a.fill(4) { |index| fail 'Cannot happen' } # => ["a", "b", "c", "d"]
```

If `start` is negative, counts from the end:

```
a = ['a', 'b', 'c', 'd']
a.fill(-2) { |index| "new_#{index}" } # => ["a", "b", "new_2", "new_3"]
```

If start is too small (`start <= -array.size`, replaces all elements:

```
a = ['a', 'b', 'c', 'd']
a.fill(-6) { |index| "new_#{index}" } # => ["new_0", "new_1", "new_2", "new_3
```

```
a = ['a', 'b', 'c', 'd']
a.fill(-50) { |index| "new_#{index}" } # => ["new_0", "new_1", "new_2", "new_
```

With arguments `start` and `length`, and a block given, calls the block for each index specified by start length; replaces the corresponding element with the block's return value.

If `start` is in range, replaces `length` elements beginning at offset `start`:

```
a = ['a', 'b', 'c', 'd']
a.fill(1, 1) { |index| "new_#{index}" } # => ["a", "new_1", "c", "d"]
```

If start is negative, counts from the end:

```
a = ['a', 'b', 'c', 'd']
a.fill(-2, 1) { |index| "new_#{index}" } # => ["a", "b", "new_2", "d"]
```

If `start` is large (`start >= array.size`), extends `self` with `nil`:

```
a = ['a', 'b', 'c', 'd']
a.fill(5, 0) { |index| "new_#{index}" } # => ["a", "b", "c", "d", nil]
a = ['a', 'b', 'c', 'd']
a.fill(5, 2) { |index| "new_#{index}" } # => ["a", "b", "c", "d", nil, "new_5
```

If `length` is zero or less, replaces no elements:

```
a = ['a', 'b', 'c', 'd']
a.fill(1, 0) { |index| "new_#{index}" } # => ["a", "b", "c", "d"]
a.fill(1, -1) { |index| "new_#{index}" } # => ["a", "b", "c", "d"]
```

With arguments `obj` and `range`, and a block given, calls the block with each index in the given range; replaces the corresponding element with the block's return value.

If the range is positive and ascending (`range 0 < range.begin <= range.end`, replaces elements from `range.begin` to `range.end`:

```
a = ['a', 'b', 'c', 'd']
a.fill(1..1) { |index| "new_#{index}" } # => ["a", "new_1", "c", "d"]
```

If `range.first` is negative, does nothing:

```
a = ['a', 'b', 'c', 'd']
a.fill(-1..1) { |index| fail 'Cannot happen' } # => ["a", "b", "c", "d"]
```

If `range.last` is negative, counts from the end:

```
a = ['a', 'b', 'c', 'd']
a.fill(0..-2) { |index| "new_#{index}" } # => ["new_0", "new_1", "new_2", "d"
a = ['a', 'b', 'c', 'd']
a.fill(1..-2) { |index| "new_#{index}" } # => ["a", "new_1", "new_2", "d"]
```

If `range.first` and `range.last` are both negative, both count from the end:

```
a = ['a', 'b', 'c', 'd']
a.fill(-1..-1) { |index| "new_#{index}" } # => ["a", "b", "c", "new_3"]
a = ['a', 'b', 'c', 'd']
a.fill(-2..-2) { |index| "new_#{index}" } # => ["a", "b", "new_2", "d"]
```

`filter()`

Calls the block, if given, with each element of `self`; returns a new Array containing those elements of `self` for which the block returns a truthy value:

```
a = [:foo, 'bar', 2, :bam]
a1 = a.select {|element| element.to_s.start_with?('b') }
a1 # => ["bar", :bam]
```

Returns a new `Enumerator` if no block given:

```
a = [:foo, 'bar', 2, :bam]
a.select # => #<Enumerator: [:foo, "bar", 2, :bam]:select>
```

Alias for: select

`filter!()`

Calls the block, if given with each element of `self`; removes from `self` those elements for which the block returns `false` or `nil`.

Returns `self` if any elements were removed:

```
a = [:foo, 'bar', 2, :bam]
a.select! {|element| element.to_s.start_with?('b') } # => ["bar", :bam]
```

Returns `nil` if no elements were removed.

Returns a new `Enumerator` if no block given:

```
a = [:foo, 'bar', 2, :bam]
a.select! # => #<Enumerator: [:foo, "bar", 2, :bam]:select!>
```

Alias for: select!

find_index -> new_enumerator

Returns the index of a specified element.

When argument `object` is given but no block, returns the index of the first element `element` for which `object == element`:

```
a = [:foo, 'bar', 2, 'bar']
a.index('bar') # => 1
```

Returns `nil` if no such element found.

When both argument `object` and a block are given, calls the block with each successive element; returns the index of the first element for which the block returns a truthy value:

```
a = [:foo, 'bar', 2, 'bar']
a.index {|element| element == 'bar' } # => 1
```

Returns `nil` if the block never returns a truthy value.

When neither an argument nor a block is given, returns a new Enumerator:

```
a = [:foo, 'bar', 2]
e = a.index
e # => #<Enumerator: [:foo, "bar", 2]:index>
e.each {|element| element == 'bar' } # => 1
```

Related: `rindex`.

Also aliased as: index

first → object or nil
first(n) → new_array

Returns elements from `self`; does not modify `self`.

When no argument is given, returns the first element:

```
a = [:foo, 'bar', 2]
a.first # => :foo
a # => [:foo, "bar", 2]
```

If `self` is empty, returns `nil`.

When non-negative `Integer` argument `n` is given, returns the first `n` elements in a new Array:

```
a = [:foo, 'bar', 2]
a.first(2) # => [:foo, "bar"]
```

If `n >= array.size`, returns all elements:

```
a = [:foo, 'bar', 2]
a.first(50) # => [:foo, "bar", 2]
```

If `n == 0` returns an new empty Array:

```
a = [:foo, 'bar', 2]
a.first(0) # []
```

Related: `last`.

flatten → new_array
flatten(level) → new_array

Returns a new Array that is a recursive flattening of `self`:

- Each non-Array element is unchanged.
- Each Array is replaced by its individual elements.

With non-negative `Integer` argument `level`, flattens recursively through `level` levels:

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

With no argument, a `nil` argument, or with negative argument `level`, flattens all levels:

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

flatten! → self or nil
flatten!(level) → self or nil

Replaces each nested Array in `self` with the elements from that Array; returns `self` if any changes, `nil` otherwise.

With non-negative `Integer` argument `level`, flattens recursively through `level` levels:

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

With no argument, a `nil` argument, or with negative argument `level`, flattens all levels:

```
a = [ 0, [ 1, [2, 3], 4 ], 5 ]
a.flatten! # => [0, 1, 2, 3, 4, 5]
[0, 1, 2].flatten! # => nil
a = [ 0, [ 1, [2, 3], 4 ], 5 ]
a.flatten!(-1) # => [0, 1, 2, 3, 4, 5]
a = [ 0, [ 1, [2, 3], 4 ], 5 ]
a.flatten!(-2) # => [0, 1, 2, 3, 4, 5]
[0, 1, 2].flatten!(-1) # => nil
```

hash → integer

Returns the integer hash value for `self`.

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

```
[0, 1, 2].hash == [0, 1, 2].hash # => true
[0, 1, 2].hash == [0, 1, 3].hash # => false
```

include?(obj) → true or false

Returns `true` if for some index `i` in `self`, `obj == self[i]`; otherwise `false`:

```
[0, 1, 2].include?(2) # => true
[0, 1, 2].include?(3) # => false
```

index(object) → integer or nil
index {|element| ... } → integer or nil
index → new_enumerator

Returns the index of a specified element.

When argument `object` is given but no block, returns the index of the first element `element` for which `object == element`:

```
a = [:foo, 'bar', 2, 'bar']
a.index('bar') # => 1
```

Returns `nil` if no such element found.

When both argument `object` and a block are given, calls the block with each successive element; returns the index of the first element for which the block returns a truthy value:

```
a = [:foo, 'bar', 2, 'bar']
a.index {|element| element == 'bar' } # => 1
```

Returns `nil` if the block never returns a truthy value.

When neither an argument nor a block is given, returns a new Enumerator:

```
a = [:foo, 'bar', 2]
e = a.index
e # => #<Enumerator: [:foo, "bar", 2]:index>
e.each {|element| element == 'bar' } # => 1
```

Related: `rindex`.

Alias for: find_index

`initialize_copy(other_array) -> self`

Replaces the content of `self` with the content of `other_array`; returns `self`:

```
a = [:foo, 'bar', 2]
a.replace(['foo', :bar, 3]) # => ["foo", :bar, 3]
```

Also aliased as: replace

`insert(index, *objects) → self`

Inserts given `objects` before or after the element at `Integer` index `offset`; returns `self`.

When `index` is non-negative, inserts all given `objects` before the element at offset `index`:

```
a = [:foo, 'bar', 2]
a.insert(1, :bat, :bam) # => [:foo, :bat, :bam, "bar", 2]
```

Extends the array if `index` is beyond the array (`index >= self.size`):

```
a = [:foo, 'bar', 2]
a.insert(5, :bat, :bam)
a # => [:foo, "bar", 2, nil, nil, :bat, :bam]
```

Does nothing if no objects given:

```
a = [:foo, 'bar', 2]
a.insert(1)
a.insert(50)
a.insert(-50)
a # => [:foo, "bar", 2]
```

When `index` is negative, inserts all given `objects` *after* the element at offset `index+self.size`:

```
a = [:foo, 'bar', 2]
a.insert(-2, :bat, :bam)
a # => [:foo, "bar", :bat, :bam, 2]
```

`inspect → new_string`

Returns the new `String` formed by calling method `#inspect` on each array element:

```
a = [:foo, 'bar', 2]
a.inspect # => "[:foo, \"bar\", 2]"
```

Also aliased as: to_s

`intersect?(other_ary) → true or false`

Returns `true` if the array and `other_ary` have at least one element in common, otherwise returns `false`:

```
a = [ 1, 2, 3 ]
b = [ 3, 4, 5 ]
c = [ 5, 6, 7 ]
a.intersect?(b)   #=> true
a.intersect?(c)   #=> false
```

Array elements are compared using `eql?` (items must also implement `hash` correctly).

`intersection(*other_arrays) → new_array`

Returns a new Array containing each element found both in `self` and in all of the given Arrays `other_arrays`; duplicates are omitted; items are compared using `eql?` (items must also implement `hash` correctly):

```
[0, 1, 2, 3].intersection([0, 1, 2], [0, 1, 3]) # => [0, 1]
[0, 0, 1, 1, 2, 3].intersection([0, 1, 2], [0, 1, 3]) # => [0, 1]
```

Preserves order from `self`:

```
[0, 1, 2].intersection([2, 1, 0]) # => [0, 1, 2]
```

Returns a copy of `self` if no arguments given.

Related: Array#&.

`join →new_string`
`join(separator = $,) → new_string`

Returns the new String formed by joining the array elements after conversion. For each element `element`:

- Uses `element.to_s` if `element` is not a `kind_of?(Array)`.
- Uses recursive `element.join(separator)` if `element` is a `kind_of?(Array)`.

With no argument, joins using the output field separator, `$,`:

```
a = [:foo, 'bar', 2]
$, # => nil
a.join # => "foobar2"
```

With string argument `separator`, joins using that separator:

```
a = [:foo, 'bar', 2]
a.join("\n") # => "foo\nbar\n2"
```

Joins recursively for nested Arrays:

```
a = [:foo, [:bar, [:baz, :bat]]]
a.join # => "foobarbazbat"
```

keep_if {|element| ... } → self
keep_if → new_enumeration

Retains those elements for which the block returns a truthy value; deletes all other elements; returns `self`:

```
a = [:foo, 'bar', 2, :bam]
a.keep_if {|element| element.to_s.start_with?('b') } # => ["bar", :bam]
```

Returns a new `Enumerator` if no block given:

```
a = [:foo, 'bar', 2, :bam]
a.keep_if # => #<Enumerator: [:foo, "bar", 2, :bam]:keep_if>
```

last → object or nil
last(n) → new_array

Returns elements from `self`; `self` is not modified.

When no argument is given, returns the last element:

```
a = [:foo, 'bar', 2]
a.last # => 2
a # => [:foo, "bar", 2]
```

If `self` is empty, returns `nil`.

When non-negative `Integer` argument `n` is given, returns the last `n` elements in a new Array:

```
a = [:foo, 'bar', 2]
a.last(2) # => ["bar", 2]
```

If `n >= array.size`, returns all elements:

```
a = [:foo, 'bar', 2]
a.last(50) # => [:foo, "bar", 2]
```

If `n == 0`, returns an new empty Array:

```
a = [:foo, 'bar', 2]
a.last(0) # []
```

Related: `first`.

length → an_integer

Returns the count of elements in `self`.

Also aliased as: size

map {|element| ... } → new_array
map → new_enumerator

Calls the block, if given, with each element of `self`; returns a new Array whose elements are the return values from the block:

```
a = [:foo, 'bar', 2]
a1 = a.map {|element| element.class }
a1 # => [Symbol, String, Integer]
```

Returns a new Enumerator if no block given:

```
a = [:foo, 'bar', 2]
a1 = a.map
a1 # => #<Enumerator: [:foo, "bar", 2]:map>
```

Alias for: collect

map! {|element| ... } → self
map! → new_enumerator

Calls the block, if given, with each element; replaces the element with the block's return value:

```
a = [:foo, 'bar', 2]
a.map! { |element| element.class } # => [Symbol, String, Integer]
```

Returns a new Enumerator if no block given:

```
a = [:foo, 'bar', 2]
a1 = a.map!
a1 # => #<Enumerator: [:foo, "bar", 2]:map!>
```

Alias for: collect!

max → element
max {|a, b| ... } → element
max(n) → new_array
max(n) {|a, b| ... } → new_array

Returns one of the following:

- The maximum-valued element from `self`.
- A new Array of maximum-valued elements selected from `self`.

When no block is given, each element in `self` must respond to method `<=>` with an `Integer`.

With no argument and no block, returns the element in `self` having the maximum value per method `<=>`:

```
[0, 1, 2].max # => 2
```

With an argument `Integer` `n` and no block, returns a new Array with at most `n` elements, in descending order per method `<=>`:

```
[0, 1, 2, 3].max(3) # => [3, 2, 1]
[0, 1, 2, 3].max(6) # => [3, 2, 1, 0]
```

When a block is given, the block must return an `Integer`.

With a block and no argument, calls the block `self.size-1` times to compare elements; returns the element having the maximum value per the block:

```
['0', '00', '000'].max {|a, b| a.size <=> b.size } # => "000"
```

With an argument `n` and a block, returns a new Array with at most `n` elements, in descending order per the block:

```
['0', '00', '000'].max(2) {|a, b| a.size <=> b.size } # => ["000", "00"]
```

`min → element`
`min { |a, b| ... } → element`
`min(n) → new_array`
`min(n) { |a, b| ... } → new_array`

Returns one of the following:

- The minimum-valued element from `self`.
- A new Array of minimum-valued elements selected from `self`.

When no block is given, each element in `self` must respond to method `<=>` with an `Integer`.

With no argument and no block, returns the element in `self` having the minimum value per method `<=>`:

```
[0, 1, 2].min # => 0
```

With Integer argument n and no block, returns a new Array with at most n elements, in ascending order per method <=> :

```
[0, 1, 2, 3].min(3) # => [0, 1, 2]
[0, 1, 2, 3].min(6) # => [0, 1, 2, 3]
```

When a block is given, the block must return an Integer .

With a block and no argument, calls the block self.size-1 times to compare elements; returns the element having the minimum value per the block:

```
['0', '00', '000'].min { |a, b| a.size <=> b.size } # => "0"
```

With an argument n and a block, returns a new Array with at most n elements, in ascending order per the block:

```
['0', '00', '000'].min(2) {|a, b| a.size <=> b.size } # => ["0", "00"]
```

minmax → [min_val, max_val]
minmax {|a, b| ... } → [min_val, max_val]

Returns a new 2-element Array containing the minimum and maximum values from self , either per method <=> or per a given block:.

When no block is given, each element in self must respond to method <=> with an Integer ; returns a new 2-element Array containing the minimum and maximum values from self , per method <=> :

```
[0, 1, 2].minmax # => [0, 2]
```

When a block is given, the block must return an Integer ; the block is called self.size-1 times to compare elements; returns a new 2-element Array containing the minimum and maximum values from self , per the block:

```
['0', '00', '000'].minmax {|a, b| a.size <=> b.size } # => ["0", "000"]
```

none? → true or false
none? {|element| ... } → true or false
none?(obj) → true or false

Returns true if no element of self meet a given criterion.

With no block given and no argument, returns `true` if `self` has no truthy elements, `false` otherwise:

```
[nil, false].none? # => true
[nil, 0, false].none? # => false
[].none? # => true
```

With a block given and no argument, calls the block with each element in `self`; returns `true` if the block returns no truthy value, `false` otherwise:

```
[0, 1, 2].none? {|element| element > 3 } # => true
[0, 1, 2].none? {|element| element > 1 } # => false
```

If argument `obj` is given, returns `true` if `obj.===` no element, `false` otherwise:

```
['food', 'drink'].none?(/bar/) # => true
['food', 'drink'].none?(/foo/) # => false
[].none?(/foo/) # => true
[0, 1, 2].none?(3) # => true
[0, 1, 2].none?(1) # => false
```

Related: Enumerable#none?

one? → true or false
one? {|element| ... } → true or false
one?(obj) → true or false

Returns `true` if exactly one element of `self` meets a given criterion.

With no block given and no argument, returns `true` if `self` has exactly one truthy element, `false` otherwise:

```
[nil, 0].one? # => true
[0, 0].one? # => false
[nil, nil].one? # => false
[].one? # => false
```

With a block given and no argument, calls the block with each element in `self`; returns `true` if the block a truthy value for exactly one element, `false` otherwise:

```
[0, 1, 2].one? {|element| element > 0 } # => false
[0, 1, 2].one? {|element| element > 1 } # => true
[0, 1, 2].one? {|element| element > 2 } # => false
```

If argument `obj` is given, returns `true` if `obj.===` exactly one element, `false` otherwise:

```
[0, 1, 2].one?(0) # => true
[0, 0, 1].one?(0) # => false
[1, 1, 2].one?(0) # => false
['food', 'drink'].one?(/bar/) # => false
['food', 'drink'].one?(/foo/) # => true
[].one?(/foo/) # => false
```

Related: Enumerable#one?

pack(template, buffer: nil) → string

Formats each element in `self` into a binary string; returns that string. See Packed Data.

permutation {|element| ... } → self
permutation(n) {|element| ... } → self
permutation → new_enumerator
permutation(n) → new_enumerator

When invoked with a block, yield all permutations of elements of `self`; returns `self`. The order of permutations is indeterminate.

When a block and an in-range positive Integer argument `n` (`0 < n <= self.size`) are given, calls the block with all `n`-tuple permutations of `self`.

Example:

```
a = [0, 1, 2]
a.permutation(2) {|permutation| p permutation }
```

Output:

```
[0, 1]
[0, 2]
[1, 0]
[1, 2]
[2, 0]
[2, 1]
```

Another example:

```
a = [0, 1, 2]
a.permutation(3) {|permutation| p permutation }
```

Output:

```
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
```

```
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

When `n` is zero, calls the block once with a new empty Array:

```
a = [0, 1, 2]
a.permutation(0) {|permutation| p permutation }
```

Output:

```
[]
```

When `n` is out of range (negative or larger than `self.size`), does not call the block:

```
a = [0, 1, 2]
a.permutation(-1) {|permutation| fail 'Cannot happen' }
a.permutation(4) {|permutation| fail 'Cannot happen' }
```

When a block given but no argument, behaves the same as `a.permutation(a.size)`:

```
a = [0, 1, 2]
a.permutation {|permutation| p permutation }
```

Output:

```
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

Returns a new `Enumerator` if no block given:

```
a = [0, 1, 2]
a.permutation # => #<Enumerator: [0, 1, 2]:permutation>
a.permutation(2) # => #<Enumerator: [0, 1, 2]:permutation(2)>
```

pop → object or nil
pop(n) → new_array

Removes and returns trailing elements.

When no argument is given and `self` is not empty, removes and returns the last element:

```
a = [:foo, 'bar', 2]
a.pop # => 2
a # => [:foo, "bar"]
```

Returns `nil` if the array is empty.

When a non-negative `Integer` argument `n` is given and is in range,

removes and returns the last `n` elements in a new Array:

```
a = [:foo, 'bar', 2]
a.pop(2) # => ["bar", 2]
```

If `n` is positive and out of range, removes and returns all elements:

```
a = [:foo, 'bar', 2]
a.pop(50) # => [:foo, "bar", 2]
```

Related: `push`, `shift`, `unshift`.

prepend(*args)

Prepends the given `objects` to `self`:

```
a = [:foo, 'bar', 2]
a.unshift(:bam, :bat) # => [:bam, :bat, :foo, "bar", 2]
```

Related: `push`, `pop`, `shift`.

Alias for: unshift

product(*other_arrays) → new_array
product(*other_arrays) {|combination| ... } → self

Computes and returns or yields all combinations of elements from all the Arrays, including both `self` and `other_arrays`:

- The number of combinations is the product of the sizes of all the arrays, including both `self` and `other_arrays`.
- The order of the returned combinations is indeterminate.

When no block is given, returns the combinations as an Array of Arrays:

```
a = [0, 1, 2]
a1 = [3, 4]
a2 = [5, 6]
p = a.product(a1)
p.size # => 6 # a.size * a1.size
```

```
p # => [[0, 3], [0, 4], [1, 3], [1, 4], [2, 3], [2, 4]]
p = a.product(a1, a2)
p.size # => 12 # a.size * a1.size * a2.size
p # => [[0, 3, 5], [0, 3, 6], [0, 4, 5], [0, 4, 6], [1, 3, 5], [1, 3, 6], [1,
```

If any argument is an empty Array, returns an empty Array.

If no argument is given, returns an Array of 1-element Arrays, each containing an element of `self`:

```
a.product # => [[0], [1], [2]]
```

When a block is given, yields each combination as an Array; returns `self`:

```
a.product(a1) {|combination| p combination }
```

Output:

```
[0, 3]
[0, 4]
[1, 3]
[1, 4]
[2, 3]
[2, 4]
```

If any argument is an empty Array, does not call the block:

```
a.product(a1, a2, []) {|combination| fail 'Cannot happen' }
```

If no argument is given, yields each element of `self` as a 1-element Array:

```
a.product {|combination| p combination }
```

Output:

```
[0]
[1]
[2]
```

`push(*objects) → self`

Appends trailing elements.

Appends each argument in `objects` to `self`; returns `self`:

```
a = [:foo, 'bar', 2]
```

```
a.push(:baz, :bat) # => [:foo, "bar", 2, :baz, :bat]
```

Appends each argument as one element, even if it is another Array:

```
a = [:foo, 'bar', 2]
a1 = a.push([:baz, :bat], [:bam, :bad])
a1 # => [:foo, "bar", 2, [:baz, :bat], [:bam, :bad]]
```

Related: pop, shift, unshift.

Also aliased as: append

rassoc(obj) → found_array or nil

Returns the first element in `self` that is an Array whose second element `==` `obj`:

```
a = [{foo: 0}, [2, 4], [4, 5, 6], [4, 5]]
a.rassoc(4) # => [2, 4]
```

Returns `nil` if no such element is found.

Related: assoc.

reject {|element| ... } → new_array
reject → new_enumerator

Returns a new Array whose elements are all those from `self` for which the block returns `false` or `nil`:

```
a = [:foo, 'bar', 2, 'bat']
a1 = a.reject {|element| element.to_s.start_with?('b') }
a1 # => [:foo, 2]
```

Returns a new Enumerator if no block given:

```
a = [:foo, 'bar', 2]
a.reject # => #<Enumerator: [:foo, "bar", 2]:reject>
```

reject! {|element| ... } → self or nil
reject! → new_enumerator

Removes each element for which the block returns a truthy value.

Returns `self` if any elements removed:

```
a = [:foo, 'bar', 2, 'bat']
a.reject! {|element| element.to_s.start_with?('b') } # => [:foo, 2]
```

Returns `nil` if no elements removed.

Returns a new `Enumerator` if no block given:

```
a = [:foo, 'bar', 2]
a.reject! # => #<Enumerator: [:foo, "bar", 2]:reject!>
```

`repeated_combination(n) {|combination| ... } → self`
`repeated_combination(n) → new_enumerator`

Calls the block with each repeated combination of length `n` of the elements of `self`; each combination is an Array; returns `self`. The order of the combinations is indeterminate.

When a block and a positive `Integer` argument `n` are given, calls the block with each `n`-tuple repeated combination of the elements of `self`. The number of combinations is `(n+1)(n+2)/2`.

`n` = 1:

```
a = [0, 1, 2]
a.repeated_combination(1) {|combination| p combination }
```

Output:

```
[0]
[1]
[2]
```

`n` = 2:

```
a.repeated_combination(2) {|combination| p combination }
```

Output:

```
[0, 0]
[0, 1]
[0, 2]
[1, 1]
[1, 2]
[2, 2]
```

If `n` is zero, calls the block once with an empty Array.

If `n` is negative, does not call the block:

```
a.repeated_combination(-1) {|combination| fail 'Cannot happen' }
```

Returns a new `Enumerator` if no block given:

```
a = [0, 1, 2]
a.repeated_combination(2) # => #<Enumerator: [0, 1, 2]:combination(2)>
```

Using Enumerators, it's convenient to show the combinations and counts for some values of `n`:

```
e = a.repeated_combination(0)
e.size # => 1
e.to_a # => [[]]
e = a.repeated_combination(1)
e.size # => 3
e.to_a # => [[0], [1], [2]]
e = a.repeated_combination(2)
e.size # => 6
e.to_a # => [[0, 0], [0, 1], [0, 2], [1, 1], [1, 2], [2, 2]]
```

`repeated_permutation(n) {|permutation| ... } → self`
`repeated_permutation(n) → new_enumerator`

Calls the block with each repeated permutation of length `n` of the elements of `self`; each permutation is an Array; returns `self`. The order of the permutations is indeterminate.

When a block and a positive `Integer` argument `n` are given, calls the block with each `n`-tuple repeated permutation of the elements of `self`. The number of permutations is `self.size**n`.

`n` = 1:

```
a = [0, 1, 2]
a.repeated_permutation(1) {|permutation| p permutation }
```

Output:

```
[0]
[1]
[2]
```

`n` = 2:

```
a.repeated_permutation(2) {|permutation| p permutation }
```

Output:

```
[0, 0]
[0, 1]
[0, 2]
[1, 0]
[1, 1]
[1, 2]
[2, 0]
[2, 1]
[2, 2]
```

If `n` is zero, calls the block once with an empty Array.

If `n` is negative, does not call the block:

```
a.repeated_permutation(-1) {|permutation| fail 'Cannot happen' }
```

Returns a new `Enumerator` if no block given:

```
a = [0, 1, 2]
a.repeated_permutation(2) # => #<Enumerator: [0, 1, 2]:permutation(2)>
```

Using Enumerators, it's convenient to show the permutations and counts for some values of `n`:

```
e = a.repeated_permutation(0)
e.size # => 1
e.to_a # => [[]]
e = a.repeated_permutation(1)
e.size # => 3
e.to_a # => [[0], [1], [2]]
e = a.repeated_permutation(2)
e.size # => 9
e.to_a # => [[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1],
```

`replace(other_array) → self`

Replaces the content of `self` with the content of `other_array`; returns `self`:

```
a = [:foo, 'bar', 2]
a.replace(['foo', :bar, 3]) # => ["foo", :bar, 3]
```

Alias for: initialize_copy

`reverse → new_array`

Returns a new Array with the elements of `self` in reverse order:

```
a = ['foo', 'bar', 'two']
a1 = a.reverse
a1 # => ["two", "bar", "foo"]
```

reverse! → self

Reverses `self` in place:

```
a = ['foo', 'bar', 'two']
a.reverse! # => ["two", "bar", "foo"]
```

reverse_each {|element| ... } → self
reverse_each → Enumerator

Iterates backwards over array elements.

When a block given, passes, in reverse order, each element to the block; returns `self`:

```
a = [:foo, 'bar', 2]
a.reverse_each {|element|  puts "#{element.class} #{element}" }
```

Output:

```
Integer 2
String bar
Symbol foo
```

Allows the array to be modified during iteration:

```
a = [:foo, 'bar', 2]
a.reverse_each {|element| puts element; a.clear if element.to_s.start_with?('
```

Output:

```
2
bar
```

When no block given, returns a new Enumerator:

```
a = [:foo, 'bar', 2]
e = a.reverse_each
e # => #<Enumerator: [:foo, "bar", 2]:reverse_each>
a1 = e.each {|element|  puts "#{element.class} #{element}" }
```

Output:

```
Integer 2
String bar
Symbol foo
```

Related: each, each_index.

rindex(object) → integer or nil
rindex {|element| ... } → integer or nil
rindex → new_enumerator

Returns the index of the last element for which `object == element`.

When argument `object` is given but no block, returns the index of the last such element found:

```
a = [:foo, 'bar', 2, 'bar']
a.rindex('bar') # => 3
```

Returns `nil` if no such object found.

When a block is given but no argument, calls the block with each successive element; returns the index of the last element for which the block returns a truthy value:

```
a = [:foo, 'bar', 2, 'bar']
a.rindex {|element| element == 'bar' } # => 3
```

Returns `nil` if the block never returns a truthy value.

When neither an argument nor a block is given, returns a new Enumerator:

```
a = [:foo, 'bar', 2, 'bar']
e = a.rindex
e # => #<Enumerator: [:foo, "bar", 2, "bar"]:rindex>
e.each {|element| element == 'bar' } # => 3
```

Related: index.

rotate → new_array
rotate(count) → new_array

Returns a new Array formed from `self` with elements rotated from one end to the other.

When no argument given, returns a new Array that is like `self`, except that the first element has been rotated to the last position:

```
a = [:foo, 'bar', 2, 'bar']
a1 = a.rotate
a1 # => ["bar", 2, "bar", :foo]
```

When given a non-negative Integer `count`, returns a new Array with `count` elements rotated from the beginning to the end:

```
a = [:foo, 'bar', 2]
a1 = a.rotate(2)
a1 # => [2, :foo, "bar"]
```

If `count` is large, uses `count % array.size` as the count:

```
a = [:foo, 'bar', 2]
a1 = a.rotate(20)
a1 # => [2, :foo, "bar"]
```

If `count` is zero, returns a copy of `self`, unmodified:

```
a = [:foo, 'bar', 2]
a1 = a.rotate(0)
a1 # => [:foo, "bar", 2]
```

When given a negative Integer `count`, rotates in the opposite direction, from end to beginning:

```
a = [:foo, 'bar', 2]
a1 = a.rotate(-2)
a1 # => ["bar", 2, :foo]
```

If `count` is small (far from zero), uses `count % array.size` as the count:

```
a = [:foo, 'bar', 2]
a1 = a.rotate(-5)
a1 # => ["bar", 2, :foo]
```

`rotate! → self`
`rotate!(count) → self`

Rotates `self` in place by moving elements from one end to the other; returns `self`.

When no argument given, rotates the first element to the last position:

```
a = [:foo, 'bar', 2, 'bar']
a.rotate! # => ["bar", 2, "bar", :foo]
```

When given a non-negative `Integer` `count`, rotates `count` elements from the beginning to the end:

```
a = [:foo, 'bar', 2]
a.rotate!(2)
a # => [2, :foo, "bar"]
```

If `count` is large, uses `count % array.size` as the count:

```
a = [:foo, 'bar', 2]
a.rotate!(20)
a # => [2, :foo, "bar"]
```

If `count` is zero, returns `self` unmodified:

```
a = [:foo, 'bar', 2]
a.rotate!(0)
a # => [:foo, "bar", 2]
```

When given a negative `Integer` `count`, rotates in the opposite direction, from end to beginning:

```
a = [:foo, 'bar', 2]
a.rotate!(-2)
a # => ["bar", 2, :foo]
```

If `count` is small (far from zero), uses `count % array.size` as the count:

```
a = [:foo, 'bar', 2]
a.rotate!(-5)
a # => ["bar", 2, :foo]
```

`sample(random: Random) → object`
`sample(n, random: Random) → new_ary`

Returns random elements from `self`.

When no arguments are given, returns a random element from `self`:

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

If `self` is empty, returns `nil`.

When argument `n` is given, returns a new Array containing `n` random elements from `self`:

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

Returns no more than `a.size` elements (because no new duplicates are introduced):

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

But `self` may contain duplicates:

```
a = [1, 1, 1, 2, 2, 3]
a.sample(a.size * 2) # => [1, 1, 3, 2, 1, 2]
```

The argument `n` must be a non-negative numeric value. The order of the result array is unrelated to the order of `self`. Returns a new empty Array if `self` is empty.

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

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

`select {|element| ... } → new_array`
`select → new_enumerator`

Calls the block, if given, with each element of `self`; returns a new Array containing those elements of `self` for which the block returns a truthy value:

```
a = [:foo, 'bar', 2, :bam]
a1 = a.select {|element| element.to_s.start_with?('b') }
a1 # => ["bar", :bam]
```

Returns a new `Enumerator` if no block given:

```
a = [:foo, 'bar', 2, :bam]
a.select # => #<Enumerator: [:foo, "bar", 2, :bam]:select>
```

Also aliased as: filter

`select! {|element| ... } → self or nil`
`select! → new_enumerator`

Calls the block, if given with each element of `self`; removes from `self` those elements for which the block returns `false` or `nil`.

Returns `self` if any elements were removed:

```
a = [:foo, 'bar', 2, :bam]
a.select! {|element| element.to_s.start_with?('b') } # => ["bar", :bam]
```

Returns `nil` if no elements were removed.

Returns a new `Enumerator` if no block given:

```
a = [:foo, 'bar', 2, :bam]
a.select! # => #<Enumerator: [:foo, "bar", 2, :bam]:select!>
```

Also aliased as: filter!

shift → object or nil
shift(n) → new_array

Removes and returns leading elements.

When no argument is given, removes and returns the first element:

```
a = [:foo, 'bar', 2]
a.shift # => :foo
a # => ['bar', 2]
```

Returns `nil` if `self` is empty.

When positive `Integer` argument `n` is given, removes the first `n` elements; returns those elements in a new Array:

```
a = [:foo, 'bar', 2]
a.shift(2) # => [:foo, 'bar']
a # => [2]
```

If `n` is as large as or larger than `self.length`, removes all elements; returns those elements in a new Array:

```
a = [:foo, 'bar', 2]
a.shift(3) # => [:foo, 'bar', 2]
```

If `n` is zero, returns a new empty Array; `self` is unmodified.

Related: `push`, `pop`, `unshift`.

shuffle(random: Random) → new_ary

Returns a new array with elements of `self` shuffled.

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

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

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

shuffle!(random: Random) → array

Shuffles the elements of `self` in place.

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

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

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

size()

Returns the count of elements in `self`.

Alias for: length

slice(index) → object or nil
slice(start, length) → object or nil
slice(range) → object or nil
slice(aseq) → object or nil

Returns elements from `self`; does not modify `self`.

When a single `Integer` argument `index` is given, returns the element at offset `index`:

```
a = [:foo, 'bar', 2]
a[0] # => :foo
a[2] # => 2
a # => [:foo, "bar", 2]
```

If `index` is negative, counts relative to the end of `self`:

```
a = [:foo, 'bar', 2]
a[-1] # => 2
```

```
a[-2] # => "bar"
```

If `index` is out of range, returns `nil`.

When two `Integer` arguments `start` and `length` are given, returns a new Array of size `length` containing successive elements beginning at offset `start`:

```
a = [:foo, 'bar', 2]
a[0, 2] # => [:foo, "bar"]
a[1, 2] # => ["bar", 2]
```

If `start + length` is greater than `self.length`, returns all elements from offset `start` to the end:

```
a = [:foo, 'bar', 2]
a[0, 4] # => [:foo, "bar", 2]
a[1, 3] # => ["bar", 2]
a[2, 2] # => [2]
```

If `start == self.size` and `length >= 0`, returns a new empty Array.

If `length` is negative, returns `nil`.

When a single `Range` argument `range` is given, treats `range.min` as `start` above and `range.size` as `length` above:

```
a = [:foo, 'bar', 2]
a[0..1] # => [:foo, "bar"]
a[1..2] # => ["bar", 2]
```

Special case: If `range.start == a.size`, returns a new empty Array.

If `range.end` is negative, calculates the end index from the end:

```
a = [:foo, 'bar', 2]
a[0..-1] # => [:foo, "bar", 2]
a[0..-2] # => [:foo, "bar"]
a[0..-3] # => [:foo]
```

If `range.start` is negative, calculates the start index from the end:

```
a = [:foo, 'bar', 2]
a[-1..2] # => [2]
a[-2..2] # => ["bar", 2]
a[-3..2] # => [:foo, "bar", 2]
```

If `range.start` is larger than the array size, returns `nil`.

```
a = [:foo, 'bar', 2]
a[4..1] # => nil
```

```
a[4..0] # => nil
a[4..-1] # => nil
```

When a single Enumerator::ArithmeticSequence argument `aseq` is given, returns an Array of elements corresponding to the indexes produced by the sequence.

```
a = ['--', 'data1', '--', 'data2', '--', 'data3']
a[(1..).step(2)] # => ["data1", "data2", "data3"]
```

Unlike slicing with range, if the start or the end of the arithmetic sequence is larger than array size, throws RangeError.

```
a = ['--', 'data1', '--', 'data2', '--', 'data3']
a[(1..11).step(2)]
# RangeError (((1..11).step(2)) out of range)
a[(7..).step(2)]
# RangeError (((7..).step(2)) out of range)
```

If given a single argument, and its type is not one of the listed, tries to convert it to Integer, and raises if it is impossible:

```
a = [:foo, 'bar', 2]
# Raises TypeError (no implicit conversion of Symbol into Integer):
a[:foo]
```

Alias for: []

slice!(n) → object or nil
slice!(start, length) → new_array or nil
slice!(range) → new_array or nil

Removes and returns elements from `self`.

When the only argument is an Integer `n`, removes and returns the *nth* element in `self`:

```
a = [:foo, 'bar', 2]
a.slice!(1) # => "bar"
a # => [:foo, 2]
```

If `n` is negative, counts backwards from the end of `self`:

```
a = [:foo, 'bar', 2]
a.slice!(-1) # => 2
a # => [:foo, "bar"]
```

If `n` is out of range, returns `nil`.

When the only arguments are Integers `start` and `length`, removes `length` elements from `self` beginning at offset `start`; returns the deleted objects in a new Array:

```
a = [:foo, 'bar', 2]
a.slice!(0, 2) # => [:foo, "bar"]
a # => [2]
```

If `start + length` exceeds the array size, removes and returns all elements from offset `start` to the end:

```
a = [:foo, 'bar', 2]
a.slice!(1, 50) # => ["bar", 2]
a # => [:foo]
```

If `start == a.size` and `length` is non-negative, returns a new empty Array.

If `length` is negative, returns `nil`.

When the only argument is a Range object `range`, treats `range.min` as `start` above and `range.size` as `length` above:

```
a = [:foo, 'bar', 2]
a.slice!(1..2) # => ["bar", 2]
a # => [:foo]
```

If `range.start == a.size`, returns a new empty Array.

If `range.start` is larger than the array size, returns `nil`.

If `range.end` is negative, counts backwards from the end of the array:

```
a = [:foo, 'bar', 2]
a.slice!(0..-2) # => [:foo, "bar"]
a # => [2]
```

If `range.start` is negative, calculates the start index backwards from the end of the array:

```
a = [:foo, 'bar', 2]
a.slice!(-2..2) # => ["bar", 2]
a # => [:foo]
```

`sort → new_array`
`sort {|a, b| ... } → new_array`

Returns a new Array whose elements are those from `self`, sorted.

With no block, compares elements using operator `<=>` (see `Comparable`):

```
a = 'abcde'.split('').shuffle
a # => ["e", "b", "d", "a", "c"]
a1 = a.sort
a1 # => ["a", "b", "c", "d", "e"]
```

With a block, calls the block with each element pair; for each element pair `a` and `b`, the block should return an integer:

- Negative when `b` is to follow `a`.
- Zero when `a` and `b` are equivalent.
- Positive when `a` is to follow `b`.

Example:

```
a = 'abcde'.split('').shuffle
a # => ["e", "b", "d", "a", "c"]
a1 = a.sort {|a, b| a <=> b }
a1 # => ["a", "b", "c", "d", "e"]
a2 = a.sort {|a, b| b <=> a }
a2 # => ["e", "d", "c", "b", "a"]
```

When the block returns zero, the order for `a` and `b` is indeterminate, and may be unstable:

```
a = 'abcde'.split('').shuffle
a # => ["e", "b", "d", "a", "c"]
a1 = a.sort {|a, b| 0 }
a1 # =>  ["c", "e", "b", "d", "a"]
```

Related: Enumerable#sort_by.

sort! → self
sort! {|a, b| ... } → self

Returns `self` with its elements sorted in place.

With no block, compares elements using operator `<=>` (see Comparable):

```
a = 'abcde'.split('').shuffle
a # => ["e", "b", "d", "a", "c"]
a.sort!
a # => ["a", "b", "c", "d", "e"]
```

With a block, calls the block with each element pair; for each element pair `a` and `b`, the block should return an integer:

- Negative when `b` is to follow `a`.
- Zero when `a` and `b` are equivalent.

- Positive when `a` is to follow `b`.

Example:

```
a = 'abcde'.split('').shuffle
a # => ["e", "b", "d", "a", "c"]
a.sort! {|a, b| a <=> b }
a # => ["a", "b", "c", "d", "e"]
a.sort! {|a, b| b <=> a }
a # => ["e", "d", "c", "b", "a"]
```

When the block returns zero, the order for `a` and `b` is indeterminate, and may be unstable:

```
a = 'abcde'.split('').shuffle
a # => ["e", "b", "d", "a", "c"]
a.sort! {|a, b| 0 }
a # => ["d", "e", "c", "a", "b"]
```

sort_by! {|element| ... } → self
sort_by! → new_enumerator

Sorts the elements of `self` in place, using an ordering determined by the block; returns self.

Calls the block with each successive element; sorts elements based on the values returned from the block.

For duplicates returned by the block, the ordering is indeterminate, and may be unstable.

This example sorts strings based on their sizes:

```
a = ['aaaa', 'bbb', 'cc', 'd']
a.sort_by! {|element| element.size }
a # => ["d", "cc", "bbb", "aaaa"]
```

Returns a new `Enumerator` if no block given:

```
a = ['aaaa', 'bbb', 'cc', 'd']
a.sort_by! # => #<Enumerator: ["aaaa", "bbb", "cc", "d"]:sort_by!>
```

sum(init = 0) → object
sum(init = 0) {|element| ... } → object

When no block is given, returns the object equivalent to:

```
sum = init
array.each {|element| sum += element }
sum
```

For example, `[e1, e2, e3].sum` returns `init + e1 + e2 + e3`.

Examples:

```
a = [0, 1, 2, 3]
a.sum # => 6
a.sum(100) # => 106
```

The elements need not be numeric, but must be `+`-compatible with each other and with `init`:

```
a = ['abc', 'def', 'ghi']
a.sum('jkl') # => "jklabcdefghi"
```

When a block is given, it is called with each element and the block's return value (instead of the element itself) is used as the addend:

```
a = ['zero', 1, :two]
s = a.sum('Coerced and concatenated: ') {|element| element.to_s }
s # => "Coerced and concatenated: zero1two"
```

Notes:

- `Array#join` and `Array#flatten` may be faster than `Array#sum` for an Array of Strings or an Array of Arrays.
- `Array#sum` method may not respect method redefinition of "+" methods such as `Integer#+`.

`take(n) → new_array`

Returns a new Array containing the first `n` element of `self`, where `n` is a non-negative `Integer`; does not modify `self`.

Examples:

```
a = [0, 1, 2, 3, 4, 5]
a.take(1) # => [0]
a.take(2) # => [0, 1]
a.take(50) # => [0, 1, 2, 3, 4, 5]
a # => [0, 1, 2, 3, 4, 5]
```

`take_while {|element| ... } → new_array`

take_while → new_enumerator

Returns a new Array containing zero or more leading elements of `self`; does not modify `self`.

With a block given, calls the block with each successive element of `self`; stops if the block returns `false` or `nil`; returns a new Array containing those elements for which the block returned a truthy value:

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

With no block given, returns a new Enumerator:

```
[0, 1].take_while # => #<Enumerator: [0, 1]:take_while>
```

to_a → self or new_array

When `self` is an instance of Array, returns `self`:

```
a = [:foo, 'bar', 2]
a.to_a # => [:foo, "bar", 2]
```

Otherwise, returns a new Array containing the elements of `self`:

```
class MyArray < Array; end
a = MyArray.new(['foo', 'bar', 'two'])
a.instance_of?(Array) # => false
a.kind_of?(Array) # => true
a1 = a.to_a
a1 # => ["foo", "bar", "two"]
a1.class # => Array # Not MyArray
```

to_ary → self

Returns `self`.

to_h → new_hash
to_h {|item| ... } → new_hash

Returns a new Hash formed from `self`.

When a block is given, calls the block with each array element; the block must return a 2-element Array whose two elements form a key-value pair in the returned Hash:

```
a = ['foo', :bar, 1, [2, 3], {baz: 4}]
h = a.to_h {|item| [item, item] }
h # => {"foo"=>"foo", :bar=>:bar, 1=>1, [2, 3]=>[2, 3], {:baz=>4}=>{:baz=>4}}
```

When no block is given, `self` must be an Array of 2-element sub-arrays, each sub-array is formed into a key-value pair in the new Hash:

```
[].to_h # => {}
a = [['foo', 'zero'], ['bar', 'one'], ['baz', 'two']]
h = a.to_h
h # => {"foo"=>"zero", "bar"=>"one", "baz"=>"two"}
```

to_s()

Returns the new `String` formed by calling method `#inspect` on each array element:

```
a = [:foo, 'bar', 2]
a.inspect # => "[:foo, \"bar\", 2]"
```

Alias for: inspect

transpose → new_array

Transposes the rows and columns in an Array of Arrays; the nested Arrays must all be the same size:

```
a = [[:a0, :a1], [:b0, :b1], [:c0, :c1]]
a.transpose # => [[:a0, :b0, :c0], [:a1, :b1, :c1]]
```

union(*other_arrays) → new_array

Returns a new Array that is the union of `self` and all given Arrays `other_arrays`; duplicates are removed; order is preserved; items are compared using `eql?`:

```
[0, 1, 2, 3].union([4, 5], [6, 7]) # => [0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 1].union([2, 1], [3, 1]) # => [0, 1, 2, 3]
[0, 1, 2, 3].union([3, 2], [1, 0]) # => [0, 1, 2, 3]
```

Returns a copy of `self` if no arguments given.

Related: Array#|.

uniq → new_array
uniq {|element| ... } → new_array

Returns a new Array containing those elements from `self` that are not duplicates, the first occurrence always being retained.

With no block given, identifies and omits duplicates using method `eql?` to compare:

```
a = [0, 0, 1, 1, 2, 2]
a.uniq # => [0, 1, 2]
```

With a block given, calls the block for each element; identifies (using method `eql?`) and omits duplicate values, that is, those elements for which the block returns the same value:

```
a = ['a', 'aa', 'aaa', 'b', 'bb', 'bbb']
a.uniq {|element| element.size } # => ["a", "aa", "aaa"]
```

uniq! → self or nil
uniq! {|element| ... } → self or nil

Removes duplicate elements from `self`, the first occurrence always being retained; returns `self` if any elements removed, `nil` otherwise.

With no block given, identifies and removes elements using method `eql?` to compare.

Returns `self` if any elements removed:

```
a = [0, 0, 1, 1, 2, 2]
a.uniq! # => [0, 1, 2]
```

Returns `nil` if no elements removed.

With a block given, calls the block for each element; identifies (using method `eql?`) and removes elements for which the block returns duplicate values.

Returns `self` if any elements removed:

```
a = ['a', 'aa', 'aaa', 'b', 'bb', 'bbb']
a.uniq! {|element| element.size } # => ['a', 'aa', 'aaa']
```

Returns `nil` if no elements removed.

unshift(*objects) → self

Prepends the given `objects` to `self`:

```
a = [:foo, 'bar', 2]
a.unshift(:bam, :bat) # => [:bam, :bat, :foo, "bar", 2]
```

Related: push, pop, shift.

Also aliased as: prepend

values_at(*indexes) → new_array

Returns a new Array whose elements are the elements of `self` at the given Integer or Range `indexes`.

For each positive `index`, returns the element at offset `index`:

```
a = [:foo, 'bar', 2]
a.values_at(0, 2) # => [:foo, 2]
a.values_at(0..1) # => [:foo, "bar"]
```

The given `indexes` may be in any order, and may repeat:

```
a = [:foo, 'bar', 2]
a.values_at(2, 0, 1, 0, 2) # => [2, :foo, "bar", :foo, 2]
a.values_at(1, 0..2) # => ["bar", :foo, "bar", 2]
```

Assigns `nil` for an `index` that is too large:

```
a = [:foo, 'bar', 2]
a.values_at(0, 3, 1, 3) # => [:foo, nil, "bar", nil]
```

Returns a new empty Array if no arguments given.

For each negative `index`, counts backward from the end of the array:

```
a = [:foo, 'bar', 2]
a.values_at(-1, -3) # => [2, :foo]
```

Assigns `nil` for an `index` that is too small:

```
a = [:foo, 'bar', 2]
a.values_at(0, -5, 1, -6, 2) # => [:foo, nil, "bar", nil, 2]
```

The given `indexes` may have a mixture of signs:

```
a = [:foo, 'bar', 2]
a.values_at(0, -2, 1, -1) # => [:foo, "bar", "bar", 2]
```

`zip(*other_arrays) → new_array`
`zip(*other_arrays) {|other_array| ... } → nil`

When no block given, returns a new Array `new_array` of size `self.size` whose elements are Arrays.

Each nested array `new_array[n]` is of size `other_arrays.size+1`, and contains:

- The *nth* element of `self`.
- The *nth* element of each of the `other_arrays`.

If all `other_arrays` and `self` are the same size:

```
a = [:a0, :a1, :a2, :a3]
b = [:b0, :b1, :b2, :b3]
c = [:c0, :c1, :c2, :c3]
d = a.zip(b, c)
d # => [[:a0, :b0, :c0], [:a1, :b1, :c1], [:a2, :b2, :c2], [:a3, :b3, :c3]]
```

If any array in `other_arrays` is smaller than `self`, fills to `self.size` with `nil`:

```
a = [:a0, :a1, :a2, :a3]
b = [:b0, :b1, :b2]
c = [:c0, :c1]
d = a.zip(b, c)
d # => [[:a0, :b0, :c0], [:a1, :b1, :c1], [:a2, :b2, nil], [:a3, nil, nil]]
```

If any array in `other_arrays` is larger than `self`, its trailing elements are ignored:

```
a = [:a0, :a1, :a2, :a3]
b = [:b0, :b1, :b2, :b3, :b4]
c = [:c0, :c1, :c2, :c3, :c4, :c5]
d = a.zip(b, c)
d # => [[:a0, :b0, :c0], [:a1, :b1, :c1], [:a2, :b2, :c2], [:a3, :b3, :c3]]
```

When a block is given, calls the block with each of the sub-arrays (formed as above); returns `nil`:

```
a = [:a0, :a1, :a2, :a3]
b = [:b0, :b1, :b2, :b3]
c = [:c0, :c1, :c2, :c3]
a.zip(b, c) {|sub_array| p sub_array} # => nil
```

Output:

```
[:a0, :b0, :c0]
[:a1, :b1, :c1]
[:a2, :b2, :c2]
[:a3, :b3, :c3]
```

`array | other_array → new_array`

Returns the union of `array` and Array `other_array`; duplicates are removed; order is preserved; items are compared using `eql?`:

```
[0, 1] | [2, 3] # => [0, 1, 2, 3]
[0, 1, 1] | [2, 2, 3] # => [0, 1, 2, 3]
[0, 1, 2] | [3, 2, 1, 0] # => [0, 1, 2, 3]
```

Related: `Array#union`.