Chapter 7

Regular Expressions

We probably spend most of our time in Ruby working with strings, so it seems reasonable for Ruby to have some great tools for working with those strings. As we've seen, the String class itself is no slouch—it has more than 100 methods. But there are still things that the basic String class can't do. For example, we might want to see whether a string contains two or more repeated characters, or we might want to replace every word longer than fifteen characters with its first five characters and an ellipsis. This is when we turn to the power of regular expressions.

Now, before we get too far in, here's a warning: there have been whole books written on regular expressions.[1] There is complexity and subtlety here that rivals that of the rest of Ruby. So if you've never used regular expressions, don't expect to read through this whole chapter the first time. In fact, you'll find two emergency exits in what follows. If you're new to regular expressions, I strongly suggest you read through to the first and then bail out. When some regular expression question next comes up, come back here and maybe read through to the next exit. Then, later, when you're feeling comfortable with regular expressions, you can give the whole chapter a read.

What Regular Expressions Let You Do

A regular expression is a pattern that can be matched against a string. It can be a simple pattern, such as *the string must contain the sequence of letters "cat"*, or the pattern can be complex, such as *the string must start with a protocol identifier, followed by two literal forward slashes, followed by...*, and so on. This is cool in theory. But what makes regular expressions so powerful is what you can do with them in practice:

- You can test a string to see whether it matches a pattern.
- You can extract from a string the sections that match all or part of a pattern.
- You can change the string, replacing parts that match a pattern.

Ruby provides built-in support that makes pattern matching and substitution convenient and concise. In this section, we'll work through the basics of regular expression patterns and

1. Such as *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools* [Fri02]

see how Ruby supports matching and replacing based on those patterns. In the sections that follow, we'll dig deeper into both the patterns and Ruby's support for them.

Ruby's Regular Expressions

There are many ways of creating a regular expression pattern. By far the most common is to write it between forward slashes. Thus, the pattern /cat/ is a regular expression literal in the same way that "cat" is a string literal.

/cat/ is an example of a simple, but very common, pattern. It matches any string that contains the substring cat. In fact, inside a pattern, all characters except ., |, (,), [,], {, }, +, \, ^, $, *, and ? match themselves. So, at the risk of creating something that sounds like a logic puzzle, here are some patterns and examples of strings they match and don't match:

/cat/ matches "dog and cat", and "catch", but not "Cat" or "c.a.t."
/123/ matches "86512312" and "abc123", but not "1.23"
/t a b/ matches "hit a ball" but not "table"

If you want to match one of the special characters literally in a pattern, precede it with a backslash, so /*/ is a pattern that matches a single asterisk, and /\// is a pattern that matches a forward slash.

Pattern literals are like double-quoted strings. In particular, you can use #{...} expression substitutions in the pattern.

Matching Strings with Patterns

The Ruby operator =~ matches a string against a pattern. It returns the character offset into the string at which the match occurred:

```
/cat/ =~ "dog and cat"   # =>  8
/cat/ =~ "catch"         # =>  0
/cat/ =~ "Cat"           # =>  nil
```

You can put the string first if you prefer:[2]

```
"dog and cat" =~ /cat/   # =>  8
"catch" =~ /cat/         # =>  0
 "Cat" =~ /cat/          # =>  nil
```

Because pattern matching returns nil when it fails and because nil is equivalent to false in Ruby, you can use the result of a pattern match as a condition in statements such as if and while.

```
str = "cat and dog"
if str =~ /cat/
  puts "There's a cat here somewhere"
end
```

2. Some folks say this is inefficient, because the string will end up calling the regular expression code to do the match. These folks are correct in theory but wrong in practice.

produces:

```
There's a cat here somewhere
```

The following code prints lines in testfile that contain on:

```
File.foreach("testfile").with_index do |line, index|
  puts "#{index}: #{line}" if line =~ /on/
end
```

produces:

```
0: This is line one
3: And so on...
```

You can test to see whether a pattern does not match a string using !~:

```
File.foreach("testfile").with_index do |line, index|
  puts "#{index}: #{line}" if line !~ /on/
end
```

produces:

```
1: This is line two
2: This is line three
```

Changing Strings with Patterns

The sub takes a pattern and some replacement text:[3] If it finds a match for the pattern in the string, it replaces the matched substring with the replacement text.

```
str = "Dog and Cat"
new_str = str.sub(/Cat/, "Gerbil")
puts "Let's go to the #{new_str} for a pint."
```

produces:

```
Let's go to the Dog and Gerbil for a pint.
```

The sub method changes only the first match it finds. To replace all matches, use gsub. (The *g* stands for global.)

```
str = "Dog and Cat"
new_str1 = str.sub(/a/, "*")
new_str2 = str.gsub(/a/, "*")
puts "Using sub: #{new_str1}"
puts "Using gsub: #{new_str2}"
```

produces:

```
Using sub: Dog *nd Cat
Using gsub: Dog *nd C*t
```

Both sub and gsub return a new string. (If no substitutions are made, that new string will just be a copy of the original.)

3. Actually, it does more than that, but we won't get to that for a while.

If you want to modify the original string, use the sub! and gsub! forms:

```
str = "now is the time"
str.sub!(/i/, "*")
str.gsub!(/t/, "T")
puts str
```

produces:

```
now *s The Time
```

Unlike sub and gsub, sub! and gsub! return the string only if the pattern was matched. If no match for the pattern is found in the string, they return nil instead. This means it can make sense (depending on your need) to use the ! forms in conditions.

So, at this point you know how to use patterns to look for text in a string and how to substitute different text for those matches. And, for many people, that's enough. So if you're itching to get on to other Ruby topics, now is a good time to move on to the next chapter. At some point, you'll likely need to do something more complex with regular expressions (for example, matching a time by looking for two digits, a colon, and two more digits). You can then come back and read the next section.

Or, you can just stay right here as we dig deeper into patterns, matches, and replacements.

Digging Deeper

Like most things in Ruby, regular expressions are just objects—they're instances of class Regexp. This means you can assign them to variables, pass them to methods, and so on:

```
str = "dog and cat"
pattern = /nd/
pattern =~ str   # =>   5
str =~ pattern   # =>   5
```

You can also create regular expression objects by calling the Regexp class's new method and by using the %r{...} syntax. The %r syntax is particularly useful when creating patterns that contain forward slashes:

```
/mm\/dd/              # =>   /mm\/dd/
Regexp.new("mm/dd")   # =>   /mm\/dd/
%r{mm/dd}             # =>   /mm\/dd/
```

Regular Expression Options

A regular expression may include one or more options that modify the way the pattern matches strings. If you're using literals to create the Regexp object, then the options are one or more characters placed immediately after the terminator. If you're using Regexp.new, the options are constants used as the second parameter of the constructor.

Playing with Regular Expressions

If you're like me, you'll sometimes get confused by regular expressions. You create something that *should* work, but it just doesn't seem to match. That's when I fall back to irb. I'll cut and paste the regular expression into irb and then try to match it against strings. I'll slowly remove portions until I get it to match my target string and add stuff back until it fails. At that point, I'll know what I was doing wrong.

i *Case insensitive*. The pattern match will ignore the case of letters in the pattern and string. (The old technique of setting $= to make matches case insensitive no longer works.)

o *Substitute once*. Any #... substitutions in a particular regular expression literal will be performed just once, the first time it is evaluated. Otherwise, the substitutions will be performed every time the literal generates a Regexp object.

m *Multiline mode*. Normally, "." matches any character except a newline. With the /m option, "." matches any character.

x *Extended mode*. Complex regular expressions can be difficult to read. The x option allows you to insert spaces and newlines in the pattern to make it more readable. You can also use # to introduce comments.

Another set of options allows you to set the language encoding of the regular expression. If none of these options is specified, the regular expression will have US-ASCII encoding if it contains only 7-bit characters. Otherwise, it will use the default encoding of the source file containing the literal: n: no encoding (ASCII), e: EUC, s: SJIS, and u: UTF-8.

Matching Against Patterns

Once you have a regular expression object, you can match it against a string using the Regexp#match(*string*) method or the match operators =~ (positive match) and !~ (negative match). The match operators are defined for both String and Regexp objects. One operand of the match operator must be a regular expression.

```
name = "Fats Waller"
name =~ /a/                     # =>  1
name =~ /z/                     # =>  nil
/a/ =~ name                     # =>  1
/a/.match(name)                 # =>  #<MatchData "a">
Regexp.new("all").match(name)   # =>  #<MatchData "all">
```

The match operators return the character position at which the match occurred, while the match method returns a MatchData object. In all forms, if the match fails, nil is returned.

After a successful match, Ruby sets a whole bunch of magic variables. For example, $& receives the part of the string that was matched by the pattern, $` receives the part of the

string that preceded the match, and $' receives the string after the match. However, these particular variables are considered to be fairly ugly, so most Ruby programmers instead use the MatchData object returned from the match method, because it encapsulates all the information Ruby knows about the match. Given a MatchData object, you can call pre_match to return the part of the string before the match, post_match for the string after the match, and index using [0] to get the matched portion.

We can use these methods to write a method, show_regexp, that illustrates where a particular pattern matches:

```
def show_regexp(string, pattern)
  match = pattern.match(string)
  if match
    "#{match.pre_match}->#{match[0]}<-#{match.post_match}"
  else
    "no match"
  end
end

show_regexp('very interesting', /t/)   # =>  very in->t<-eresting
show_regexp('Fats Waller', /a/)        # =>  F->a<-ts Waller
show_regexp('Fats Waller', /lle/)      # =>  Fats Wa->lle<-r
show_regexp('Fats Waller', /z/)        # =>  no match
```

Deeper Patterns

We said earlier that, within a pattern, all characters match themselves except ., |, (,), [,], {, }, +, \, ^, $, *, and ?. Let's dig a bit deeper into this.

First, always remember that you need to escape any of these characters with a backslash if you want them to be treated as regular characters to match:

```
show_regexp('yes | no', /\|/)         # =>  yes ->|<- no
show_regexp('yes (no)', /\(no\)/)     # =>  yes ->(no)<-
show_regexp('are you sure?', /e\?/)   # =>  are you sur->e?<-
```

Now let's see what some of these characters mean if you use them without escaping them.

Anchors

By default, a regular expression will try to find the first match for the pattern in a string. Match /iss/ against the string "Mississippi," and it will find the substring "iss" starting at position 1 (the second character in the string). But what if you want to force a pattern to match only at the start or end of a string?

The patterns ^ and $ match the beginning and end of a line, respectively. These are often used to *anchor* a pattern match; for example, /^option/ matches the word *option* only if it appears at the start of a line. The sequence \A matches the beginning of a string, and \z and \Z match the end of a string. (Actually, \Z matches the end of a string *unless* the string ends with a \n, in which case it matches just before the \n.)

```
str = "this is\nthe time"
show_regexp(str, /^the/)     # =>   this is\n->the<- time
show_regexp(str, /is$/)      # =>   this ->is<-\nthe time
show_regexp(str, /\Athis/)   # =>   ->this<- is\nthe time
show_regexp(str, /\Athe/)    # =>   no match
```

Similarly, the patterns \b and \B match word boundaries and nonword boundaries, respectively. Word characters are ASCII letters, numbers, and underscores:

```
show_regexp("this is\nthe time", /\bis/)   # =>   this ->is<-\nthe time
show_regexp("this is\nthe time", /\Bis/)   # =>   th->is<- is\nthe time
```

Character Classes

A *character class* is a set of characters between brackets: [*characters*] matches any single character between the brackets. [aeiou] will match a vowel, [,.:;!?] matches some punctuation, and so on. The significance of the special regular expression characters—.|(){+^$*?—is turned off inside the brackets. However, normal string substitution still occurs, so (for example) \b represents a backspace character and \n a newline (see Table 22.2 on page 329). In addition, you can use the abbreviations shown in Table 7.1 on page 125 so that (for example) \s matches any whitespace character, not just a literal space:

```
show_regexp('Price $12.', /[aeiou]/)   # =>   Pr->i<-ce $12.
show_regexp('Price $12.', /[\s]/)      # =>   Price-> <-$12.
show_regexp('Price $12.', /[$.]/)      # =>   Price ->$<-12.
```

Within the brackets, the sequence c_1-c_2 represents all the characters from c_1 to c_2 in the current encoding:

```
a = 'see [The PickAxe-page 123]'
show_regexp(a, /[A-F]/)        # =>   see [The Pick->A<-xe-page 123]
show_regexp(a, /[A-Fa-f]/)     # =>   s->e<-e [The PickAxe-page 123]
show_regexp(a, /[0-9]/)        # =>   see [The PickAxe-page ->1<-23]
show_regexp(a, /[0-9][0-9]/)   # =>   see [The PickAxe-page ->12<-3]
```

You can negate a character class by putting an up arrow or caret (^) immediately after the opening bracket:

```
show_regexp('Price $12.', /[^A-Z]/)        # =>   P->r<-ice $12.
show_regexp('Price $12.', /[^\w]/)         # =>   Price-> <-$12.
show_regexp('Price $12.', /[a-z][^a-z]/)   # =>   Pric->e <-$12.
```

The POSIX character classes in Table 7.2 on page 125 correspond to the ctype(3) macros of the same names. They can also be negated by putting an up arrow (or caret) after the first colon:

```
show_regexp('Price $12.', /[aeiou]/)           # =>   Pr->i<-ce $12.
show_regexp('Price $12.', /[[:digit:]]/)       # =>   Price $->1<-2.
show_regexp('Price $12.', /[[:space:]]/)       # =>   Price-> <-$12.
show_regexp('Price $12.', /[[:^alpha:]]/)      # =>   Price-> <-$12.
show_regexp('Price $12.', /[[:punct:]aeiou]/)  # =>   Pr->i<-ce $12.
```

If you want to include the literal characters] and - within a character class, put them at the start or escape them with \:

```
a = 'see [The PickAxe-page 123]'
show_regexp(a, /[]]/)       # =>   see [The PickAxe-page 123->]<-
show_regexp(a, /[0-9\]]/)   # =>   see [The PickAxe-page ->1<-23]
show_regexp(a, /[\d\-]/)    # =>   see [The PickAxe->-<-page 123]
```

Some character classes are used so frequently that Ruby provides abbreviations for them. These abbreviations are listed in Table 7.1 on the next page—they may be used both within brackets and in the body of a pattern.

```
show_regexp('It costs $12.', /\s/)   # =>   It-> <-costs $12.
show_regexp('It costs $12.', /\d/)   # =>   It costs $->1<-2.
```

You can create the intersection of character classes using &&. So, to match all lowercase ASCII letters that aren't vowels, you could use this:

```
str = "now is the time"
str.gsub(/[a-z&&[^aeiou]]/, '*')   # =>   "*o* i* **e *i*e"
```

1.9 The \p construct is new with Ruby 1.9. It gives you an encoding-aware way of matching a character with a particular Unicode property (shown in Table 7.3 on page 126):

```
# encoding: utf-8
string = "δy/δx = 2πx"
show_regexp(string, /\p{Alnum}/)   # =>   δ->y<-/δx = 2πx
show_regexp(string, /\p{Digit}/)   # =>   δy/δx = ->2<-πx
show_regexp(string, /\p{Space}/)   # =>   δy/δx-> <-= 2πx
show_regexp(string, /\p{Greek}/)   # =>   δy/δx = 2->π<-x
show_regexp(string, /\p{Graph}/)   # =>   ->δ<-y/δx = 2πx
```

Finally, a period (.) appearing outside brackets represents any character except a newline (though in multiline mode it matches a newline, too):

```
a = 'It costs $12.'
show_regexp(a, /c.s/)   # =>   It ->cos<-ts $12.
show_regexp(a, /./)     # =>   ->I<-t costs $12.
show_regexp(a, /\./)    # =>   It costs $12->.<-
```

Repetition

When we specified the pattern that split the song list line, /\s*\|\s*/, we said we wanted to match a vertical bar surrounded by an arbitrary amount of whitespace. We now know that the \s sequences match a single whitespace character and \| means a literal vertical bar, so it seems likely that the asterisks somehow mean "an arbitrary amount." In fact, the asterisk is one of a number of modifiers that allow you to match multiple occurrences of a pattern.

If *r* stands for the immediately preceding regular expression within a pattern, then

*r**	Matches zero or more occurrences of *r*.
r+	Matches one or more occurrences of *r*.
r?	Matches zero or one occurrence of *r*.
r{m,n}	Matches at least m and at most n occurrences of *r*.
r{m,}	Matches at least m occurrences of *r*.
r{,n}	Matches at most n occurrences of *r*.
r{m}	Matches exactly m occurrences of *r*.

Table 7.1. Character Class Abbreviations

Text in parentheses indicates the Unicode classes. These apply if the regular expression's encoding is one of the Unicode encodings.

Sequence	As [. . .]	Meaning (Unicode)
\d	[0-9]	Decimal digit character (*Decimal_Number*)
\D	[^0-9]	Any character except a digit
\h	[0-9a-fA-F]	Hexadecimal digit character
\H	[^0-9a-fA-F]	Any character except a hex digit
\s	[␣\t\r\n\f]	Whitespace character (+ *Line_Separator*)
\S	[^␣\t\r\n\f]	Any character except whitespace
\w	[A-Za-z0-9_]	Word character (+ *Connector_Punctuation*, *Letter*, *Mark*, and *Number*)
\W	[^A-Za-z0-9_]	Any character except a word character

Table 7.2. Posix Character Classes

Text in parentheses indicates the Unicode classes. These apply if the regular expression's encoding is one of the Unicode encodings.

POSIX Character Classes (Unicode)	
[:alnum:]	Alphanumeric (*Letter* | *Mark* | *Decimal_Number*)
[:alpha:]	Uppercase or lowercase letter (*Letter* | *Mark*)
[:ascii:]	7-bit character including nonprinting
[:blank:]	Blank and tab (+ *Space_Separator*)
[:cntrl:]	Control characters—at least 0x00–0x1f, 0x7f (*Control* | *Format* | *Unassigned* | *Private_Use* | *Surrogate*)
[:digit:]	Digit (*Decimal_Number*)
[:graph:]	Printable character excluding space (Unicode also excludes *Control, Unassigned,* and *Surrogate*)
[:lower:]	Lowercase letter (*Lowercase_Letter*)
[:print:]	Any printable character (including space)
[:punct:]	Printable character excluding space and alphanumeric. Unicode: (*Connector_Punctuation* | *Dash_Punctuation* | *Close_Punctuation* | *Final_Punctuation* | *Initial_Punctuation* | *Other_Punctuation* | *Open_Punctuation*)
[:space:]	Whitespace (same as \s)
[:upper:]	Uppercase letter (*Uppercase_Letter*)
[:xdigit:]	Hex digit (0–9, a–f, A–F)
[:word:]	Alphanumeric, underscore, and multibyte (*Letter* | *Mark* | *Decimal_Number* | *Connector_Punctuation*)

Table 7.3. Unicode Character Properties

Character Properties	
\p{*name*}	Matches character with named property
\p{^*name*}	Matches any character except named property
\P{*name*}	Matches any character except named property
Property names	
All encodings	Alnum, Alpha, Blank, Cntrl, Digit, Graph, Lower, Print, Punct, Space, Upper, XDigit, Word, ASCII
EUC and SJIS	Hiragana, Katakana
UTF-n	Any, Assigned, C, Cc, Cf, Cn, Co, Cs, L, Ll, Lm, Lo, Lt, Lu, M, Mc, Me, Mn, N, Nd, Nl, No, P, Pc, Pd, Pe, Pf, Pi, Po, Ps, S, Sc, Sk, Sm, So, Z, Zl, Zp, Zs, Arabic, Armenian, Bengali, Bopomofo, Braille, Buginese, Buhid, Canadian_Aboriginal, Cherokee, Common, Coptic, Cypriot, Cyrillic, Deseret, Devanagari, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Inherited, Kannada, Katakana, Kharoshthi, Khmer, Lao, Latin, Limbu, Linear_B, Malayalam, Mongolian, Myanmar, New_Tai_Lue, Ogham, Old_Italic, Old_Persian, Oriya, Osmanya, Runic, Shavian, Sinhala, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Yi

These repetition constructs have a high precedence—they bind only to the immediately preceding matching construct in the pattern. /ab+/ matches an *a* followed by one or more *b*'s, not a sequence of *ab*'s.

These patterns are called *greedy*, because by default they will match as much of the string as they can. You can alter this behavior, and have them match the minimum, by adding a question mark suffix. The repetition is then called *lazy*—it stops once it has done the minimum amount of work required.

```
a = "The moon is made of cheese"
show_regexp(a, /\w+/)            # =>   ->The<- moon is made of cheese
show_regexp(a, /\s.*\s/)         # =>   The-> moon is made of <-cheese
show_regexp(a, /\s.*?\s/)        # =>   The-> moon <-is made of cheese
show_regexp(a, /[aeiou]{2,99}/)  # =>   The m->oo<-n is made of cheese
show_regexp(a, /mo?o/)           # =>   The ->moo<-n is made of cheese
# here's the lazy version
show_regexp(a, /mo??o/)          # =>   The ->mo<-on is made of cheese
```

(There's an additional modifier, +, that makes them greedy and also stops backtracking, but that will have to wait until the advanced section of the chapter.)

Be very careful when using the * modifier. It matches zero or more occurences. I know that I personally often forget about the zero part. In particular, a pattern that contains just a * repetition will always match, whatever string you pass it. The pattern /a*/ will always match, because every string contains zero or more a's.

```
a = "The moon is made of cheese"
# both of these match an empty substring at the start of the string
show_regexp(a, /m*/)   # =>   ->><-The moon is made of cheese
show_regexp(a, /Z*/)   # =>   ->><-The moon is made of cheese
```

Alternation

We know that the vertical bar is special, because our line-splitting pattern had to escape it with a backslash. That's because an unescaped vertical bar | matches either the construct that precedes it or the construct that follows it:

```
a = "red ball blue sky"
show_regexp(a, /d|e/)                 # =>   r->e<-d ball blue sky
show_regexp(a, /al|lu/)               # =>   red b->al<-l blue sky
show_regexp(a, /red ball|angry sky/)  # =>   ->red ball<- blue sky
```

There's a trap for the unwary here, because | has a very low precedence. The last example in the previous lines matches *red ball* or *angry sky*, not *red ball sky* or *red angry sky*. To match *red ball sky* or *red angry sky*, you'd need to override the default precedence using grouping.

Grouping

You can use parentheses to group terms within a regular expression. Everything within the group is treated as a single regular expression.

```
# This matches an 'a' followed by one or more 'n's
show_regexp('banana', /an+/)     # =>   b->an<-ana
# This matches the sequence 'an' one or more times
show_regexp('banana', /(an)+/)   # =>   b->anan<-a
```

```
a = 'red ball blue sky'
show_regexp(a, /blue|red/)         # =>   ->red<- ball blue sky
show_regexp(a, /(blue|red) \w+/)   # =>   ->red ball<- blue sky
show_regexp(a, /(red|blue) \w+/)   # =>   ->red ball<- blue sky
show_regexp(a, /red|blue \w+/)     # =>   ->red<- ball blue sky

show_regexp(a, /red (ball|angry) sky/)   # =>   no match
a = 'the red angry sky'
show_regexp(a, /red (ball|angry) sky/)   # =>   the ->red angry sky<-
```

Parentheses also collect the results of pattern matching. Ruby counts opening parentheses and for each stores the result of the partial match between it and the corresponding closing parenthesis. You can use this partial match both within the rest of the pattern and in your Ruby program. Within the pattern, the sequence \1 refers to the match of the first group, \2 the second group, and so on. Outside the pattern, the special variables $1, $2, and so on, serve the same purpose.

```
/(\d\d):(\d\d)(..)/ =~ "12:50am"     # =>   0
"Hour is #$1, minute #$2"            # =>   "Hour is 12, minute 50"
/((\d\d):(\d\d))(..)/ =~ "12:50am"   # =>   0
"Time is #$1"                        # =>   "Time is 12:50"
"Hour is #$2, minute #$3"            # =>   "Hour is 12, minute 50"
"AM/PM is #$4"                       # =>   "AM/PM is am"
```

If you're using the MatchData object returned by the match method, you can index into it to get the corresponding subpatterns:

```
md = /(\d\d):(\d\d)(..)/.match("12:50am")
"Hour is #{md[1]}, minute #{md[2]}"   # =>   "Hour is 12, minute 50"
md = /((\d\d):(\d\d))(..)/.match("12:50am")
"Time is #{md[1]}"                    # =>   "Time is 12:50"
"Hour is #{md[2]}, minute #{md[3]}"   # =>   "Hour is 12, minute 50"
"AM/PM is #{md[4]}"                   # =>   "AM/PM is am"
```

The ability to use part of the current match later in that match allows you to look for various forms of repetition:

```
# match duplicated letter
show_regexp('He said "Hello"', /(\w)\1/)   # =>   He said "He->ll<-o"
# match duplicated substrings
show_regexp('Mississippi', /(\w+)\1/)      # =>   M->ississ<-ippi
```

Rather than use numbers, you can also use names to refer to previously matched content. You give a group a name by placing ?<*name*> immediately after the opening parenthesis. You can subsequently refer to this named group using \k<*name*> (or \k'*name*').

```
# match duplicated letter
str = 'He said "Hello"'
show_regexp(str, /(?<char>\w)\k<char>/)   # =>   He said "He->ll<-o"

# match duplicated adjacent substrings
str = 'Mississippi'
show_regexp(str, /(?<seq>\w+)\k<seq>/)    # =>   M->ississ<-ippi
```

The named matches in a regular expression are also available as local variables:[4]

```
/(?<hour>\d\d):(?<min>\d\d)(..)/ =~ "12:50am"   # =>   0
"Hour is #{hour}, minute #{min}"                # =>   "Hour is 12, minute
                                                       50"
```

Once you use named matches in a particular regular expression, Ruby no longer bothers to capture unnamed groups. Thus, in the previous example, you couldn't refer to the last group (which matches am) as $3.

Pattern-Based Substitution

We've already seen how sub and gsub replace the matched part of a string with other text. In those previous examples, the pattern was always fixed text, but the substitution methods work equally well if the pattern contains repetition, alternation, and grouping.

4. Note that this works only with literal regular expressions (so you can't, for example, assign a regular expression object to a variable, match the contents of that variable against a string, and expect the local variables to be set).

```
a = "quick brown fox"
a.sub(/[aeiou]/,  '*')   # =>   "q*ick brown fox"
a.gsub(/[aeiou]/, '*')   # =>   "q**ck br*wn f*x"
a.sub(/\s\S+/,  '')      # =>   "quick fox"
a.gsub(/\s\S+/, '')      # =>   "quick"
```

The substitution methods can take a string or a block. If a block is used, it is passed the matching substring, and the block's value is substituted into the original string.

```
a = "quick brown fox"
a.sub(/^./) {|match| match.upcase }         # =>   "Quick brown fox"
a.gsub(/[aeiou]/) {|vowel| vowel.upcase }   # =>   "qUIck brOwn fOx"
```

Maybe we want to normalize names entered by users into a web application. They may enter DAVE THOMAS, dave thomas, or dAvE tHoMas, and we'd like to store it as Dave Thomas. The following method is a simple first iteration. The pattern that matches the first character of a word is \b\w—look for a word boundary followed by a word character. Combine this with gsub, and we can hack the names:

```
def mixed_case(name)
  name.downcase.gsub(/\b\w/) {|first| first.upcase }
end

mixed_case("DAVE THOMAS")   # =>   "Dave Thomas"
mixed_case("dave thomas")   # =>   "Dave Thomas"
mixed_case("dAvE tHoMas")   # =>   "Dave Thomas"
```

1.9

There's an idiomatic way to write the substitution in Ruby 1.9, but we'll have to wait until Chapter 23 on page 379 to see why it works:

```
def mixed_case(name)
  name.downcase.gsub(/\b\w/, &:upcase)
end

mixed_case("dAvE tHoMas")   # =>   "Dave Thomas"
```

You can also give sub and gsub a hash as the replacement parameter, in which case they will look up matched groups and use the corresponding values as replacement text:

```
replacement = { "cat" => "feline", "dog" => "canine" }
replacement.default = "unknown"

"cat and dog".gsub(/\w+/, replacement)   # =>   "feline unknown canine"
```

Backslash Sequences in the Substitution

Earlier we noted that the sequences \1, \2, and so on, are available in the pattern, standing for the *n*th group matched so far. The same sequences can be used in the second argument of sub and gsub.

```
puts "fred:smith".sub(/(\w+):(\w+)/, '\2, \1')
puts "nercpyitno".gsub(/(.)(.)/, '\2\1')
```

produces:

```
smith, fred
encryption
```

You can also reference named groups:

```
puts "fred:smith".sub(/(?<first>\w+):(?<last>\w+)/, '\k<last>, \k<first>')
puts "nercpyitno".gsub(/(?<c1>.)(?<c2>.)/, '\k<c2>\k<c1>')
```

produces:

```
smith, fred
encryption
```

Additional backslash sequences work in substitution strings: \& (last match), \+ (last matched group), \` (string prior to match), \' (string after match), and \\ (a literal backslash).

It gets confusing if you want to include a literal backslash in a substitution. The obvious thing is to write this:

```
str.gsub(/\\/, '\\\\')
```

Clearly, this code is trying to replace each backslash in str with two. The programmer doubled up the backslashes in the replacement text, knowing that they'd be converted to \\ in syntax analysis. However, when the substitution occurs, the regular expression engine performs another pass through the string, converting \\ to \, so the net effect is to replace each single backslash with another single backslash. You need to write gsub(/\\/, '\\\\\\\\')!

```
str = 'a\b\c'                # =>   "a\b\c"
str.gsub(/\\/, '\\\\\\\\')   # =>   "a\\b\\c"
```

However, using the fact that \& is replaced by the matched string, you could also write this:

```
str = 'a\b\c'            # =>   "a\b\c"
str.gsub(/\\/, '\&\&')   # =>   "a\\b\\c"
```

If you use the block form of gsub, the string for substitution is analyzed only once (during the syntax pass), and the result is what you intended:

```
str = 'a\b\c'               # =>   "a\b\c"
str.gsub(/\\/) { '\\\\' }   # =>   "a\\b\\c"
```

At the start of this chapter, we said that it contained two emergency exits. The first was after we discussed basic matching and substitution. This is the second: you now know as much about regular expressions as the vast majority of Ruby developers. Feel free to break away and move on to the next chapter. But if you're feeling brave...

Advanced Regular Expressions

You may never need the information in the rest of this chapter. But, at the same time, knowing some of the real power in the Ruby regular expression implementation might just dig you out of a hole.

Regular Expression Extensions

Ruby uses the Oniguruma regular expression library. This offers a large number of extensions over traditional Unix regular expressions. Most of these extensions are written between the characters (? and). The parentheses that bracket these extensions are groups, but they do not necessarily generate backreferences. Some do not set the values of \1 and $1, and so on.

The sequence (?# *comment*) inserts a comment into the pattern. The content is ignored during pattern matching. As we'll see, commenting complex regular expressions can be as helpful as commenting complex code.

(?:*re*) makes *re* into a group without generating backreferences. This is often useful when you need to group a set of constructs but don't want the group to set the value of $1 or whatever. In the example that follows, both patterns match a date with either colons or slashes between the month, day, and year. The first form stores the separator character (which can be a slash or a colon) in $2 and $4, but the second pattern doesn't store the separator in an external variable.

```
date = "12/25/2008"

date =~ %r{(\d+)(/|:)(\d+)(/|:)(\d+)}
[$1,$2,$3,$4,$5]   # =>   ["12", "/", "25", "/", "2008"]

date =~ %r{(\d+)(?:/|:)(\d+)(?:/|:)(\d+)}
[$1,$2,$3]         # =>   ["12", "25", "2008"]
```

Lookahead and Lookbehind

You'll sometimes want to match a pattern only if the matched subsubtring is preceded by or followed by some other pattern. That is, you want to set some context for your match but don't want to capture that context as part of the match.

For example, you might want to match every word in a string that is followed by a comma, but you don't want the comma to form part of the match. Here you could use the charmingly named *zero-width positive lookahead* extension. (?=*re*) matches *re* at this point but does not consume it—you can look forward for the context of a match without affecting $&. In this example, we'll use scan to pick out the words:

```
str = "red, white, and blue"
str.scan(/[a-z]+(?=,)/)   # =>   ["red", "white"]
```

You can also match before the pattern using (?<=*re*) (*zero-width positive lookbehind*). This lets you look for characters that precede the context of a match without affecting $&. The following example matches the letters *dog* but only if they are preceded by the letters *hot*:

```
show_regexp("seadog hotdog", /(?<=hot)dog/)   # =>   seadog hot->dog<-
```

For the lookbehind extension, *re* either must be a fixed length or consist of a set of fixed length alternatives. That is, (?<=aa) and (?<=aa|bbb) are valid, but (?<=a+b) is not.

Both forms have negated versions, (?!*re*) and (?<!*re*), which are true if the context is not present in the target string.

Controlling Backtracking

Say you're given the problem of searching a string for a sequence of *X*s not followed by an *O*. You know that a string of *X*s can be represented as (X+), and you can use a lookahead to check that it isn't followed by an *O*, so you code up the pattern /(X+)(?!O)/. Let's try it:

```
re = /(X+)(?!O)/

# This one works
re =~ "test XXXY"   # =>   5
$1                  # =>   "XXX"

# But, unfortunately, so does this one
re =~ "test XXXO"   # =>   5
$1                  # =>   "XX"
```

Why did the second match succeed? Well, the regular expression engine saw the X+ in the pattern and happily gobbled up all the *X*s in the string. It then saw the pattern (?!O), saying that it should not now be looking at an *O*. Unfortunately, it is looking at an *O*, so the match doesn't succeed. But the engine doesn't give up. No sir! Instead it says, "Maybe I was wrong to consume every single *X* in the string. Let's try consuming one less and see what happens." This is called *backtracking*—when a match fails, the engine goes back and tries to match a different way. In this case, by backtracking past a single character, it now finds itself looking at the last *X* in the string (the one before the final *O*). And that *X* is not an *O*, so the negative lookahead succeeds, and the pattern matches. Look carefully at the output of the previous program: there are three *X*s in the first match but only two in the second.

But this wasn't the intent of our regexp. Once it finds a sequence of *X*s, those *X*s should be locked away. We don't want one of them being the terminator of the pattern. We can get that behavior by telling Ruby not to backtrack once it finds a string of *X*s. There are a couple of ways of doing this.

The sequence (?>*re*) nests an independent regular expression within the first regular expression. This expression is anchored at the current match position. If it consumes characters, these will no longer be available to the higher-level regular expression. This construct therefore inhibits backtracking.

Let's try it with our previous code:

```
re = /((?>X+))(?!O)/

# This one works
re =~ "test XXXY"         # =>   5
$1                        # =>   "XXX"

# Now this doesn't match
re =~ "test XXXO"         # =>   nil
$1                        # =>   nil

# And this finds the second string of Xs
re =~ "test XXXO XXXXY"   # =>   10
$1                        # =>   "XXXX"
```

You can also control backtracking by using a third form of repetition. We're already seen greedy repetition, such as *re*+, and lazy repetition, *re*+?. The third form is called *possessive*. You code it using a plus sign after the repetition character. It behaves just like greedy repetition, consuming as much of the string as it can. But once consumed, that part of the string can never be reexamined by the pattern—the regular expression engine can't backtrack past a possessive qualifier. This means we could also write our code as this:

```
re = /(X++)(?!O)/

re =~ "test XXXY"         # =>   5
$1                        # =>   "XXX"

re =~ "test XXXO"         # =>   nil
$1                        # =>   nil

re =~ "test XXXO XXXXY"   # =>   10
$1                        # =>   "XXXX"
```

Backreferences and Named Matches

Within a pattern, the sequences \n, \k'n', and \k<n> all refer to the n^{th} captured subpattern. Thus, the expression /(...)\1/ matches six characters with the first three characters being the same as the last three.

Rather than refer to matches by their number, you can give them names and then refer to those names. A subpattern is named using either of the syntaxes (?<name>...) or (?'name'...). You then refer to these named captures using either \k<name> or \k'name'.

For example, the following shows different ways of matching a time range (in the form hh:mm-hh:mm) where the hour part is the same:

```
same   = "12:15-12:45"
differ = "12:45-13:15"

# use numbered backreference
same   =~ /(\d\d):\d\d-\1:\d\d/   # =>   0
differ =~ /(\d\d):\d\d-\1:\d\d/   # =>   nil


# use named backreference
same   =~ /(?<hour>\d\d):\d\d-\k<hour>:\d\d/   # =>   0
differ =~ /(?<hour>\d\d):\d\d-\k<hour>:\d\d/   # =>   nil
```

Negative backreference numbers refer count backward from the place they're used, so they are relative, not absolute numbers. The following pattern matches four-letter palindromes:[5]

```
"abab" =~ /(.)(.)\k<-1>\k<-2>/   # =>   nil
"abba" =~ /(.)(.)\k<-1>\k<-2>/   # =>   0
```

5. These are words that read the same forward and backward.

You can invoke a named subpattern using \g<name> or \g<number>. Note that this reexecutes the match in the subpattern, in contrast to \k<name>, which matches whatever is matched by the subpattern:

```
re = /(?<color>red|green|blue) \w+ \g<color> \w+/

re =~ "red sun blue moon"    # =>  0
re =~ "red sun white moon"   # =>  nil
```

You can use \g recursively, invoking a subpattern within that pattern. The following code matches a string in which braces are properly nested:

```
re = /
  \A
    (?<brace_expression>
      {
        (
          [^{}]       # anything other than braces
        |
          \g<brace_expression>
        )*
      }
    )
  \Z
/x
```

We use the x option to allow us to write the expression with lots of space, which makes it easier to understand. We also indent it, just as we would indent Ruby code. And we can also use Ruby-style comments to document the tricky stuff. You can read this regular expression as follows: a brace expression is an open brace, then a sequence of zero or more characters or brace expressions, and then a closing brace.

Nested Groups

The ability to invoke subpatterns recursively means that backreferences can get tricky. Ruby solves this by allowing you to refer to a named or numbered group at a particular level of the recursion—simply (!) add a +n or -n to refer to the capture at the given level relative to the current level.

Here's an example from the Oniguruma cheat sheet. It matches palindromes:

```
/\A(?<a>|.|(?:(?<b>.)\g<a>\k<b+0>))\z/
```

That's pretty hard to read, so let's spread it out.

```
palindrome_matcher = /
\A
 (?<palindrome>
              # nothing, or
  |
    \w        # a single character, or
  |
    (?:       # x <palindrome> x
      (?<some_letter>\w)
      \g<palindrome>
      \k<some_letter+0>
    )
 )
\z
/x

palindrome_matcher.match "madam"   # =>   #<MatchData "madam"
                                          palindrome:"madam"
                                          some_letter:"a">
palindrome_matcher.match "m"       # =>   #<MatchData "m" palindrome:"m"
                                          some_letter:nil>
palindrome_matcher.match "adam"    # =>   nil
```

So, a palindrome is an empty string, a string containing a single character, or a character followed by a palindrome, followed by that same character. The notation \k<some_letter+0> means that the letter matched at the end of the inner palindrome will be the same letter that was at the start of it. Inside the nesting, however, a different letter may wrap the interior palindrome.

Named Subroutines

There's a trick that allows us to write subroutines inside regular expressions. Recall that we can invoke a named group using \g<name>, and we define the group using (?<name>...). Normally, the definition of the group is itself matched as part of executing the pattern. However, if you add the suffix {0} to the group, it means "zero matches of this group," so the group is not executed when first encountered:

```
sentence = %r{
  (?<subject>   cat   | dog   | gerbil    ){0}
  (?<verb>      eats  | drinks| generates ){0}
  (?<object>    water | bones | PDFs      ){0}
  (?<adjective> big   | small | smelly    ){0}
  (?<opt_adj>   (\g<adjective>\s)?        ){0}
  The\s\g<opt_adj>\g<subject>\s\g<verb>\s\g<opt_adj>\g<object>
}x
md = sentence.match("The cat drinks water")
puts "The subject is #{md[:subject]} and the verb is #{md[:verb]}"
md = sentence.match("The big dog eats smelly bones")
puts "The last adjective in the second sentence is #{md[:adjective]}"
sentence =~ "The gerbil generates big PDFs"
puts "And the object in the last sentence is #{$~[:object]}"
```

produces:

```
The subject is cat and the verb is drinks
The last adjective in the second sentence is smelly
And the object in the last sentence is PDFs
```

Setting Options

As we saw at the start of this chapter, you can add one or more of the options i (case insensitive), m (multiline), and x (allow spaces) to the end of a regular expression literal. You can also embed these options within the pattern itself.

(?imx)	Turns on the corresponding i, m, or x option. If used inside a group, the effect is limited to that group.
(?-imx)	Turns off the i, m, or x option.
(?imx:*re*)	Turns on the i, m, or x option for *re*.
(?-imx:*re*)	Turns off the i, m, or x option for *re*.

So, that's it. If you've made it this far, consider yourself a regular expression ninja. Get out there and match some strings.