Chapter 4

Take Advantage of Ruby's Smart Strings

Ask a nonprogrammer what we software types do all day and their answer will probably involve numbers. It's true: Programs are often knee deep in numerical values, everything from account balances to the number of milliseconds it takes to store a balance in a database. It ain't called computing for nothing. The thing that non-coders don't realize is just how much text processing goes on alongside all the number-crunching. Real-world programs tend to overflow with text, everything from the name of the account owner, to the name of the table that stores the balance, to the name of the program doing the storing. Because text processing is so central to just about every programming problem, every modern programming language comes equipped with some kind of string facility, typically a container that will not just store your text but also help you to fold, bend, and mutilate it.

The place for text in a Ruby program is in instances of the `String` class. In this chapter we will take a quick tour of Ruby's strings, starting with the surprising number of ways that you create a string and progressing to a fast look at the String API. We will also explore some of the ways that newcomers to Ruby can come to grief with strings and look at a few of the powerful things that Ruby applications do with just a pinch of string processing.

Coming Up with a String

What strikes many new Ruby coders about the language's strings is not the strings themselves but the number of ways that you can write a string literal. For example, just about every programming language features quoted strings. Some languages use single quotes as delimiters; others rely on double quotes. Ruby uses both kinds: There is the single-quoted string literal that is very literal indeed, doing almost no interpretation on the text between the quotes. In fact, the only fancy things you can do in a single-quoted string are to embed a literal quote in it, escaped by a backslash:

```
a_string_with_a_quote = 'Say it ain\'t so!'
```

And embed a literal backslash, also escaped by a backslash:[1]

```
a_string_with_a_backslash = 'This is a backslash: \\'
```

Double-quoted strings do quite a bit more interpretation: You can put characters like tabs and newlines in a double-quoted string with the appropriate character after a backslash:

```
double_quoted = "I have a tab: \t and a newline: \n"
```

You can also embed arbitrary expressions in a double-quoted string by enclosing the expression between `#{` and `}`, so that this:

```
author = "Ben Bova"
title = "Mars"
puts "#{title} is written by #{author}"
```

Will print:

```
Mars is written by Ben Bova
```

1. Do be aware that Ruby will print backslashes embedded in strings in different ways, depending on how you output the string. If you use `puts` to print out a string with a backslash, Ruby will just print the backslash. If you use `pp`, however, Ruby will print the same double backslash you would use to embed the backslash character in the string. Either way, it's the same string with the same (single) backslash.

Keep in mind that since a single quote is not special in a doubled-quoted string and vice versa, you can sometimes avoid a lot of quote escaping by surrounding your string with a different quote flavor, so that instead of this:

```
str = "\"Stop\", she said, \"I cannot deal with the backslashes.\""
```

You can simply say:

```
str = '"Stop", she said, "I cannot deal with the backslashes."'
```

Although simple single- and double-quoted strings will suffice for about 98% of your string literal needs, occasionally you do need something a little more exotic. What if, for example, you had a string full of both types of quotation marks:

```
str = '"Stop", she said, "I can\'t live without \'s and "s."'
```

Ruby offers a couple of ways out of this kind of backslash Hell. In this case, the best choice is probably the arbitrary quote mechanism, which looks like this:

```
str = %q{"Stop", she said, "I can't live without 's and "s."}
```

Arbitrarily quoted strings always start with a percent sign (`%`) followed by the letter `q`. The character after the `q` is the actual string delimiter—we used a brace (`{`) in the example above. If your delimiter has a natural partner, the way `}` goes with `{`, then that's the character you use to close the string. In the example above we could have used `(` and `)` instead of `{` and `}`:

```
str = %q("Stop", she said, "I can't live without 's and "s.")
```

Or < and >:

```
str = %q<"Stop", she said, "I can't live without 's and "s.">
```

The `[` and `]` characters also work this way. Alternatively, you can use any of the other special characters as your delimiter, ending the string with the same character. Here we use a dollar sign:

```
str = %q$"Stop", she said, "I can't live without 's and "s."$
```

The case of the letter `q` that leads off your string also matters: If it is lowercase, as it has been in all of our examples so far, the string gets the limited interpretation, single-quote style treatment. A string with an uppercase `Q` gets the more liberal double-quoted interpretation:

```
str = %Q<The time in now #{Time.now}>
```

One nice feature of all Ruby strings is that they can span lines:

```
a_multiline_string = "a multi-line
string"

another_one = %q{another multi-line
string}
```

Keep in mind that if you don't do anything about it, you will end up with embedded newlines in those multiline strings. You can cancel out the newlines in double-quoted strings (and the equivalent `%Q` form) with a trailing backslash:

```
yet_another = %Q{another multi-line string with \
no newline}
```

Finally, if you happen to have a very long multiline string, consider using a here document. A here document allows you to create a (usually long) string by sticking it right in your code:

```
heres_one = <<EOF
This is the beginning of my here document.
And this is the end.
EOF
```

Literal strings are frequently necessary and sometimes very messy. You can reduce the messiness by picking the best form for the situation.

Another API to Master

The real utility in strings lies in the things you can do with—and to—them. Like collections, Ruby strings support a very extensive API, an API that every Ruby programmer needs to master. For example, once you have your string you can call the `lstrip` method, which will return a copy of the string with all of the leading whitespace clipped off, so that `'  hello'.lstrip` will return `'hello'`.

Similarly, there is `rstrip`, which will peel the white space off of the end of your string as well as plain old `strip`, which will take the white space off of both ends. Similar to the trio of strip methods are `chomp` and `chop`. The `chomp` method is useful for those times when you are reading lines from a text file, something that tends to produce strings with unwanted line-terminating characters at the end. The `chomp` method will return a copy of the string with at most one newline character[2] lopped from the end. Thus,

```
"It was a dark and stormy night\n".chomp
```

Will give you the dark and stormy night without the newline. Note that `chomp` only does one newline at a time, so that `"hello\n\n\n".chomp` will return a string ending with two newlines. Be careful not to confuse `chomp` with the `chop` method. The `chop` method will simply knock off the last character of the string, no matter what it is, so that `"hello".chop` is just `"hell"`.

If the case of your characters concerns you, then you can use the `upcase` method to get a copy of your string with all of the lowercase letters made uppercase or the `downcase` method to go the opposite direction, or you can use `swapcase` to go from `'Hello'` to `'hELLO'`.

If you need to make more extensive changes to your string you might reach for the `sub` method. The name of this method is short for substitute, and the method will enable you to search for some substring and replace it with another. Thus, this:

```
'It is warm outside'.sub( 'warm', 'cold' )
```

2. Strictly speaking, `chomp` removes at most one record separator character from your string. Since the record separator is by default your system's newline character, most of the time this is a distinction without a difference.

Will evaluate to `'It is cold outside'`. If you need to be more enthusiastic about your substitutions, you might turn to `gsub`. While `sub` does at most one substitution, `gsub` will replace as many substrings as it possibly can. Thus, this:

```
puts 'yes yes'.sub( 'yes', 'no' )
puts 'yes yes'.gsub( 'yes', 'no' )
```

Will print out:

```
no yes
no no
```

Sometimes it's handy to be able to break your strings into smaller strings, and for that we have `split`. Call `split` with no arguments and it will return an array containing all the bits of your string that were separated by white space. Thus:

```
'It was a dark and stormy night'.split
```

Will return the following array:

```
["It", "was", "a", "dark", "and", "stormy", "night"]
```

Pass a string argument to split and it will break things up using that string as a delimiter, so that:

```
'Bill:Shakespeare:Playwright:Globe'.split( ':' )
```

Will evaluate to:

```
["Bill", "Shakespeare", "Playwright", "Globe"]
```

Like the collection classes, many of the string methods have counterparts whose names end with a `!`, which modify the original string instead of returning a modified copy. Thus, along with `sub`, we have `sub!`. So if you run this:

```
title = 'It was a dark and stormy night'
title.sub!( 'dark', 'bright' )
title.sub!( 'stormy', 'clear' )
```

Then `title` will end up as `'It was a bright and clear night'`.

If your interest runs more towards searching for things than changing them, you can locate a string within a bigger string with the `index` method:

```
"It was a dark and stormy night".index( "dark" )  # Returns 9
```

The String: A Place for Your Lines, Characters, and Bytes

Along with being things themselves, strings are also collections of other things. One odd aspect of strings is that they can't quite make up their minds as to the kinds of things they collect. Most commonly we think of strings as being collections of characters—after all, if you say `@author[3]` you are trying to get at the fourth character of the `@author` string. If you do think of your strings as collections of characters, you can use the `each_char` method to iterate. Thus, if `@author` is `'Clarke'`, this:

```
@author.each_char {|c| puts c}
```

Will produce:

```
C
l
a
r
k
e
```

You can also look at a string as a collection of bytes—after all, behind all those pretty characters are some utilitarian bytes. If you want to look at the bytes behind your string, you can loop through them with the aptly named `each_byte` method:

```
@author.each_byte { |b| puts b }
```

This code will print out a series of numbers, one for each byte:[3]

```
67
108
97
114
107
101
```

Strings can also hold more than one line, and if you find yourself with such a string, then you can look at it as a collection of lines:

```
@content.each_line { |line| puts line }
```

Interestingly, since the "collection of what?" question is not one that can really be answered for strings, Ruby's string class omits the plain old `each` method.[4]

In the Wild

As I said at the beginning of this chapter, most real-world programs spend their lives swimming through a sea of strings. For example, five minutes of poking around in the Ruby standard library uncovered this:

```
def html_escape(s)
  s.to_s.gsub(/&/, "&amp;").gsub(/\"/, "&quot;").
    gsub(/>/, "&gt;").gsub(/</, "&lt;")
end
```

This code,[5] which decontaminates a string so that it is suitable for use in HTML and XML, is from the RSS library. The slashes around the first arguments to `gsub` mark those arguments as regular expressions, which we will look at in the next chapter.

3. The difference between bytes and characters isn't just the difference between a single number and an associated character. If you happen to be dealing with the multi-byte characters—common in many Asian languages—then your strings can have many more bytes than characters.

4. This was not always the case. In pre-1.9 versions of Ruby, strings did have an each method that iterated over—wait for it—lines! This somewhat less than intuitive method disappeared in 1.9.

5. The code is slightly edited to fit on the page.

If making text safe for XML seems a bit pedestrian, consider the inflection code in Rails. Rails uses the **inflection** facility to figure out that the class contained within `current_employee.rb` should be `CurrentEmployee` and the database table associated with `CurrentEmployee` is `current_employees`. Although this sounds sophisticated in a very AI way, it's all done with simple string processing.

The key data structure behind the Rails inflections is a set of rules, where each rule consists of a pattern and a replacement. The easiest inflection rules to understand are ones that handle the irregular cases, like pluralizing `'person'` into `'people'`. Here's the code that creates a bunch of special pluralizing cases:

```
inflect.irregular('person', 'people')
inflect.irregular('man', 'men')
inflect.irregular('child', 'children')
inflect.irregular('sex', 'sexes')
```

And how does Rails actually apply those rules? With a call to `gsub`[6]:

```
inflections.plurals.each do |(rule, replacement)|
  break if result.gsub!(rule, replacement)
end
```

The inflections code is one of the things that gives Rails its wonderful human-centered feeling, and it's all built around `gsub`.

Staying Out of Trouble

Ruby strings are mutable. Since this aspect of the language takes many newcomers by surprise, let me say it again: Ruby strings are mutable. Like the collections, it is perfectly possible to change a Ruby string out from under code that is expecting the string to remain stable. Keep in mind that if you have these two strings:

```
first_name = 'Karen'
given_name = first_name
```

6. Again, the code is edited to fit on the page.

In fact you only have *one* string: Modify `first_name`:

```
first_name[0] = 'D'
```

And you have also changed `given_name`. You should treat Ruby strings like any other mutable data structure. Get in the habit of saying this:

```
first_name = first_name.upcase
```

Instead of this:

```
first_name.upcase!
```

Finally, you can sometimes make your string code more comprehensible by taking advantage of the flexibility that Ruby gives you between those string indexing brackets. You can, for example, use negative numbers to index from the end of the string, with `-1` being the last character in the string, so that you can turn this:

```
first_name[first_name.size - 1]
```

Into

```
first_name[-1]
```

You can also use ranges to index into your strings so that `"abcde"[3..4]` will evaluate to `"de"`.

Wrapping Up

As with the collection classes, Ruby strings are a good news/bad news proposition. The good news is that strings are very powerful objects that allow you to do all sorts of interesting things. You have a range of string literal forms to choose from, everything from `'simple quoted strings'` to `%Q{much more exotic forms}`. The bad news is that strings are very powerful objects *that you have to learn how to use*. The programming power that flows out of really mastering the string API is well worth the effort—especially when you couple your strings to regular expressions, which is where we turn next.