**12/03/2011:** This book is in the very preliminary stages. None of the content is guaranteed to be thorough or accurate. See the About Page.
The Bastards Book of Ruby
A Programming Primer for Counting and Other Unconventional Tasks

- Home
- About
- Contents
- Resources
- Blog
- Contact

The Fundamentals

# Collections

How to store lists of data.

The American Wing of the Metropolitan Museum of Art. Photo by Dan Nguyen

**Chapter outline**

**Previous Chapter:** Loops
**Next Chapter:** Enumerables

We've learned how loops can make life easier through repetition. **Collections** in Ruby allow for the organizing of large amounts of data, most of which you'll end up looping through.

When practicing loops, we encountered one kind of collection: the `Range`, which is a collection of numbers in order:

```
(1..5).each{ |num| puts num }
```

```
1
2
3
4
5
```

The two types of collections covered in this chapter can hold any kind of Ruby data object.

## Arrays

The `Array` is a data object that stores an **ordered list** of values. In fact, arrays are referred to as "lists" in other programming languages, including Python.

The following example shows how to initialize an array, fill it, read a value from it, reverse it, and then print it out as a single `String`:

This is how you initialize an array:

```
my_first_array = []
```

You can fill an array with any kind of data object:

```
my_first_array[0] = "Alpha"
my_first_array[1] = "Beta"
my_first_array[2] = 3

# initializing and filling the array in one step:
my_first_array = ["Alpha", "Beta", 3]
```

Elements in arrays are accessed by addressing their numerical place in the array:

```
puts my_first_array[1]
#=> Beta
```

Some common Array methods include `reverse` and `length`:

```
puts my_first_array.length
#=> 3

my_first_array = my_first_array.reverse
puts my_first_array.join(", ")
#=> 3, Beta, Alpha
```

- Square brackets, `[]`, denote the beginning and end of an array.
- Each spot in an array can point to any data object you want. You're not mixing different data types, per se, you're making a **collection** of them.
- Use integers to address locations in the array.
- Ruby arrays are **zero-based** indexed, meaning that the **first item in the Array is at position 0**, the second item is in position 1, and so on.

### Loop iteration

You can use the loops we learned in the last chapter – `for` or `while` – like so:

```
arr = [1,2,3,4,5,6]
for x in 0..(arr.length-1)
    puts arr[x]
end

# or, with while:
x = 0
while x < arr.length
    puts arr[x]
    x += 1
end
```

However, as I briefly mentioned in the last chapter, most iteration in Ruby is done using the `each` method.

The `each` method provides a tighter syntax:

```
arr.each{ |x| puts x}

# using the equivalent do..end construct:
arr.each do |x|
    puts x
end
```

The `each` method yields a reference to **each element** in the collection, rather than a reference to the **element's numerical index** in the array.

However, sometimes a reference to the current element's index is useful. Collections also have the `each_with_index` method available to them:

```
arr = ['cat', 'dog', 'pony', 'bird']
arr.each_with_index do |str, idx|
    puts "#{str} and #{arr[idx-1]} show"
end

#=> cat and bird show
#=> dog and cat show
#=> pony and dog show
#=> bird and pony show
```

### Adding to an Array with `push`

Collections can expand and contract. As arrays are their own data type, you can't use the + operator to add non-arrays and arrays together

```
# Attempting to add a Fixnum to an Array:
[3,2,1] + 4   # TypeError: can't convert Fixnum into Array

# Adding an array with one Fixnum to another Array:
[3,2,1] + [4]   # [3,2,1,4]
```

**Note:** Adding a new element to an array, as in the above example, will *add it to the end* of the left-side array. Arrays *do not automatically sort* themselves or maintain any kind of alphabetical/numerical order (but they do have methods for that).

It's a little clunky to put square brackets around a single object just to add it to an array. The `push` method (other languages also use the word `push` for this functionality) adds objects to the end of an array:

```
arr = ['dog', 'cat', 'bear']
arr.push('penguin', 'fox')

puts arr.join("-")   #=> dog-cat-bear-penguin-fox
puts arr[0]   # dog
```

Unlike most of the methods we've covered so far, `push` **modifies** the array that calls it. A new element is added to that same array, rather than a new array. So there's no need to do the assignment operation in the second line here:

```
arr = ['dog', 'cat', 'bear']
arr.push('penguin', 'fox')
puts arr.join(',')
#=> dog,cat,bear,penguin,fox
```

In Ruby, you can also use two left-angled brackets << to push objects onto an array:

```
new_arr = ["dog", 7, 42, "cat"]
new_arr << 42
puts new_arr.join(",")
#=> dog,7,42,cat,42
```

### Remove with `pop`

To take an item out of an array, use `pop`, which removes (and returns) the last item in the array.

Just as with `push`, `pop` operates at the *end of the array*. In computer science lingo, this is referred to as **LIFO** – Last In, First Out:

And, like `push`, `pop` also modifies its array.

```
arr = ["Atlanta", "Boston", "Chicago", 42]
arr.pop   # 42
puts arr.join(', ')   #=> Atlanta, Boston, Chicago
```

##### Exercise: Reverse an array using a loop and `push` and `pop`

**Note:** This exercise goes into a lot of detail about thinking through the most efficient algorithm. Again, understanding how arrays work is the most important thing. I go into extra detail to give you more ways to observe how arrays work.

A well-known technical interview question is: **write some code to reverse the order of elements in an array**.

You *could* say something like, "I would use the array's `reverse` method" to show that you have indeed read the Ruby Array docs. But that doesn't give much insight on your actual comprehension of collections.

As with any coding problem, there are multiple solutions. Try doing it with `push`, `pop`, and a loop. It is not necessarily the ideal solution but serves as good practice with these methods.

If you are using a `for` loop, you will have to use the `length` method of `Array`. This returns the number of elements in an array.

Also, keep in mind that `array.length - 1` is the index of the final element, as the first element begins at `0`.

###### Solution

Use a second placeholder array to do a reversing with the limitations I placed:

```
arr = [1,2,3,4,5]   # given an arbitrary array

temparr = []
for x in 0..(arr.length-1)   # or: arr.length.times
   temparr.push(arr.pop)
end
arr = temparr
```

Remember that `x` is a reference to the current element in the array. However, we never need to use this reference because `pop` always acts on the very last element in the array.

A `while`/`until` loop may be more graceful in this case:

```
temparr = []
temparr.push( arr.pop) until arr.empty?   # alternatively: while !arr.empty?
arr = temparr
```

##### Exercise: Reverse an array without a temporary array

With today's computers, you can pretty much declare all the arrays you think you'll need without worrying about memory limitations. However, this isn't the devil-may-care attitude that employers are seeking.

So try reversing an array in-place without a second temporary array. While I said not to use a *temporary array*, you're welcome to use a placeholder *variable*.

###### Solution

One solution is to access the array's elements by indices. Start at the 0-index and swap that element with the element in the last index.

If the array has a `length` of `5`, then the `array[0]` is swapped with `array[4]`. Then we go to `array[1]`, which is swapped with `array[3]`.

See a pattern here? For any given index, `x`, we are swapping it with the element at the position of the **final index** *minus* `x`:

```
array[x] = array[final_index-x]
```

What happens if we loop through the **entire** array and `x` is equal to `final_index`? Well, `final_index - x` will be equal to `0`.

This means that at the very end of the loop, we switch what is currently in the array's first element with what is in its final element. But remember that we switched those two elements *in the first iteration* of the loop!.

In other words, if we run the loop the same number of times as elements in the array, the array will end up in the same order as before. So we have to stop halfway:

```
arr.length/2
```

```
arr = [1,2,3,4,5]   # given an arbitrary array

final_idx = arr.length-1
for x in 0..(arr.length/2)
   temp = arr[x]
   arr[x] = arr[final_idx - x]
   arr[final_idx - x] = temp
end
```

You could also do without the `temp` variable to use Ruby's special notation for assigning values to multiple variables at once:

```
arr = [1,2,3,4,5]   # given an arbitrary array

final_idx = arr.length-1
for x in 0..(arr.length/2)
   arr[x], arr[final_idx - x] = arr[final_idx - x], arr[x]
end
```

I don't know enough of the details of how Ruby executes that one-line swap, but it appears to be significantly slower than using a temporary variable. But it at least looks prettier.

In fact, using the temporary array with the `while`, `push` and `pop` operations appears to be the fastest. When I say "fastest", I mean a **tenth of a second** faster when doing the reversing operation **10,000 times**.

However, the fastest method by far – on the order of 100 times faster– is to use the array's `reverse` method. Why? It uses memory pointers and low-level arithmetic to swap the references. But this is done in C, the language that Ruby is written in. You can read the source for yourself, but it probably won't make immediate sense (it also uses a few macros defined else where in Ruby's source code).

Return to chapter outline

## Other array methods

Arrays come with very useful methods to read, sort, and manipulate them. I'll cover more of the transformative methods in the Enumerables chapter. Check out the Ruby documentation for the full list.

### Sorting arrays

As you insert elements into an `Array`, they will remain in the order in which they were added. You can call the `sort` or `sort_by` methods, both of which return a sorted *copy* of that an array.

###### The default `sort`

When your array contains a uniform set of standard objects, such as numbers or strings, calling `sort` without any arguments will arrange the array's elements in ascending order:

```
puts [8, 0, 15, -42].sort.join(', ')   #=> -42, 0, 8, 15

arr = ["cat", "zebra", "dog", "alpaca"]
puts arr.sort.join(', ')   #=> alpaca, cat, dog, zebra
```

##### Sort an array in descending order

The method `sort` arranges elements in ascending order. Arrange an array of numbers in descending order.

###### Solution

There's variations on calls to `sort` that can do this. But let's do it the no-brainer way, with a subsequent call to `Array`'s `reverse` method.

```
[6, -30, 0, 100, 2].sort.reverse   #=> [100, 6, 2, 0, -30]
```

###### Custom sorting with `sort_by`

Alphabetical letters *of different cases* do not sort alphabetically. Try converting a letter into a number to see the standard numerical code that represents it:

```
puts 'a'[0]   #=> 97
puts 'B'[0]   #=> 66
puts 'c'[0]   #=> 99
puts 'Z'[0]   #=> 90

puts ['a', 'B', 'c', 'Z'].sort.join(', ')   #=> B, Z, a, c
```

If we want to sort a collection's elements *by values derived from the elements*, we can use `sort_by`. This method takes in a block; the block's return value is what ends up being compared.

To sort strings in alphabetical order without regard to case, we pass a block into `sort_by` that either lowercases or uppercases each string for the comparison.

```
arr = ['Bob', 'Zed', 'cat', 'apple'].sort_by do |x|
   x.upcase
end

puts arr.join(', ')   #=> apple, Bob, cat, Zed
```

**Note:** Despite the upcase operation done in the block passed to `sort_by`, the **elements** in `arr` themselves **are not modified** during the sort.

#### String.split

There are also methods to convert other objects into arrays. One we'll use frequently is `split` from the `String` class, which turns a string into an array of elements, separated by a delimiter string you specify.

```
"Smith,John,Ithica,NY".split(',')   #=> ["Smith", "John", "Ithica", "NY"]
"She sells seashells".split(' ')   #=> ['She', 'sells', 'seashells']
"She sells seashells".split('se')   #=> ["She ", "lls ", "ashells"]
```

#### Exercise: Sort names by last name

Given an array of strings that represent names in "FIRSTNAME LASTNAME" form, use `sort_by` and `split` to return an array sorted by last names. For simplicity's sake, assume that each name consists of only two words separated by space (i.e. only "John Doe" and not "Mary Jo Doe").

```
names = ["John Smith", "Dan Boone", "Jennifer Jane", "Charles Lindy", "Jennifer Eight", "Rob Roy"]
```

#### Solution

Pass a block into `sort_by` in which each name is `split` by a space character. The last element of the `Array` created by `split` is the last name. However, if two names have the same last name, you'll need to use their first names as a tiebreaker.

So the comparison value will consists of the first and last name components split apart, reversed in order, and then joined together and uppercased:

```
names = ["John Smith", "Dan Boone", "Jennifer Jane", "Charles Lindy", "Jennifer Eight", "Rob Roy"]
sorted_names = names.sort_by do |name|
   name.split(" ").reverse.join.upcase
end

puts sorted_names.join('; ')
#=> Dan Boone; Jennifer Eight; Jennifer Jane; Charles Lindy; Rob Roy; John Smith
```

To do it in one line with the curly braces notation:

```
sorted_names = names.sort_by{ |name| name.split(" ").reverse.join.upcase }
```

Return to chapter outline

## Working with partial arrays

To use just part of an `Array`, you can use its `slice` method. The **first argument** to `slice` is the index of the original array that you want to start the sub-array at. The **second argument** is the number of elements to include, starting from *and including* the element at the specified index (i.e. a second argument of 0 will return an empty array)

**Note:** The `slice` method will create a new `Array` with the specified elements from the original without removing anything from the original.

```
arr = [0,1,2,3,4,5]
puts arr.slice(3, 2).join(',')   #=> 3,4
puts arr.slice(0,1).join(',')   #=> 0
arr.slice(1,0).join(',')   #=> []
```

#### Partial arrays with ranges

**Note:** Ranges, and their use in addressing partial arrays, is something that's not common among other programming languages. It's an extremely useful convention for Rubyists but your mileage will vary in other languages.

I prefer using bracket notation with a `Range` to excerpt an array. To specify a group of consecutive elements, **use a `Range` instead of a number** inside square brackets. The end of the range specified is the index of the last element you want to include:

```
arr = ['a', 'b', 'c', 'd', 'e', 'f']
puts arr[3..4].join(',')   #=> d,e
puts arr[0..0].join(',')   #=> a
puts arr[0..(arr.length-1)].join(',')   #=> a,b,c,d,e,f
```

#### Negative indices

What happens when you use a negative integer as an index? Ruby will count from the *end of the array*. This is extremely useful for getting the last element in a given array without having to know its length.

```
arr = "160,Ninth Avenue,Apt 42,10015,New York,NY,USA".split(",")
# ["160", "Ninth Avenue", "Apt 42", "10015", "New York", "NY", "USA"]

puts arr[-1]   #=> USA
```

#### Partial strings with arrays and ranges

Strings can be treated as **arrays** of characters when addressed with the brackets-and-range notation:

```
puts "Hello world"[0..4]
#=> Hello
```

Negative indices also work:

```
puts "Synecdoche, NY"[-2..-1]
#=> NY
```

What if we want just a **single character**, such as the first or last character in a string? **Using a single integer as an index will not do this**. Instead, that notation will get us the character's ASCII value:

```
puts "J"[0]
#=> 74
```

However, by using a range in which the range spans just a single value, then we're in business:

#### A summary of partial indexing

This table summarizes how partial indices affect a given array or string. The elements affected by the given index are highlighted.

| Array/String | Index |
|---|---|
| [1,2,3,4,5] | [0] |
| [1,2,3,4,5] | [-1] |
| ["alpha", "beta", "charlie", "delta", "echo"] | [2..3] |
| [0,1,2,3,4,5] | [1..-1] |
| ["AL", "AK", "CA", "FL", "HI"] | [-2..-1] |
| "The sun also rises" | [4..6] |
| "DON'T PANIC" | [-1] *returns `67`, the ASCII value* |
| "Mostly harmless." | [-1..-1] |
| "She sells seashells".split(' ') | [-1][-6..-1] |

#### Exercise: Ignore the header and footer of a table

Oftentimes in a spreadsheet of data, the first row specifies the names of columns and the final row specifies the totals of that column:

| Name | Profit | Loss |
|---|---|---|
| James | 100 | 12 |
| Bob | 20 | 42 |
| Sara | 50 | 0 |
| **Totals** | **170** | **(54)** |

Write an expression that captures only the rows **in between** the first and last rows.

The data will be a single string containing line breaks for each row, so you will also have to use the `split` method:

```
table="Name,Profit,Loss
James,100,12
Bob,20,42
Sara,50,0
Totals,170,(54)"
```

**Hint**: `\n` is the special character used to represent line breaks (i.e. **n**ew lines).

#### Solution

Use a negative number in to refer to the elements at the end of an Array. Remember that `-1` will access the very last element, so `-2` will get the next-to-last element.

```
table="Name,Profit,Loss
James,100,12
Bob,20,42
Sara,50,0
Totals,170,(54)"

puts table.split("\n")[1..-2]
#=> James,100,12
#=> Bob,20,42
#=> Sara,50,0
```

Return to chapter outline

## Hashes

Think of the `Hash` object as an `Array` that can use `any` data object as a **key** (the Hash version of an index). In Python, this is referred to as a **Dictionary**.

A hash uses **curly brackets `{ }`** around its elements. The notation is a little more complicated than it is for an `Array`. To associate a **key** with a **value**, use the `=>` (referred to as a "**hash rocket**" (*however, in **Ruby 1.9+**, the hash rocket can be replaced with a simple colon* `:`).

```
hsh = {1=>"Alaska", "blue"=>"orange", "The answer to life"=>42, -100=>200}
```

Just like with arrays, use square brackets `[ ]` to get a hash's value at a particular key:

```
puts hsh[1]
```

```
#=> Alaska

puts hsh["blue"]
#=> orange
```

**Note:** While it's *possible*, it is inadvisable to use something like an `Array` as a key unless you really have to. Remember how operations like `push` and `pop` modify an array in-place? Pushing an element onto an Array will essentially change the value it has as a Hash key. Read this article if you really care. I can't recall the last time I used something other than a string or number as hash keys.

## Comparing Hash and Array operations

Here's a table showing the common collection-type operations for arrays and hashes.

| Operation | Array | Hash |
|---|---|---|
| Basic initialization | `arr = []` | `hsh = {}` |
| Initialize with values | `arr = [`<br>`      ["dog", "Fido"],`<br>`      ["cat", "Whiskers"]`<br>`]` | `hsh = {`<br>`   "dog"=>"Fido",`<br>`   "cat"=>"Whiskers"`<br>`}` |
| Add new element | `arr << ["pig", "Wilbur"]` | `hsh["pig"] = "Wilbur"` |
| Get last element | `puts arr[-1].join(': ')`<br>`#=> pig: Wilbur` | N/A |
| Get the *name* of the "cat" | `arr.assoc('cat')[1]` | `hsh['cat']` |
| Iterate and print strings in the format of `"ANIMAL: NAME"` | `arr.each{\|a\| puts a.join(': ')}` | `hsh.each_pair{ \|k,v\| puts "#{k}: #{v}"}` |

## Hash chaos

You already know that you use `push` or << to add an element to the end of on an existing `Array`. But these methods do not work for a `Hash`.

So how do you add a new element (also referred to as a **key-value pair** for hashes) to the end of a `Hash`?

You don't. Actually, you just *can't* add something to the **end** of a `Hash`.

The key word is **"end"**. Hash elements (before Ruby 1.9x) do not maintain the order in which they were added.

So, in the table of comparisons in the previous section, even though `"pig"=>"Wilbur"` was the last key-value pair to be added to `hsh`, it will not necessarily appear last when iterating through that Hash with `each`.

As I noted in the aside above, Ruby 1.9 hashes do keep their order. This is a nice optional cherry-on-the-top. Since they've been unordered in Ruby 1.8 and below, I've generally used hashes when the order of the elements is not important.

## Exercise: Convert an array into a hash

Write a loop to convert `arr_pets` into the equivalent `hash`:

```
arr_pets = [["dog", "Fido"], ["cat", "Whiskers"], ["pig", "Wilbur"]]
```

###### Solution

Initialize an empty `Hash` object and fill it by iterating through `arr_pets` with `each`.

Here's the verbose way to do it:

```
hsh_pets = {}
arr_pets.each do |x|
   key = x[0]
   val = x[1]
   hsh_pets[key] = val
end

#=> hsh_pets = {"cat"=>"Whiskers", "pig"=>"Wilbur", "dog"=>"Fido"}
```

And here's the one-line version:

```
hsh_pets = {}
arr_pets.each{|x| hsh_pets[x[0]] = x[1]}
```

## Why choose a `Hash` over an `Array`?

Anything you use a `Hash` for, you can generally do through an equivalent `Array`. But a Hash can be easier to maintain and manipulate. Because when it comes to addresses – and indices – humans remember words better than they do numbers.

Let's use as an example a dataset of campaign contributors:

| Name | State | Candidate | Amount |
|---|---|---|---|
| John Doe | IA | Rep. Smithers | $300 |
| Mary Smith | CA | Sen. Nando | $1,000 |
| Sue Johnson | TX | Rep. Nguyen | $200 |

This can be represented as **an array nested in an array**. Each row is represented as an array, with each field appearing in the array in the same order it does in the table. And all these arrays are themselves stored in an array, in the same order they are listed in original table:

```
rows = [["Name", "State", "Candidate", "Amount"],
["John Doe", "IA", "Rep. Smithers", "$300"],
["Mary Smith", "CA", "Sen. Nando", "$1,000"],
["Sue Johnson", "TX", "Rep. Nguyen", "$200"]]
```

To iterate through all the **states**, for example, you just have to remember that the **State** column is in the **`1`-index** spot of each sub-array. For the **Amount** values, their position is in the **last index** of each sub-array.

```
puts "Candidate #{rows[2][2]} received #{rows[2][-1]} from a donor named #{rows[2][0]} from the state of #{rows[2][1]}"

#=> Candidate Sen. Nando received $1,000 from a donor named Mary Smith from the state of CA
```

Here's the table as an **array of hashes**:

```
rows = [
{"Name"=>"John Doe", "State"=>"IA", "Candidate"=>"Rep. Smithers", "Amount"=>"$300"},
{"Name"=>"Mary Smith", "State"=>"CA", "Candidate"=>"Sen. Nando", "Amount"=>"$1,000"},
{"Name"=>"Sue Johnson", "State"=>"TX", "Candidate"=>"Rep. Nguyen", "Amount"=>"$200"}]
```

Here's the output of a row (note that there is no headers row):

```
puts "Candidate #{rows[0]['Candidate']} received #{rows[0]['Amount']} from a donor named #{rows[0]['Name']} from the state of #{rows[0]['State']}"

Candidate Rep. Smithers received $300 from a donor named John Doe from the state of IA
```

Here's the loop for the **array-of-arrays**:

```
rows.each do |row|
   puts "#{row[0]} paid #{row[3]} to #{row[2]}"
end

# Output:
#=> John Doe paid $300 to Rep. Smithers
#=> Mary Smith paid $1,000 to Sen. Nando
#=> Sue Johnson paid $200 to Rep. Nguyen
```

This is the loop for an **array-of-hashes** – which is more readable to you?

```
rows.each do |row|
   puts "#{row["Name"]} paid #{row["Amount"]} to #{row["Candidate"]}"
end
```

This example may seem simple enough. But what if the data format contained many more columns? Keeping track of how the numerical indices map to the data fields positions can be maddening.

To add to the confusion, consider what happens if you want to split the **Name** field into **First name** and **Last name** fields?

In the array-of-arrays version, if you replace the **Name** column with a **LastName** and **FirstName** columns, there needs to be an extra column in the table. If you insert that extra column in place, then every column after that is moved over. For example, the **State** column would now be in the `2`-index position of each subarray.

Which means that everywhere in the code that you used `row[1]` to refer to the `State` column has to be changed to `row[2]`

The exercise below is probably more confusing than helpful. I might remove it in future editions.

##### Exercise: Convert an array of arrays into an array of hashes

In the previous exercise involving pets, it was a simple matter of knowing that the `0`-index and `1`-index corresponded to **pet type** and **name**, respectively.

Using the sample campaign finance data from above, change each row (sub-array) into an equivalent hash:

```
data_arr = [
   ["Name", "State", "Candidate", "Amount"],
   ["John Doe", "IA", "Rep. Smithers", "$300"],
   ["Mary Smith", "CA", "Sen. Nando", "$1,000"],
   ["Sue Johnson", "TX", "Rep. Nguyen", "$200"]
]
```

###### Hints:

- You'll need at least two `each` loops to do this (sanely).
- Store the first row of `data_arr` in a variable to refer to later

###### Solution

The following code may seem overly complicated, but the concept is simple: The keys needed for the new hashes are in the **first** row of the dataset. This row is itself an array. So iterate through that array of headers to create each new hash.

Here's the code:

```
# first row contains column headers
headers_row = data_arr[0]
```

```
new_data_arr = []
data_arr[1..-1].each do |row|   # [1..-1] skips the first row
   hsh = {}
   current_column_index = 0
   headers_row.each do |header|
      hsh[header] = row[current_column_index]
      current_column_index += 1
   end
   new_data_arr << hsh
end

new_data_arr
#=> [{"Name"=>"John Doe", "Amount"=>"$300", "Candidate"=>"Rep. Smithers", "State"=>"IA"}, {"Name"=>"Mary Smith", "Amount"=>"$1,000", "Candidate"=>"Sen. Nando", "State"=>"CA"}, {"Name"=>"Sue Johnson", "Amount"=>"$200", "Candidate"=>"Rep. Nguyen", "State"=>"TX"}]
```

The main loop will iterate through each of `data_arr`'s non-headers rows. At the beginning of this loop, create a new empty hash: `hsh`.

Also, we declare a new variable, `current_column_index`, which will be set to `0` on each iteration. This variable is useful in the inner loop:

The second `each` call iterates through the array of column headers, `headers_row`. Each value in `headers_row` is used as a *key* for `hsh`.

So where does the *value* for the **key-value pair** come from? From the corresponding column in `row`, of course.

The variable `current_column_index` keeps track of which index of `headers_row` that `header` corresponds to.

The value in `row[current_column_index]` corresponds to `header` (i.e. `headers_row[current_column_index]`) because, well, that's how tables of data work.

At the end of each iteration in the inner loop, we increment `current_column_index`. When the inner loop finishes, we go back to the top of the main loop, and `current_column_index` is set to 0 as the inner loop starts all over again.

Whew. That was a lot of words to awkwardly explain a pretty simple operation.

We can simplify this a little by, instead of `each`, using `each_with_index`. This provides the block both a reference to the value in the collection *and* its numerical index. Thus, no need to keep `current_column_index` around:

```
headers_row = data_arr[0]
new_data_arr = []
data_arr[1..-1].each do |row|   # [1..-1] skips the first row
   hsh = {}
   headers_row.each_with_index{ |header, idx|   hsh[header] = row[idx] }
   new_data_arr << hsh
end
```

### Using `each` and `each_pair` for hash iteration

The `each` method works for a `Hash`. However, it converts each key-value pair into an array like so: `[key, value]`

```
{"CA"=>"California"}.each do |pair|
   puts pair[0]
   puts pair[1]
end

#=> CA
#=> California
```

To more conveniently access keys and values of a `Hash`, use the `each_pair` method:

```
{'New York'=>'Knicks', 'Miami'=>'Heat', 'Boston'=>'Celtics',
 'Dallas'=>'Mavericks'}.each_pair |key, val| do
   puts "#{key} has the #{val}"
end
#=> New York has the Knicks
#=> Miami has the Heat
#=> Boston has the Celtics
#=> Dallas has the Mavericks
```

Hashes also have the `each_key` and `each_value`, which each return their respective part of key-value pair.

###### Exercise: Count frequencies of letters

Given an arbitrary string, write a method that returns a hash containing:

- **Each unique character** found in the string
- The **frequency** that each type of character appears in that string

Use a `Hash` and its ability to use anything, including strings, as keys. Use the `split` method of `String` to convert a string into an array of letters.

Also, you may find the hash's `fetch` method useful. It takes two arguments: 1) a key and 2) a value to return *if there is nothing* in the hash at that key. If a value at `hash[key]` exists, then that is what `fetch` returns.

###### Solution

```
def count_freq(str)
   hsh = {}
   lth = str.length
   str.split('').each do |letter|
      hsh[letter] = hsh.fetch(letter,0) + 1
   end

   # The each_pair method, when given a block, returns the Hash that invoked it
   # i.e. this will be count_freq's return value

   hsh.each_pair{ |key,val|
      hsh[key] = (val/lth.to_f)
   }

end


count_freq("abcd")
#=> {"a"=>0.25, "b"=>0.25, "c"=>0.25, "d"=>0.25}

count_freq("eeeieeeoueeeiaaaieeeuauaieeieaau")
#=> {"e"=>0.46875, "i"=>0.15625, "o"=>0.03125, "u"=>0.125, "a"=>0.21875}
```

Return to chapter outline

## Nested collections

Get used to seeing collections inside collections. There's no new concepts to know, it's just takes more careful reading. You'll chain together the bracket notation used to access indices:

```
two_dimensional_array = [["apples", "bananas", "cherries"], [1,2,3], ["Alpha", "Beta", "Charlie"]]

puts two_dimensional_array[2][1]   #=> Beta
puts two_dimensional_array[1].join(',')   #=> 1,2,3

dbl_hash = {
   "Chicago"=>{
      "nickname"=>"The Windy City",
      "state"=>"Illinois",
      "area in sq. mi."=> 234.0},
   "New York City"=>{
         "nickname"=>"The Big Apple",
         "state"=>"New York",
         "area in sq. mi."=> 468.9}
}

puts dbl_hash["Chicago"]["area in sq. mi."]   #=> 234.0
```

An array in which each element is an array can be referred to as a two-dimensional array. Multi-dimensional arrays are often referred to as **matrices**.

###### Exercise: Make a multiplication table

Using loops, write a script that will fill a two-dimensional array with the product of each row-index and column-index, for integers 0 through 9.

The value in `arr[4][6]` should be `24`. And `arr[0][4]` should be `0`. And so on.

###### Solution

A two-dimensional array has an array nested at each index of another array. So we'll use a nested `for` loop.

```
arr = []
for i in 0..9
   arr[i] = []
   for j in 0..9
      arr[i][j] = i*j
   end
end
```

###### Exercise: Sort an array of hashes

Starting from this array of U.S. presidents:

```
presidents = [
 {:last_name=>"Clinton", :first_name=>"Bill", :party=>"Democrat", :year_elected=>1992, :state_residence=>"Arkansas"},
 {:last_name=>"Obama", :first_name=>"Barack", :party=>"Democrat", :year_elected=>2008, :state_residence=>"Illinois"},
 {:last_name=>"Bush", :first_name=>"George", :party=>"Republican", :year_elected=>2000, :state_residence=>"Texas"},
 {:last_name=>"Lincoln", :first_name=>"Abraham", :party=>"Republican", :year_elected=>1860, :state_residence=>"Illinois"},
 {:last_name=>"Eisenhower", :first_name=>"Dwight", :party=>"Republican", :year_elected=>1952, :state_residence=>"New York"}
]
```

Write the code in order to:

1. Sort `presidents` by **last name** in reverse alphabetical order.
2. Create a :full_name key for each hash and set its value to a string in "LASTNAME, FIRSTNAME" format. Then sort the collection, first by party affiliation and then by the full name field.

I've surreptitiously introduced Ruby's `Symbol` class. Instead of allocating a whole `String` to use as a key, such as `"last_name"`, I use a `Symbol`:

`:last_name`

Symbols aren't strings. They have less functionality and share the same restrictive naming policies as variables (basically, just use lowercase alphabet letters and underscores). But they are more lightweight and are just about as readable as strings.

Read more about symbols here. They don't really affect anything we do. Use strings if you want. But at least you know what they look like.

###### Answer

1. Use `sort_by` and the `:last_name` key of each hash. Reverse the result:

```
presidents.sort_by{ |hsh| hsh[:last_name] }.reverse
```

2. Use `sort_by`; add the `:full_name` key inside the block

```
presidents.sort_by do |p|
   p[:full_name] = "#{p[:last_name]}, #{p[:first_name]}"
   "#{p[:party]}-#{p[:full_name]}"
end
```

Since `sort_by` only compares the return values of each block, you're free to modify each hash as you like before the final line.

Return to chapter outline

## Revisiting the `crack` gem

Back in the methods and gems chapter, we briefly made contact with hashes. Hopefully, hashes make a little more sense after this formal if brief introduction.

Given an XML file like this:

```
<result>
  <type>street_address</type>
  <formatted_address>900 Broadway, New York, NY 10010, USA</formatted_address>
  <address_component>
   <long_name>900</long_name>
   <short_name>900</short_name>
   <type>street_number</type>
  </address_component>
<result>
```

– And then passing it to `Crack::XML.parse` method, we get a hash that can be accessed like this:

```
parsed_xml = Crack::XML.parse(thexml)
puts parsed_xml["result"]["type"]   # street_address
```

I've highlighted below which nodes are accessed:

```
<result>
  <type>street_address</type>
  <formatted_address>900 Broadway, New York, NY 10010, USA</formatted_address>
  <address_component>
   <long_name>900</long_name>
   <short_name>900</short_name>
   <type>street_number</type>
  </address_component>
<result>
```

Another example:

```
puts parsed_xml["result"]["address_component"]["short_name"]
#=> 900
```

```
<result>
     <type>street_address</type>
     <formatted_address>900 Broadway, New York, NY 10010, USA</formatted_address>
     <address_component>
      <long_name>900</long_name>
      <short_name>900</short_name>
      <type>street_number</type>
     </address_component>
   <result>
```

###### Multiple elements as an array

Given an XML document in which there are multiple elements with **the same name at the same nested-level**, the crack `parse` method will structure those elements as an **array**:

```
parsed_xml = Crack::XML.parse(thexml)
puts parsed_xml["users"]["user"].class   #=> Array
puts parsed_xml["users"]["user"][-1]["last_name"]   #=> Doe
```

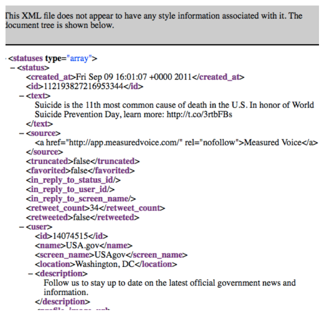
```
<users>
   <user>
      <first_name>Dan</first_name>
      <last_name>Nguyen</last_name>
   </user>
   <user>
      <first_name>Jeremy</first_name>
      <last_name>Smith</last_name>
   </user>
   <user>
      <first_name>Jane</first_name>
      <last_name>Doe</last_name>
   </user>
</users>
```

#### Exercise: Parse Tweets

Let's try to parse the Twitter API response that we played around with at the beginning of this book

I have a cached version here for the user USAGov that you can hit up.

Open the XML up in your browser. You might find it easier to copy and paste it into your text-editor and save it as XML to get a color-coded view of it:



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<statuses type="array">
 – <status>
     <created_at>Fri Sep 09 16:01:07 +0000 2011</created_at>
     <id>112193827216953344</id>
   – <text>
       Suicide is the 11th most common cause of death in the U.S. In honor of World
       Suicide Prevention Day, learn more: http://t.co/3rtbFBs
     </text>
   – <source>
       <a href="http://app.measuredvoice.com/" rel="nofollow">Measured Voice</a>
     </source>
     <truncated>false</truncated>
     <favorited>false</favorited>
     <in_reply_to_status_id/>
     <in_reply_to_user_id/>
     <in_reply_to_screen_name/>
     <retweet_count>34</retweet_count>
     <retweeted>false</retweeted>
   – <user>
       <id>14074515</id>
       <name>USA.gov</name>
       <screen_name>USAgov</screen_name>
       <location>Washington, DC</location>
     – <description>
         Follow us to stay up to date on the latest official government news and
         information.
       </description>
```

Write a script using the **crack** gem that loops through each tweets and prints to screen the date and text of each tweet.

Remember **RestClient** from the methods and gems chapter? I'll provide the lines that will retrieve the tweet XML from the remote address and you can take it from there.

```
require 'rubygems'
require 'rest-client'
require 'crack'

URL = "http://ruby.bastardsbook.com/files/tweet-fetcher/tweets-data/USAGov-tweets-page-2.xml"
response = RestClient.get(URL)
if response.code == 200
   xml = response.body
   # do your parsing here
   # ...
else
   puts "Service is currently down!"
end
```

###### Solution

```
require 'rubygems'
require 'rest-client'
require 'crack'

URL = "http://ruby.bastardsbook.com/files/tweet-fetcher/tweets-data/USAGov-tweets-page-2.xml"
response = RestClient.get(URL)

if response.code == 200
   xml = Crack::XML.parse(response.body)
   xml["statuses"].each do |status_el|
      puts status_el["text"]
      puts status_el["created_at"]
      puts "--- \n"
   end
else
   puts "Service is currently down!"
end
```

Using `each`, we iterate through each status node in the XML. The current status node in each iteration is pointed to by the variable `status_el`. `status_el` is a hash with keys representing the child elements `text` and `created_at`

- **Next Chapter**
  Enumerables
- **Previous Chapter**
  Loops

---

- Chapter Outline
- Table of Contents

###### The Book

**Version:** 0.1

- Home
- About
- Contents
- Resources
- Blog

- Twitter: @bastardsbook
- Facebook

**Author Info**

Dan Nguyen is a journalist in New York

- Email: dan@danwin.com
- Blog: danwin.com
- Twitter: @dancow
- Tumblr: eyeheartnewyork
- Flickr

Copyright 2011