Table of Contents

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

4. Arrays

Like all high-level languages, Ruby has built-in support for *arrays*, objects that contain ordered lists of other objects. You can use arrays (often in conjunction with hashes) to build and use complex data structures without having to define any custom classes.

An array in Ruby is an ordered list of *elements*. Each element is a reference to some object, the way a Ruby variable is a reference to some object. For convenience, throughout this book we usually talk about arrays as though the array elements were the actual objects, not references to the objects. Since Ruby (unlike languages like C) gives no way of manipulating object references directly, the distinction rarely matters.

The simplest way to create a new array is to put a comma-separated list of object references between square brackets. The object references can be predefined variables (`my_var`), anonymous objects created on the spot (`'my string'`, `4.7`, or `MyClass.new`), or expressions (`a+b, object.method`). A single array can contain references to objects of many different types:

```
a1 = []                                 # => []
a2 = [1, 2, 3]                          # => [1, 2, 3]
a3 = [1, 2, 3, 'a', 'b', 'c', nil]      # => [1, 2, 3, "a", "b", "c", nil]

n1 = 4
n2 = 6
sum_and_difference = [n1, n2, n1+n2, n1-n2]
# => [4, 6, 10, -2]
```

If your array contains only strings, you may find it simpler to build your array by enclosing the strings in the `w{}` syntax, separated by whitespace. This saves you from having to write all those quotes and comma:

```
%w{1 2 3}                               # => ["1", "2", "3"]
%w{The rat sat
   on the mat}
# => ["The", "rat", "sat", "on", "the", "mat"]
```

The << operator is the simplest way to add a value to an array. Ruby dynamically resizes arrays as elements are added and removed.

```
a = [1, 2, 3]                   # => [1, 2, 3]
a << 4.0                        # => [1, 2, 3, 4.0]
a << 'five'                     # => [1, 2, 3, 4.0, "five"]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

An array element can be any object reference, including a reference to another array. An array can even contain a reference to itself, though this is usually a bad idea, since it can send your code into infinite loops.

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

As in most other programming languages, the elements of an array are numbered with *indexes* starting from zero. An array element can be looked up by passing its index into the array index operator `[]`. The first element of an array can be accessed with `a[0]`, the second with `a[1]`, and so on.

Negative indexes count from the end of the array: the last element of an array can be accessed with `a[-1]`, the second-to-last with `a[-2]`, and so on. See Recipe 4.13 for more ways of using the array indexing operator.

The size of an array is available through the `Array#size` method. Because the index numbering starts from zero, the index of the last element of an array is the size of the array, minus one.

```
a = [1, 2, 3, [4, 5, 6]]
a.size                         # => 4
a << a                         # => [1, 2, 3, [4, 5, 6], […]]
a.size                         # => 5

a[0]                           # => 1
a[3]                           # => [4, 5, 6]
a[3][0]                        # => 4
a[3].size                      # => 3

a[-2]                          # => [4, 5, 6]
a[-1]                          # => [1, 2, 3, [4, 5, 6], […]]
a[a.size-1]                    # => [1, 2, 3, [4, 5, 6], […]]

a[-1][-1]                      # => [1, 2, 3, [4, 5, 6], […]]
a[-1][-1][-1]                  # => [1, 2, 3, [4, 5, 6], […]]
```

All languages with arrays have constructs for iterating over them (even if it's just a `for` loop). Languages like Java and Python have general iterator methods similar to Ruby's, but they're usually used for iterating over arrays. In Ruby, iterators are the standard way of traversing all data structures: array iterators are just their simplest manifestation.

Ruby's array iterators deserve special study because they're Ruby's simplest and most accessible iterator methods. If you come to Ruby from another language, you'll probably start off thinking of iterator methods as letting you treat aspects of a data structure "like an array." Recipe 4.1 covers the basic array iterator methods, including ones in the `Enumerable` module that you'll encounter over and over again in different contexts.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `Set` class, included in Ruby's standard library, is a useful alternative to the `Array` class for many basic algorithms. A Ruby set models a mathematical set: sets are not ordered, and cannot contain more than one reference to the same object. For more about sets, see Recipes 4.14 and 4.15.

Recipe 4.1. Iterating Over an Array

Problem

You want to perform some operation on each item in an array.

Solution

Iterate over the array with `Enumerable#each`. Put into a block the code you want to execute for each item in the array.

```
[1, 2, 3, 4].each { |x| puts x }
# 1
# 2
# 3
# 4
```

If you want to produce a new array based on a transformation of some other array, use `Enumerable#collect` along with a block that takes one element and transforms it:

```
[1, 2, 3, 4].collect { |x| x ** 2 }             # => [1, 4, 9, 16]
```

Discussion

Ruby supports `for` loops and the other iteration constructs found in most modern programming languages, but its prefered idiom is a code block fed to an method like `each` or `collect`.

Methods like `each` and `collect` are called *generators* or *iterators*: they iterate over a data structure, `yield`ing one element at a time to whatever code block you've attached. Once your code block completes, they continue the iteration and `yield` the next item in the data structure (according to whatever definition of "next" the generator supports). These methods are covered in detail in Chapter 7.

In a method like `each`, the return value of the code block, if any, is ignored. Methods like `collect` take a more active role. After they `yield` an element of a data structure to a code block, they use the return value in some way. The `collect` method uses the return value of its attached block as an element in a new array.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Although commonly used in arrays, the `collect` method is actually defined in the `Enumerable` module, which the `Array` class includes. Many other Ruby classes (`Hash` and `Range` are just two) include the `Enumerable` methods; it's a sort of baseline for Ruby objects that provide iterators. Though `Enumerable` does not define the `each` method, it must be defined by any class that includes `Enumerable`, so you'll see that method a lot, too. This is covered in Recipe 9.4.

If you need to have the array indexes along with the array elements, use `Enumerable#each_with_index`.

```
['a', 'b', 'c'].each_with_index do |item, index|
  puts "At position #{index}: #{item}"
end
# At position 0: a
# At position 1: b
# At position 2: c
```

Ruby's `Array` class also defines several generators not seen in `Enumerable` . For instance , to iterate over a list in reverse order, use the `reverse_each` method:

```
[1, 2, 3, 4].reverse_each { |x| puts x }
# 4
# 3
# 2
# 1
```

`Enumerable#collect` has a destructive equivalent: `Array#collect`!, also known as `Arary#map`! (a helpful alias for Python programmers). This method acts just like `collect`, but instead of creating a new array to hold the return values of its calls to the code block, it *replaces* each item in the old array with the corresponding value from the code block. This saves memory and time, but it destroys the old array:

```
array = ['a', 'b', 'c']
array.collect! { |x| x.upcase }
array                                  # => ["A", "B", "C"]
array.map! { |x| x.downcase }
array                                  # => ["a", "b", "c"]
```

If you need to skip certain elements of an array, you can use the iterator methods `Range#step` and `Integer#upto` instead of `Array#each`. These methods generate a sequence of numbers that you can use as successive indexes into an array.

```
array = ['junk', 'junk', 'junk', 'val1', 'val2']
3.upto(array.length-1) { |i| puts "Value #{array[i]}" }
# Value val1
# Value val2

array = ['1', 'a', '2', 'b', '3', 'c']
(0..array.length-1).step(2) do |i|
  puts "Letter #{array[i]} is #{array[i+1]}"
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
end
# Letter 1 is a
# Letter 2 is b
# Letter 3 is c
```

Like most other programming languages, Ruby lets you define `for`, `while`, and `until` loops—but you shouldn't need them very often. The `for` construct is equivalent to `each`, whether it's applied to an array or a range:

```
for element in ['a', 'b', 'c']
  puts element
end
# a
# b
# c

for element in (1..3)
  puts element
end
# 1
# 2
# 3
```

The `while` and `until` constructs take a boolean expression and execute the loop while the expression is true (`while`)or until it becomes true (`until`). All three of the following code snippets generate the same output:

```
array = ['cherry', 'strawberry', 'orange']

for index in (0…array.length)
  puts "At position #{index}: #{array[index]}"
end

index = 0
while index < array.length
  puts "At position #{index}: #{array[index]}"
  index += 1
end

index = 0
until index == array.length
  puts "At position #{index}: #{array[index]}"
  index += 1
end

# At position 0: cherry
# At position 1: strawberry
# At position 2: orange
```

These constructs don't make for very idiomatic Ruby. You should only need to use them when you're iterating over a data structure in a way that doesn't already have an iterator method (for instance, if you're traversing a custom tree structure). Even then, it's more idiomatic if you only use them to define your own iterator methods.

The following code is a hybrid of `each` and `each_reverse`. It switches back and forth between iterating from the beginning of an array and iterating from its end.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
array = [1,2,3,4,5]
new_array = []
front_index = 0

back_index = array.length-1
while front_index <= back_index
  new_array << array[front_index]
  front_index += 1
  if front_index <= back_index
   new_array << array[back_index]
    back_index -= 1
  end
end
new_array                              # => [1, 5, 2, 4, 3]
```

That code works, but it becomes reusable when defined as an iterator. Put it into the `Array` class, and it becomes a universally accessible way of doing iteration, the colleague of `each` and `reverse_each`:

```
class Array
 def each_from_both_sides
    front_index = 0
    back_index = self.length-1
    while front_index <= back_index
      yield self[front_index]
      front_index += 1
      if front_index <= back_index
    yield self[back_index]
        back_index -= 1
      end
    end
  end
end

new_array = []
[1,2,3,4,5].each_from_both_sides { |x| new_array << x }
new_array                              # => [1, 5, 2, 4, 3]
```

This "burning the candle at both ends" behavior can also be defined as a `collect`type method: one which constructs a new array out of multiple calls to the attached code block. The implementation below delegates the actual iteration to the `each_from_both_sides` method defined above:

```
class Array
  def collect_from_both_sides
    new_array = []
    each_from_both_sides { |x| new_array << yield(x) }
    return new_array
  end
end

["ham", "eggs", "and"].collect_from_both_sides { |x| x.capitalize }
# => ["Ham", "And", "Eggs"]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Chapter 7, especially Recipe 7.5, "Writing an Iterator Over a Data Structure," and Recipe 7.9, "Looping Through Multiple Iterables in Parallel"

Recipe 4.2. Rearranging Values Without Using Temporary Variables

Problem

You want to rearrange a number of variables, or assign the elements of an array to individual variables.

Solution

Use a single assignment statement. Put the destination variables on the left-hand side, and line each one up with a variable (or expression) on the right side.

A simple swap:

```
a = 1
b = 2
a, b = b, a
a                                 # => 2
b                                 # => 1
```

A more complex rearrangement:

```
a, b, c = :red, :green, :blue
c, a, b = a, b, c
a                                 # => :green
b                                 # => :blue
c                                 # => :red
```

You can split out an array into its components:

```
array = [:red, :green, :blue]
c, a, b = array
a                                 # => :green
b                                 # => :blue
c                                 # => :red
```

You can even use the splat operator to extract items from the front of the array:

```
a, b, *c = [12, 14, 178, 89, 90]
a                                  # => 12
b                                  # => 14
c                                  # => [178, 89, 90]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

Ruby assignment statements are very versatile. When you put a comma-separated list of variables on the left-hand side of an assignment statement, it's equivalent to assigning each variable in the list the corresponding right-hand value. Not only does this make your code more compact and readable, it frees you from having to keep track of temporary variables when you swap variables.

Ruby works behind the scenes to allocate temporary storage space for variables that would otherwise be overwritten, so you don't have to do it yourself. You don't have to write this kind of code in Ruby:

```
a, b = 1, 2
x = a
a = b
b = x
```

The right-hand side of the assignment statement can get almost arbitrarily complicated:

```
a, b = 5, 10
a, b = b/a, a-1                        # => [2, 4]

a, b, c = 'A', 'B', 'C'
a, b, c = [a, b], { b => c }, a
a                                      # => ["A", "B"]
b                                      # => {"B"=>"C"}
c                                      # => "A"
```

If there are more variables on the left side of the equal sign than on the right side, the extra variables on the left side get assigned `nil`. This is usually an unwanted side effect.

```
a, b = 1, 2
a, b = b
a                                      # => 2
b                                      # => nil
```

One final nugget of code that is interesting enough to mention even though it has no legitimate use in Ruby: it doesn't save enough memory to be useful, and it's slower than doing a swap with an assignment. It's possible to swap two integer variables using bitwise XOR, without using any additional storage space at all (not even implicitly):

```
a, b = rand(1000), rand(1000)          # => [595, 742]
a = a ^ b                              # => 181
b = b ^ a                              # => 595
a = a ^ b                              # => 742

[a, b]                                 # => [742, 595]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

In terms of the cookbook metaphor, this final snippet is a dessert—no nutritional value, but it sure is tasty.

Recipe 4.3. Stripping Duplicate Elements from an Array

Problem

You want to strip all duplicate elements from an array, or prevent duplicate elements from being added in the first place.

Solution

Use `Array#uniq` to create a new array, based on an existing array but with no duplicate elements. `Array#uniq`! strips duplicate elements from an existing array.

```
survey_results = [1, 2, 7, 1, 1, 5, 2, 5, 1]
distinct_answers = survey_results.uniq        # => [1, 2, 7, 5]
survey_results.uniq!
survey_results                                # => [1, 2, 7, 5]
```

To ensure that duplicate values never get into your list, use a `Set` instead of an array. If you try to add a duplicate element to a `Set`, nothing will happen.

```
require 'set'
survey_results = [1, 2, 7, 1, 1, 5, 2, 5, 1]
distinct_answers = survey_results.to_set
# => #<Set: {5, 1, 7, 2}>

games = [["Alice", "Bob"], ["Carol", "Ted"],
         ["Alice", "Mallory"], ["Ted", "Bob"]]
players = games.inject(Set.new) { |set, game| game.each { |p| set << p }; set }
# => #<Set: {"Alice", "Mallory", "Ted", "Carol", "Bob"}>

players << "Ted"
# => #<Set: {"Alice", "Mallory", "Ted", "Carol", "Bob"}>
```

Discussion

The common element between these two solutions is the hash (see Chapter 5). `Array#uniq` iterates over an array, using each element as a key in a hash that it always checks to see if it encountered an element earlier in the iteration. A `Set` keeps the same kind of hash from the beginning, and rejects elements already in the hash. You see something that acts like an array, but it won't accept duplicates. In either case, two objects are considered "duplicates" if they have the same result for ==.

The return value of `Array#uniq` is itself an array, and nothing prevents you from adding duplicate elements to it later on. If you want to start enforcing uniqueness in perpetuity,

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

you should turn the array into a `Set` instead of calling `uniq`. Requiring the `set` library will define a new method `Enumerable#to_set`, which does this.

`Array#uniq` preserves the original order of the array (that is, the first instance of an object remains in its original location), but a `Set` has no order, because its internal implementation is a hash. To get array-like order in a `Set`, combine this recipe with Recipe 5.8 and subclass `Set` to use an `OrderedHash`:

```
class OrderedSet < Set
  def initialize
    @hash ||= OrderedHash.new
  end
end
```

Needing to strip all instances of a particular value from an array is a problem that often comes up. Ruby provides `Array#delete` for this task, and `Array#compact` for the special case of removing `nil` values.

```
a = [1, 2, nil, 3, 3, nil, nil, nil, 5]
a.compact                                 # => [1, 2, 3, 3, 5]

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

Recipe 4.4. Reversing an Array

Problem

Your array is the wrong way around: the last item should be first and the first should be last.

Solution

Use `reverse` to create a new array with the items reversed. Internal subarrays will not themselves be reversed.

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

Discussion

Like many operations on basic Ruby types, `reverse` has a corresponding method, `reverse!`, which reverses an array in place:

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


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Don't reverse an array if you just need to iterate over it backwards. Don't use a `for` loop either; the `reverse_each` iterator is more idiomatic.

See Also

- Recipe 1.4, "Reversing a String by Words or Characters"
- Recipe 4.1, "Iterating Over an Array," talks about using `Array#reverse_each` to iterate over an array in reverse order
- Recipe 4.2, "Rearranging Values Without Using Temporary Variables"

Recipe 4.5. Sorting an Array

Problem

You want to sort an array of objects, possibly according to some custom notion of what "sorting" means.

Solution

Homogeneous arrays of common data types, like strings or numbers, can be sorted "naturally" by just calling `Array#sort`:

```
[5.01, -5, 0, 5].sort                                  # => [-5, 0, 5, 5.01]
["Utahraptor", "Ankylosaur", "Maiasaur"].sort
# => ["Ankylosaur", "Maiasaur", "Utahraptor"]
```

To sort objects based on one of their data members, or by the results of a method call, use `Array#sort_by`. This code sorts an array of arrays by size, regardless of their contents:

```
arrays = [[1,2,3], [100], [10,20]]
arrays.sort_by { |x| x.size }                    # => [[100], [10, 20], [1, 2, 3]]
```

To do a more general sort, create a code block that compares the relevant aspect of any two given objects. Pass this block into the `sort` method of the array you want to sort.

This code sorts an array of numbers in ascending numeric order, except that the number 42 will always be at the end of the list:

```
[1, 100, 42, 23, 26, 10000].sort do |x, y|
  x == 42 ? 1 : x <=> y
end
# => [1, 23, 26, 100, 10000, 42]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

If there is one "canonical" way to sort a particular class of object, then you can have that class implement the <=> comparison operator. This is how Ruby automatically knows how to sort numbers in ascending order and strings in ascending ASCII order: `Numeric` and `String` both implement the comparison operator.

The `sort_by` method sorts an array using a Schwartzian transform (see Recipe 4.6 for an in-depth discussion). This is the most useful customized sort, because it's fast and easy to define. In this example, we use `sort_by` to sort on any one of an object's fields.

```
class Animal
 attr_reader :name, :eyes, :appendages

 def initialize(name, eyes, appendages)
  @name, @eyes, @appendages = name, eyes, appendages
 end

 def inspect
   @name
 end
end

animals = [Animal.new("octopus", 2, 8),
           Animal.new("spider", 6, 8),
           Animal.new("bee", 5, 6),
           Animal.new("elephant", 2, 4),
           Animal.new("crab", 2, 10)]

animals.sort_by { |x| x.eyes }
# => [octopus, elephant, crab, bee, spider]

animals.sort_by { |x| x.appendages }
# => [elephant, bee, octopus, spider, crab]
```

If you pass a block into sort, Ruby calls the block to make comparisons instead of using the comparison operator. This is the most general possible sort, and it's useful for cases where `sort_by` won't work.

The comparison operator and a `sort` code block both take one argument: an object against which to compare `self`. A call to <=> (or a `sort` code block) should return –1 if `self` is "less than" the given object (and should therefore show up before it in a sorted list). It should return 1 if `self` is "greater than" the given object (and should show up after it in a sorted list), and 0 if the objects are "`equal`" (and it doesn't matter which one shows up first). You can usually avoid remembering this by delegating the return value to some other object's <=> implementation.

See Also

- Recipe 4.6, "Ignoring Case When Sorting Strings," covers the workings of the Schwartzian Transform

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

- Recipe 4.7, "Making Sure a Sorted Array Stays Sorted"
- Recipe 4.10, "Shuffling an Array"
- If you need to find the minimum or maximum item in a list according to some criteria, don't sort it just to save writing some code; see Recipe 4.11, "Getting the N Smallest Items of an Array," for other options

Recipe 4.6. Ignoring Case When Sorting Strings

Problem

When you sort a list of strings, the strings beginning with uppercase letters sort before the strings beginning with lowercase letters.

```
list = ["Albania", "anteater", "zorilla", "Zaire"]
list.sort
# => ["Albania", "Zaire", "anteater", "zorilla"]
```

You want an alphabetical sort, regardless of case.

Solution

Use `Array#sort_by`. This is both the fastest and the shortest solution.

```
list.sort_by { |x| x.downcase }
# => ["Albania", "anteater", "Zaire", "zorilla"]
```

Discussion

The `Array#sort_by` method was introduced in Recipe 4.5, but it's worth discussing in detail because it's so useful. It uses a technique called a Schwartzian Transform. This common technique is like writing the following Ruby code (but it's a lot faster, because it's implemented in C):

```
list.collect { |s| [s.downcase, s] }.sort.collect { |subarray| subarray[1] }
```

It works like this: Ruby creates a new array containing two-element subarrays. Each subarray contains a value of `String#downcase`, along with the original string. This new array is sorted, and then the original strings (now sorted by their values for `String#downcase`) are recovered from the subarrays. `String#downcase` is called only once for each string.

A sort is the most common occurance of this pattern, but it shows up whenever an algorithm calls a particular method on the same objects over and over again. If you're not

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

sorting, you can't use Ruby's internal Schwartzian Transform, but you can save time by caching, or *memoizing*, the results of each distinct method call.

If you need to implement a Schwartzian Transform in Ruby, it's faster to use a hash than an array:

```
m = {}
list.sort { |x,y| (m[x] ||= x.downcase) <=> (m[y] ||= y.downcase) }
```

This technique is especially important if the method you need to call has side effects. You certainly don't want to call such methods more than once!

See Also

- The Ruby FAQ, question 9.15
- Recipe 4.5, "Sorting an Array"

Recipe 4.7. Making Sure a Sorted Array Stays Sorted

Problem

You want to make sure an array stays sorted, even as you replace its elements or add new elements to it.

Solution

Subclass `Array` and override the methods that add items to the array. The new implementations add every new item to a position that maintains the sortedness of the array.

As you can see below, there are a lot of these methods. If you can guarantee that a particular method will never be called, you can get away with not overriding it.

```
class SortedArray < Array

  def initialize(*args, &sort_by)
    @sort_by = sort_by || Proc.new { |x,y| x <=> y }
    super(*args)
    sort! &sort_by
  end

  def insert(i, v)
    # The next line could be further optimized to perform a
    # binary search.
    insert_before = index(find { |x| @sort_by.call(x, v) == 1 })
    super(insert_before ? insert_before : -1, v)
  end

  def <<(v)
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
    insert(0, v)
  end

  alias push <<
  alias unshift <<
```

Some methods, like `collect!`, can modify the items in an array, taking them out of sort order. Some methods, like `flatten!`, can add new elements to strange places in an array. Rather than figuring out a way to implement these methods in a way that preserves the sortedness of the array, we'll just let them run and then re-sort the array.[1]

[1] We can't use `define_method` to define these methods because in Ruby 1.8 you can't use `define_method` to create a method that takes a block argument. See Chapter 10 for more on this.

```
  ["collect!", "flatten!", "[]="].each do |method_name|
    module_eval %{
      def #{method_name}(*args)
        super
        sort! &@sort_by
      end
    }
  end

  def reverse!
    #Do nothing; reversing the array would disorder it.
  end
end
```

A `SortedArray` created from an unsorted array will end up sorted:

```
a = SortedArray.new([3,2,1])         # => [1, 2, 3]
```

Discussion

Many methods of `Array` are much faster on sorted arrays, so it's often useful to expend some overhead on keeping an array sorted over time. Removing items from a sorted array won't unsort it, but adding or modifying items can. Keeping a sorted array sorted means intercepting and reimplementing every sneaky way of putting objects into the array.

The `SortedArray` constructor accepts any code block you can pass into `Array#sort`, and keeps the array sorted according to that code block. The default code block uses the comparison operator (<=>) used by `sort`.

```
unsorted= ["b", "aa", "a", "cccc", "1", "zzzzz", "k", "z"]
strings_by_alpha = SortedArray.new(unsorted)
# => ["1", "a", "aa", "b", "cccc", "k", "z", "zzzzz"]
strings_by_length = SortedArray.new(unsorted) do |x,y|
  x.length <=> y.length
end
# => ["b", "z", "a", "k", "1", "aa", "cccc", "zzzzz"]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The methods that add elements to an array specify where in the array they operate: `push` operates on the end of the array, and `insert` operates on a specified spot. `SortedArray` responds to these methods but it ignores the caller's request to put elements in a certain place. Every new element is inserted into a position that keeps the array sorted.

```
a << -1                          # => [-1, 1, 2, 3]
a << 1.5                         # => [-1, 1, 1.5, 2, 3]
a.push(2.5)                      # => [-1, 1, 1.5, 2, 2.5, 3]
a.unshift(1.6)                   # => [-1, 1, 1.5, 1.6, 2, 2.5, 3]
```

For methods like `collect!` and array assignment (`[]=`)that allow complex changes to an array, the simplest solution is to allow the changes to go through and then re-sort:

```
a = SortedArray.new([10, 6, 4, -4, 200, 100])
# => [-4, 4, 6, 10, 100, 200]
a.collect! { |x| x * -1 }          # => [-200, -100, -10, -6, -4, 4]

a[3] = 25
a                                  # => [-200, -100, -10, -4, 4, 25]
# That is, -6 has been replaced by 25 and the array has been re-sorted.

a[1..2] = [6000, 10, 600, 6]
a                                  # => [-200, -4, 4, 6, 10, 25, 600, 6000]
# That is, -100 and -10 have been replaced by 6000, 10, 600, and 6,
# and the array has been re-sorted.
```

But with a little more work, we can write a more efficient implementation of array assignment that gives the same behavior. What happens when you run a command like `a[0]= 10` on a `SortedArray`? The first element in the `SortedArray` is replaced by 10, and the `SortedArray` is re-sorted. This is equivalent to removing the first element in the array, then adding the value 10 to a place in the array that keeps it sorted.

`Array#[]=` implements three different types of array assignment, but all three can be modeled as a series of removals followed by a series of insertions. We can use this fact to implement a more efficient version of `SortedArray#[]=`:.

```
class SortedArray
  def []=(*args)
    if args.size == 3
      #e.g. "a[6,3] = [1,2,3]"
      start, length, value = args
      slice! Range.new(start, start+length, true)
      (value.respond_to? :each) ? value.each { |x| self << x } : self << value
    elsif args.size == 2
      index, value = args
      if index.is_a? Numeric
        #e.g. "a[0] = 10" (the most common form of array assignment)
        delete_at(index)
        self << value
      elsif index.is_a? Range
        #e.g. "a[0..3] = [1,2,3]"
        slice! index
        (value.respond_to? :each) ? value.each { |x| self << x } : self << value
      else
        #Not supported. Delegate to superclass; will probably give an error.
        super
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
        sort!(&sort_by)
      end
    else
      #Not supported. Delegate to superclass; will probably give an error.
      super
      sort!(&sort_by)
    end
  end
end
```

Just as before, the sort will be maintained even when you use array assignment to replace some of a `SortedArray`'s elements with other objects. But this implementation doesn't have to re-sort the array every time.

```
a = SortedArray.new([1,2,3,4,5,6])
a[0] = 10
a                                    # => [2, 3, 4, 5, 6, 10]

a[0, 2] = [100, 200]
a                                    # => [4, 5, 6, 10, 100, 200]

a[1..2] = [-4, 6]
a                                    # => [-4, 4, 6, 10, 100, 200]
```

It's possible to subvert the sortedness of a `SortedArray` by modifying an object in place in a way that changes its sort order. Since the `SortedArray` never hears about the change to this object, it has no way of updating itself to move that object to its new sort position: [2]

[2] One alternative is to modify `SortedArray[]` so that when you look up an element of the array, you actually get a delegate object that intercepts all of the element's method calls, and re-sorts the array whenever the user calls a method that modifies the element in place. This is probably overkill.

```
stripes = SortedArray.new(["aardwolf", "zebrafish"])
stripes[1].upcase!
stripes                              # => ["aardwolf", "ZEBRAFISH"]
stripes.sort!                        # => ["ZEBRAFISH", "aardwolf"]
```

If this bothers you, you can make a `SortedArray` keep frozen copies of objects instead of the objects themselves. This solution hurts performance and uses more memory, but it will also prevent objects from being modified after being put into the `SortedArray`. This code adds a convenience method to `Object` that makes a frozen copy of the object:

```
class Object
  def to_frozen
    f = self
    unless frozen?
       begin
         f = dup.freeze
       rescue TypeError
         #This object can't be duped (e.g. Fixnum); fortunately,
         #it usually can't be modified either
       end
    end
    return f
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `FrozenCopySortedArray` stores frozen copies of objects instead of the objects themselves:

```
class FrozenCopySortedArray < SortedArray
  def insert(i, v)
    insert_before = index(find { |x| x > v })
    super(insert_before ? insert_before : -1, v.to_frozen)
  end

  ["initialize", "collect!", "flatten!"].each do |method_name|
    define_method(method_name) do
      super
      each_with_index { |x, i| self[i] = x.to_frozen }
      # No need to sort; by doing an assignment to every element
      # in the array, we've made #insert keep the array sorted.
    end
  end
end

stripes = SortedArray.new(["aardwolf", "zebrafish"])
stripes[1].upcase!
# TypeError: can't modify frozen string
```

Unlike a regular array, which can have elements of arbitrarily different data classes, all the elements of a `SortedArray` must be mutually comparable. For instance, you can mix integers and floating-point numbers within a `SortedArray`, but you can't mix integers and strings. Any data set that would cause `Array#sort` to fail makes an invalid `SortedArray`:

```
[1, "string"].sort
# ArgumentError: comparison of Fixnum with String failed

a = SortedArray.new([1])
a << "string"
# ArgumentError: comparison of Fixnum with String failed
```

One other pitfall: operations that create a new object, such as `|=`, `+=`, and `to_a` will turn an `SortedArray` into a (possibly unsorted) array.

```
a = SortedArray.new([3, 2, 1])       # => [1, 2, 3]
a += [1, -10]                        # => [1, 2, 3, 1, -10]
a.class                              # => Array
```

The simplest way to avoid this is to override these methods to transform the resulting array back into a `SortedArray`:

```
class SortedArray
  def + (other_array)
    SortedArray.new(super)
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- Recipe 4.11, "Getting the N Smallest Items of an Array," uses a `SortedArray`
- If you're going to do a lot of insertions and removals, a red-black tree may be faster than a `SortedArray`; you can choose from a pure Ruby implementation (http://www.germane-software.com/software/Utilities/RBTree/) and one that uses a C extension for speed (http://www.geocities.co.jp/SiliconValley-PaloAlto/3388/rbtree/README.html)

Recipe 4.8. Summing the Items of an Array

Problem

You want to add together many objects in an array.

Solution

There are two good ways to accomplish this in Ruby. Plain vanilla iteration is a simple way to approach the problem:

```
collection = [1, 2, 3, 4, 5]
sum = 0
collection.each {|i| sum += i}
sum                                     # => 15
```

However this is such a common action that Ruby has a special iterator method called `inject`, which saves a little code:

```
collection = [1, 2, 3, 4, 5]
collection.inject(0) {|sum, i| sum + i}      # => 15
```

Discussion

Notice that in the `inject` solution, we didn't need to define the variable `total` variable outside the scope of iteration. Instead, its scope moved into the iteration. In the example above, the initial value for `total` is the first argument to `inject`. We changed the += to + because the block given to `inject` is evaluated on each value of the collection, and the `total` variable is set to its output every time.

You can think of the `inject` example as equivalent to the following code:

```
collection = [1, 2, 3, 4, 5]
sum = 0
sum = sum + 1
sum = sum + 2
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
sum = sum + 3
sum = sum + 4
sum = sum + 5
```

Although `inject` is the preferred way of summing over a collection, `inject` is generally a few times slower than `each`. The speed difference does not grow exponentially, so you don't need to always be worrying about it as you write code. But after the fact, it's a good idea to look for `inject` calls in crucial spots that you can change to use faster iteration methods like `each`.

Nothing stops you from using other kinds of operators in your `inject` code blocks. For example, you could multiply:

```
collection = [1, 2, 3, 4, 5]
collection.inject(1) {|total, i| total * i}  # => 120
```

Many of the other recipes in this book use `inject` to build data structures or run calculations on them.

See Also

- Recipe 2.8, "Finding Mean, Median, and Mode"
- Recipe 4.12, "Building Up a Hash Using Injection"
- Recipe 5.12, "Building a Histogram"

Recipe 4.9. Sorting an Array by Frequency of Appearance

Problem

You want to sort an array so that its least-frequently-appearing items come first.

Solution

Build a histogram of the frequencies of the objects in the array, then use it as a lookup table in conjunction with the `sort_by` method.

The following method puts the least frequently-appearing objects first. Objects that have the same frequency are sorted normally, with the comparison operator.

```
module Enumerable
  def sort_by_frequency
    histogram = inject(Hash.new(0)) { |hash, x| hash[x] += 1; hash}
    sort_by { |x| [histogram[x], x] }
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
[1,2,3,4,1,2,4,8,1,4,9,16].sort_by_frequency
# => [3, 8, 9, 16, 2, 2, 1, 1, 1, 4, 4, 4]
```

Discussion

The `sort_by_frequency` method uses `sort_by`, a method introduced in Recipe 4.5 and described in detail in Recipe 4.6. The technique here is a little different from other uses of `sort_by`, because it sorts by two different criteria. We want to first compare the relative frequencies of two items. If the relative frequencies are equal, we want to compare the items themselves. That way, all the instances of a given item will show up together in the sorted list.

The block you pass to `Enumerable#sort_by` can return only a single sort key for each object, but that sort key can be an array. Ruby compares two arrays by comparing their corresponding elements, one at a time. As soon as an element of one array is different from an element of another, the comparison stops, returning the comparison of the two different elements. If one of the arrays runs out of elements, the longer one sorts first. Here are some quick examples:

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

In our case, all the arrays contain two elements: the relative frequency of an object in the array, and the object itself. If two objects have different frequencies, the first elements of their arrays will differ, and the items will be sorted based on their frequencies. If two items have the same frequency, the first element of each array will be the same. The comparison method will move on to the second array element, which means the two objects will be sorted based on their values.

If you don't mind elements with the same frequency showing up in an unsorted order, you can speed up the sort a little by comparing only the histogram frequencies:

```
module Enumerable
  def sort_by_frequency_faster
    histogram = inject(Hash.new(0)) { |hash, x| hash[x] += 1; hash}
    sort_by { |x| histogram[x] }
  end
end

[1,2,3,4,1,2,4,8,1,4,9,16].sort_by_frequency_faster
# => [16, 8, 3, 9, 2, 2, 4, 1, 1, 4, 4, 1]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

To sort the list so that the most-frequently-appearing items show up first, either invert the result of `sort_by_frequency`, or multiply the histogram values by –1 when passing them into `sort_by`:

```
module Enumerable
  def sort_by_frequency_descending
    histogram = inject(Hash.new(0)) { |hash, x| hash[x] += 1; hash}
    sort_by { |x| [histogram[x] * -1, x]}
  end
end

[1,2,3,4,1,2,4,8,1,4,9,16].sort_by_frequency_descending
# => [1, 1, 1, 4, 4, 4, 2, 2, 3, 8, 9, 16]
```

If you want to sort a list by the frequency of its elements, but not have repeated elements actually show up in the sorted list, you can run the list through `Array#uniq` after sorting it. However, since the keys of the histogram are just the distinct elements of the array, it's more efficient to sort the keys of the histogram and return those:

```
module Enumerable
  def sort_distinct_by_frequency
    histogram = inject(Hash.new(0)) { |hash, x| hash[x] += 1; hash }
    histogram.keys.sort_by { |x| [histogram[x], x] }
  end
end

[1,2,3,4,1,2,4,8,1,4,9,16].sort_distinct_by_frequency
# => [3, 8, 9, 16, 2, 1, 4]
```

See Also

- Recipe 4.5, "Sorting an Array"
- Recipe 5.12, "Building a Histogram"

Recipe 4.10. Shuffling an Array

Problem

You want to put the elements of an array in random order.

Solution

The simplest way to shuffle an array (in Ruby 1.8 and above) is to sort it randomly:

```
[1,2,3].sort_by { rand } # => [1, 3, 2]
```

This is not the fastest way, though.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

It's hard to beat a random sort for brevity of code, but it does a lot of extra work. Like any general sort, a random sort will do about *n log n* variable swaps. But to shuffle a list, it suffices to put a randomly selected element in each position of the list. This can be done with only *n* variable swaps.

```
class Array
  def shuffle!
    each_index do |i|
      j = rand(length-i) + i
      self[j], self[i] = self[i], self[j]
    end
  end

  def shuffle
    dup.shuffle!
  end
end
```

If you're shuffling a very large list, either `Array#shuffle` or `Array#shuffle!` will be significantly faster than a random sort. Here's a real-world example of shuffling using `Array#shuffle`:

```
class Card
  def initialize(suit, rank)
    @suit = suit
    @rank = rank
  end

  def to_s
    "#{@suit} of #{@rank}"
  end
end

class Deck < Array
  attr_reader :cards
  @@suits = %w{Spades Hearts Clubs Diamonds}
  @@ranks = %w{Ace 2 3 4 5 6 7 8 9 10 Jack Queen King}

  def initialize
    @@suits.each { |suit| @@ranks.each { |rank| self << Card.new(rank, suit) } }
  end
end

deck = Deck.new
deck.collect { |card| card.to_s }
# => ["Ace of Spades", "2 of Spades", "3 of Spades", "4 of Spades",…]
deck.shuffle!
deck.collect { |card| card.to_s }
# => ["6 of Clubs", "8 of Diamonds", "2 of Hearts", "5 of Clubs",…]
```

See Also

- Recipe 2.5, "Generating Random Numbers"
- The Facets Core library provides implementations of `Array#shuffle` and `Array#shuffle!`

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Recipe 4.11. Getting the N Smallest Items of an Array

Problem

You want to find the smallest few items in an array, or the largest, or the most extreme according to some other measure.

Solution

If you only need to find the single smallest item according to some measure, use `Enumerable#min`. By default, it uses the <=> method to see whether one item is "smaller" than another, but you can override this by passing in a code block.

```
[3, 5, 11, 16].min
# => 3
["three", "five", "eleven", "sixteen"].min
# => "eleven"
["three", "five", "eleven", "sixteen"].min { |x,y| x.size <=> y.size }
# => "five"
```

Similarly, if you need to find the single largest item, use `Enumerable#max`.

```
[3, 5, 11, 16].max
# => 16
["three", "five", "eleven", "sixteen"].max
# => "three"
["three", "five", "eleven", "sixteen"].max { |x,y| x.size <=> y.size }
# => "sixteen"
```

By default, arrays are sorted by their natural order: numbers are sorted by value, strings by their position in the ASCII collating sequence (basically alphabetical order, but all lowercase characters precede all uppercase characters). Hence, in the previous examples, "three" is the largest string, and "eleven" the smallest.

It gets more complicated when you need to get a number of the smallest or largest elements according to some measurement: say, the top 5 or the bottom 10. The simplest solution is to sort the list and skim the items you want off of the top or bottom.

```
l = [1, 60, 21, 100, -5, 20, 60, 22, 85, 91, 4, 66]
sorted = l.sort

#The top 5
sorted[-5…sorted.size]
# => [60, 66, 85, 91, 100]

#The bottom 5
sorted[0…5]
# => [-5, 1, 4, 20, 21]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Despite the simplicity of this technique, it's inefficient to sort the entire list unless the number of items you want to extract approaches the size of the list.

Discussion

The `min` and `max` methods work by picking the first element of the array as a "champion," then iterating over the rest of the list trying to find an element that can beat the current champion on the appropriate metric. When it finds one, that element becomes the new champion. An element that can beat the old champion can also beat any of the other contenders seen up to that point, so one run through the list suffices to find the maximum or minimum.

The naive solution to finding more than one smallest item is to repeat this process multiple times. Iterate over the `Array` once to find the smallest item, then iterate over it again to find the next-smallest item, and so on. This is naive for the same reason a bubble sort is naive: you're repeating many of your comparisons more times than necessary. Indeed, if you run this algorithm once for every item in the array (trying to find the n smallest items in an array of *n* items), you get a bubble sort.

Sorting the list beforehand is better when you need to find more than a small fraction of the items in the list, but it's possible to do better. After all, you don't really want to sort the whole list: you just want to sort the bottom of the list to find the smallest items. You don't care if the other elements are unsorted because you're not interested in those elements anyway.

To sort only the smallest elements, you can keep a sorted "stable" of champions, and kick the largest champion out of the stable whenever you find an element that's smaller. If you encounter a number that's too large to enter the stable, you can ignore it from that point on. This process rapidly cuts down on the number of elements you must consider, making this approach faster than doing a sort.

The `SortedList` class from Recipe 4.7 is useful for this task. The `min_n` method below creates a `SortedList` "stable" that keeps its elements sorted based on the same block being used to find the minimum. It keeps the stable at a certain size by kicking out the largest item in the stable whenever a smaller item is found. The `max_n` method works similarly, but the comparisons are reversed, and the smallest element in the stable is kicked out when a larger element is found.

```
module Enumerable
  def min_n(n, &block)
    block = Proc.new { |x,y| x <=> y } if block == nil
    stable = SortedArray.new(&block)
    each do |x|
      stable << x if stable.size < n or block.call(x, stable[-1]) == -1
      stable.pop until stable.size <= n
    end
    return stable
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
  end

  def max_n(n, &block)
    block = Proc.new { |x,y| x <=> y } if block == nil
    stable = SortedArray.new(&block)
    each do |x|
      stable << x if stable.size < n or block.call(x, stable[0]) == 1
      stable.shift until stable.size <= n
    end
    return stable
  end
end

l = [1, 60, 21, 100, -5, 20, 60, 22, 85, 91, 4, 66]
l.max_n(5)
# => [60, 66, 85, 91, 100]
l.min_n(5)
# => [-5, 1, 4, 20, 21]

l.min_n(5) { |x,y| x.abs <=> y.abs }
# => [1, 4, -5, 20, 21]
```

See Also

- Recipe 4.7, "Making Sure a Sorted Array Stays Sorted"

Recipe 4.12. Building Up a Hash Using Injection

Problem

You want to create a hash from the values in an array.

Solution

As seen in Recipe 4.8, the most straightforward way to solve this kind of problem is to use `Enumerable#inject`. The `inject` method takes one parameter (the object to build up, in this case a hash), and a block specifying the action to take on each item. The block takes two parameters: the object being built up (the hash), and one of the items from the array.

Here's a straightforward use of `inject` to build a hash out of an array of key-value pairs:

```
collection = [ [1, 'one'], [2, 'two'], [3, 'three'],
               [4, 'four'], [5, 'five']
             ]

collection.inject({}) do |hash, value|
  hash[value.first] = value.last
  hash
end
# => {5=>"five", 1=>"one", 2=>"two", 3=>"three", 4=>"four"}
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Discussion

Why is there that somewhat incongrous expression `hash` at the end of the `inject` block above? Because the next time it calls the block, `inject` uses the value it got from the block the last time it called the block. When you're using `inject` to build a data structure, the last line of code in the block should evaluate to the object you're building up: in this case, our hash.

This is probably the most common `inject`-related gotcha. Here's some code that doesn't work:

```
collection.dup.inject({}) { |hash, value| hash[value.first] = value.last }
# IndexError: index 3 out of string
```

Why doesn't this work? Because hash assignment returns the assigned value, not the hash.

```
Hash.new["key"] = "some value"       # => "some value"
```

In the broken example above, when `inject` calls the code block for the second and subsequent times, it does not pass the hash as the code block's first argument. It passes in the last value to be assigned to the hash. In this case, that's a string (maybe "one" or "four"). The hash has been lost forever, and the `inject` block crashes when it tries to treat a string as a hash.

`Hash#update` can be used like hash assignment, except it returns the hash instead of the assigned value (and it's slower). So this code will work:

```
collection.inject({}) do |hash, value|
  hash.update value.first => value.last
end
# => {5=>"five", 1=>"ontwo", 2=>"two", 3=>"three", 4=>"four"}
```

Ryan Carver came up with a more sophisticated way of building a hash out of an array: define a general method for all arrays called `to_h`.

```
class Array
  def to_h(default=nil)
    Hash[ *inject([]) { |a, value| a.push value, default || yield(value) } ]
  end
end
```

The magic of this method is that you can provide a code block to customize how keys in the array are mapped to values.

```
a = [1, 2, 3]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
a.to_h(true)
# => {1=>true, 2=>true, 3=>true}

a.to_h { |value| [value * -1, value * 2] }
# => {1=>[-1, 2], 2=>[-2, 4], 3=>[-3, 6]}
```

References

- Recipe 5.3, "Adding Elements to a Hash"
- Recipe 5.12, "Building a Histogram"
- The original definition of `Array#to_h`:(http://fivesevensix.com/posts/2005/05/20/array-to_h)

Recipe 4.13. Extracting Portions of Arrays

Problem

Given an array, you want to retrieve the elements of the array that occupy certain positions or have certain properties. You might to do this in a way that removes the matching elements from the original array.

Solution

To gather a chunk of an array without modifying it, use the array retrieval operator `Array#[]`, or its alias `Array#slice`.

The array retrieval operator has three forms, which are the same as the corresponding forms for substring accesses. The simplest and most common form is `array[index]`.It takes a number as input, treats it as an index into the array, and returns the element at that index. If the input is negative, it counts from the end of the array. If the array is smaller than the index, it returns `nil`. If performance is a big consideration for you, `Array#at` will do the same thing, and it's a little faster than `Array#[]`:

```
a = ("a".."h").to_a               # => ["a", "b", "c", "d", "e", "f", "g", "h"]

a[0]                              # => "a"
a[1]                              # => "b"

a.at(1)                           # => "b"
a.slice(1)                        # => "b"
a[-1]                             # => "h"
a[-2]                             # => "g"
a[1000]                           # => nil
a[-1000]                          # => nil
```

The second form is `array[range]`. This form retrieves every element identified by an index in the given range, and returns those elements as a new array.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

A range in which both numbers are negative will retrieve elements counting from the end of the array. You can mix positive and negative indices where that makes sense:

```
a[2..5]                                 # => ["c", "d", "e", "f"]
a[2…5]                                # => ["c", "d", "e"]
a[0..0]                                 # => ["a"]
a[1..-4]                                # => ["b", "c", "d", "e"]
a[5..1000]                              # => ["f", "g", "h"]

a[2..0]                                 # => []
a[0…0]                                # => []

a[-3..2]                                # => []
```

The third form is `array[start_index, length]`. This is equivalent to `array[range.new(start_index…start_index+length)]`.

```
a[2, 4]                                 # => ["c", "d", "e", "f"]
a[2, 3]                                 # => ["c", "d", "e"]
a[0, 1]                                 # => ["a"]
a[1, 2]                                 # => ["b", "c"]
a[-4, 2]                                # => ["e", "f"]
a[5, 1000]                              # => ["f", "g", "h"]
```

To remove a slice from the array, use `Array#slice!`. This method takes the same arguments and returns the same results as `Array#slice`, but as a side effect, the objects it retrieves are removed from the array.

```
a.slice!(2..5)                          # => ["c", "d", "e", "f"]
a                                       # => ["a", "b", "g", "h"]

a.slice!(0)                             # => "a"
a                                       # => ["b", "g", "h"]

a.slice!(1,2)                           # => ["g", "h"]
a                                       # => ["b"]
```

Discussion

The `Array` methods `[]`, `slice`, and `slice!` work well if you need to extract one particular elements, or a set of adjacent elements. There are two other main possibilities: you might need to retrieve the elements at an arbitrary set of indexes, or (a catch-all) you might need to retrieve all elements with a certain property that can be determined with a code block.

To nondestructively gather the elements at particular indexes in an array, pass in any number of indices to `Array#values_at`. Results will be returned in a new array, in the same order they were requested.

```
a = ("a".."h").to_a             # => ["a", "b", "c", "d", "e", "f", "g", "h"]
a.values_at(0)                  # => ["a"]
a.values_at(1, 0, -2)           # => ["b", "a", "g"]
a.values_at(4, 6, 6, 7, 4, 0, 3)# => ["e", "g", "g", "h", "e", "a", "d"]
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`Enumerable#find_all` finds all elements in an array (or other class with `Enumerable` mixed in)for which the specified code block returns true. `Enumerable#reject` will find all elements for which the specified code block returns false.

```
a.find_all { |x| x < "e" }       # => ["a", "b", "c", "d"]
a.reject { |x| x < "e" }         # => ["e", "f", "g", "h"]
```

To find all elements in an array that match a regular expression, you can use `Enumerable#grep` instead of defining a block that does the regular expression match:

```
a.grep /[aeiou]/                 # => ["a", "e"]
a.grep /[^g]/                    # => ["a", "b", "c", "d", "e", "f", "h"]
```

It's a little tricky to implement a destructive version of `Array#values_at`, because removing one element from an array changes the indexes of all subsequent elements. We can let Ruby do the work, though, by replacing each element we want to remove with a dummy object that we know cannot already be present in the array. We can then use the C-backed method `Array#delete` to remove all instances of the dummy object from the array. This is much faster than using `Array#slice`! to remove elements one at a time, because each call to `Array#slice`! forces Ruby to rearrange the array to be contiguous.

If you know that your array contains no `nil` values, you can set your undesired values to `nil`, then use use `Array#compress`! to remove them. The solution below is more general.

```
class Array
  def strip_values_at!(*args)
      #For each mentioned index, replace its value with a dummy object.
      values = []
      dummy = Object.new
      args.each do |i|
        if i < size
         values << self[i]
          self[i] = dummy
      end
      #Strip out the dummy object.
      delete(dummy)
      return values
    end
  end
end

a = ("a".."h").to_a
a.strip_values_at!(1, 0, -2)         # => ["b", "a", "g"]
a                                    # => ["c", "d", "e", "f", "h"]

a.strip_values_at!(1000)             # => []
a                                    # => ["c", "d", "e", "f", "h"]
```

`Array#reject`! removes all items from an array that match a code block, but it doesn't return the removed items, so it won't do for a destructive equivalent of

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

`Enumerable#find_all`. This implementation of a method called `extract!` picks up where `Array#reject!` leaves off:

```
class Array
  def extract!
    ary = self.dup
    self.reject! { |x| yield x }
    ary - self
  end
end

a = ("a".."h").to_a
a.extract! { |x| x < "e" && x != "b" }    # => ["a", "c", "d"]
a                                         # => ["b", "e", "f", "g", "h"]
```

Finally, a convenience method called `grep_extract!` provides a method that destructively approximates the behavior of `Enumerable#grep`.

```
class Array
   def grep_extract!(re)
   extract! { |x| re.match(x) }
   end
end

a = ("a".."h").to_a
a.grep_extract!(/[aeiou]/)           # => ["a", "e"]
a                                    # => ["b", "c", "d", "f", "g", "h"]
```

See Also

- Strings support the array lookup operator, `slice, slice!`, and all the methods of `Enumerable`, so you can treat them like arrays in many respects; see Recipe 1.13, "Getting the Parts of a String You Want"

Recipe 4.14. Computing Set Operations on Arrays

Problem

You want to find the union, intersection, difference, or Cartesian product of two arrays, or the complement of a single array with respect to some universe.

Solution

`Array` objects have overloaded arithmetic and logical operators to provide the three simplest set operations:

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

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


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

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

`Set` objects overload the same operators, as well as the exclusive-or operator (`^`).If you already have `Array`s, though, it's more efficient to deconstruct the XOR operation into its three component operations.

```
require 'set'
a = [1,2,3]
b = [3,4,5]
a.to_set ^ b.to_set                 # => #<Set: {5, 1, 2, 4}>
(a | b) - (a & b)                   # => [1, 2, 4, 5]
```

Discussion

`Set` objects are intended to model mathematical sets: where arrays are ordered and can contain duplicate entries, `Set`s model an unordered collection of unique items. `Set` not only overrides operators for set operations, it provides English-language aliases for the three most common operators: `Set#union`, `Set#intersection`, and `Set#difference`. An array can only perform a set operation on another array, but a `Set` can perform a set operation on any `Enumerable`.

```
array = [1,2,3]
set = [3,4,5].to_s
array & set                     # => TypeError: can't convert Set into Array
set & array                     # => #<Set: {3}>
```

You might think that `Set` objects would be optimized for set operations, but they're actually optimized for constant-time membership checks (internally, a `Set` is based on a hash). Set union is faster when the left-hand object is a `Set` object, but intersection and difference are significantly faster when both objects are arrays. It's not worth it to convert arrays into `Set`s just so you can say you performed set operations on `Set` objects.

The union and intersection set operations remove duplicate entries from arrays. The difference operation does not remove duplicate entries from an array except as part of a subtraction.

```
[3,3] & [3,3]                       # => [3]
[3,3] | [3,3]                       # => [3]
[1,2,3,3] - [1]                     # => [2, 3, 3]
[1,2,3,3] - [3]                     # => [1, 2]
[1,2,3,3] - [2,2,3]                 # => [1]
```

Complement

If you want the complement of an array with respect to some small universe, create that universe and use the difference operation:

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

```
u = [:red, :orange, :yellow, :green, :blue, :indigo, :violet]
a = [:red, :blue]
u - a                    # => [:orange, :yellow, :green, :indigo, :violet]
```

More often, the relevant universe is infinite (the set of natural numbers)or extremely large (the set of three-letter strings). The best strategy here is to define a generator and use it to iterate through the complement. Be sure to break when you're done; you don't want to iterate over an infinite set.

```
def natural_numbers_except(exclude)
  exclude_map = {}
  exclude.each { |x| exclude_map[x] = true }
  x = 1
  while true
    yield x unless exclude_map[x]
    x = x.succ
  end
end

natural_numbers_except([2,3,6,7]) do |x|
  break if x > 10
  puts x
end
# 1
# 4
# 5
# 8
# 9
# 10
```

Cartesian product

To get the Cartesian product of two arrays, write a nested iteration over both lists and append each pair of items to a new array. This code is attached to `Enumerable` so you can also use it with `Set`s or any other `Enumerable`.

```
module Enumberable
  def cartesian(other)
    res = []
    each { |x| other.each { |y| res << [x, y] } }
    return res
  end
end

[1,2,3].cartesian(["a",5,6])
# => [[1, "a"], [1, 5], [1, 6],
#     [2, "a"], [2, 5], [2, 6],
#     [3, "a"], [3, 5], [3, 6]
```

This version uses `Enumerable#inject` to make the code more concise; however, the original version is more efficient.

```
module Enumerable
  def cartesian(other)
    inject([]) { |res, x| other.inject(res) { |res, y| res << [x,y] } }
  end
end
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

See Also

- See Recipe 2.5, "Generating Random Numbers," for an example (constructing a deck of cards from suits and ranks)that could benefit from a function to calculate the Cartesian product
- Recipe 2.10, "Multiplying Matrices"

Recipe 4.15. Partitioning or Classifying a Set

Problem

You want to partition a `Set` or array based on some attribute of its elements. All elements that go "together" in some code-specific sense should be grouped together in distinct data structures. For instance, if you're partitioning by color, all the green objects in a `Set` should be grouped together, separate from the group of all the red objects in the `Set`.

Solution

Use `Set#divide`, passing in a code block that returns the partition of the object it's passed. The result will be a new `Set` containing a number of partitioned subsets of your original `Set`.

The code block can accept either a single argument or two arguments.[3] The single-argument version examines each object to see which subset it should go into.

[3] This is analogous to the one-argument code block passed into `Enumerable#sort_by` and the two-argument code block passed into `Array#sort`.

```
require 'set'
s = Set.new((1..10).collect)
# => #<Set: {5, 6, 1, 7, 2, 8, 3, 9, 4, 10}>

# Divide the set into the "true" subset and the "false" subset: that
# is, the "less than 5" subset and the "not less than 5" subset.
s.divide { |x| x < 5 }
# => #<Set: {#<Set: {5, 6, 7, 8, 9, 10}>, #<Set: {1, 2, 3, 4}>}>

# Divide the set into the "0" subset and the "1" subset: that is, the
# "even" subset and the "odd" subset.
s.divide { |x| x % 2 }
# => #<Set: {#<Set: {6, 2, 8, 4, 10}>, #<Set: {5, 1, 7, 3, 9}>}>

s = Set.new([1, 2, 3, 'a', 'b', 'c', -1.0, -2.0, -3.0])
# Divide the set into the "String subset, the "Fixnum" subset, and the
# "Float" subset.
s.divide { |x| x.class }
# => #<Set: {#<Set: {"a", "b", "c"}>,
# =>       #<Set: {1, 2, 3}>,
# =>       #<Set: {-1.0, -3.0, -2.0}>}>
```


Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

For the two-argument code block version of `Set#divide`, the code block should return true if both the arguments it has been passed should be put into the same subset.

```
s = [1, 2, 3, -1, -2, -4].to_set

# Divide the set into sets of numbers with the same absolute value.
s.divide { |x,y| x.abs == y.abs }
# => #<Set: {#<Set: {-1, 1}>,
# =>         #<Set: {2, -2}>,
# =>         #<Set: {-4}>,
# =>         #<Set: {3}>}>

# Divide the set into sets of adjacent numbers
s.divide { |x,y| (x-y).abs == 1 }
# => #<Set: {#<Set: {1, 2, 3}>,
# =>         #<Set: {-1}>,
# =>         #<Set: {-4, -3}>}>
```

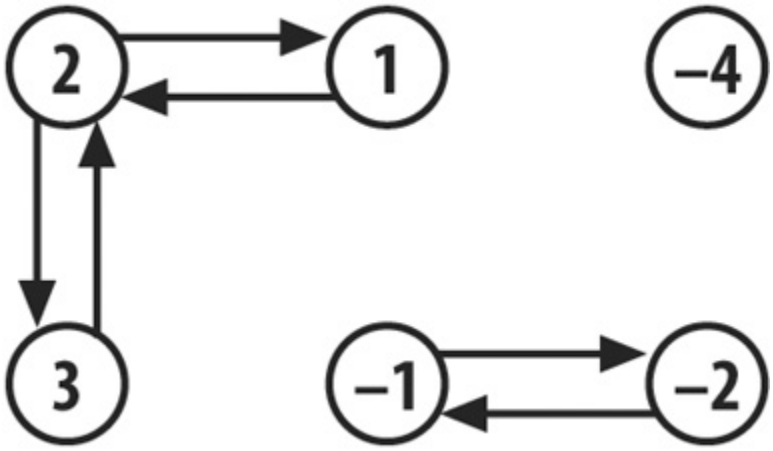
If you want to classify the subsets by the values they have in common, use `Set#classify` instead of `Set#divide`. It works like `Set#divide`, but it returns a hash that maps the names of the subsets to the subsets themselves.

```
s.classify { |x| x.class }
# => {String=>#<Set: {"a", "b", "c"}>,
# =>  Fixnum=>#<Set: {1, 2, 3}>,
# =>  Float=>#<Set: {-1.0, -3.0, -2.0}>}
```

Discussion

The version of `Set#divide` that takes a two-argument code block uses the `tsort` library to turn the `Set` into a directed graph. The nodes in the graph are the items in the `Set`. Two nodes `x` and `y` in the graph are connected with a vertex (one-way arrow) if the code block returns true when passed `|x,y|`. For the `Set` and the two-argument code block given in the example above, the graph looks like Figure 4-1.

Figure 4-1. The set {1, 2, 3, -1, -2, -4} graphed according to the code block that checks adjacency



Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

The `Set` partitions returned by `Set#divide` are the *strongly connected components* of this graph, obtained by iterating over `TSort#each_strongly_connected_component`. A strongly connected component is a set of nodes such that, starting from any node in the component, you can follow the one-way arrows and get to any other node in the component.

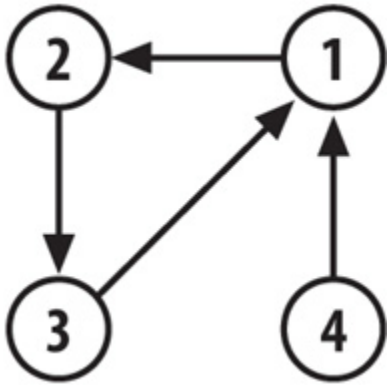
Visually speaking, the strongly connected components are the "clumps" in the graph. 1 and 3 are in the same strongly connected component as 2, because starting from 3 you can follow one-way arrows through 2 and get to 1. Starting from 1, you can follow one-way arrows through 2 and get to 3. This makes 1, 2, and 3 part of the same `Set` partition, even though there are no direct connections between 1 and 3.

In most real-world scenarios (including all the examples above), the one-way arrows will be symmetrical: if the code returns true for `|x,y|`, it will also return true for `|y,x|`. `Set#divide` will work even if this isn't true. Consider a `Set` and a `divide` code block like the following:

```
connections = { 1 => 2, 2 => 3, 3 => 1, 4 => 1 }
[1,2,3,4].to_set.divide { |x,y| connections[x] == y }
# => #<Set: {#<Set: {1, 2, 3}>, #<Set: {4}>}>
```

The corresponding graph looks like Figure 4-2.

Figure 4-2. The set {1,2,3,4} graphed according to the connection hash



You can get to any other node from 4 by following one-way arrows, but you can't get to 4 from any of the other nodes. This puts 4 is in a strongly connected component—and a `Set` partition—all by itself. 1, 2, and 3 form a second strongly connected component—and a second `Set` partition—because you can get from any of them to any of them by following one-way arrows.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01

Implementation for arrays

If you're starting with an array instead of a `Set`, it's easy to simulate `Set#classify` (and the single-argument block form of `Set#divide`)with a hash. In fact, the code below is almost identical to the current Ruby implementation of `Set#classify`.

```
class Array
  def classify
    require 'set'
    h = {}
    each do |i|
      x = yield(i)
      (h[x] ||= self.class.new) << i
  end
  h
end

  def divide(&block)
    Set.new(classify(&block).values)
  end
end

[1,1,2,6,6,7,101].divide { |x| x % 2 }
# => #<Set: {[2, 6, 6], [1, 1, 7, 101]}>
```

There's no simple way to implement a version of `Array#divide` that takes a two-argument block. The `TSort` class is `Set`-like, in that it won't create two different nodes for the same object. The simplest solution is to convert the array into a `Set` to remove any duplicate values, `divide` the `Set` normally, then convert the partitioned subsets into arrays, adding back the duplicate values as you go:

```
class Array
  def divide(&block)
    if block.arity == 2
     counts = inject({}) { |h, x| h[x] ||= 0; h[x] += 1; h}
     to_set.divide(&block).inject([]) do |divided, set|
       divided << set.inject([]) do |partition, e|
         counts[e].times { partition << e }
         partition
       end
     end
   else
     Set.new(classify(&block).values)
   end
  end
end

[1,1,2,6,6,7,101].divide { |x,y| (x-y).abs == 1 }
# => [[101], [1, 1, 2], [6, 6, 7]]
```

Is it worth it? You decide.

Ruby Cookbook By Lucas Carlson, Leonard Richardson ISBN: 0596523696 Publisher: O'Reilly
Print Publication Date: 2006/07/01