SAMPLE CHAPTER

THE WELL-GROUNDED Rubyist

SECOND EDITION

David A. Black

MANNING

The Well-Grounded Rubyist, Second Edition

by David A. Black

Chapter 11

Copyright 2014 Manning Publications

brief contents

11 Regular expressions and regexp-based string operations

This chapter covers

- Regular expression syntax
- Pattern-matching operations
- The `MatchData` class
- Built-in methods based on pattern matching

In this chapter, we'll explore Ruby's facilities for pattern matching and text processing, centering around the use of regular expressions. A *regular expression* in Ruby serves the same purposes it does in other languages: it specifies a pattern of characters, a pattern that may or may not correctly predict (that is, match) a given string. Pattern-match operations are used for conditional branching (match/no match), pinpointing substrings (parts of a string that match parts of the pattern), and various text-filtering techniques.

Regular expressions in Ruby are objects. You send messages *to* a regular expression. Regular expressions add something to the Ruby landscape but, as objects, they also fit nicely into the landscape.

We'll start with an overview of regular expressions. From there, we'll move on to the details of how to write them and, of course, how to use them. In the latter category, we'll look at using regular expressions both in simple match operations and

in methods where they play a role in a larger process, such as filtering a collection or repeatedly scanning a string.

11.1 What are regular expressions?

Regular expressions appear in many programming languages, with minor differences among the incarnations. Their purpose is to specify character patterns that subsequently are determined to match (or not match) strings. Pattern matching, in turn, serves as the basis for operations like parsing log files, testing keyboard input for validity, and isolating substrings—operations, in other words, of frequent and considerable use to anyone who has to process strings and text.

Regular expressions have a weird reputation. Using them is a powerful, concentrated technique; they burn through a large subset of text-processing problems like acid through a padlock. They're also, in the view of many people (including people who understand them well), difficult to use, difficult to read, opaque, unmaintainable, and ultimately counterproductive.

You have to judge for yourself. The one thing you should *not* do is shy away from learning at least the basics of how regular expressions work and how to use the Ruby methods that utilize them. Even if you decide you aren't a "regular expression person," you need a reading knowledge of them. And you'll by no means be alone if you end up using them in your own programs more than you anticipated.

A number of Ruby built-in methods take regular expressions as arguments and perform selection or modification on one or more string objects. Regular expressions are used, for example, to *scan* a string for multiple occurrences of a pattern, to *substitute* a replacement string for a substring, and to *split* a string into multiple substrings based on a matching separator.

If you're familiar with regular expressions from Perl, sed, vi, Emacs, or any other source, you may want to skim or skip the expository material here and pick up in section 11.5, where we talk about Ruby methods that use regular expressions. But note that Ruby regular expressions aren't identical to those in any other language. You'll almost certainly be able to read them, but you may need to study the differences (such as whether parentheses are special by default or special when escaped) if you get into writing them.

Let's turn now to writing some regular expressions.

11.2 Writing regular expressions

Regular expressions are written with familiar characters—of course—but you have to learn to read and write them as things unto themselves. They're not strings, and their meaning isn't always as obvious as that of strings. They're representations of *patterns*.

11.2.1 Seeing patterns

A regular expression (regexp or regex) specifies a pattern. For every such pattern, every string in the world either matches the pattern or doesn't match it. The Ruby

methods that use regular expressions use them either to determine whether a given string matches a given pattern or to make that determination and also take some action based on the answer.

Patterns of the kind specified by regular expressions are most easily understood, initially, in plain language. Here are several examples of patterns expressed this way:

- The letter a, followed by a digit
- Any uppercase letter, followed by at least one lowercase letter
- Three digits, followed by a hyphen, followed by four digits

A pattern can also include components and constraints related to positioning inside the string:

- The beginning of a line, followed by one or more whitespace characters
- The character . (period) at the end of a string
- An uppercase letter at the beginning of a word

Pattern components like "the beginning of a line," which match a condition rather than a character in a string, are nonetheless expressed with characters or sequences of characters in the regexp.

Regular expressions provide a language for expressing patterns. Learning to write them consists principally of learning how various things are expressed inside a regexp. The most commonly applied rules of regexp construction are fairly easy to learn. You just have to remember that a regexp, although it contains characters, isn't a string. It's a special notation for expressing a pattern that may or may not correctly describe some or all of any given string.

11.2.2 Simple matching with literal regular expressions

Regular expressions are instances of the `Regexp` class, which is one of the Ruby classes that has a literal constructor for easy instantiation. The regexp literal constructor is a pair of forward slashes:

```
//
```

As odd as this may look, it really is a regexp, if a skeletal one. You can verify that it gives you an instance of the `Regexp` class in irb:

```
>> //.class
=> Regexp
```

The specifics of the regexp go between the slashes. We'll start to construct a few simple regular expressions as we look at the basics of the matching process.

Any pattern-matching operation has two main players: a regexp and a string. The regexp expresses predictions about the string. Either the string fulfills those predictions (matches the pattern) or it doesn't.

The simplest way to find out whether there's a match between a pattern and a string is with the `match` method. You can do this either direction—regexp objects

and string objects both respond to `match`, and both of these examples succeed and print `"Match!"`:

```
puts "Match!" if /abc/.match("The alphabet starts with abc.")
puts "Match!" if "The alphabet starts with abc.".match(/abc/)
```

The string version of `match` (the second line of the two) differs from the regexp version in that it converts a string argument *to* a regexp. (We'll return to that a little later.) In the example, the argument is already a regexp (`/abc/`), so no conversion is necessary.

In addition to the `match` method, Ruby also features a pattern-matching operator, `=~` (equal sign and tilde), which goes between a string and a regexp:

```
puts "Match!" if /abc/ =~ "The alphabet starts with abc."
puts "Match!" if "The alphabet starts with abc." =~ /abc/
```

As you might guess, this pattern-matching "operator" is an instance method of both the `String` and `Regexp` classes. It's one of the many Ruby methods that provide the syntactic sugar of an infix-operator usage style.

The `match` method and the `=~` operator are equally useful when you're after a simple yes/no answer to the question of whether there's a match between a string and a pattern. If there's no match, you get back `nil`. That's handy for conditionals; all four of the previous examples test the results of their match operations with an `if` test. Where `match` and `=~` differ from each other chiefly is in what they return when there *is* a match: `=~` returns the numerical index of the character in the string where the match started, whereas `match` returns an instance of the class `MatchData`:

```
>> "The alphabet starts with abc" =~ /abc/
=> 25
>> /abc/.match("The alphabet starts with abc.")
=> #<MatchData "abc">
```

The first example finds a match in position 25 of the string. In the second example, the creation of a `MatchData` object means that a match was found.

We'll examine `MatchData` objects a little further on. For the moment, we'll be concerned mainly with getting a yes/no answer to an attempted match, so any of the techniques shown thus far will work. For the sake of consistency, and because we'll be more concerned with `MatchData` objects than numerical indexes of substrings, most of the examples in this chapter will stick to the `Regexp#match` method.

Now, let's look in more detail at the composition of a regexp.

11.3 Building a pattern in a regular expression

When you write a regexp, you put the definition of your pattern between the forward slashes. Remember that what you're putting there isn't a string but a set of predictions and constraints that you want to look for in a string.

The possible components of a regexp include the following:

- *Literal characters*, meaning "match this character"
- The *dot wildcard character (.)*, meaning "match any character" (except \n, the newline character)
- *Character classes*, meaning "match one of these characters"

We'll discuss each of these in turn. We'll then use that knowledge to look more deeply at match operations.

11.3.1 Literal characters in patterns

Any literal character you put in a regexp matches *itself* in the string. Thus the regexp

```
/a/
```

matches any string containing the letter a.

Some characters have special meanings to the regexp parser (as you'll see in detail shortly). When you want to match one of these special characters as itself, you have to escape it with a backslash (\). For example, to match the character ? (question mark), you have to write this:

```
/\?/
```

The backslash means "don't treat the next character as special; treat it as itself."

The special characters include those listed between the parentheses here: (^ $? . / \ [ ] { } ( ) + *). Among them, as you can see, is the dot, which is a special character in regular expressions.

11.3.2 The dot wildcard character (.)

Sometimes you'll want to match *any character* at some point in your pattern. You do this with the special dot wildcard character (.). A dot matches any character with the exception of a newline. (There's a way to make it match newlines too, which you'll see a little later.)

The pattern in this regexp matches both "dejected" and "rejected":

```
/.ejected/
```

It also matches "%ejected" and "8ejected":

```
puts "Match!" if /.ejected/.match("%ejected")
```

The wildcard dot is handy, but sometimes it gives you more matches than you want. You can impose constraints on matches while still allowing for multiple possible strings, using character classes.

11.3.3 Character classes

A *character class* is an explicit list of characters placed inside the regexp in square brackets:

```
/[dr]ejected/
```

This means "match either *d* or *r*, followed by *ejected*." This new pattern matches either "dejected" or "rejected" but not "&ejected." A character class is a kind of partial or constrained wildcard: it allows for multiple possible characters, but only a limited number of them.

Inside a character class, you can also insert a *range* of characters. A common case is this, for lowercase letters:

```
/[a-z]/
```

To match a hexadecimal digit, you might use several ranges inside a character class:

```
/[A-Fa-f0-9]/
```

This matches any character *a* through *f* (upper- or lowercase) or any digit.

Character classes are longer than what they match

Even a short character class like `[a]` takes up more than one space in a regexp. But remember, each character class matches *one character* in the string. When you look at a character class like `/[dr]/`, it may look like it's going to match the substring `dr`. But it isn't: it's going to match either `d` or `r`.

Sometimes you need to match any character *except* those on a special list. You may, for example, be looking for the first character in a string that is *not* a valid hexadecimal digit.

You perform this kind of negative search by negating a character class. To do so, you put a caret (`^`) at the beginning of the class. For example, here's a character class that matches any character except a valid hexadecimal digit:

```
/[^A-Fa-f0-9]/
```

And here's how you might find the index of the first occurrence of a non-hex character in a string:

```
>> string = "ABC3934 is a hex number."
=> "ABC3934 is a hex number."
>> string =~ /[^A-Fa-f0-9]/
=> 7
```

A character class, positive or negative, can contain any characters. Some character classes are so common that they have special abbreviations.

Special escape sequences for common character classes

To match any digit, you can do this:

```
/[0-9]/
```

You can also accomplish the same thing more concisely with the special escape sequence `\d`:

```
/\d/
```

Notice that there are no square brackets here; it's just `\d`. Two other useful escape sequences for predefined character classes are these:

- `\w` matches any digit, alphabetical character, or underscore (_).
- `\s` matches any whitespace character (space, tab, newline).

Each of these predefined character classes also has a negated form. You can match any character that isn't a digit by doing this:

```
/\D/
```

Similarly, `\W` matches any character other than an alphanumeric character or underscore, and `\S` matches any non-whitespace character.

A successful call to `match` returns a `MatchData` object. Let's look at `MatchData` objects and their capabilities up close.

11.4 Matching, substring captures, and MatchData

So far, we've looked at basic match operations:

```
regex.match(string)
string.match(regex)
```

These are essentially true/false tests: either there's a match or there isn't. Now we'll examine what happens on successful and unsuccessful matches and what a match operation can do for you beyond the yes/no answer.

11.4.1 Capturing submatches with parentheses

One of the most important techniques of regexp construction is the use of parentheses to specify *captures*.

The idea is this. When you test for a match between a string—say, a line from a file—and a pattern, it's usually because you want to do something with the string or, more commonly, with part of the string. The capture notation allows you to isolate and save substrings of the string that match particular subpatterns.

For example, let's say we have a string containing information about a person:

```
Peel,Emma,Mrs.,talented amateur
```

From this string, we need to harvest the person's last name and title. We know the fields are comma separated, and we know what order they come in: last name, first name, title, occupation.

To construct a pattern that matches such a string, we might think in English along the following lines:

First `some alphabetical characters,`
then `a comma,`
then `some alphabetical characters,`
then `a comma,`
then `either 'Mr.' or 'Mrs.'`

We're keeping it simple: no hyphenated names, no doctors or professors, no leaving off the final period on Mr. and Mrs. (which would be done in British usage). The regexp, then, might look like this:

```
/[A-Za-z]+,[A-Za-z]+,Mrs?\./
```

(The question mark after the *s* means *match zero or one s*. Expressing it that way lets us match either "Mr." and "Mrs." concisely.) The pattern matches the string, as irb attests:

```
>> /[A-Za-z]+,[A-Za-z]+,Mrs?\./.match("Peel,Emma,Mrs.,talented amateur")
=> #<MatchData "Peel,Emma,Mrs.">
```

We got a `MatchData` object rather than `nil`; there was a match.

But now what? How do we isolate the substrings we're interested in (`"Peel"` and `"Mrs."`)?

This is where parenthetical groupings come in. We want two such groupings: one around the subpattern that matches the last name, and one around the subpattern that matches the title:

```
/([A-Za-z]+),[A-Za-z]+,(Mrs?\.)/
```

Now, when we perform the match

```
/([A-Za-z]+),[A-Za-z]+,(Mrs?\.)/.match("Peel,Emma,Mrs.,talented amateur")
```

two things happen:

- We get a `MatchData` object that gives us access to the submatches (discussed in a moment).
- Ruby automatically populates a series of variables for us, which also give us access to those submatches.

The variables that Ruby populates are global variables, and their names are based on numbers: `$1`, `$2`, and so forth. `$1` contains the substring matched by the subpattern inside the *first* set of parentheses from the left in the regexp. Examining `$1` after the previous match (for example, with `puts $1`) displays `Peel`. `$2` contains the substring matched by the *second* subpattern; and so forth. In general, the rule is this: after a successful match operation, the variable `$n` (where *n* is a number) contains the substring matched by the subpattern inside the *n*th set of parentheses from the left in the regexp.

NOTE If you've used Perl, you may have seen the variable `$0`, which represents not a specific captured subpattern but the entire substring that has been successfully matched. Ruby uses `$0` for something else: it contains the name of the Ruby program file from which the current program or script was initially started up. Instead of `$0` for pattern matches, Ruby provides a method; you call `string` on the `MatchData` object returned by the match. You'll see an example of the `string` method in section 11.4.2.

We can combine these techniques with string interpolation to generate a salutation for a letter, based on performing the match and grabbing the `$1` and `$2` variables:

```
line_from_file = "Peel,Emma,Mrs.,talented amateur"
/([A-Za-z]+),[A-Za-z]+,(Mrs?\.)/.match(line_from_file)
puts "Dear #{$2} #{$1},"
```

Output: Dear Mrs. Peel,

The `$n`-style variables are handy for grabbing submatches. But you can accomplish the same thing in a more structured, programmatic way by querying the `MatchData` object returned by your match operation.

11.4.2 Match success and failure

Every match operation either succeeds or fails. Let's start with the simpler case: failure. When you try to match a string to a pattern and the string doesn't match, the result is always `nil`:

```
>> /a/.match("b")
=> nil
```

Unlike `nil`, the `MatchData` object returned by a successful match has a Boolean value of true, which makes it handy for simple match/no-match tests. Beyond this, it also stores information about the match, which you can pry out with the appropriate methods: where the match began (at what character in the string), how much of the string it covered, what was captured in the parenthetical groups, and so forth.

To use the `MatchData` object, you must first save it. Consider an example where you want to pluck a phone number from a string and save the various parts of it (area code, exchange, number) in groupings. The following listing shows how you might do this. It's also written as a clinic on how to use some of `MatchData`'s more common methods.

Listing 11.1 Matching a phone number and querying the resulting `MatchData` object

```
string = "My phone number is (123) 555-1234."
phone_re = /\((\d{3})\)\s+(\d{3})-(\d{4})/
m = phone_re.match(string)
unless m
  puts "There was no match—sorry."
  exit
end
print "The whole string we started with: "
puts m.string                                          ❶
print "The entire part of the string that matched: "
puts m[0]                                              ❷
puts "The three captures: "
3.times do |index|                                     ❸
  puts "Capture ##{index + 1}: #{m.captures[index]}"
end
puts "Here's another way to get at the first capture:"
print "Capture #1: "
puts m[1]                                              ❹
```

Terminates program

In this code, we've used the `string` method of `MatchData` ❶, which returns the entire string on which the match operation was performed. To get the part of the string that matched our pattern, we address the `MatchData` object with square brackets, with an index of 0 ❷. We also use the nifty `times` method ❸ to iterate exactly three times through a code block and print out the submatches (the parenthetical captures) in succession. Inside that code block, a method called `captures` fishes out the substrings that matched the parenthesized parts of the pattern. Finally, we take another look at the first capture, this time through a different technique ❹: indexing the `MatchData` object directly with square brackets and positive integers, each integer corresponding to a capture.

Here's the output of the listing:

```
The whole string we started with: My phone number is (123) 555-1234.
The entire part of the string that matched: (123) 555-1234
The three captures:
Capture #1: 123
Capture #2: 555
Capture #3: 1234
Here's another way to get at the first capture:
Capture #1: 123
```

This gives you a taste of the kinds of match data you can extract from a `MatchData` object. You can see that there are two ways of retrieving captures. Let's zoom in on those techniques.

11.4.3 Two ways of getting the captures

One way to get the parenthetical captures from a `MatchData` object is by directly indexing the object, array-style:

```
m[1]
m[2]
#etc.
```

The first line will show the first capture (the first set of parentheses from the left), the second line will show the second capture, and so on.

As listing 11.1 shows, an index of 0 gives you the entire string that was matched. From 1 onward, an index of *n* gives you the *n*th capture, based on counting opening parentheses from the left. (And *n*, where $n > 0$, always corresponds to the number in the global `$n` variable.)

The other technique for getting the parenthetical captures from a `MatchData` object is the `captures` method, which returns all the captured substrings in a single array. Because this is a regular array, the first item in it—essentially, the same as the global variable `$1`—is item 0, not item 1. In other words, the following equivalencies apply:

```
m[1] == m.captures[0]
m[2] == m.captures[1]
```

and so forth.

A word about this recurrent "counting parentheses from the left" thing. Some regular expressions can be confusing as to their capture parentheses if you don't know the rule. Take this one, for example:

```
/((a)((b)c))/.match("abc")
```

What will be in the various captures? Well, just count opening parentheses from the left. For each opening parenthesis, find its counterpart on the right. Everything inside that pair will be capture number *n*, for whatever *n* you've gotten up to.

That means the first capture will be `"abc"`, because that's the part of the string that matches the pattern between the outermost parentheses. The next parentheses surround `"a"`; that will be the second capture. Next comes `"bc"`, followed by `"b"`. And that's the last of the opening parentheses.

The string representation of the `MatchData` object you get from this match will obligingly show you the captures:

```
>> /((a)((b)c))/.match("abc")
=> #<MatchData "abc" 1:"abc" 2:"a" 3:"bc" 4:"b">
```

Sure enough, they correspond rigorously to what was matched between the pairs of parentheses counted off from the left.

NAMED CAPTURES

Capturing subexpressions indexed by number is certainly handy, but there's another, sometimes more reader-friendly way, that you can label subexpressions: named captures.

Here's an example. This regular expression will match a name of the form "David A. Black" or, equally, "David Black" (with no middle initial):

```
>> re = /(?<first>\w+)\s+((?<middle>\w\.)\s+)?(?<last>\w+)/
```

What are the words `first`, `middle`, and `last` doing there? They're providing named captures: parenthetical captures that you can recover from the `MatchData` object using words instead of numbers.

If you perform a match using this regular expression, you'll see evidence of the named captures in the screen output representing the `MatchData` object:

```
>> m = re.match("David A. Black")
=> #<MatchData "David A. Black" first:"David" middle:"A." last:"Black">
```

Now you can query the object for its named captures:

```
>> m[:first]
=> "David"
```

Named captures can bulk up your regular expressions, but with the payback that the semantics of retrieving captures from the match become word-based rather than number-based, and therefore potentially clearer and more self-documenting. You also don't have to count pairs of parentheses to derive a reference to your captured substrings.

`MatchData` objects provide information beyond the parenthetical captures, information you can take and use if you need it.

11.4.4 Other MatchData information

The code in the following listing, which is designed to be grafted onto listing 11.1, gives some quick examples of several further `MatchData` methods.

Listing 11.2 Supplemental code for phone number–matching operations

```
print "The part of the string before the part that matched was: "
puts m.pre_match
print "The part of the string after the part that matched was: "
puts m.post_match
print "The second capture began at character "
puts m.begin(2)
print "The third capture ended at character "
puts m.end(3)
```

The output from this supplemental code is as follows:

```
The string up to the part that matched was: My phone number is
The string after the part that matched was: .
The second capture began at character 25
The third capture ended at character 33
```

The `pre_match` and `post_match` methods you see in this listing depend on the fact that when you successfully match a string, the string can then be thought of as being made up of three parts: the part before the part that matched the pattern; the part that matched the pattern; and the part after the part that matched the pattern. Any or all of these can be an empty string. In this listing, they're not: the `pre_match` and `post_match` strings both contain characters (albeit only one character in the case of `post_match`).

You can also see the `begin` and `end` methods in this listing. These methods tell you where the various parenthetical captures, if any, begin and end. To get the information for capture *n*, you provide *n* as the argument to `begin` and/or `end`.

The `MatchData` object is a kind of clearinghouse for information about what happened when the pattern met the string. With that knowledge in place, let's continue looking at techniques you can use to build and use regular expressions. We'll start with a fistful of important regexp components: quantifiers, anchors, and modifiers. Learning about these components will help you both with the writing of your own regular expressions and with your regexp literacy. If matching `/abc/` makes sense to you now, matching `/^x?[yz]{2}.*\z/i` will make sense to you shortly.

The global `MatchData` object `$~`

Whenever you perform a successful match operation, using either `match` or `=~`, Ruby sets the global variable `$~` to a `MatchData` object representing the match. On an unsuccessful match, `$~` gets set to `nil`. Thus you can always get at a `MatchData` object, for analytical purposes, even if you use `=~`.

11.5 Fine-tuning regular expressions with quantifiers, anchors, and modifiers

Quantifiers let you specify how many times in a row you want something to match. *Anchors* let you stipulate that the match occur at a certain structural point in a string (beginning of string, end of line, at a word boundary, and so on). *Modifiers* are like switches you can flip to change the behavior of the regexp engine; for example, by making it case-insensitive or altering how it handles whitespace.

We'll look at quantifiers, anchors, and modifiers here, in that order.

11.5.1 Constraining matches with quantifiers

Regexp syntax gives you ways to specify not only what you want but also how many: exactly one of a particular character, 5–10 repetitions of a subpattern, and so forth.

All the quantifiers operate either on a single character (which may be represented by a character class) or on a parenthetical group. When you specify that you want to match, say, three consecutive occurrences of a particular subpattern, that subpattern can thus be just one character, or it can be a longer subpattern placed inside parentheses.

ZERO OR ONE

You've already seen a zero-or-one quantifier example. Let's review it and go a little more deeply into it.

You want to match either "Mr" or "Mrs"—and, just to make it more interesting, you want to accommodate both the American versions, which end with periods, and the British versions, which don't. You might describe the pattern as follows:

```
the character M, followed by the character r, followed by
zero or one of the character s, followed by
zero or one of the character '.'
```

Regexp notation has a special character to represent the zero-or-one situation: the question mark (`?`). The pattern just described would be expressed in regexp notation as follows:

```
/Mrs?\.?/
```

The question mark after the *s* means that a string with an *s* in that position will match the pattern, and so will a string without an *s*. The same principle applies to the literal period (note the backslash, indicating that this is an actual period, not a special wildcard dot) followed by a question mark. The whole pattern, then, will match "Mr," "Mrs," "Mr.," or "Mrs." (It will also match "ABCMr." and "Mrs!," but you'll see how to delimit a match more precisely when we look at anchors in section 11.5.3.)

The question mark is often used with character classes to indicate zero or one of any of a number of characters. If you're looking for either one or two digits in a row, for example, you might express that part of your pattern like this:

```
\d\d?
```

This sequence will match "1," "55," "03," and so forth.

Along with the zero-or-one, there's a zero-or-more quantifier.

ZERO OR MORE

A fairly common case is one in which a string you want to match contains whitespace, but you're not sure how much. Let's say you're trying to match closing `</poem>` tags in an XML document. Such a tag may or may not contain whitespace. All of these are equivalent:

```
</poem>
< /poem>
</    poem>
</poem
>
```

In order to match the tag, you have to allow for unpredictable amounts of whitespace in your pattern—including none.

This is a case for the *zero-or-more* quantifier—the asterisk or star (`*`):

```
/<\s*\/\s*poem\s*>/
```

Each time it appears, the sequence `\s*` means the string being matched is allowed to contain zero or more whitespace characters at this point in the match. (Note the necessity of escaping the forward slash in the pattern with a backslash. Otherwise, it would be interpreted as the slash signaling the end of the regexp.)

Regular expressions, it should be noted, can't do everything. In particular, it's a commonplace and correct observation that you can't parse arbitrary XML with regular expressions, for reasons having to do with the nesting of elements and the ways in which character data is represented. Still, if you're scanning a document because you want to get a rough count of how many poems are in it, and you match and count poem tags, the likelihood that you'll get the information you're looking for is high.

Next among the quantifiers is one or more.

ONE OR MORE

The one-or-more quantifier is the plus sign (`+`) placed after the character or parenthetical grouping you wish to match one or more of. The match succeeds if the string contains at least one occurrence of the specified subpattern at the appropriate point. For example, the pattern

```
/\d+/
```

matches any sequence of one or more consecutive digits:

```
/\d+/.match("There's a digit here somewh3re...")
/\d+/.match("No digits here. Move along.")
/\d+/.match("Digits-R-Us 2345")
```

Succeeds

Fails

Succeeds

Of course, if you throw in parentheses, you can find out what got matched:

```
/(\d+)/.match("Digits-R-Us 2345")
puts $1
```

The output here is `2345`.

Here's a question, though. The job of the pattern `\d+` is to match one or more digits. That means as soon as the regexp engine (the part of the interpreter that's doing all this pattern matching) sees that the string has the digit 2 in it, it has enough information to conclude that yes, there's a match. Yet it clearly keeps going; it doesn't stop matching the string until it gets all the way to the 5. You can deduce this from the value of `$1`: the fact that `$1` is `2345` means that the subexpression `\d+`, which is what's in the first set of parentheses, is considered to have matched that substring of four digits.

But why match four digits when all you need to prove you're right is one digit? The answer, as it so often is in life as well as regexp analysis, is greed.

11.5.2 Greedy (and non-greedy) quantifiers

The `*` (zero-or-more) and `+` (one-or-more) quantifiers are *greedy*. This means they match as many characters as possible, consistent with allowing the rest of the pattern to match.

Look at what `.*` matches in this snippet:

```
string = "abc!def!ghi!"
match = /.+!/.match(string)
puts match[0]
```

Output:
abc!def!ghi!

We've asked for one or more characters (using the wildcard dot) followed by an exclamation point. You might expect to get back the substring `"abc!"`, which fits that description.

Instead, we get `"abc!def!ghi!"`. The `+` quantifier greedily eats up as much of the string as it can and only stops at the *last* exclamation point, not the first.

We can make `+` as well as `*` into non-greedy quantifiers by putting a question mark after them. Watch what happens when we do that with the last example:

```
string = "abc!def!ghi!"
match = /.+?!/.match(string)
puts match[0]
```

Output: abc!

This version says, "Give me one or more wildcard characters, but only as many as you see up to the *first* exclamation point, which should also be included." Sure enough, this time we get `"abc!"`.

If we add the question mark to the quantifier in the digits example, it will stop after it sees the 2:

```
/(\d+?)/.match("Digits-R-Us 2345")
puts $1
```

In this case, the output is `2`.

What does it mean to say that greedy quantifiers give you as many characters as they can, "consistent with allowing the rest of the pattern to match"?

Consider this match:

```
/\d+5/.match("Digits-R-Us 2345")
```

If the one-or-more quantifier's greediness were absolute, the `\d+` would match all four digits—and then the `5` in the pattern wouldn't match anything, so the whole match would fail. But greediness always subordinates itself to ensuring a successful match. What happens, in this case, is that after the match fails, the regexp engine backtracks: it unmatches the `5` and tries the pattern again. This time, it succeeds: it has satisfied both the `\d+` requirement (with `234`) and the requirement that `5` follow the digits that `\d+` matched.

Once again, you can get an informative X-ray of the proceedings by capturing parts of the matched string and examining what you've captured. Let's let irb and the `MatchData` object show us the relevant captures:

```
>> /(\d+)(5)/.match("Digits-R-Us 2345")
=> #<MatchData "2345" 1:"234" 2:"5">
```

The first capture is `"234"` and the second is `"5"`. The one-or-more quantifier, although greedy, has settled for getting only three digits, instead of four, in the interest of allowing the regexp engine to find a way to make the whole pattern match the string.

In addition to using the zero-/one-or-more-style modifiers, you can also require an exact number or number range of repetitions of a given subpattern.

SPECIFIC NUMBERS OF REPETITIONS

To specify exactly how many repetitions of a part of your pattern you want matched, put the number in curly braces ({}) right after the relevant subexpression, as this example shows:

```
/\d{3}-\d{4}/
```

This example matches exactly three digits, a hyphen, and then four digits: 555-1212 and other phone number–like sequences.

You can also specify a range inside the braces:

```
/\d{1,10}/
```

This example matches any string containing 1–10 consecutive digits. A single number followed by a comma is interpreted as a minimum (*n* or more repetitions). You can therefore match "three or more digits" like this:

```
/\d{3,}/
```

Ruby's regexp engine is smart enough to let you know if your range is impossible; you'll get a fatal error if you try to match, say, `{10,2}` (at least 10 but no more than 2) occurrences of a subpattern.

You can specify that a repetition count not only for single characters or character classes but also for any regexp *atom*—the more technical term for "part of your pattern." Atoms include parenthetical subpatterns and character classes as well as individual characters. Thus you can do this, to match five consecutive uppercase letters:

```
/([A-Z]){5}/.match("David BLACK")
```

But there's an important potential pitfall to be aware of in cases like this.

THE LIMITATION ON PARENTHESES

If you run that last line of code and look at what the `MatchData` object tells you about the first capture, you may expect to see `"BLACK"`. But you don't:

```
>> /([A-Z]){5}/.match("David BLACK")
=> #<MatchData "BLACK" 1:"K">
```

It's just `"K"`. Why isn't `"BLACK"` captured in its entirety?

The reason is that the parentheses don't "know" that they're being repeated five times. They just know that they're the first parentheses from the left (in this particular case) and that what they've captured should be stashed in the first capture slot (`$1`, or `captures[1]` of the `MatchData` object). The expression inside the parentheses, `[A-Z]`, can only match one character. If it matches one character five times in a row, it's still only matched one at a time—and it will only "remember" the last one.

In other words, matching one character five times isn't the same as matching five characters one time.

If you want to capture all five characters, you need to move the parentheses so they enclose the entire five-part match:

```
>> /([A-Z]{5})/.match("David BLACK")
=> #<MatchData "BLACK" 1:"BLACK">
```

Be careful and literal-minded when it comes to figuring out what will be captured.

We'll look next at ways in which you can specify conditions under which you want matches to occur, rather than the content you expect the string to have.

11.5.3 Regular expression anchors and assertions

Assertions and anchors are different types of creatures from characters. When you match a character (even based on a character class or wildcard), you're said to be *consuming* a character in the string you're matching. An assertion or an anchor, on the other hand, doesn't consume any characters. Instead, it expresses a *constraint*: a condition that must be met before the matching of characters is allowed to proceed.

The most common anchors are *beginning of line* (`^`) and *end of line* (`$`). You might use the beginning-of-line anchor for a task like removing all the comment lines from a Ruby program file. You'd accomplish this by going through all the lines in the file and printing out only those that did *not* start with a hash mark (`#`) or with whitespace followed by a hash mark. To determine which lines are comment lines, you can use this regexp:

```
/^\s*#/
```

The `^` (caret) in this pattern *anchors* the match at the beginning of a line. If the rest of the pattern matches, but *not* at the beginning of the line, that doesn't count—as you can see with a couple of tests:

```
>> comment_regexp = /^\s*#/
=> /^\s*#/
>> comment_regexp.match("  # Pure comment!")
```

```
=> #<MatchData "  #">
>> comment_regexp.match("  x = 1  # Code plus comment!")
=> nil
```

Only the line that starts with some whitespace and the hash character is a match for the comment pattern. The other line doesn't match the pattern and therefore wouldn't be deleted if you used this regexp to filter comments out of a file.

Table 11.1 shows a number of anchors, including start and end of line and start and end of string.

Table 11.1 Regular expression anchors

Notation	Description	Example	Sample matching string
^	Beginning of line	/^\s*#/	"# A Ruby comment line with leading spaces"
$	End of line	/\.$/	"one\ntwo\nthree.\nfour"
\A	Beginning of string	/\AFour score/	"Four score"
\z	End of string	/from the earth.\z/	"from the earth."
\Z	End of string (except for final newline)	/from the earth.\Z/	"from the earth\n"
\b	Word boundary	/\b\w+\b/	"!!!word***" (matches "word")

Note that \z matches the absolute end of the string, whereas \Z matches the end of the string except for an optional trailing newline. \Z is useful in cases where you're not sure whether your string has a newline character at the end—perhaps the last line read out of a text file—and you don't want to have to worry about it.

Hand-in-hand with anchors go *assertions*, which, similarly, tell the regexp processor that you want a match to count only under certain conditions.

LOOKAHEAD ASSERTIONS

Let's say you want to match a sequence of numbers only if it ends with a period. But you don't want the period itself to count as part of the match.

One way to do this is with a *lookahead assertion*—or, to be complete, a zero-width, positive lookahead assertion. Here, followed by further explanation, is how you do it:

```
str = "123 456. 789"
m = /\d+(?=\.)/.match(str)
```

At this point, m[0] (representing the entire stretch of the string that the pattern matched) contains 456—the one sequence of numbers that's followed by a period.

Here's a little more commentary on some of the terminology:

- *Zero-width* means it doesn't consume any characters in the string. The presence of the period is noted, but you can still match the period if your pattern continues.

- *Positive* means you want to stipulate that the period be present. There are also *negative* lookaheads; they use `(?!...)` rather than `(?=...)`.
- *Lookahead assertion* means you want to know that you're specifying what *would* be next, without matching it.

When you use a lookahead assertion, the parentheses in which you place the lookahead part of the match don't count; `$1` won't be set by the match operation in the example. And the dot after the 6 won't be consumed by the match. (Keep this last point in mind if you're ever puzzled by lookahead behavior; the puzzlement often comes from forgetting that looking ahead isn't the same as moving ahead.)

LOOKBEHIND ASSERTIONS

The lookahead assertions have lookbehind equivalents. Here's a regexp that matches the string `BLACK` only when it's preceded by "David ":

```
re = /(?<=David )BLACK/
```

Conversely, here's one that matches it only when it isn't preceded by "David ":

```
re = /(?<!David )BLACK/
```

Once again, keep in mind that these are zero-width assertions. They represent constraints on the string (*"David " has to be before it, or this "BLACK" doesn't count as a match*), but they don't match or consume any characters.

Non-capturing parentheses

If you want to match something—not just assert that it's next, but actually match it—using parentheses, but you don't want it to count as one of the numbered parenthetical captures resulting from the match, use the `(?:...)` construct. Anything inside a `(?:)` grouping will be matched based on the grouping, but not saved to a capture. Note that the `MatchData` object resulting from the following match only has two captures; the `def` grouping doesn't count, because of the `?:` notation:

```
>> str = "abc def ghi"
=> "abc def ghi"
>> m = /(abc) (?:def) (ghi)/.match(str)
=> #<MatchData "abc def ghi" 1:"abc" 2:"ghi">
```

Unlike a zero-width assertion, a `(?:)` group does consume characters. It just doesn't save them as a capture.

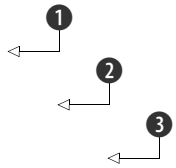
There's also such a thing as a *conditional match*.

CONDITIONAL MATCHES

While it probably won't be among your everyday regular expression practices, it's interesting to note the existence of conditional matches in Ruby 2.0's regular expression engine (project name Onigmo). A conditional match tests for a particular capture (by number or name), and matches one of two subexpressions based on whether or not the capture was found.

Here's a simple example. The conditional expression `(?(1)b|c)` matches `b` if capture number `1` is matched; otherwise it matches `c`:

```
>> re = /(a)?(?(1)b|c)/
=> /(a)?(?(1)b|c)/
>> re.match("ab")
=> #<MatchData "ab" 1:"a">
>> re.match("b")
=> nil
>> re.match("c")
=> #<MatchData "c" 1:nil>
```



The regular expression `re` matches the string `"ab"` ❶, with `"a"` as the first parenthetical capture and the conditional subexpression matching `"a"`. However, `re` doesn't match the string `"b"` ❷. Because there's no first parenthetical capture, the conditional subexpression tries to match `"c"`, and fails ❸. That's also why `re` *does* match the string `"c"`: the condition `(?(1)…)` isn't met, so the expression tries to match the "else" part of itself, which is the subexpression `/c/`.

You can also write conditional regular expressions using named captures. The preceding example would look like this:

```
/(?<first>a)?(?(<first>)b|c)/
```

and the results of the various matches would be the same.

Anchors, assertions, and conditional matches add richness and granularity to the pattern language with which you express the matches you're looking for. Also in the language-enrichment category are regexp modifiers.

11.5.4 Modifiers

A regexp *modifier* is a letter placed after the final, closing forward slash of the regex literal:

```
/abc/i
```

The `i` modifier shown here causes match operations involving this regexp to be case-insensitive. The other most common modifier is `m`. The `m` (multiline) modifier has the effect that the wildcard dot character, which normally matches *any character except newline*, will match *any character, including newline*. This is useful when you want to capture everything that lies between, say, an opening parenthesis and a closing one, and you don't know (or care) whether they're on the same line.

Here's an example; note the embedded newline characters (`\n`) in the string:

```
str = "This (including\nwhat's in parens\n) takes up three lines."
m = /\(.*?\)/m.match(str)
```

The non-greedy wildcard subpattern `.*?` matches:

```
(including\nwhat's in parens\n)
```

Without the `m` modifier, the dot in the subpattern wouldn't match the newline characters. The match operation would hit the first newline and, not finding a `)` character by that point, would fail.

Another often-used regexp modifier is `x`. The `x` modifier changes the way the regexp parser treats whitespace. Instead of including it literally in the pattern, it ignores it unless it's escaped with a backslash. The point of the `x` modifier is to let you add comments to your regular expressions:

```
/
 \((\d{3})\)  # 3 digits inside literal parens (area code)
   \s         # One space character
 (\d{3})      # 3 digits (exchange)
    -         # Hyphen
 (\d{4})      # 4 digits (second part of number
/x
```

The previous regexp is exactly the same as this one but with expanded syntax and comments:

```
/\((\d{3})\)\s(\d{3})-(\d{4})/
```

Be careful with the `x` modifier. When you first discover it, it's tempting to bust all your patterns wide open:

```
/  (?<=   David\ )    BLACK  /x
```

(Note the backslash-escaped literal space character, the only such character that will be considered part of the pattern.) But remember that a lot of programmers have trained themselves to understand regular expressions without a lot of ostensibly user-friendly extra whitespace thrown in. It's not easy to un-x a regexp as you read it, if you're used to the standard syntax.

For the most part, the `x` modifier is best saved for cases where you want to break the regexp out onto multiple lines for the sake of adding comments, as in the telephone number example. Don't assume that whitespace automatically makes regular expressions more readable.

We'll look next at techniques for converting back and forth between two different but closely connected classes: `String` and `Regexp`.

11.6 Converting strings and regular expressions to each other

The fact that regular expressions aren't strings is easy to absorb at a glance in the case of regular expressions like this:

```
/[a-c]{3}/
```

With its special character-class and repetition syntax, this pattern doesn't look much like any of the strings it matches (`"aaa"`, `"aab"`, `"aac"`, and so forth).

It gets a little harder *not* to see a direct link between a regexp and a string when faced with a regexp like this:

```
/abc/
```

This regexp isn't the string `"abc"`. Moreover, it matches not only `"abc"` but any string with the substring `"abc"` somewhere inside it (like "Now I know my abcs."). There's no unique relationship between a string and a similar-looking regexp.

Still, although the visual resemblance between some strings and some regular expressions doesn't mean they're the same thing, regular expressions and strings do interact in important ways. Let's look at some flow in the string-to-regexp direction and then some going the opposite way.

11.6.1 String-to-regexp idioms

To begin with, you can perform string (or string-style) interpolation inside a regexp. You do so with the familiar `#{...}` interpolation technique:

```
>> str = "def"
=> "def"
>> /abc#{str}/
=> /abcdef/
```

The value of `str` is dropped into the regexp and made part of it, just as it would be if you were using the same technique to interpolate it into a string.

The interpolation technique becomes more complicated when the string you're interpolating contains regexp special characters. For example, consider a string containing a period (`.`). As you know, the period or dot has a special meaning in regular expressions: it matches any single character except newline. In a string, it's just a dot. When it comes to interpolating strings into regular expressions, this has the potential to cause confusion:

```
>> str = "a.c"
=> "a.c"
>> re = /#{str}/
=> /a.c/
>> re.match("a.c")
=> #<MatchData "a.c">
>> re.match("abc")
=> #<MatchData "abc">
```

Both matches succeed; they return `MatchData` objects rather than `nil`. The dot in the pattern matches a dot in the string `"a.c"`. But it also matches the b in `"abc"`. The dot, which started life as just a dot inside `str`, takes on special meaning when it becomes part of the regexp.

But you can *escape* the special characters inside a string before you drop the string into a regexp. You don't have to do this manually: the `Regexp` class provides a `Regexp.escape` class method that does it for you. You can see what this method does by running it on a couple of strings in isolation:

```
>> Regexp.escape("a.c")
=> "a\\.c"
>> Regexp.escape("^abc")
=> "\\^abc"
```

(irb doubles the backslashes because it's outputting double-quoted strings. If you wish, you can `puts` the expressions, and you'll see them in their real form with single backslashes.)

As a result of this kind of escaping, you can constrain your regular expressions to match exactly the strings you interpolate into them:

```
>> str = "a.c"
=> "a.c"
>> re = /#{Regexp.escape(str)}/
=> /a\.c/
>> re.match("a.c")
=> #<MatchData "a.c">
>> re.match("abc")
=> nil
```

This time, the attempt to use the dot as a wildcard match character fails; `"abc"` isn't a match for the escaped, interpolated string.

It's also possible to instantiate a regexp from a string by passing the string to `Regexp.new`:

```
>> Regexp.new('(.*)\s+Black')
=> /(.*)\s+Black/
```

The usual character-escaping and/or regexp-escaping logic applies:

```
>> Regexp.new('Mr\. David Black')
=> /Mr\. David Black/
>> Regexp.new(Regexp.escape("Mr. David Black"))
=> /Mr\.\ David\ Black/
```

Notice that the literal space characters have been escaped with backslashes—not strictly necessary unless you're using the `x` modifier, but not detrimental either.

You can also pass a literal regexp to `Regexp.new`, in which case you get back a new, identical regexp. Because you can always just use the literal regexp in the first place, `Regexp.new` is more commonly used to convert strings to regexps.

The use of single-quoted strings makes it unnecessary to double up on the backslashes. If you use double quotes (which you may have to, depending on what sorts of interpolation you need to do), remember that you need to write `Mr\\.` so the backslash is part of the string passed to the regexp constructor. Otherwise, it will only have the effect of placing a literal dot in the string—which was going to happen anyway—and that dot will make it into the regexp without a slash and will therefore be interpreted as a wildcard dot.

Now let's look at some conversion techniques in the other direction: regexp to string. This is something you'll do mostly for debugging and analysis purposes.

11.6.2 Going from a regular expression to a string

Like all Ruby objects, regular expressions can represent themselves in string form. The way they do this may look odd at first:

```
>> puts /abc/
(?-mix:abc)
```

This is an alternate regexp notation—one that rarely sees the light of day except when generated by the `to_s` instance method of regexp objects. What looks like *mix* is a list of modifiers (`m`, `i`, and `x`) with a minus sign in front indicating that the modifiers are all switched off.

You can play with `puts`ing regular expressions in irb, and you'll see more about how this notation works. We won't pursue it here, in part because there's another way to get a string representation of a regexp that looks more like what you probably typed—by calling `inspect` or `p` (which in turn calls `inspect`):

```
>> /abc/.inspect
=> "/abc/"
```

Going from regular expressions to strings is useful primarily when you're studying and/or troubleshooting regular expressions. It's a good way to make sure your regular expressions are what you think they are.

At this point, we'll bring regular expressions full circle by examining the roles they play in some important methods of other classes. We've gotten this far using the `match` method almost exclusively; but `match` is just the beginning.

11.7 Common methods that use regular expressions

The payoff for gaining facility with regular expressions in Ruby is the ability to use the methods that take regular expressions as arguments and do something with them.

To begin with, you can always use a `match` operation as a test in, say, a `find` or `find_all` operation on a collection. For example, to find all strings longer than 10 characters and containing at least 1 digit, from an array of strings called `array`, you can do this:

```
array.find_all {|e| e.size > 10 and /\d/.match(e) }
```

But a number of methods, mostly pertaining to strings, are based more directly on the use of regular expressions. We'll look at several of them in this section.

11.7.1 String#scan

The `scan` method goes from left to right through a string, testing repeatedly for a match with the pattern you specify. The results are returned in an array.

For example, if you want to harvest all the digits in a string, you can do this:

```
>> "testing 1 2 3 testing 4 5 6".scan(/\d/)
=> ["1", "2", "3", "4", "5", "6"]
```

Note that `scan` jumps over things that don't match its pattern and looks for a match later in the string. This behavior is different from that of `match`, which stops for good when it finishes matching the pattern completely once.

If you use parenthetical groupings in the regexp you give to `scan`, the operation returns an array of arrays. Each inner array contains the results of one scan through the string:

```
>> str = "Leopold Auer was the teacher of Jascha Heifetz."
=> "Leopold Auer was the teacher of Jascha Heifetz."
>> violinists = str.scan(/([A-Z]\w+)\s+([A-Z]\w+)/)
=> [["Leopold", "Auer"], ["Jascha", "Heifetz"]]
```

This example nets you an array of arrays, where each inner array contains the first name and the last name of a person. Having each complete name stored in its own array makes it easy to iterate over the whole list of names, which we've conveniently stashed in the variable `violinists`:

```
violinists.each do |fname,lname|
  puts "#{lname}'s first name was #{fname}."
end
```

The output from this snippet is as follows:

```
Auer's first name was Leopold.
Heifetz's first name was Jascha.
```

The regexp used for names in this example is, of course, overly simple: it neglects hyphens, middle names, and so forth. But it's a good illustration of how to use captures with `scan`.

`String#scan` can also take a code block—and that technique can, at times, save you a step. `scan` yields its results to the block, and the details of the yielding depend on whether you're using parenthetical captures. Here's a scan-block-based rewrite of the previous code:

```
str.scan(/([A-Z]\w+)\s+([A-Z]\w+)/) do |fname, lname|
  puts "#{lname}'s first name was #{fname}."
end
```

Each time through the string, the block receives the captures in an array. If you're not doing any capturing, the block receives the matched substrings successively. Scanning for clumps of `\w` characters (`\w` is the character class consisting of letters, numbers, and underscore) might look like this

```
"one two three".scan(/\w+/) {|n| puts "Next number: #{n}" }
```

which would produce this output:

```
Next number: one
Next number: two
Next number: three
```

Note that if you provide a block, `scan` doesn't store the results up an array and return them; it sends each result to the block and then discards it. That way, you can scan through long strings, doing something with the results along the way, and avoid taking up memory with the substrings you've already seen and used.

Another common regexp-based string operation is `split`.

Even more string scanning with the `StringScanner` class

The standard library includes an extension called `strscan`, which provides the `StringScanner` class. `StringScanner` objects extend the available toolkit for scanning and examining strings. A `StringScanner` object maintains a pointer into the string, allowing for back-and-forth movement through the string using position and pointer semantics.

Here are some examples of the methods in `StringScanner`:

```
>> require 'strscan'
=> true
>> ss = StringScanner.new("Testing string scanning")
=> #<StringScanner 0/23 @ "Testi...">
>> ss.scan_until(/ing/)
=> "Testing"
>> ss.pos
=> 7
>> ss.peek(7)
=> " string"
>> ss.unscan
=> #<StringScanner 0/23 @ "Testi...">
>> ss.pos
=> 0
>> ss.skip(/Test/)
=> 4
>> ss.rest
=> "ing string scanning"
```

Loads scanner library

Creates scanner

Scans string until regexp matches

Examines new pointer position

Looks at next 7 bytes (but doesn't advance pointer)

Undoes previous scan

Moves pointer past regexp

Examines part of string to right of pointer

Using the notion of a pointer into the string, `StringScanner` lets you traverse across the string as well as examine what's already been matched and what remains. `StringScanner` is a useful complement to the built-in string scanning facilities.

11.7.2 String#split

In keeping with its name, `split` splits a string into multiple substrings, returning those substrings as an array. `split` can take either a regexp or a plain string as the separator for the split operation. It's commonly used to get an array consisting of all the characters in a string. To do this, you use an empty regexp:

```
>> "Ruby".split(//)
=> ["R", "u", "b", "y"]
```

`split` is often used in the course of converting flat, text-based configuration files to Ruby data structures. Typically, this involves going through a file line by line and converting each line. A single-line conversion might look like this:

```
line = "first_name=david;last_name=black;country=usa"
record = line.split(/=|;/)
```

This leaves `record` containing an array:

```
["first_name", "david", "last_name", "black", "country", "usa"]
```

With a little more work, you can populate a hash with entries of this kind:

```
data = []
record = Hash[*line.split(/=|;/)]
data.push(record)
```

Use * to turn array into bare list to feed to Hash[ ]

If you do this for every line in a file, you'll have an array of hashes representing all the records. That array of hashes, in turn, can be used as the pivot point to a further operation—perhaps embedding the information in a report or feeding it to a library routine that can save it to a database table as a sequence of column/value pairs.

You can provide a second argument to `split`; this argument limits the number of items returned. In this example

```
>> "a,b,c,d,e".split(/,/,3)
=> ["a", "b", "c,d,e"]
```

`split` stops splitting once it has three elements to return and puts everything that's left (commas and all) in the third string.

In addition to breaking a string into parts by scanning and splitting, you can also change parts of a string with substitution operations, as you'll see next.

11.7.3 sub/sub! and gsub/gsub!

`sub` and `gsub` (along with their bang, in-place equivalents) are the most common tools for changing the contents of strings in Ruby. The difference between them is that `gsub` (*g*lobal *sub*stitution) makes changes throughout a string, whereas `sub` makes at most one substitution.

SINGLE SUBSTITUTIONS WITH SUB

`sub` takes two arguments: a regexp (or string) and a replacement string. Whatever part of the string matches the regexp, if any, is removed from the string and replaced with the replacement string:

```
>> "typigraphical error".sub(/i/,"o")
=> "typographical error"
```

You can use a code block *instead of* the replacement-string argument. The block is called (yielded to) if there's a match. The call passes in the string being replaced as an argument:

```
>> "capitalize the first vowel".sub(/[aeiou]/) {|s| s.upcase }
=> "cApitalize the first vowel"
```

If you've done any parenthetical grouping, the global `$n` variables are set and available for use inside the block.

GLOBAL SUBSTITUTIONS WITH GSUB

`gsub` is like `sub`, except it keeps substituting as long as the pattern matches anywhere in the string. For example, here's how you can replace the first letter of every word in a string with the corresponding capital letter:

```
>> "capitalize every word".gsub(/\b\w/) {|s| s.upcase }
=> "Capitalize Every Word"
```

As with `sub`, `gsub` gives you access to the `$n` parenthetical capture variables in the code block.

USING THE CAPTURES IN A REPLACEMENT STRING

You can access the parenthetical captures by using a special notation consisting of backslash-escaped numbers. For example, you can correct an occurrence of a lowercase letter followed by an uppercase letter (assuming you're dealing with a situation where this is a mistake) like this:

```
>> "aDvid".sub(/([a-z])([A-Z])/, '\2\1')
=> "David"
```

Note the use of single quotation marks for the replacement string. With double quotes, you'd have to double the backslashes to escape the backslash character.

To double every word in a string, you can do something similar, but using `gsub`:

```
>> "double every word".gsub(/\b(\w+)/, '\1 \1')
=> "double double every every word word"
```

A global capture variable pitfall

Beware: You can use the global capture variables (`$1`, etc.) in your substitution string, but they may not do what you think they will. Specifically, you'll be vulnerable to leftover values for those variables. Consider this example:

```
>> /(abc)/.match("abc")
=> #<MatchData "abc" 1:"abc">
>> "aDvid".sub(/([a-z])([A-Z])/, "#{$2}#{$1}")
=> "abcvid"
```

Here, `$1` from the previous match (`"abc"`) ended up infiltrating the substitution string in the second match. In general, sticking to the `\1`-style references to your captures is safer than using the global capture variables in `sub` and `gsub` substitution strings.

We'll conclude our look at regexp-based tools with two techniques having in common their dependence on the case equality operator (`===`): case statements (which aren't method calls but which do incorporate calls to the threequal operator) and `Enumerable#grep`.

11.7.4 Case equality and grep

As you know, all Ruby objects understand the `===` message. If it hasn't been overridden in a given class or for a given object, it's a synonym for `==`. If it has been overridden, it's whatever the new version makes it be.

Case equality for regular expressions is a match test: for any given *regexp* and *string*, *regexp* `===` *string* is true if *string* matches *regexp*. You can use `===` explicitly as a match test:

```
puts "Match!" if re.match(string)
puts "Match!" if string =~ re
puts "Match!" if re === string
```

And, of course, you have to use whichever test will give you what you need: `nil` or `MatchData` object for `match`; `nil` or integer offset for `=~`; `true` or `false` for `===`.

In case statements, `===` is used implicitly. To test for various pattern matches in a case statement, proceed along the following lines:

```
print "Continue? (y/n) "
answer = gets
case answer
when /^y/i
  puts "Great!"
when /^n/i
  puts "Bye!"
  exit
else
  puts "Huh?"
end
```

Each `when` clause is a call to `===`: `/^y/i === answer`, and so forth.

The other technique you've seen that uses the `===` method/operator, also implicitly, is `Enumerable#grep`. You can refer back to section 10.3.3. Here, we'll put the spotlight on a couple of aspects of how it handles strings and regular expressions.

`grep` does a filtering operation from an enumerable object based on the case equality operator (`===`), returning all the elements in the enumerable that return a true value when threequaled against `grep`'s argument. Thus if the argument to `grep` is a regexp, the selection is based on pattern matches, as per the behavior of `Regexp#===`:

```
>> ["USA", "UK", "France", "Germany"].grep(/[a-z]/)
=> ["France", "Germany"]
```

You can accomplish the same thing with `select`, but it's a bit wordier:

```
["USA", "UK", "France", "Germany"].select {|c| /[a-z]/ === c }
```

`grep` uses the generalized threequal technique to make specialized `select` operations, including but not limited to those involving strings, concise and convenient.

You can also supply a code block to `grep`, in which case you get a combined `select/map` operation: the results of the filtering operation are yielded one at a time to the block, and the return value of the whole `grep` call is the cumulative result of those yields. For example, to select countries and then collect them in uppercase, you can do this:

```
>> ["USA", "UK", "France", "Germany"].grep(/[a-z]/) {|c| c.upcase }
=> ["FRANCE", "GERMANY"]
```

Keep in mind that `grep` selects based on the case equality operator (`===`), so it won't select anything other than strings when you give it a regexp as an argument—and there's no automatic conversion between numbers and strings. Thus if you try this

```
[1,2,3].grep(/1/)
```

you get back an empty array; the array has no string element that matches the regexp `/1/`, no element for which it's true that `/1/ === element`.

This brings us to the end of our survey of regular expressions and some of the methods that use them. There's more to learn; pattern matching is a sprawling subject. But this chapter has introduced you to much of what you're likely to need and see as you proceed with your study and use of Ruby.

11.8 Summary

In this chapter you've seen

- The underlying principles behind regular expression pattern matching
- The `match` and `=~` techniques
- Character classes
- Parenthetical captures
- Quantifiers
- Anchors
- `MatchData` objects
- String/regexp interpolation and conversion
- Ruby methods that use regexps: `scan`, `split`, `grep`, `sub`, `gsub`

This chapter has introduced you to the fundamentals of regular expressions in Ruby, including character classes, parenthetical captures, and anchors. You've seen that regular expressions are objects—specifically, objects of the `Regexp` class—and that they respond to messages (such as "match"). We looked at the `MatchData` class, instances of which hold information about the results of a `match` operation. You've also learned how to interpolate strings into regular expressions (escaped or unescaped, depending on whether you want the special characters in the string to be treated as special in the regexp), how to instantiate a regexp from a string, and how to generate a string representation of a regexp.

Methods like `String#scan`, `String#split`, `Enumerable#grep`, and the "sub" family of `String` methods use regular expressions and pattern matching as a way of determining how their actions should be applied. Gaining knowledge about regular expressions gives you access not only to relatively simple matching methods but also to a suite of string-handling tools that otherwise wouldn't be usable.

As we continue our investigation of Ruby's built-in facilities, we'll move in chapter 12 to the subject of I/O operations in general and file handling in particular.

RUBY

THE WELL-GROUNDED Rubyist Second Edition

David A. Black

This is a good time for Ruby! It's powerful like Java or C++, and has dynamic features that let your code react gracefully to changes at runtime. And it's elegant, so creating applications, development tools, and administrative scripts is easier and more straightforward. With the long-awaited Ruby 2, an active development community, and countless libraries and productivity tools, Ruby has come into its own.

The Well-Grounded Rubyist, Second Edition is a beautifully written tutorial that begins with your first Ruby program and goes on to explore sophisticated topics like callable objects, reflection, and threading. The book concentrates on the language, preparing you to use Ruby in any way you choose. This second edition includes coverage of new Ruby features such as keyword arguments, lazy enumerators, and Module#prepend, along with updated information on new and changed core classes and methods.

What's Inside

- Clear explanations of Ruby concepts
- Numerous simple examples
- Updated for Ruby 2.1
- Prepares you to use Ruby anywhere for any purpose

David A. Black is an internationally known Ruby developer, author, trainer, speaker, event organizer, and founder of Ruby Central as well as a Lead Consultant at Cyrus Innovation.

"Once again, the definitive book on Ruby from David Black. A must-have!"
—William Wheeler, TekSystems

"All wheat, no chaff—takes you from Ruby programmer to full-fledged Rubyist."
—Doug Sparling
Andrews McMeel Universal

"Provides powerful insights and digs into Ruby's quirks. Revelatory."
—Ted Roche
Ted Roche & Associates, LLC

"The best way to learn Ruby fundamentals."
—Derek Sivers, sivers.org

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/TheWellGroundedRubyistSecondEdition

ISBN 13: 978-1-617291-69-2
ISBN 10: 1-617291-69-2
54499
9 781617 291692

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]