CHAPTER 5

Find the Right String with Regular Expressions

I'd like to pose a problem for you: Imagine that you have some text stored in a string, and somewhere in that text hides a time, something like `09:24 AM`. You need to write a program that will locate the time. How do you do it? You might, using the methods we talked about in the last chapter, scan the string for a digit and then check to see whether it is followed by another digit and then see whether *that* is followed by a colon, then two more digits . . . well, you get the picture. Doing all this with the `String` methods is certainly possible. Tedious, but possible.

Sometimes a problem speaks to you. This time-finding problem is trying to tell you that you are using the wrong tool. The tool this problem is crying out for is the regular expression. The idea behind the regular expression—that you construct a pattern that either will or will not match some string—is as simple as regular expressions are powerful. Unfortunately, regular expressions have a reputation for being complex and obscure, and many engineers shy away from them. But shying away from regular expressions is not really an option: As we saw in the last chapter, Ruby programs do a lot of string processing, and where you find strings you will find regular expressions. To help with the shyness, we'll spend the first half of this chapter going over the regular expression basics before turning to their use in Ruby code. As usual, we will finish with a look at some real-world uses for regular expressions as well as some of the sharp edges that you would do well to avoid.

Matching One Character at a Time

Although whole books[1] have been written about regular expressions, the basics of this very useful tool are not very complex. For example, in a regular expression, letters and numbers match themselves. Thus:

- The regular expression `x` will match `x`.
- The regular expression `aaa` will match three a's all in a row.
- The regular expression `123` will match the first three numbers.
- The regular expression `R2D2` will match the name of a certain sci-fi robot.

By default, case is important in regular expressions, so the last expression will *not* match `r2d2` nor will it match `R2d2`.[2]

Unlike letters and numbers, most of the punctuation characters—things like `.` and `*`—have special meanings in regular expressions. For example, the period or dot character matches *any* single character.[3] Thus:

- The regular expression `.` will match any single-character string including `r` and `%` and `~`.
- In the same way, two periods (`..`) will match any two characters, perhaps `xx` or `4F` or even `[!`, but won't match `Q` since it's one, not two, characters long.

You use a backslash to turn off the special meanings of the punctuation characters. Thus:

- The regular expression `\.` will match a literal dot.
- `3\.14` will match the string version of PI to two decimal places, complete with the decimal point: `3.14`
- `Mr\. Olsen` will match exactly one thing: `Mr. Olsen`

1. See the Appendix for the names of a couple of books on regular expressions.
2. As we will see in a bit, you can turn off regular expression case sensitivity.
3. Well, any single character except a newline character, but keep reading.

One of the beauties of regular expressions is that you can combine the different kinds of expressions to build up more complex patterns. For example:

- The regular expression `A.` will match any two-character string that starts with a capital `A`, including `AM`, `An`, `At`, and even `A=`.
- Similarly, `...X` will match any four-character string that *ends* with an `X`, including `UVWX` and `XOOX`.
- The regular expression `.r\. Smith` will match both `Dr. Smith` as well as `Mr. Smith` but not `Mrs. Smith`.

It's easy to see how the dot, with its talent for matching any single character, extends the reach of regular expressions. But what if you want to match something less than any? What if you want to write an expression that would match only the letters, or only the vowels, or maybe just the numbers?

Sets, Ranges, and Alternatives

Enter the **set**. Sets match any one of a bunch of characters. To create a regular expression set, you wrap the characters in square brackets: Thus the regular expression `[aeiou]` will match any single lowercase vowel. The key word there is *single*: Although `[aeiou]` will match the single character `a` or the single character `i`, it will not match `ai`, since `ai` is two characters long. Similarly:

- The regular expression `[0123456789]` will match any single digit.
- `[0123456789abdef]` will match any single hexadecimal digit,[4] like `7` or `f`.

Once you understand how sets work, you can start to build up some fairly complex regular expressions. For example:

- The regular expression `[Rr]uss [Oo]lsen` will match my name, with or without leading capitals.
- More practically, you could use `[0123456789abcdef][0123456789abcdef]` to pick out a two-digit hexadecimal number like `3e` or `ff`.

4. As long as the letter, if there is one, is lowercase.

- You can also use `[aApP][mM]` to match `am` or `PM` and anything in between, like `aM` or `Pm`.

There is one problem with simple sets: They don't scale well at the keyboard. No one wants to type `[0123456789abcdef]`, let alone `[abcdefghijklmnopqrstuvwxyz]` or `[abcdefghijklmnopqrstuvwxyz0123456789]`. Fortunately, there is a special regular expression syntax for building just this sort of large set of continuous characters: the range.

As the name suggests, you define a **range** by specifying the beginning and end of a sequence of characters, separated by a dash. So the range `[0–9]` will match exactly what you expect: any decimal digit. Similarly, `[a-z]` will match any lowercase letter. You can also combine several ranges together and mix them with the regular set notation, so that:

- `[0-9abcdef]` will match a single hexadecimal digit.
- `[0-9a-f]` will also match a single hexadecimal digit.
- `[0-9a-zA-Z_]` will match any letter, number, or the underscore character.

If even `[0-9]` seems like too much work, there are some shorter shortcuts for common sets:

- `\d` will match any digit, so that `\d\d` will match any two digit number from `00` to `99`.
- `\w`, where the `w` stands for "word character," will match any letter, number or the underscore.
- `\s` will match any white space character including the vanilla space, the tab, and the newline.

Another way to extend the power of your regular expressions is by using alternatives. If you separate the different parts of your expression with the vertical bar character `|`, the expression will match either the thing before the bar *or* the thing after it:

- `A|B` will match either `A` or `B`.
- `AM|PM` will match either `AM` or `PM`.
- `Batman|Spiderman` will match the name of one of the two superheros.

The sky is the limit with alternatives: You can specify as many choices as you like. Thus `A\.M\.|AM|P\.M\.|PM` will match `A.M.` or `AM`, or `P.M.` or `PM`. You can also surround your alternatives in parentheses to set them off from the rest of the pattern, so that:

```
The (car|boat) is red
```

Will match both `The car is red` as well as `The boat is red`.

Pull all of this together and you have enough regular-expression moxie to solve the time-finding problems that we opened this chapter with:

```
\d\d:\d\d (AM|PM)
```

Translated into English, the expression above says "Any string that starts with two digits, followed by a colon, followed by two more digits, followed by a space, followed by either `AM` or `PM`."

The Regular Expression Star

Now that we have the basics of regular expressions down it's time to move on to the interesting part: the asterisk. In regular expressions, an asterisk (`*`) matches zero or more of the thing that came just before it. Pause and think that through for a minute . . . zero or more of the thing that came just before the asterisk. What this means is that `A*` will match zero or more `A`'s. The `A` is the thing that came before the star, so the pattern will match zero or more `A`'s. Similarly:

- `AB*` will match `AB`—that's an `A` followed by one `B`.
- `AB*` will also match `ABB` as well as `ABBBBBBBB`—remember, it's an `A` followed by any number of `B`'s.
- Don't forget that `AB*` will also match plain old `A`—any number of `B`'s includes no `B`'s at all.

Although our examples so far have the star at the end of the expression, you can put it anywhere: It can be up at the front, so that `R*uby` means any number of `R`'s followed by `uby`. So `uby`, `Ruby`, `RRuby`, and `RRRRRRRuby` all match. The star can also be in the middle of your expression, so that you can use `Rub*y` to match `Ruy`, `Ruby`, as well as `Rubbbbbbbbby`. There's also no limit to the number of stars that you can use in

a regular expression so that `R*u*by` will match any number of `R`'s followed by any number of `u`'s followed by `by`.

You can also use the star in combination with sets, so that:

- The expression `[aeiou]*` will match any number of vowels: The whole `[aeiou]` set is the thing that came before the star.
- Likewise, the expression `[0–9]*` will match any number of digits.
- And `[0-9a-f]*` will match any number of hexadecimal digits.

Finally, we can combine the idea of a dot matching any single character and the `*` matching zero or more of the thing that came before into one of the most widely used of all regular expressions:

```
.*
```

This little gem—just a dot followed by an asterisk—will match any number of any characters, or to put it another way, *anything*. This works because the star matches any number of what came before, so that `.*` is the same as `.` and `..` and `...` and so on. But `.` matches any single character, while `..` matches any two characters, and so on. So `.*` will match anything.

Frequently you combine `.*` with other regular expression bits to make extremely elastic patterns. For example:

- The regular expression `George.*` will match the full name of anyone whose first name is `George`.
- In contrast, `.*George` will match the name of anyone whose last name is `George`.
- Finally `.*George.*` will match the name of anyone who has `George` in his name somewhere.

Regular Expressions in Ruby

In Ruby, the regular expression, or `Regexp` for short,[5] is one of the built-in data types, with its own special literal syntax. To make a Ruby regular expression you encase your pattern between forward slashes. So in Ruby our time regular expression would be:

5. `Regexp` is the name of the Ruby regular expression class.

```
/\d\d:\d\d (AM|PM)/
```

You use the =~ operator to test whether a regular expression matches a string. Thus, if we wanted to match the regular expression above with an actual time we would run:

```
puts /\d\d:\d\d (AM|PM)/ =~ '10:24 PM'
```

Which would print out:

```
0
```

That zero is trying to tell us a couple of things. First, it is saying that the regular expression matched, starting at index zero. Second, the zero is telling us is that when you match a regular expression, Ruby scans along the string, searching for a match *anywhere* in the string. We can see the scanning in action with this next example:

```
puts /PM/ =~ '10:24 PM'
```

Run the code shown here and it will print:

```
6
```

That six is an indication that the `Regexp` did match, but only after Ruby scanned well into the string. If there is no match, then you will get a `nil` back for your trouble, so that this:

```
/May/ =~ 'Sometime in June'
```

Will return a `nil`. Since =~ returns a number when it finds a match and `nil` if it doesn't, you can use regular expression matches as booleans:

```
the_time = '10:24 AM'
puts "It's morning!" if /AM/ =~ the_time
```

The =~ operator is also ambidextrous: It doesn't matter whether the string or the regular expression comes first, so we could rephrase the last example as:

```
puts "It's morning!" if '10:24 AM' =~ /AM/
```

As I mentioned earlier, regular expressions are by default case sensitive: `/AM/` will not match `'am'`. Fortunately, you can turn that case sensitivity off my sticking an `i` on the end of your expression, so that this:

```
puts "It matches!" if /AM/i =~ 'am'
```

Will print something.

Aside from their more or less stand-alone use with the =~ operator, regular expressions also come into play in the string methods that involve searching. Thus, you can pass a regular expression into the string `gsub` method, perhaps to blot out all of the times in the content of a document:

```
class Document
  # Most of the class omitted...

  def obscure_times!
    @content.gsub!( /\d\d:\d\d (AM|PM)/, '**:** **' )
  end
end
```

Taken together, strings and regular expressions make for a very powerful text-processing toolkit.

Beginnings and Endings

The fact that the =~ operator scans for a match anywhere in the string raises an interesting question: What if you only want your regular expression to match at the beginning of the string? For example, what if you were looking for the words `Once upon a time`? No problem: Simply cook up a straightforward regular expression:

```
/Once upon a time/
```

But what if you were looking specifically for fairy tales, which typically *start* with those immortal words? Again, there's a special regular expression for that, `\A`. Thus:

```
/\AOnce upon a time/
```

Will match a string only if it *begins* like a fairy tale. Note that the `\A` doesn't match the first *character*. Instead, it matches the unseen leading edge of the string. Similarly, `\z` (note the lower case) matches the end of the string, so that:

```
/and they all lived happily ever after\z/
```

Will only match a string that *ends* like a classic fairy tale.

Multiline strings, the ones full of embedded newlines, present some interesting challenges when you are doing this sort of work. Imagine that our alleged fairy tale is stored in a multiline string like this:

```
content = 'The Princess And The Monkey

Once upon a time there was a princess...
...and they all lived happily ever after.

The End'
```

Now *you* know, and *I* know, this is a fairy tale, but how do we write a regular expression that will know it? Our previous try, `/\AOnce upon a time/` isn't going to work because the string above doesn't *start* with the right words. Instead, inside of the string is a *line* that starts with the magic words. Fortunately, there a Regexp for that too: the circumflex ^. The circumflex character matches two things: the beginning of the string *or* the beginning of any line within the string. Just the thing to ferret out a fairy tale:

```
puts "Found it" if content =~ /^Once upon a time/
```

Similarly, the dollar sign `$` matches the end of the string *or* the end of any line within the string, so that we could also say:

```
puts "Found it" if content =~ /happily ever after\.$/
```

Multiline strings pose one more challenge to regular expressions, specifically to the dot: By default, the dot will match any character *except* the newline character. So if we were trying to match the whole text of fairy tale, less the title and `The End`, we might try:

```
/^Once upon a time.*happily ever after\.$/
```

But this attempt is doomed because the `.*` won't match across the lines . . . unless we simply turn off this behavior by adding an `m` to our expression:

```
/^Once upon a time.*happily ever after\.$/m
```

And then they did live happily ever after.

In the Wild

You can find a great example of the practical use of regular expressions in the `time.rb` file, which comes with your Ruby installation. The code in `time.rb` knows how to parse a string containing a date and time in any of a number of standard formats[6] and turn it into an instance of the `Time` class. Unlike the simple example that kicked off this chapter, `time.rb` needs to deal with all of the glorious complexities of real dates and times. Early on in the file you will find the `zone_offset` method, which attempts to parse the time zone section of a date and time string, figuring out if this string is supposed to represent 12:01 AM in Greenwich or Green Bay or Greenland. Among other things, the method needs to be able to understand time zones expressed as names like `'UTC'`, or in numeric offsets like `'-07:00'`.

The `zone_offset` method starts out by switching the whole zone string to uppercase, mixing in a pinch of simplicity right there at the start:

```
def zone_offset(zone, year=self.now.year)
  # ...
  zone = zone.upcase
  # ...
```

Next the code tests the zone against a number of regular expressions, trying to figure out if the zone is one of the numeric forms:

```
if /\A([+-])(\d\d):?(\d\d)\z/ =~ zone
```

This expression makes use of a regular expression feature that we haven't seen yet: Similar to the asterisk, the question mark matches zero or *one* of the things that came

6. It can also go the other way, from the Time object to a string.

before. Thus `:?` allows the regular expression to match zone offsets with or without a colon.

If the regular expression doesn't match, the `zone_offset` method goes on to do some plain old string comparison-based processing, looking for things like `'GMT'` or `'EST'`, all of which are stored in the `ZoneOffset` collection:

```
elsif ZoneOffset.include?(zone)
```

The `zone_offset` method takes a wonderfully pragmatic approach to an annoyingly complex problem: It uses regular expressions where they work best and the simple string methods where they work best.

Staying Out of Trouble

There are a couple of easy-to-make but also easy-to-avoid mistakes that you can perpetrate when working with regular expressions. The first is to write something like this:

```
puts /abc*/ == "abccccc"
```

The problem with this code is that I've let my fingers do the thinking as well as the typing. The regular expression `/abc*/` will never, ever be equal to `"abccccc"`, at least according to the `==` operator. Remember that the regular expression match operator is `=~` and not `==`. The correct way to say it is:

```
puts /abc*/ =~ "abccccc"
```

The second boneheaded thing that you can do, particularly if your background includes C or C++,[7] is to look at the output of the last example:

```
0
```

And decide that there was no match. Zero always seems so negative to C++ programmers. What the result is trying to tell you is that the regular expression matched the string starting at the beginning, or the zero-th index of the string. As we saw earlier, regular expression matches return `nil` when there is no match.

7. As does mine, so I know the bonehead of which I speak.

Wrapping Up

When it comes to serious string processing, there is only one word for regular expressions, and that word is *gift*. Regular expressions, with their stars, dollar signs, and little hat characters, can be intimidating, but the fundamental ideas are well within the reach of any coder.