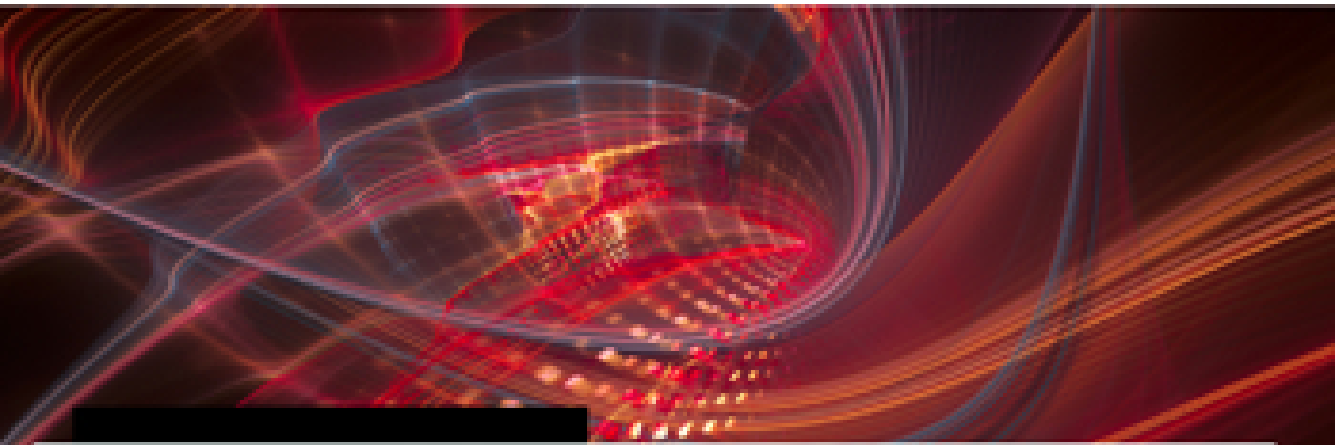

Regular Expressions

Pocket Primer

O. Campesato

REGULAR EXPRESSIONS

Pocket Primer

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and its companion files (the "Work"), you agree that this license grants permission to use the contents contained herein, including the companion files, but does not give you the right of ownership to any of the textual content in the book / files or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion files, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and/or companion files, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

The companion files are available for downloading by writing to the publisher at info@merclearning.com.

REGULAR EXPRESSIONS

Pocket Primer

Oswald Campesato

MERCURY LEARNING AND INFORMATION
Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
800-232-0223

O. Campesato. *Regular Expressions Pocket Primer.*
ISBN: 978-1-68392-227-8

Library of Congress Control Number: 2018943739

181920321 Printed on acid-free paper in the United States of America.

I'd like to dedicate this book to my parents –
may this bring joy and happiness into their lives.

CONTENTS

PREFACE

WHAT IS THE GOAL?

The goal of this book is to introduce readers to regular expressions in several technologies. While the material is primarily for people who have little or no experience with regular expressions, there is also some content that may be suitable for intermediate users, or for people who wish to understand how to translate what they know about regular expressions from prior experience into any of the languages discussed in this book. Hence, this is more suitable as an introductory "how-to" book than a reference book. *Keep in mind that this book will not make you an expert in creating regular expressions.*

If you are interested in applying regular expressions to tasks that involve some type of data cleaning, *Data Cleaning Pocket Primer* might be a good fit for you.

IS THIS BOOK IS FOR ME AND WHAT WILL I LEARN?

This book is intended for data scientists, data analysts, and other people who want to understand regular expressions to perform various tasks. As such, no prior knowledge of regular expressions is required (but can obviously be helpful).

You will acquire an understanding of how to create an assortment of regular expressions, such as filtering data for strings containing uppercase or lowercase letters; matching integers, decimals, hexadecimal, and scientific numbers; and context-dependent pattern matching expressions.

Some chapters contain use cases, such as replacing non-alphabetic characters with a white space (Chapter 1), how to switch columns in a text file (Chapter 5), and how to reverse the order of the fields of a record in a text file

(Chapter 5). Moreover, the Appendix contains Perl-based regular expressions that are taken from Chapter 1 (and portions of Chapter 2).

This book saves you the time required to search for relevant code samples, adapting them to your specific needs, which is a potentially time-consuming process.

HOW WERE THE CODE SAMPLES CREATED?

The code samples in this book were created and tested using bash on a Macbook Pro with OS X 10.12.6 (macOS Sierra). Regarding their content: the regular expressions are derived primarily from the author, and in some cases there are code samples that incorporate short sections of code from discussions in online forums. The key point to remember is that the overwhelming majority of the code samples follow the "Four Cs": they must be Clear, Concise, Complete, and Correct to the extent that it's possible to do so, given the size of this book.

WHAT YOU NEED TO KNOW FOR THIS BOOK

You need some familiarity with working from the command line in a Unix-like environment. However, there are subjective prerequisites, such as a strong desire to learn regular expressions, along with the motivation and discipline to read and understand the code samples. In any case, if you're not sure whether or not you can absorb the material in this book, glance through the code samples to get a feel for the level of complexity.

WHICH REGULAR EXPRESSIONS ARE EXCLUDED?

Although there isn't a specific list, this book does not cover the REs that are very complex and contain "corner cases" that are useful for expert-level developers. The purpose of the material in the chapters is to illustrate how to use create a variety of regular expressions for handling common data-related tasks with datasets, after which you can do further reading to deepen your knowledge.

HOW DO I SET UP A COMMAND SHELL?

If you are a Mac user, there are three ways to do so. The first method is to use `Finder` to navigate to `Applications > Utilities` and then double click on the `Utilities` application. Next, if you already have a command shell available, you can launch a new command shell by typing the following command:

```
open /Applications/Utilities/Terminal.app
```

A second method for Mac users is to open a new command shell on a Macbook from a command shell that is already visible simply by clicking

`command+n` in that command shell, and your Mac will launch another command shell.

If you are a PC user, you can install Cygwin (open source *https://cygwin.com/*) that simulates bash commands, or use another toolkit such as MKS (a commercial product). Please read the online documentation that describes the download and installation process.

If you use RStudio, you launch a command shell inside of RStudio by navigating to `Tools > Command Line`, and then you can launch bash commands. *Note* that custom aliases are not automatically set if they are defined in a file other than the main start-up file (such as .bash_login).

WHY IS PERL IN AN APPENDIX AND NOT IN A CHAPTER?

Although Perl has fantastic support for regular expressions (and peerless for many years), Perl has become a sort of "niche" language. Since Perl appeals to a much smaller audience, it makes more sense to include Perl regular expressions in an Appendix instead of a chapter.

However, it's worth spending a few minutes to skim through the first portion of the Perl Appendix: the examples of regular expressions are modeled after the material in Chapter 1 and the syntax is very similar.

In addition, if you are a front-end Web developer (or perhaps a full-stack developer), you will benefit from the Appendix because the Perl examples are more similar to JavaScript than other scripting languages. Furthermore, if you work with R, you can leverage your knowledge of Perl regular expressions because the Perl syntax is supported in R.

WHAT ARE THE "NEXT STEPS" AFTER FINISHING THIS BOOK?

The answer to this question varies widely, mainly because the answer depends heavily on your objectives. The best answer is to try a new tool or technique from the book out on a problem or task you care about, professionally or personally. Precisely what that might be depends on who you are, as the needs of a data scientist, manager, student or developer are all different. In addition, keep what you learned in mind as you tackle new data cleaning or manipulation challenges. Sometimes knowing a technique is possible makes finding a solution easier, even if you have to re-read the section to remember exactly how the syntax works.

If you have reached the limits of what you have learned here and want to get further technical depth about regular expressions, there are various online resources and literature describing how to create complex and arcane regular expressions.

CHAPTER 1

INTRODUCTION TO REGULAR EXPRESSIONS

This chapter introduces you to basic Regular Expressions, often abbreviated as REs, that will prepare you for the material in subsequent chapters. The REs in this chapter are illustrated via the Unix `grep` utility that is available on any Unix-related platform, including Linux and MacBook (OS X). If you are a complete neophyte, you'll learn a decent variety of REs by the time you have finished reading this chapter.

In fact, this chapter does not require you to understand any of the deeper theory that underlies REs: simply launch the `grep` (or `egrep`) utility from the command line to see the result of matching REs to various strings. In most cases, the text strings are placed in text files so that the REs can be tested against multiple strings simultaneously.

In essence, this chapter acts as "ground zero" for REs, starting from the simplest search strings (i.e., hard-coded strings), to search strings that contain REs involving uppercase letters, lowercase letters, numbers, special characters, and various combinations of such strings.

If you have some experience working with REs, skim through the code samples in this chapter (you might find something new to you). If you are impatient, see if you can explain the purpose of the following RE: `[^ ]*?@[^ ]*?\.[^ ]*`. If you know the answer, then you can probably go directly to Chapter 2.

The first section in this chapter (which comprises most of the chapter) contains code snippets that illustrate how to perform very simple pattern matching with lines of text in a text file. This section also introduces the metacharacters `^, $, ., \,` and `?`, along with code snippets that illustrate how to use these metacharacters in REs (and also their nuances). The purpose of this section is to provide a myriad of concrete examples of REs, after which the more abstract descriptions of metacharacters will be more meaningful to you.

The second section in this chapter contains a summary of metacharacters and character classes, along with code snippets that illustrate how to use them. For example, you will see how to match alphabetic characters (uppercase, lowercase, or a combination of both types), pure digits, and REs with combinations of digits and alphabetic characters.

WHAT ARE REs?

An RE (Regular Expression) is a text string that describes a pattern match or a search pattern where the text string can include metacharacters. In simplified terms, a metacharacter is a character that represents something other than itself to help match patterns. If you have ever used "*" in a Find tool (ctrl-F on your browser) to represent a wildcard, you have used * as a metacharacter. Here are some examples of REs (you will see them again later in this chapter), each of which represents a different search pattern:

```
grey
gr[a-z]y
^the
^[the]
[^the]
^[^z]
^t.*gray
^the.*gray.$
```

A pattern match can involve a character, a word, a group of words, or an entire sentence. Many REs are of the form "find the lines in a text file that contain the word (or pattern) ___."

In order to illustrate the output from matching REs with text strings, we'll use the well-known Unix `grep` utility (and sometimes the Unix `egrep` utility) for the code samples in this chapter. If you work on a PC, please read the Preface for information about software to download to your PC so that you can run the `grep` and `egrep` commands. If you are unfamiliar with Unix, then you can learn about these two commands from short online tutorials. However, we won't discuss the various options that `grep` and `egrep` support because we're only interested in seeing the result of invoking these command line utilities with REs.

Listing 1.1 displays the contents of `lines1.txt`, which contains several lines of text that are relevant to various REs in this section.

LISTING 1.1: lines1.txt

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

As you can see, the word `grey` appears in the first and second lines, the word `gray` appears in the first and third lines, and all three lines contain either `grey` or `gray`.

Here are the tasks that we want to perform in this section (and also the next section):

1. find the lines that contain grey
2. find the lines that contain gray
3. find the lines that contain either grey or gray

The solutions to the three preceding tasks are very easy. The following command performs the first task:

```
grep grey lines1.txt
```

The output of the preceding command is here:

```
the dog is grey and the cat is gray.
this dog is grey
```

The following command performs the second task:

```
grep gray lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
that cat is gray
```

The third task can be solved using the metacharacter "|" (logical "or" in egrep syntax) and the `egrep` utility, as shown here:

```
egrep "gray|grey" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

The third task can also be solved with a character class, which is the topic of the next section.

Your First Character Class

The examples in the previous section show you how to search for hard-coded strings in a text file. Sometimes you can combine two (or more) search expressions into one by using a character class. Specifically, suppose you want to search for either `gray` or `grey` in a text file, which means matching with the vowel a or the vowel e. Square brackets provide this functionality: the term `[ae]` means "use either a or e" (and later you'll see other variations, such as a range of letters or numbers).

The following command performs the third task listed in the previous section:

```
grep gr[ae]y lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

The term `gr[ae]y` is an RE, and it's a compact way of representing the two strings `gray` and the string `grey`. The order of the letters in the square brackets is irrelevant, which means that the third task can also be solved with this command:

```
grep gr[ea]y lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

Specifying a Range of Letters

We can "expand" the RE in the preceding code snippet to include all the lowercase letters of the alphabet: `[a-z]`. We can find all the lines that contain a string of the form `gr[a-z]y`, which matches any string that meets the following conditions:

1. start with the letters `gr`
2. followed by any single letter `a, b, c, …, z`
3. end with the letter `y`

Just to confirm, launch the following command:

```
grep gr[a-z]y lines1.txt
```

The output of the preceding command is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

The matching lines contain either `grey` and `gray`, and if the text file included a line with the string `grzy`, then such a line would appear in the previous output.

We can also specify a single letter inside the square brackets. For example, the term `[a]` is an RE that matches the letter `a`. Now launch this command from the command line:

```
grep [a] lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
that cat is gray
```

If we specify a vowel that does not appear in any word in `lines1.txt`, then there is no output. An example is here:

```
grep [u] lines1.txt
```

We can specify different ranges of letters. For example, suppose we want to find the lines that contain words with any of the vowels `e, o,` or `u`. This expression will do the job:

```
grep "[eou]" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
```

Once again, the order of the letters in the square brackets is irrelevant, which means that the following commands have the same output:

```
grep "[eou]" lines1.txt
grep "[oeu]" lines1.txt
grep "[oue]" lines1.txt
```

One other point regarding the preceding RE should be mentioned before we complete this section. Suppose that the file `abc.txt` consists of four lines that contain the words `day`, `dog`, `den`, and `dupe`. Then the following RE will not match lines with words that contain the vowels a or i:

```
grep "[eou]" abc.txt
```

The output is here:

```
dog
den
dupe
```

WORKING WITH THE "^" AND "$" METACHARACTERS

The special character ^ is called a *metacharacter* (discussed in more detail later), and it's used for matching a pattern that starts from the beginning of a line. For example, the RE `^the` matches any lines that start with the string `the`:

```
grep "^the" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

On the other hand, the RE `^[the]` matches any lines that start with one of the letters `t`, `h`, or `e`, as shown here:

```
grep "^[the]" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray
this dog is grey
that cat is gray
```

Excluding Matches with the "^" Metacharacter

The metacharacter ^ has two different interpretations, based on whether it's specified inside or before a pair of square brackets. If it's inside the square brackets, it means do *not* use any of the letters that appear inside the square brackets, and if it's outside the brackets (as you saw in the previous section), it means "a matching line must *start* with the RE that immediately follows the ^ character."

For example, the following RE matches any lines that start with the letter `t`:

```
grep "^[t]" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

By contrast, the following expression matches any lines that do *not* start with the letter `t` (and in this case, there are no matching lines):

```
grep "^[^t]" lines1.txt
```

Here are additional examples of using "^" with character classes:

1. `^[a-z]` matches any lowercase letter at the beginning of a line of text
2. `^[^a-z]` matches any line of text that does *not* start with a lowercase letter

Based on what you have learned thus far, you know the meaning of the following REs:

3. `([a-z]|[A-Z])`: either a lowercase letter or an uppercase letter
4. `(^[a-z][a-z])`: an initial lowercase letter followed by another lowercase letter
5. `(^[^a-z][A-Z])`: anything other than a lowercase letter followed by an uppercase letter

Later in this book you will learn how to match more complex expressions, such as zip codes for different countries, email addresses, phone numbers, and ISBNs.

Matches with the "$" Metacharacter

The metacharacter $ enables you to match letters or words that appear at the end of a text string or a line of text. For example, the following expression matches any lines in the file `lines1.txt` that end with the word `gray`:

```
grep "gray$" lines1.txt
```

The output is here:

```
that cat is gray
```

Notice that the first line in the file `lines1.txt` is excluded, because although `gray` is in the line, the line ends with a period instead of `gray`.

WORKING WITH ".", "*", AND "\" METACHARACTERS

The metacharacter "." matches any single character (except a linefeed). At the other extreme is the metacharacter "*" that matches zero or more occurrences of any character. In addition, this * is often called a wildcard, and it behaves the same way in most regular expression syntax as it does in common Find/Replace tools. The metacharacter "*" is useful when you want to match the intervening letters between a start character (or word) and an end character (or word).

For example, if you want to match the lines that start with the letter `t`, followed by any letters, and then followed by an occurrence of the word `gray`, use this expression:

```
grep "^t.*gray" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray
that cat is gray
```

Notice how the metacharacters ".*" enable you to "match" the intervening characters between the initial `t` and the occurrence of the word `gray` somewhere else in a line. In this example, `gray` appears at the end of both matching lines, but a line containing the word `gray` somewhere "in the middle" would also have matched the RE.

If you want to match the lines that start with the word the, followed by an occurrence of the word `gray`, use this RE:

```
grep "^the.*gray" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

You can match the final "." character by using the "escape" metacharacter "\". This tells the expression that it should treat something that is normally a

metacharacter as an actual period, and to match it as if it was a normal character, such as "a" or "x". That means the following RE says "find lines that starts with "t" and end in "grey." The escape character is covered in more detail later in this chapter.

The following RE also match the final "." character:

```
grep "^t.*gray.$" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

The following RE also matches the final "." character, because a period is a legitimate match, but it would also match a line that ends in "grayx" or "gray!"

```
grep "^t.*gray.$" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

However, the following RE does *not* match the final "." character, but it will match a line that ends in "gray," because "." as a metacharacter matches the final "y":

```
grep "^t.*gra.$" lines1.txt
```

The output is here:

```
that cat is gray
```

Finally, the following expression only matches the first line, because you need one and only one additional character after "grey" to match:

```
grep "^the.*gray.$" lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

Checking for Whitespaces

Listing 1.2 displays the contents of `spaces.txt`, which contains several lines of text consisting mainly of whitespaces.

LISTING 1.2: spaces.txt

```
                    x
y
z                   w

a
```

Match lines that contain a whitespace with this expression:

```
grep " " spaces.txt
```

The output is here:

```
                    x
y
z                   w
```

Match lines that start with a whitespace with this expression:

```
grep "^ " spaces.txt
```

The output (of two lines) is here:

```
                    x
```

Match lines that end with a whitespace with this expression:

```
grep " $" spaces.txt
```

The output consisting of two lines (the second line is blank) is here:

```
y
```

Match lines that contain only whitespaces with this expression, which literally means "match lines that begin with whitespace, and end in one or more instances of whitespace." The "+" metacharacter means "match one or more instances of the prior element":

```
egrep "^[ ][ ]+$" spaces.txt
```

The output consists of one blank line.

Note that the following REs will not match just the lines that contain only whitespaces:

```
egrep "[ ][ ]+.*$" spaces.txt
egrep "^[ ][ ].*$" spaces.txt
egrep "[ ][ ].*$"  spaces.txt
egrep "^[ ]*"      spaces.txt
egrep "^[ ].*$"    spaces.txt
```

Test your understanding of the metacharacters in this section by figuring out why the preceding REs also match lines that contain characters other than whitespaces.

Match empty lines with this very simple expression:

```
grep "^$" spaces.txt
```

The output is a blank line, which you will see on the screen. *Note* that matching an empty line is different from matching a line containing only whitespaces.

ESCAPING A METACHARACTER

If you need to treat a metacharacter as a literal character, use the backslash "\" character to "escape" its interpretation as a metacharacter. As we saw earlier, the term "\." escapes the "." and matches a "." that appears inside a line.

Listing 1.3 displays the contents of `lines2.txt`, which contains several lines of text and an embedded "." character.

LISTING 1.3: lines2.txt

```
the dog is grey. the cat is gray.
this dog is called doc.
that cat is called .doc
```

Recall that if you want to match the lines that start with the letter t and also end with the word `gray`, use this expression:

```
grep "^t.*gray\.$" lines2.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

If you want to match the lines that contain a ".", use this expression:

```
grep "\." lines2.txt
```

The output is here:

```
the dog is grey. the cat is gray.
this dog is called doc.
that cat is called .doc
```

If you want to match the lines that match `.doc`, use this expression:

```
grep "\.doc" lines2.txt
```

The output is here:

```
that cat is called .doc
```

The following expression matches the lines that end with `.doc`:

```
grep "\.doc$" lines2.txt
```

The output is here:

```
that cat is called .doc
```

EXAMPLES OF MIXING (AND ESCAPING) METACHARACTERS

Listing 1.4 displays the contents of `lines3.txt`, which is used in code snippets in this section.

LISTING 1.4: lines3.txt

```
grey.
.gray
dog
doggy

cat
catty
catfish
small catfish
```

If you want to match the lines that contain `dog`, use this expression:

```
grep "dog" lines3.txt
```

The output is here:

```
dog
doggy
```

If you want to match the lines that start with the word `dog`, use this expression:

```
grep "^dog" lines3.txt
```

The output is here:

```
dog
doggy
```

If you want to match the lines that end with the word `dog`, use this expression:

```
grep "dog$" lines3.txt
```

The output is here:

```
dog
```

If you want to match the lines that start and also end with the word `dog`, use this expression:

```
grep "^dog$" lines3.txt
```

The output is here:

```
dog
```

If you want to match the lines that start with a blank space, use this expression:

```
grep "^ " lines3.txt
```

The output is here:

```
 catfish
```

If you want to match the lines that start with a period, use this expression:

```
grep "^\." lines3.txt
```

The output is here:

```
.gray
```

If you want to match the lines with any occurrence of a period, use this expression:

```
grep "\." lines3.txt
```

The output is here:

```
grey.
.gray
```

By contrast, the following expression matches all lines because the "." metacharacter has not been escaped (so you are now telling it to match lines that begin with any character at all. Only an empty line would fail to match):

```
grep "^." lines3.txt
```

The output is here:

```
grey.
.gray
dog
doggy
cat
catty
 catfish
 small catfish
```

The following expression matches lines that start with a space, followed by any characters, and then followed by the string `cat`:

```
egrep "[ ].*cat" lines3.txt
```

The output is here:

```
 catfish
 small catfish
```

The following expression matches lines that contain the letter `r` or the letter `e`:

```
grep "[re]" lines3.txt
```

The output is here:

```
grey.
.gray
```

The following expression matches lines that contain the letter g, followed by either the letter r or the letter e:

```
grep "g[re]" lines3.txt
```

The output is here:

```
grey.
```

The following three REs match the word .grey:

```
grep "^[.g][re]"  lines3.txt
grep "^[\.g][re]" lines3.txt
grep "^[^.][re]"  lines3.txt
```

Note that the third RE in the preceding list matches other words (e.g., are, bre, cre, and so forth) that are not contained in lines3.txt, and it's just happenstance that the RE matches the string .grey.

This RE matches the word .gray:

```
grep "^.[g][re]" lines3.txt
```

THE EXTENDED "?" METACHARACTERS "+", "?", AND "|"

This section only provides a brief description of some extended metacharacters; however, you will see examples (in various contents) of these metacharacters throughout this chapter.

Here is the difference between egrep and grep: the former uses the "extended" standard, and grep uses the "basic" standard for regular expressions. In the "extended" standard, three additional metacharacters are allowed, two of which we touched on in previous sections.

- "?" means "match exactly zero or one instance of the previous element"
- "+" means "match one or more instances of the previous element"
- "|"is used as a "logical or" in an extended regular expression

Note that in many modern operating systems or environments, one or more of the previous extended elements can be accessed by using "grep -e" (for "extended") if egrep is not available. Most other languages that use regular expressions will have these metacharacters available in some form, the "extended" syntax being more common than the basic in most modern languages. Some examples containing metacharacters are shown as follows:

The expression a? matches the string a and also the string a followed by a single character, such as a1, a2, …, aa, ab, ac, and so forth. However, abc and a12 do not match the expression a?.

The expression a+ matches the string a followed by one or more characters, such as a1, a2, …, aa, ab, ac, abc, a12, and so forth.

The expression a* matches the string a followed by zero or more characters, such as a, a1, a2, …, aa, ab, ac, and so forth.

The pipe "|" metacharacter (which has a different context from the pipe symbol in the command line: REs have their own syntax, which does not match that of the operating system a lot of the time) provides a choice of options. For example, the expression `a|b` means `a` or `b`, and the expression `a|b|c` means `a` or `b` or `c`.

The "`$`" metacharacter refers to the end of a line of text, and in REs inside the vi editor, the "`$`" metacharacter refers to the last line in a file.

The "`^`" metacharacter refers to the beginning of a string or a line of text. For example:

```
*a$ matches "Mary Anna" but not "Anna Mary"
^A* matches "Anna Mary" but not "Mary Anna"
```

In the case of REs, the "`^`" metacharacter can also mean "does not match": the context determines which interpretation to use for the "`10`" metacharacter.

This chapter contains multiple sections with examples of these metacharacters, as well as "`^`", "`$`", "`*`", and "`\`" metacharacters.

MIXED-CASE STRINGS AND REs

This section contains examples of REs that match mixed-case strings (typically user names or text that has proper sentences). Listing 1.5 displays the contents of `lines5.txt`, which is used in code snippets in this section.

LISTING 1.5: lines5.txt

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
He is John smith. the cat is gray.
He is john smith. the cat is gray.
that cat is called .doc
```

The following RE matches lines that contain mixed-case strings, but not lines that fail to have mixed-case strings. Recall that `[A-Z]` is the character class that matches any capital letter, and `[a-z]` is the character class that matches any lowercase letter:

```
egrep "[A-Z][a-z]+" lines5.txt
```

The output is here:

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
```

The following RE matches mixed-case strings that end with a period ".":

```
egrep "[A-Z][a-z]+\." lines5.txt
```

The output is here:

```
He is John Smith. the cat is gray.
```

The following RE matches mixed-case strings that start with an uppercase letter:

```
egrep "^[A-Z][a-z]+" lines5.txt
```

The output is here:

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
He is John smith. the cat is gray.
He is john smith. the cat is gray.
```

The following RE matches strings that start with an uppercase letter followed by a space and another lowercase string, and end in a period ".":

```
egrep "[A-Z][a-z]+ [a-z]+\." lines5.txt
```

The output is here:

```
He is John smith. the cat is gray.
```

The following RE matches strings that start with an uppercase or lowercase `J`, followed by the letters `ohn`:

```
grep "[Jj]ohn" lines5.txt
```

The output is here:

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
He is John smith. the cat is gray.
He is john smith. the cat is gray.
```

Another RE that uses the "|" metacharacter to match strings that contain either `John` or `john` is here:

```
egrep "(John|john)" lines5.txt
```

The output is here:

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
He is John smith. the cat is gray.
He is john smith. the cat is gray.
```

The following RE matches strings that do *not* start with an uppercase or lowercase `J`, followed by the letters `ohn`:

```
grep "[^Jj]ohn" lines5.txt
```

There is no output for the preceding RE because there are no matching lines.

USING \S AND \S IN REs

For your convenience, here are the contents of `lines3.txt` (displayed in Listing 1.4) that are used for the REs in this section:

```
grey.
.gray
dog
doggy

cat
catty
 catfish
 small catfish
```

NOTE *The examples in this section require `egrep` because `grep` does not support + on all operating systems.*

The expression `\s` matches a single whitespace. For example, the following expression matches lines that start with one or more whitespaces, followed by the string `cat`:

```
egrep "\s+cat" lines3.txt
```

The output is here:

```
catfish
```

The following expression matches lines that start with one or more whitespaces then any number of characters, and are then followed by the string `cat`:

```
egrep "\s+.*cat" lines3.txt
```

The output is here:

```
 catfish
 small catfish
```

Use `\S` when you want to match non-whitespace characters. For example, the following expression matches lines that do not start with a whitespace:

```
egrep "^\S+" lines3.txt
```

The output is here:

```
grey.
.gray
dog
doggy
cat
catty
```

USING \W AND \w IN REs

The expression \w matches a single word, where a "word" consists only of letters, digits, or underscores. To match, a "word" must either start the line or be preceded by a non-word character, such as whitespace or a period. The following expression matches lines that start with a word:

```
grep "^\w" lines3.txt
```

The output is here:

```
grey.
dog
doggy
cat
catty
```

The expression \W matches a non-word. The following expression matches lines that do not start with a word:

```
grep "^\W" lines3.txt
```

The output is here:

```
.gray
 catfish
 small catfish
```

The following expression matches lines that do not start with a word, followed by the string `cat`:

```
egrep "^\Wcat" lines3.txt
```

The output is here:

```
 catfish
```

USING \B AND \b IN REs

The simplest scenario for \b involves an exact match of a string without matching longer strings that contain the given string. For example, use \b in a RE if you need to match the string `see` but *not* the strings `seen`, `foreseen`, or `Pharisee`. Notice that the string `see` occurs in the beginning, middle, and end, respectively, of the preceding three strings.

As another example, the following RE matches lines that contain the string that starts and ends with `cat`:

```
grep "\bcat\b" lines3.txt
```

The output is here:

```
cat
```

The following expression matches lines that do not start with a word, and contain the string `cat` somewhere in the line:

```
egrep "^\W.*\bcat" lines3.txt
```

The output is here:

```
 catfish
 small catfish
```

You can use \b as a boundary marker to match email addresses that occur somewhere in a text string:

```
grep "\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b" lines6.txt
```

MATCHING DATE STRINGS

Listing 1.6 displays the contents of `lines6.txt`, which is used in code snippets in this section. As you can see, some of the lines contain valid dates and others contain invalid date formats. Recall that `[0-9]` is the character class that matches any integer.

LISTING 1.6: lines6.txt

```
05/12/18
05/12/2018
05912918
05.12.18
05.12.2018
0591292018
```

The following expression matches lines that start with two digits (followed by anything):

```
grep "^[0-9][0-9]." lines6.txt
```

The output is here:

```
05/12/18
05/12/2018
05912918
05.12.18
05.12.2018
0591292018
```

The following expression matches lines that start with two digits (followed by a forward slash):

```
grep "^[0-9][0-9]/" lines6.txt
```

The output is here:

```
05/12/18
05/12/2018
```

The following expression matches lines that start with two digits (followed by a forward slash or period):

```
grep "^[0-9][0-9][\/.]" lines6.txt
```

The output is here:

```
05/12/18
05/12/2018
05.12.18
05.12.2018
```

The following expression matches lines that end with a forward slash or period, followed by two digits:

```
grep "[\/.][0-9][0-9]$" lines6.txt
```

The output is here:

```
05/12/18
05/12/2018
05.12.18
05.12.2018
```

By contrast, the following expression matches lines that *contain* a forward slash or period, followed by two digits:

```
grep "[\/.][0-9][0-9]" lines6.txt
```

The output is here:

```
05/12/18
05.12.18
```

The following expression matches lines that contain four consecutive digits:

```
grep "[0-9][0-9][0-9][0-9]" lines6.txt
```

The output is here:

```
05/12/2018
05912918
05.12.2018
0591292018
```

The following expression matches lines that end with four consecutive digits that are preceded by a forward slash or period:

```
grep "[\/.][0-9][0-9][0-9][0-9]" lines6.txt
```

The output is here:

```
05/12/2018
05.12.2018
```

Remove the "." in the preceding RE to obtain the following expression that matches the pattern `mm/dd/yyyy`:

```
grep "[\/][0-9][0-9][0-9][0-9]" lines6.txt
```

The output is here:

```
05/12/2018
```

Keep in mind that it's common for people to start a date with a one-character month (for example, 5 instead of 05, which requires matching one digit instead of two digits). A single RE for this scenario probably requires the "or" operator | to handle both possibilities for the first nine months of the year.

Working with \d and \D in REs

There is a simpler way to match a digit: use the `\d` character class (`d` is for digit). The following RE matches lines that contain three consecutive digits:

```
grep "\d\d\d" lines6.txt
```

The output is here:

```
05/12/2018
05912918
05.12.2018
0591292018
```

There is also a simpler way to match multiple consecutive digits via the `\d` character class. The following expression uses `egrep` to match lines that contain three consecutive digits:

```
egrep "\d{3}" lines6.txt
```

```
05/12/2018
05912918
05.12.2018
0591292018
```

Here are some REs that use `egrep` in order to match some common patterns:

`[0-9][0-9]` matches a consecutive pair of digits
`[0-9[0-9][0-9]` matches three consecutive digits
`\d{3}` also matches three consecutive digits

The following expression uses `egrep` to match lines that contain any pair of digits followed by a non-digit character:

```
egrep "\d{2}\D" lines6.txt
```

```
05/12/18
05/12/2018
05.12.18
05.12.2018
```

The following expression uses `egrep` to match lines that contain three pairs of digits that are separated by a non-digit character:

```
egrep "\d{2}\D\d{2}\D\d{2}" lines6.txt

05/12/18
05/12/2018
05.12.18
05.12.2018
```

The following expression uses `egrep` to match lines that contain three pairs of digits that are separated by a non-digit character, and also *exclude* four-digit sequences:

```
egrep "\d{2}\D\d{2}\D\d{2}$" lines6.txt

05/12/18
05.12.18
```

The following RE matches the pattern `mm/dd/yyyy`, which is similar to the example in the previous section, except that now we are using a "cleaner" syntax:

```
egrep "\d{2}/\d{2}/\d{4}$" lines6.txt
```

The output is here:

```
05/12/2018
```

The following RE matches US social security numbers (SSNs) consisting of three digits followed by a hyphen, two digits followed by a hyphen, and ending with three digits:

```
egrep "^\d{3}-\d{2}-\d{4}" lines6.txt
```

The output is here (the following string is not a real SSN):

```
123-45-6789
```

Now that you've seen some working examples of REs, let's summarize our understanding of metacharacters.

SUMMARY OF METACHARACTERS AND CHARACTER CLASSES

Metacharacters can be thought of as a complex set of wildcards. An RE is a "search pattern" which is a combination of normal text and metacharacters. In concept it is much like a "find" tool (press `ctrl-f` on your search engine), but `bash` (and Unix in general) allows for much more complex pattern matching because of its rich metacharacter set. There are entire books devoted to REs, but this section contains enough information to get started, as well as the key concepts needed for data manipulation and cleansing.

The following metacharacters are useful with REs:
The ? metacharacter refers to 0 or 1 occurrences of something
The + metacharacter refers to 1 or more occurrences of something
The * metacharacter refers to 0 more occurrences of something
The ^ metacharacter matches the beginning of a line (or excludes characters)
The $ metacharacter matches the end of a line
The | metacharacter is an "OR" operator that allows for alternatives
The \ metacharacter "escapes" metacharacters and treats them as normal characters

Note that "something" in the preceding descriptions can refer to a digit, letter, word, or more complex combinations. Some examples are shown earlier in this chapter.

Character classes enable you to express a range of digits, letters, or a combination of both. For example, the character class `[0-9]` matches any single digit; `[a-z]` matches any lowercase letter; and `[A-Z]` matches any uppercase letter. You can also specify subranges of digits or letters, such as `[3-7]`, `[g-p]`, and `[F-X]`, as well as other combinations:

[0-9][0-9] matches a consecutive pair of digits
[0-9[0-9][0-9] matches three consecutive digits
\d{3} also matches three consecutive digits

A USE CASE: COLUMN SPLITTING IN PERL (OPTIONAL)

This section is marked optional because it's more complex than the other examples in this chapter, and also because the solution involves Perl. If you have some knowledge of Perl, then you will probably be comfortable with this example. If you are new to Perl, you might benefit from reading at least a portion of the Perl Appendix before delving into the code sample in this section.

This example illustrates how to use a basic RE to solve a common data-related task. Until now we've focused on the "Find" applications, but REs are just as useful in Find/Replace scenarios. For our first example of this type, we will show how it works in Perl. In later chapters we'll show how to do this exact use case in other languages and environments. If you are interested in more Perl examples, the appendix contains many, including both Perl versions of this chapter's examples as well as some of the more advanced concepts from later chapters.

Listing 1.7 displays the contents of `alphanums.txt`, which consists of two comma-separated fields in each row. Question: how would you split each row into three fields consisting of numbers and alphabetic characters?

LISTING 1.7: alphanums.txt

```
"AAA_1234_4",1XY
"BBB_5678_3",2YX
```

```
"CCC_9012_2",3YZ
"DDD_3456_1",4WX
```

One Perl-based solution for replacing everything except letters and digits with blank spaces is shown here:

```
perl -pln -e 's/[^a-zA-Z0-9]/ /g' alphanums.txt
```

The result of the preceding code snippet is here:

```
 AAA 1234 4  1XY
 BBB 5678 3  2YX
 CCC 9012 2  3YZ
 DDD 3456 1  4WX
```

A Perl-based solution for replacing non-digits with blank spaces is shown here:

```
perl -pln -e 's/[^0-9]/ /g' alphanums.txt
```

The result of the preceding code snippet is here:

```
1234 4  1
5678 3  2
9012 2  3
3456 1  4
```

Notice that every line in the preceding output starts with five blank spaces, because the original lines start with five non-digits. Read the Appendix that contains an assortment of Perl-based REs, many of which are counterparts to the code snippets in this chapter.

The key point of this section is that you can apply your knowledge of the REs to solve a variety of tasks in multiple programming languages, but there may be (usually small) language-specific syntax differences in both the command and in how the output is presented.

USEFUL LINKS

Although this chapter (and the next one as well) uses the `grep` and `egrep` commands for testing REs, there are also websites that enable you to test whether or not a text string matches an RE. For example, the following website provides an interface for testing REs:

https://regex101.com/

Navigate to the preceding website, enter an RE in the "Regular Expression" field, and then specify a text string in the "Test String" field. The right panel displays whether or not a full or partial match succeeded, along with a description of the details of the RE.

A search for "regular expressions in <language>" will always turn up useful syntax links, beyond what is covered in this text.

CHAPTER SUMMARY

This chapter started with an introduction to some basic REs, followed by examples that illustrate how to match (or how to not match) characters or words by combining the most commonly used character classes and metacharacters. A Perl use case showed how a regular expression could be embedded in a programming command to accomplish a common text manipulation problem. Finally, you saw a summary of metacharacters, followed by a summary of character classes, which consolidated the information of the entire chapter.

CHAPTER 2

COMMON REGEX TASKS

This chapter extends the material in Chapter 1, with examples of interesting and more sophisticated REs that match ISBNs, email addresses, and so forth. The REs in this chapter also use the `grep` (or `egrep`) command, just as we did in Chapter 1.

The first (short yet relevant) section of this chapter contains tips for "thinking in REs", specifically designed to help you solve new tasks involving REs. Although this section will not make you an expert in REs, you will learn useful guidelines for creating REs to solve a variety of tasks.

The second section in this chapter contains REs that match dates, phone numbers, and zip codes. You will also see REs that match various types of numbers, such as integers, decimals, hexadecimals, octals, and binary numbers. In addition, you will learn how to create REs for scientific numbers.

The third section contains REs that match IP addresses and simple comment strings (in source code), as well as REs for matching proper names and ISBNs. The final section discusses capture groups and back references, which are useful for more complex pattern matches.

SOME TIPS FOR "THINKING" IN REs

This section provides a simple methodology for creating the REs in this chapter as well as REs for your own projects. After you have crafted an RE for a task, then you can focus on simplifying that RE. However, there is often a trade-off: compact REs that are also complex and sophisticated tend to be more difficult for other people to understand (and hence more difficult to debug and to enhance), whereas lengthier REs that are based on a combination of simpler REs can be simpler to manage in applications. When in doubt, include a well-structured comment block that concisely explains the purpose of the RE.

Another point to consider: favor the use of well-tested RE libraries over writing custom REs. This provides several advantages: bugs in REs are less likely to occur (especially in more mature libraries), a broader user community can share information about the libraries, and functionality that is added in future releases will benefit everyone.

Regardless of the strategy that you adopt, keep in mind that it's easier to understand REs written by other people if you have extensive experience writing REs (which might seem contradictory, but it's actually true).

Here is the key idea for creating REs: use a "divide and conquer" strategy, which involves a bottom-up (instead of top-down) approach. In general terms, suppose that your task can be described in terms of three patterns P1, P2, and P3. If you can find REs RE1, RE2, and RE3 that correspond to P1, P2, and P3, respectively, then a candidate solution to the original task looks like this:

```
egrep "^(RE1|RE2|RE3)$" input.txt
```

As a simple illustration, suppose you need an RE for positive and negative integers, which consists of one pattern for positive numbers (P1) and another pattern for the (optional) positive sign or negative sign (P2).

The following RE matches positive integers (P1). As you recall from Chapter 1, the regular expression argument to `egrep` means literally "begins with and ends with one or more digits":

```
egrep "^\d+$" numbers.txt
```

Since the RE that matches an optional positive sign or a negative sign is `[-|+]?` (again from Chapter 1, it literally means zero or one "+" or "-" character), let's combine this RE with the RE for P1, as shown here:

```
egrep "^[-|+]?\d+$" numbers.txt
```

Someone looking at this later should be able to parse out "begins with either a +, a –, or neither, and the rest of the characters on the line must be one or more digits" by working from left to right.

Although the preceding example is simple, you can use the same type of analysis to solve more complex problems, along with the following points:

1. The RE that you produce is not necessarily perfectly efficient in terms of time to execute or computer resources used.
2. You might be able to extract additional common sub-patterns.
3. Extracting sub-patterns can create REs that are difficult to understand.
4. Difficult REs can be more error-prone and more difficult to debug.
5. The initial solution can be straightforward (quick to create).
6. Refining an RE can consume a great deal of time.

For example, consider the RE for ISBNs, which we will develop later in this chapter. It consists of the concatenation of four REs. The solution presented in this chapter is lengthy, yet it requires about five minutes to create

a solution if you adopt the divide-and-conquer strategy. While this solution is not necessarily optimal (and perhaps merely one of several possible solutions), the final RE is easily decomposed into its four components and therefore more likely to be understood by another person (or even yourself six months later when looking at the code).

As another example, suppose we want to construct an RE that matches positive integers whose digit count is a prime number between three and eleven inclusive. In other words, the numbers we are looking for contain three digits, five digits, seven digits, or eleven digits. The straightforward solution is as follows:

```
egrep "^(d\{3}|d\{5}|d{7}\d{11})$" input.txt
```

The solution for the preceding (albeit contrived) task is a straightforward concatenation of four disjoint REs with an "or" operator connecting them. Our first example just combined the two REs, and when you do that the "and" (all conditions apply to match) is assumed. More difficult tasks are of the form "it's an E1 or E2 but not an E3", where E3 partially overlaps with either E1 or E2 (or both).

REs AND PHONE NUMBERS

This section contains REs that match simple phone numbers, followed by some more complex REs that handle phone extensions. You will also learn about some of the features of a Google library (written in multiple programming languages) that supports international phone numbers. This section is surprisingly long (phone numbers are simple, n'est-ce pas?) and contains some nice REs that will help you hone your ability to differentiate between phone formats that have very slight differences.

As you might already know, different countries have special cases for their phone numbers. In the USA, `(408) 974-3218` is a valid U.S. number, whereas `(999) 974-3218` is invalid. Meanwhile, the numbers `0404 999 999` and `(02) 9999 9999` are valid numbers in Australia, but the sequence `(09) 9999 9999` is invalid. In the United States, any number beginning with a 555 prefix at the local level (e.g., [405] 555-3212) is fake, used only for movies or similar public art, to avoid a random person being bothered by fans dialing the number.

Listing 2.1 displays the contents of `phonenumbers.txt`, which contains various patterns for phone numbers.

LISTING 2.1: phonenumbers.txt

```
1-234-567-8901
1-234-567-8901 x1234
1-234-567-8901 ext1234
1 (234) 567-8901
1.234.567.8901
1/234/567/8901
12345678901
```

The following RE matches U.S. phone numbers of the form `ddd ddd dddd`:

```
egrep "^\d{3} \d{3} \d{4}" phonenumbers.txt
```

The output is here:

```
650 123 4567
```

The following RE matches U.S. phone numbers of the form `ddd ddd-dddd`:

```
egrep "^\d{3} \d{3}-\d{4}" phonenumbers.txt
```

The output is here:

```
650 123-4567
```

The following RE matches U.S. phone numbers of the form `(ddd) ddd-dddd`:

```
egrep "^\(\d{3}\) \d{3}\d{4}" phonenumbers.txt
```

The output is here:

```
(650) 123-4567
```

The following RE matches U.S. phone numbers of the form `1-ddd ddd-dddd`:

```
egrep "^1-\d{3} \d{3}-\d{4}" phonenumbers.txt
```

The output is here:

```
1-(650) 123-4567
```

The following RE checks for numbers that have an optional dash "-" between the three groups of digits:

```
egrep "^\d{3}-?\d{3}-?\d{4}" phonenumbers.txt
```

The output is here:

```
9405306123
```

The following RE checks for numbers that have an optional dash "-" *or* a blank between the three groups of digits:

```
egrep "^\d{3}[- ]?\d{3}[- ]?\d{4}" phonenumbers.txt
```

The output is here:

```
9405306123
650 123-4567
650 123 4567
```

Compare the preceding pair of similar REs to make sure that you understand how (and why) they produce a different set of matching phone numbers.

The following RE matches numbers with seven digits and also numbers with ten digits, with extensions allowed (and delimiters are spaces, dashes, or periods):

```
^(?:(?:\+?1\s*(?:[.-]\s*)?)?(?:\(\s*([2-9]1[02-9]|[2-9][02-8]1|
[2-9][02-8][02-9])\s*\)|([2-9]1[02-9]|[2-9][02-8]1|[2-9][02-8]
[02-9]))\s*(?:[.-]\s*)?)?([2-9]1[02-9]|[2-9][02-9]1|[2-9][02-9]
{2})\s*(?:[.-]\s*)?([0-9]{4})(?:\s*(?:#|x\.?|ext\.?|extension)
\s*(\d+))?$
```

On the other hand, an RE that does *not* match extensions is here:

```
^(?:(?:\+?1\s*(?:[.-]\s*)?)?(?:(\s*([2-9]1[02-9]|[2-9][02-8]1|
[2-9][02-8][02-9])\s*)|([2-9]1[02-9]|[2-9][02-8]1|[2-9][02-8]
[02-9]))\s*(?:[.-]\s*)?)?([2-9]1[02-9]|[2-9][02-9]1|[2-9][02-9]
{2})\s*(?:[.-]\s*)?([0-9]{4})$
```

Imagine yourself having to maintain or debug either of the two preceding REs (good luck!). What is more common in real code is to check each possibility until you match a valid pattern, raising an error if it matches none of them. This not only allows code that is simpler to understand, but also easier maintenance if the rules or laws change. It is easier to comment out or add a new block of code than to tamper with a long regular expression, and it's far less likely to lead to unexpected subtle errors on a change.

The libphonenumber Library

The need to really validate phone numbers without having to know all the arcane rules across the world and maintain complex code has led to shared resources. One alternative to creating your own REs for matching phone numbers is the Google library (in Java, C++ and JavaScript) for parsing, formatting, and validating international phone numbers, found at *https://github.com/googlei18n/libphonenumber*.

For example, the preceding recognizes that the sequence `15555555555` is a possible number but not a valid number. This library provides many features, including the following:

- validation for phone numbers in every country
- detection of phone types (fixed-line, mobile, toll-free, and so forth)
- APIs to provide valid phone numbers (per country/region)
- full validation of phone numbers
- formats numbers on the fly as users enter digits
- geographical information for phone numbers

There are phone-related libraries for other languages that rely on the Google `i18n` phone number dataset:

PHP: *https://github.com/giggsey/libphonenumber-for-php*
Python: *https://github.com/daviddrysdale/python-phonenumbers*

Ruby: *https://github.com/sstephenson/global_phone*
C#: *https://github.com/erezak/libphonenumber-csharp*
Objective-C: *https://github.com/iziz/libPhoneNumber-iOS*

The following website provides a PHP script that validates phone numbers based on a list of acceptable formats: *http://www.bitrepository.com/how-to-validate-a-telephone-number.html*.

REs AND ZIP CODES (U.S. AND CANADIAN)

Listing 2.2 displays the contents of `lines1.txt`, which is used in code snippets in this section.

LISTING 2.2: lines1.txt

```
94053
94053-06123
9405306123
V6K8Z3
36K8Z3
123-45-6789
650 123-4567
650 123 4567
(650) 123 4567
1-650 123-4567
jsmith@acme.com
```

The following RE matches strings that contain five digits (which is a common U.S. zip code pattern):

```
egrep "\d{5}" lines1.txt
```

The output is here:

```
94053
94053-06123
9405306123
```

However, the third string in the preceding output is an invalid U.S. zip code. Let's see how to match either of the first two zip codes and exclude the third (invalid) zip code.

The following expression matches U.S. zip codes consisting of five digits:

```
egrep "^\d{5}$" lines1.txt
```

The output is here:

```
94053
```

The following expression matches U.S. zip codes consisting of five digits followed by a hyphen, and then followed by another five digits:

```
egrep "^\d{5}-\d{5}$" lines1.txt
```

The output is here:

```
94053-06123
```

Recall from earlier examples that the "or" operator lets you combine both expressions to properly sort out both valid U.S. zip code options:

```
egrep "^\d{5} | \d{5}-\d{5}$" lines1.txt

94053
94053-06123
```

You can also define REs that match zip codes that end in a fixed pattern. For example, the following RE matches U.S. zip codes that end with `43` or `58`:

```
egrep "^\d{3}(43|58)$" lines1.txt
```

The preceding RE matches the zip code `94043` as well as the `94058` zip code. On the other hand, the following RE matches zip codes that start with three digits and end in either `53` or `23`:

```
egrep "^[0-9]{3}(53|23)" lines1.txt
```

The output is here:

```
94053
94053-06123
9405306123
```

Valid Canadian postal codes are significantly different from U.S. zip codes: they have the form `A1A 1A1`, where A is a capital letter and 1 is a digit (with a space between the two triplets). The following RE matches Canadian zip codes:

```
egrep "^[A-Z][0-9][A-Z] [0-9][A-Z][0-9]" lines1.txt
```

The output is here:

```
V6K 8Z3
```

Most applications that handle truly international addresses do not try to validate postal codes, instead providing a free-form field outside of the United States and sometimes a few other countries. There is no shared resource similar to the Google i18n phone number dataset.

REs AND EMAIL ADDRESSES

Matching email addresses is a complex task. This section provides REs that match common email addresses that have the following pattern:

1. an initial string having at least four characters and at most twelve characters (which can be any combination of lowercase letters, uppercase letters, or digits), then

2. followed by the "@" symbol, then
3. a string having at least four characters and at most twelve characters (which can be any combination of lowercase letters, uppercase letters, or digits), then
4. followed by the string ".com"

Here is the RE that has the structure described in the preceding list (which also requires `egrep` instead of `grep`) that matches an email address:

```
egrep "^[A-Za-z0-9]{4,12}\@[A-Za-z0-9]{4,8}\.com$" lines1.txt
```

The output of the preceding RE is here:

```
jsmith@acme.com
```

There are a few points to keep in mind regarding the preceding RE. First, it only matches email addresses with the suffix ".com". Second, longer (yet still valid) email addresses are excluded, such as the one shown here:

```
myverylongemailaddress@acme.com
```

Consequently, you need to make decisions about the allowable set of email addresses that you want to match with your RE.

The following RE that has the structure described in the preceding list also allows a dot "." as in the initial portion of the email address:

```
egrep "^[A-Za-z0-9]{4,12}\.[A-Za-z0-9]{4,12}\@[A-Za-z0-9]{4,8}\.
com$" lines1.txt
```

The output is here:

```
john.smith@acme.com
```

The section shown in bold in the preceding RE shows you how to match the dot "." character, followed by an alphanumeric string that has at least four characters and at most twelve characters.

There are other combinations of characters that form valid email addresses that have not been discussed in this section. For example, consider the following email addresses:

```
dave.edward.smith@gmail.com
dave-777-smith@artist.net
Dave-777-Smith@artist.net
```

The REs that match the preceding email addresses are an exercise for you.

Most applications manage the complexity by only focusing on the following patterns:

- It must have one and only one @ in the string.
- It must have at least one character before the @.

- It must have a period after the @, and at least one character between the @ and the period.
- It must have at least one character after the period.

According to the preceding list of rules, the patterns `brad@flick@com`, `@flick.com.`, and `@flick.com` are detected as invalid email addresses. However, `a@b.c` and `-@_.1` would then pass most validations and be assumed to be an email address.

As you can see, there are many "corner cases" to check to validate email addresses, and while it's an interesting exercise, it's probably better to perform an Internet search to find add-ons that have been tested and documented.

HEXADECIMAL COLOR SEQUENCES

Hexadecimal colors start with the # symbol, followed by any combination of the digits 0 through 9, the letters a through f, and the letters A through F.

Listing 2.3 displays the contents of `hexcolors.txt`, which is used in code snippets in this section.

LISTING 2.3: hexcolors.txt

```
#abd
#a1b2d3
#A1B2D4
#A3b5D7
#acbedf
#AcBeDf
#ABD
#fad
#f00
#F00
#FF0000
#ABCDEF
#123456
```

The following RE matches hexadecimal colors with six lowercase letters:

```
egrep '^#[a-f]{6}$' hexcolors.txt
```

The output is here:

```
#acbedf
```

The following RE matches some three-character hexadecimal colors and six-character hexadecimal numbers (but only lowercase letters):

```
egrep '^#([a-f]{3}){1,2}$' hexcolors.txt
```

The output is here:

```
#abd
#acbedf
#fad
```

The following RE matches some three-character hexadecimal colors or six-character hexadecimal numbers containing lowercase letters or uppercase letters (or both):

```
egrep '^#([a-fA-F]{3}){1,2}$' hexcolors.txt
```

The output is here:

```
#abd
#acbedf
#AcBeDf
#ABD
#fad
#ABCDEF
```

The following RE matches six-character hexadecimal numbers that consist of the pattern lowercase letter plus digit, repeated three times:

```
egrep '^#(([a-f]\d){3}){1,2}$' hexcolors.txt
```

The output is here:

```
#a1b2d3
```

Strangely, the following RE does not match six-character hexadecimal numbers that consist of the pattern any letter plus digit, repeated three times (a bug in egrep?):

```
egrep '^#(([a-fA-F]\d){3}){1,2}$' hexcolors.txt
```

Fortunately, the following RE matches everything in hexcolors.txt after replacing \d with the range [0-9]:

```
egrep '^#(([a-fA-F0-9]){3}){1,2}$' hexcolors.txt
```

The output is here:

```
#abd
#a1b2d3
#A1B2D4
#A3b5D7
#acbedf
#AcBeDf
#ABD
#fad
#f00
#F00
#FF0000
#ABCDEF
#123456
```

We were fortunate to define a single RE that is simple, compact, and easy to understand that captures all the combinations of hexadecimal values contained in hexcolors.txt.

WORKING WITH NUMBERS

This section contains examples of REs that match integers, floating point numbers, hexadecimal numbers, octal numbers, and binary numbers. The subsequent section discusses REs for scientific numbers, which are a "generalization" of decimal numbers: they are more complex, and so they merit their own section.

Listing 2.4 displays the contents of `numbers.txt`, which is used in code snippets in this section.

LISTING 2.4: numbers.txt

```
#integers
1234
-123

#floating point numbers
1234.432
-123.528
0.458

#hexadecimal numbers
12345
FA4389
0xFA4389
0X4A3E5C

#octal numbers
1234
03434

#binary numbers
010101
110101
0b010101
```

REs, Integers, and Decimal Numbers

The following RE matches positive integers and negative integers:

```
egrep "^[-|+]?\d+$" numbers.txt
```

The output is here:

```
1234
-123
1234
1234.432 why?
0.458    why?
010101
110101
0b010101 why?
1234
03434
```

The following RE matches positive integers, negative integers, and decimal numbers:

```
egrep "^[-|+]?\d+([\.]?\d*)$" numbers.txt
```

The output is here:

```
1234
-123
1234.432
-123.528
0.458
010101
110101
1234
03434
```

The following RE matches only decimal numbers:

```
egrep "^[-|+]?\d+([\.]\d*)$" numbers.txt
```

The output is here:

```
1234.432
-123.528
0.458
```

REs and Hexadecimal Numbers

Hexadecimal numbers can contain the digits 0 through 9 and also the letters A through F, and they can also start with `0x` or `0X` (both of which are optional).

The following RE matches hexadecimal numbers (and other patterns as well) without a `0x` or `0X` prefix:

```
egrep '^[a-fA-F0-9]+$' numbers.txt
```

The output is here:

```
1234
12345
FA4389
1234
03434
010101
110101
0b010101
```

The last string in the preceding output matches the initial pattern (because of the lowercase "b"). Remove the `a-f` section of the preceding RE if you want to exclude strings that contain lowercase letters.

Notice that numbers that are integers, octal numbers, and binary numbers also appear in the preceding list (because they are valid hexadecimal numbers).

The following RE matches hexadecimal numbers that start with either `0x` or `0X`:

```
egrep '^(0x|0X)[a-fA-F0-9]+$' numbers.txt
```

The output is here:

```
1234
12345
FA4389
0xFA4389
0X4A3E5C
1234
03434
010101
110101
0b010101
```

Once again, notice that numbers that are integers, octal numbers, and binary numbers also appear in the preceding list (because they are valid hexadecimal numbers).

You can also match "couplets" of hexadecimal numbers that are separated by a blank space. For example, the following RE matches the string `A3 B6 3F 62`: (the Bash `echo` command "echoes" the string in quotes):

```
echo "A3 B6 3F 62" | egrep "^([0-9A-F]{2} )+"
```

REs and Octal Numbers

Octal numbers can start with an optional digit 0. The following RE matches octal numbers without a `0` prefix:

```
egrep "^[1-7][0-7]+$" numbers.txt
```

The output is here:

```
1234
110101
1234
```

Notice that there are two occurrences of the number `1234`: the first one appears as an integer (and it's a valid octal number) and the second one appears in the section with octal numbers. Moreover, the number `110101` from the binary section is also a valid octal number.

The following RE matches octal numbers with a `0` prefix:

```
egrep "^0?[1-7]+$" numbers.txt
```

The output is here:

```
1234
1234
03434
```

Once again, there are two occurrences of the number `1234`: the first one appears as an integer (and it's a valid octal number) and the second one appears in the section with octal numbers.

REs and Binary Numbers

The following RE matches binary numbers without a `0b` prefix:

```
egrep "^[0-1]+$" numbers.txt
```

The output is here:

```
010101
110101
```

The following RE matches binary numbers with or without a 0b prefix:

```
egrep "(^[0-1]+|0b[0-1]+)$" numbers.txt
```

The output is here:

```
010101
110101
0b010101
```

WORKING WITH SCIENTIFIC NUMBERS

This section contains examples of REs that match scientific numbers. Listing 2.5 displays the contents of `numbers2.txt` that will be used in code snippets in this section.

LISTING 2.5: numbers2.txt

```
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
```

REs and Scientific Numbers

Matching all scientific numbers (and nothing else) is rather complex, and this section contains some REs that partially succeed in this task.

Option #1: the following RE matches some scientific numbers and other numbers as well:

```
egrep '^[+-]?\d*(([,.]\d{3})+)?([,.]\d+)?([eE][+-]?\d+)?$'
numbers.txt
```

The output is here:

```
1234
-123

1234.432
-123.528
0.458

12345


1234
03434

010101
110101
```

However, the preceding RE provides a better match with `numbers2.txt`:

```
egrep '^[+-]?\d*(([,.]\d{3})+)?([,.]\d+)?([eE][+-]?\d+)?$'
numbers2.txt
```

The output is here:

```
0.123
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
```

For your convenience, Listing 2.6 displays the contents of `scientific.sh`, which contains an assortment of REs that check for scientific numbers, tested against the files `numbers.txt` and `numbers2.txt`.

LISTING 2.6: scientific.sh

```
echo "*** Option #1:"
echo "--------------"
egrep '^[+-]?\d*(([,.]\d{3})+)?([,.]\d+)?([eE][+-]?\d+)?$'
numbers2.txt
echo "*** Option #2:"
echo "--------------"
egrep '^[+-]?\d*(([,.]\d{3})+)?([,.]\d+)?([eE][+-]?\d+)?$'
numbers.txt
echo "*** Option #3:"
echo "--------------"
egrep '[-]?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?' numbers2.txt
echo "*** Option #4:"
echo "--------------"
egrep '[-]?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?' numbers.txt
echo "*** Option #5:"
echo "--------------"
egrep '[-]?(?:0|[1-9]\d*)(?:\.\d*)?(?:[eE][+\-]?\d+)?' numbers2.txt
echo "*** Option #6:"
echo "--------------"
```

```
egrep '[-]?(?:0|[1-9]\d*)(?:\.\d*)?(?:[eE][+\-]?\d+)?' numbers.txt
echo "*** Option #7:"
echo "--------------"
egrep '[-]?(?:0|[1-9]\d*)(?:\.\d*)?(?:[eE][+\-]?\d+)?' numbers2.txt
echo "*** Option #8:"
echo "--------------"
egrep '[+\-]?(?:0|[1-9]\d*)(?:\.\d*)?(?:[eE][+\-]?\d+)?' numbers2.txt
```

Launch the code in Listing 2.6 with this command:

```
./scientific.sh > scientific.out
```

Listing 2.7 displays the contents of `scientific.out`, which shows the result of launching the shell script in Listing 2.6.

LISTING 2.7: scientific.out

```
*** Option #1:
--------------
0.123
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
*** Option #2:
--------------
1234
-123

1234.432
-123.528
0.458

12345

1234
03434

010101
110101

*** Option #3:
--------------
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
*** Option #4:
--------------
1234
```

```
-123
1234.432
-123.528
0.458
12345
FA4389
0xFA4389
0X4A3E5C
1234
03434
010101
110101
0b010101
*** Option #5:
--------------
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
*** Option #6:
--------------
1234
-123
1234.432
-123.528
0.458
12345
FA4389
0xFA4389
0X4A3E5C
1234
03434
010101
110101
0b010101
*** Option #7:
--------------
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
*** Option #8:
--------------
0.123
z = 0xFFFF00;
+13
```

```
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
```

As you can see, the REs in Listing 2.7 have varying degrees of success in terms of matching scientific numbers. In general, they err by matching "false positives" (numbers that are *not* valid scientific numbers) instead of excluding "false negatives" (numbers that *are* valid scientific numbers).

Most real-world applications and programming languages use various numeric variable "types" to deal with calculations, and when forced to translate a number which is stored as text in scientific notation, they require you to either use a predefined format or instruct the command which format is being used. This means most of the complexity of the matching problem is limited to some kind of reasonable subset that can be matched with a simpler regular expression. Unless you are trying to do something like scan documents and pull out numbers from free-form text without knowing ahead of time what format the numbers used, you should not normally encounter this level of complexity.

As with any problem of this type, it is often easier to run several "match" expressions and then filter out duplicates and false positives with other program logic than to try to match every possibility with a single regular expression.

REs AND COMMENTS

This section contains examples of REs that match IP addresses and simple comment strings (in source code). Listing 2.8 displays the contents of `lines2.txt` that will be used in code snippets in this section.

LISTING 2.8: lines2.txt

```
192.168.3.99
192.168.123.065
// this is a comment
v = 7; // this is also a comment
/* the third comment */
x = 7; /* the fourth comment */
y = \uFFEA;
      z = 0xFFFF00;
```

The following RE matches lines that start with `//`:

```
grep "^//" lines2.txt
```

The output is here:

```
// this is a comment
```

The following RE matches lines that contain an occurrence of // (anywhere in the string):

```
grep "//" lines2.txt
```

The output is here:

```
// this is a comment
v = 7; // this is also a comment
```

The following RE matches lines that start with /*:

```
grep "^\/\*" lines2.txt
```

The output is here:

```
/* the third comment */
```

The following RE matches lines that contain an occurrence of /* (anywhere in the string):

```
grep "\/\*" lines2.txt
```

The output is here:

```
/* the third comment */
x = 7; /* the fourth comment */
```

REs AND IP ADDRESSES

This section contains examples of REs that match IP addresses, based on the strings contained in Listing 2.8 in the previous section.

The following RE matches arbitrary valid IP addresses:

```
egrep "^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}" lines2.txt
```

The output is here:

```
192.168.3.99
192.168.123.065
```

The following RE matches valid IP addresses that contain three digits in all four components:

```
egrep "^\d{3}\.\d{3}\.\d{3}\.\d{3}" lines2.txt
```

The output is here:

```
192.168.123.065
```

DETECTING FTP AND HTTP LINKS

Listing 2.9 displays the contents of `lines11.txt`, which contains examples of URLs.

LISTING 2.9: urls.txt

```
ftp://www.acme.com
http://www.bdnf.com
https://www.ceog.com
a line with https://www.ceog.com embedded in it
```

The following snippet matches the lines that contain the string `http` or `https`:

```
grep "http" urls.txt
```

The output is here:

```
ftp://www.acme.com
http://www.bdnf.com
https://www.ceog.com
a line with https://www.ceog.com embedded in it
```

The following snippet matches the lines that contain the string `http` or `https`:

```
egrep "https?" urls.txt
```

The output is here:

```
http://www.bdnf.com
https://www.ceog.com
a line with https://www.ceog.com embedded in it
```

The following snippet matches the lines that contain the string `ftp`, `http`, or `https`:

```
egrep "ftp|http|https" urls.txt
```

The output is here:

```
ftp://www.acme.com
http://www.bdnf.com
https://www.ceog.com
a line with https://www.ceog.com embedded in it
```

The following snippet matches the lines that contain the string `http` embedded in the line of text:

```
egrep "^([ a-z]+)https?://[a-z\.]*" urls.txt
```

The output is here:

```
a line with https://www.ceog.com embedded in it
```

REs AND PROPER NAMES

Listing 2.10 displays the contents of `ProperNames.txt`, which contains a list of proper names along with various titles.

LISTING 2.10: ProperNames.txt

```
Mr. Smith
Mr Smith
Mr. J
Mr J
Mrs Smith
Mrs. Smith
Mrs. S
Mrs S
Ms Smith
Ms. Smith
Ms S
Ms. S
Mr. John Smith
Mr. John Edward David Smith
```

Let's consider the following REs that match male names having only a last name (we'll handle the proper names with a first name and multiple middle names later in this section):

```
^Mr\.?\s[A-Z][a-z]+$
^Mr\.?\s[A-Z]$
```

Now let's consider the following REs that match female names.

```
^Ms\.?\s[A-Z][a-z]+$
^Ms\.?\s[A-Z]$

^Mrs\.?\s[A-Z][a-z]+$
^Mrs\.?\s[A-Z]$
```

Finally, let's consider the following REs that match male names as well as female names, and let's see how they differ:

```
^M([rs]|(rs))\.?\s[A-Z]([a-z]+)?$
^Mr\.?\s[A-Z]\w*$
^M(r|s|rs)\.?\s[A-Z]\w*$
```

Now we can match proper names that contain a first name with the following RE:

```
^M([rs]|(rs))\.?\s[A-Z]([a-z]+\s+[A-Za-z]*)$
```

The following RE matches one or more optional middle names:

```
^M([rs]|(rs))\.?\s[A-Z]([a-z]+(\s*[A-Za-z])+)$
```

This section illustrates the technique discussed at the beginning of this chapter: start with REs that match simple proper names for males and females, and then combine them to into a single RE. As a result, the last step (i.e., a first name and multiple middle names) was a *much* simpler task. Incidentally, this type of task is a "confidence builder" when you can quickly create REs for tasks of moderate complexity.

There are additional cases to consider for proper names. For example, you might need to match suffixes such as Jr., Sr., or Esq. In addition, you might need to consider prefixes such as Sir, Count, Lord, Dr, Dr., Prof, and Professor (among others). Thus, REs that match proper names can involve many nuances, and it's a good idea to determine (to the extent that it's possible to do so) which prefixes and suffices that you need to match before you embark on the task of creating the appropriate REs.

The next section contains REs for matching ISBNs (that are more complex than the REs in this section) and also illustrates the same divide-and-conquer technique.

REs AND ISBNS

The REs in this section involve multiple simpler REs that are concatenated using the pipe "|" symbol, which indicates an "OR" operation (as described in the first section of this chapter).

As a simplified example, suppose we want to construct an RE that matches the following strings:

```
a123
ab123
abc123
abcd123
```

The solution is easy to construct when you describe the strings in an English sentence: the strings start with either an `a` OR an `ab` OR an `abc` OR an `abcd`, AND all of them have the number `123` as the rightmost portion. Using the "|" symbol we can construct the RE like this:

```
^(a|ab|abc|abcd)123$
```

Now let's consider valid ISBNs, which can start with the optional string ISBN, and also contain either ten-digit sequences or thirteen-digit sequences. Listing 2.11 displays the contents of `ISBN.txt`, which contains examples of valid ISBN numbers.

LISTING 2.11: ISBN.txt

```
ISBN 978-0-596-52068-7
ISBN-13: 978-0-596-52068-7
ISBN-10 0-596-52068-9
978 0 596 52068 7
9780596520687
0-596-52068-9
```

Notice that the first line in Listing 2.11 contains the string ISBN followed by a blank space, and the next two lines contain the string ISBN followed by a hyphen, and then two more digits, and then either a colon ":" or a blank space. Those two lines end with a hyphenated thirteen-digit number and a hyphenated ten-digit number, respectively.

The fourth line in Listing 2.11 contains a thirteen-digit number with white spaces; the fifth line contains a "pure" thirteen-digit number; and the sixth line contains a hyphenated ten-digit number.

Now let's see how to match the numeric portion of the ISBNs in Listing 2.11. The following RE matches the digits in the first and the second line:

```
\d{3}-\d-\d{3}-\d{5}-\d
```

The following RE matches the digits in the third line as well as the sixth line:

```
\d-\d{3}-\d{5}-\d
```

The following RE matches the digits in the fourth line:

```
\d{3} \d \d{3} \d{5} \d
```

The following RE matches the digits in the fifth line:

```
\d{13}
```

Now let's create REs for the text prefix (when present) and combine them with the earlier list of REs to match all of the lines in Listing 2.11. The result involves four REs, as shown in the following:

1. the RE (`^([A-Z]{4}[ -]?)? \d{3}-\d-\d{3}-\d{5}-\d`) matches:

```
ISBN 978-0-596-52068-7
ISBN-13: 978-0-596-52068-7
ISBN-10 0-596-52068-9
```

2. the RE (`\d{3} \d \d{3} \d{5} \d`) matches:

```
978-0-596-52068-7
978 0 596 52068 7
```

3. the RE (`\d{13}`) matches:

```
9780596520687
```

4. 4. the RE (`\d-\d{3}-\d{5}-\d`) matches:

```
0-596-52068-9
```

Now we can combine the preceding four REs to create a single RE that matches every valid ISBN in the text file `ISBN.txt`:

```
egrep "^((([A-Z]{4}-\d{2}: \d{3}-\d-)|([A-Z]{4}-\d{2} \d-))?\d{3}-
\d{5}-\d)|(^([A-Z]{4}[ -]?)? \d{3}-\d-\d{3}-\d{5}-\d)|(\d{3} \d
\d{3} \d{5} \d)|(\d{13})|(\d-\d{3}-\d{5}-\d)" ISBN.txt
```

If you decide to use the preceding RE in a bash script, include a comment block to explain how it has been constructed, which will help other people understand your code (and they will appreciate your efforts).

WORKING WITH BACKSLASHES AND LINEFEED (OPTIONAL)

This section is intended for readers with an intermediate level of knowledge regarding REs. If you are a beginner, you can skip this section with no loss of continuity.

Listing 2.12 displays the contents of `linefeeds.txt`, which contains examples of the character sequences `\n` and `\r` in lines of text. Keep in mind that these character sequences involve two characters, and hence they are not the same as a linefeed or a carriage return, which are single characters ^J (hexadecimal 0xA for a linefeed) and ^M (hexadecimal 0xD for a carriage return). If you see either of these characters in a text file, it means that the file was saved using a DOS-based format. Incidentally, the ^M character will also appear in text files that were created from Excel spreadsheets.

One other detail: the combination of a carriage return followed by a linefeed is the end-of-line marker for text files in DOS-like operating systems, whereas text files in Unix-based systems only require a linefeed.

LISTING 2.12: linefeeds.txt

```
\nthe dog is grey/n and the cat\n is gray.
/nthis dog \ris grey\n
that cat is gray/n
/r/nthat cat is gray/n
```

Listing 2.13 displays the contents of the bash script `linefeeds.sh`, which contains various REs that match some of the lines in `linefeeds.txt`.

LISTING 2.13: linefeeds.sh

```
echo 'line 1: egrep \/n.*'
egrep "\/n.*" linefeeds.txt
echo ""
echo 'line 2: egrep ^\/n.*'
egrep "^\/n.*" linefeeds.txt
echo ""
echo 'line 3: egrep .*\/n.*'
egrep ".*\/n.*" linefeeds.txt
echo ""
echo 'line 4: egrep .*\/n.$'
egrep ".*\/n$" linefeeds.txt
echo ""
echo 'line 5: egrep ^\/r.*'
egrep "^\/r.*" linefeeds.txt
echo ""
echo 'line 6: egrep \\r'       #[nothing]
egrep "\\r" linefeeds.txt
echo ""
```

```
echo 'line 7: egrep .*\\n.*'
egrep ".*\\n.*" linefeeds.txt  #[nothing]
echo ""
echo 'line 8: egrep \\n'
egrep "\\n" linefeeds.txt      #[nothing]
```

Listing 2.14 displays the contents of the output file `linefeeds.out`, which contains the output from launching the bash script linefeeds.sh.

LISTING 2.14: linefeeds.out

```
echo 'line 1: egrep \/n.*'
line 1: egrep \/n.*
\nthe dog is grey/n and the cat\n is gray.
/nthis dog \ris grey\n
that cat is gray/n
/r/nthat cat is gray/n

line 2: egrep ^\/n.*
/nthis dog \ris grey\n

line 3: egrep .*\/n.*
\nthe dog is grey/n and the cat\n is gray.
/nthis dog \ris grey\n
that cat is gray/n
/r/nthat cat is gray/n

line 4: egrep .*\/n.$
that cat is gray/n
/r/nthat cat is gray/n

line 5: egrep ^\/r.*
/r/nthat cat is gray/n

line 6: egrep \\r

line 7: egrep .*\\n.*

line 8: egrep \\n
```

As you can see, the first five REs match some of the lines in `linefeeds.txt`, whereas the last three REs do not match anything (contrary to what you might have expected).

WORKING WITH CAPTURE GROUPS

REs support the notion of "capturing" a group of characters, which occurs whenever you parenthesize a subexpression in an RE. Although it hasn't been explicitly mentioned, you have already worked with several REs that contain capture groups.

For example, the following RE does not contain a capture group:

```
^[A-Z]-\d{3}
```

However, the following RE contains a capture group that consists of one or more consecutive capital letters:

```
^([A-Z]+)-\d{3}
```

Note that the capture group consists of one or more capital letters that appear at the beginning of a line because of the ^ metacharacter. You can reference this capture group as \1. You can define nine capture groups, designated as \1 through \9. Here is another example:

```
^([A-Z]+)-(\d{3})-\d{4}
```

In the preceding code snippet, the first capture group, \1, refers to `^([A-Z]+)` and consists of one or more capital letters that appear at the beginning of a line. The second capture group, \2, refers to `-(\d{3})` and consists of three consecutive digits that appear after the first capture group (and also a hyphen). More information about capture groups is available here:

http://www.rexegg.com/regex-capture.html
http://www.rexegg.com/regex-lookarounds.html

WORKING WITH BACK REFERENCES

A back reference in REs means that an RE references a group that has been previously captured. Let's look at some examples. Listing 2.15 displays the contents of `duplicates.txt` that are referenced by the REs in this section.

LISTING 2.15: duplicates.txt

```
this this appears twice
THIS THIS APPEARS TWICE AGAIN
this has no duplicates
we're going back back to cali
```

Now consider the following RE that uses a back reference in order to detect duplicate (consecutive) words (with uppercase letters):

```
\b([A-Z]+)\s+\1\b
```

The preceding RE uses `\b` to ensure word boundaries, followed by `[A-Z]+` that matches alphabetic characters. In order to reference the occurrence of those alphabetic characters, simply enclose that subexpression in a pair of round parentheses, like this:

```
([A-Z]+)
```

Finally, use the term `\1` later in the RE in order to back reference that matched pattern.

If you are unfamiliar with back references, they might require some practice to become comfortable with them and to see when they can be advantageous. For example, the following pair of REs match the same patterns:

```
"\b([A-Z]+)\s+\1\b"
"\b([A-Z]+)\s+([A-Z]+)\b"
```

Even though both of the preceding REs produce the same result, it's arguably easier to read the first RE containing a back reference than the second RE.

Just to be sure, now let's test the preceding RE with the `egrep` utility to see if it finds duplicate (uppercase) words in the text file `duplicates.txt`:

```
egrep "\b([A-Z]+)\s+\1\b" duplicates.txt
```

The output is here:

```
THIS THIS APPEARS TWICE AGAIN
```

Now let's test the following RE that searches for duplicate words containing lowercase letters as well as uppercase letters:

```
egrep "\b([A-Za-z]+)\s+\1\b" duplicates.txt
```

The output is here, which confirms that the preceding RE is correct:

```
this this appears twice
THIS THIS APPEARS TWICE AGAIN
we're going back back to cali
```

TESTING REs: ARE THEY ALWAYS CORRECT?

The concepts in this section are applicable to other programming languages. The key point to remember: if slightly different REs generate the same output, is this due to the contents of the dataset? Phrased in a slightly different way: how do you know that your dataset contains a sufficient variety of text strings to ensure that differences in the output of slightly different REs is not due to your specific dataset?

This is an important question, and the following scenario is from an actual application. Consider a production application that has worked correctly for months with user-based input. Suddenly the application fails and, after much effort, you discover that an RE in the application does not handle a rarely encountered character sequence.

Unfortunately, each "rare" occurrence resulted in a false alarm involving the local fire department, which cost USD $10,000 each time that a fire truck arrived at the premises. After multiple false alarms, the problem was traced to an invalid RE in a Web application.

Listing 2.16 displays the contents of the shell script `similar-res.sh` that contains a collection of REs, and Listing 2.17 displays the contents of `similar-res.out` that displays the result of launching the shell script. *Note* that the file `lines3.txt` is the same as the file `lines1.txt` in Chapter 1.

LISTING 2.16: similar-res.sh

```
echo "line1:"
egrep '\bg\w+' lines3.txt
echo "line2:"
egrep 'g\w+\b' lines3.txt
echo "line3:"
egrep 'g\w+\b ' lines3.txt
echo "line4:"
egrep 'g\w+\b ' lines3.txt
echo "line5:"
egrep ' g\w+\b' lines3.txt
echo "line6:"
egrep '[^g ]g\w+\b' lines3.txt
echo "line7:"
egrep '[ ]g\w+\b' lines3.txt
echo "line8:"
egrep '[ ]g\S+\b' lines3.txt
echo "line9:"
egrep '[^ ]g\w*\b' lines3.txt
echo "line10:"
egrep '\bg\w+' lines3.txt
echo "line11:"
egrep 'g\w+\b ' lines3.txt
echo "line12:"
egrep 'g\w+\b ' lines3.txt
echo "line13:"
egrep ' g\w+\b' lines3.txt
echo "line14:"
egrep '[ ]g\w+\b' lines3.txt
echo "line15:"
egrep ' g\w+\b' lines3.txt
echo "line16:"
egrep '[^g ]g\w+\b' lines3.txt
#[empty output]
echo "line17:"
egrep '[^ ]g\w*\b' lines3.txt
```

LISTING 2.17: similar-res.txt

```
echo "line1:"
line1:
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
line2:
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
line3:
the dog is grey and the cat is gray.
this dog is grey
line4:
the dog is grey and the cat is gray.
this dog is grey
line5:
```

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
line6:
line7:
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
line8:
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
line9:
the dog is grey and the cat is gray.
this dog is grey
line10:
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
line11:
the dog is grey and the cat is gray.
this dog is grey
line12:
the dog is grey and the cat is gray.
this dog is grey
line13:
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
line14:
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
line15:
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
line16:
line17:
the dog is grey and the cat is gray.
this dog is grey
```

The meaning of every RE in Listing 2.17 has been discussed in Chapter 1 and the initial portion of Chapter 2.

Read each one carefully to determine the output, and compare your prediction with the actual output.

The code snippets are grouped in blocks, and in each block the code snippets look very similar, but they have subtle differences that require a solid understanding of metacharacters in REs.

While it's virtually impossible to check all possible combinations of characters in a text string, you need to be vigilant and test your REs on a large variety of patterns to minimize the likelihood of matching (or not matching) an "outlier" RE.

WHAT ABOUT PERFORMANCE FACTORS?

The following list contains guidelines for improving the performance of REs:

avoid greedy quantifiers
minimize backtracking
more specific is better
use the anchors ^ and $
non-capturing groups are preferable

However, keep in mind that readability is a very important factor, because that will affect the time spent on debugging and enhancing REs. If you need to balance performance versus readability, then favor greater readability (which includes good documentation) over better performance, but without taking this rule to an extreme.

Furthermore, keep in mind that most of the RE syntax was developed at a time when computing power and resources were scarce, where the largest mainframe was significantly less powerful than a modern smartphone. Most commands such as `grep` which use REs start out pretty efficient, because the underlying utilities that use them were developed when resources were much more scarce than they are today.

Normally you will not have to worry about the performance of your REs unless you are working on extremely large datasets or processing extremely large volumes of transactions in a short period of time. In most situations, you are more likely to cause a problem by having a confusing or fragile RE than you are by choosing a less efficient match solution.

CHAPTER SUMMARY

In this chapter you learned some tips for "thinking in REs", which will be helpful when you are faced with new tasks involving REs. Next you saw a variety of real-world RE applications that seem simple but turn out to be more complex when applied to the real world, such as with phone numbers and scientific notation.

Finally, you were exposed to important concepts about testing REs and some basic rules on RE performance.

CHAPTER 3

REs IN PYTHON

This chapter introduces you to REs in Python, with a mixture of code blocks and complete code samples that cover many of the topics that are discussed in Chapter 1. Since the details about metacharacters and character classes in Python are virtually identical to the information that you learned in Chapter 1, you can probably read this chapter quickly (even if you are only interested in a cursory view of Python and REs). If you are interested in learning more about Python, perhaps after you become comfortable with Python syntax, a Python Pocket Primer is available here: *https://www.amazon.com/dp/B00KGF0PJA*.

The first part of this chapter shows you how to define REs with digits and letters (uppercase as well as lowercase), and also how to use character classes in REs. You will also learn about character sets and character classes.

The second portion discusses the Python `re` module, which contains several useful methods, such as the `re.match()` method for matching groups of characters, the `re.search()` method to perform searches in character strings, and the `findAll()` method. You will also learn how to use character classes (and how to group them) in REs.

The final portion of this chapter contains an assortment of code samples, such as modifying text strings, splitting text strings with the `re.split()` method, and substituting text strings with the `re.sub()` method.

There are several points to keep in mind before you read this chapter and after you have installed Python on your machine. First, you need basic proficiency in Python to be comfortable with the code samples. If necessary, read some rudimentary online Python tutorials in preparation for this chapter. Second, the code samples were written for Python 2.7, and some fairly minor changes to the code samples are necessary in order to convert them to Python 3.x. A good starting point is the Python 3 documentation for REs: *https://docs.python.org/3/howto/regex.html*.

Third, this chapter contains a mixture of complete Python code samples in "py" files and code snippets (or blocks) in the Python REPL. Hence, you need some familiarity with the REPL (accessible simply by typing the word python on the command line) and how to launch Python scripts (which is also straightforward). *Note* that whenever you see the character sequence >>> it means that the code snippets have been entered manually inside the Python REPL.

If you're not sure about your level of preparation, skim through the code samples in this chapter to determine which Python features you need to learn.

WHAT ARE REs IN PYTHON?

As you know from Chapter 2, you can define REs to match characters, digits, telephone numbers, zip codes, or email addresses. The re module (added in Python 1.5) provides Perl-style RE patterns (Perl REs are discussed in an Appendix). *Note* that earlier versions of Python provided the regex module that was removed in Python 2.5. The re module provides an assortment of methods (discussed later in this chapter) for searching text strings or replacing text strings, which is similar to the basic search and/or replace functionality that is available in word processors (but usually without RE support). The re module also provides methods for splitting text strings based on REs.

Before delving into the methods in the re module, let's quickly review the various metacharacters and character classes for Python.

METACHARACTERS IN PYTHON

Python supports a set of metacharacters, most of which are the same as the metacharacters in other scripting languages such as Perl, as well as programming languages such as JavaScript and Java. The complete list of metacharacters in Python is here:

```
. ^ $ * + ? { } [ ] \ | ( )
```

The meaning of the preceding metacharacters is here:

? (matches 0 or 1): the expression ca?t matches ct or cat but not caat
* (matches 0 or more): the expression ca*t matches ct or cat or caat
+ (matches 1 or more): the expression a+ matches cat or caat but not ct
^ (beginning of line): the expression ^[a] matches the string abc (but not bc)
$ (end of line): [c]$ matches the string abc (but not cab)
. (a single dot): matches any single character (except a newline)

Sometimes you need to match the metacharacters themselves rather than their representation, which can be done in two ways. The first way involves

"escaping" their symbolic meaning with the backslash ("\") character. Thus, the sequences `\?`, `\*`, `\+`, `\^`, `\$`, and `\.` represent the literal characters instead of their symbolic meaning. You can also "escape" the backslash character with the sequence "\\". If you have two consecutive backslash characters, you need an additional backslash for each of them, which means that "\\\\" is the "escaped" sequence for "\\".

The second way is to list the metacharacters inside a pair of square brackets. For example, `[+?]` treats the two characters "+" and "?" as literal characters instead of metacharacters. The second approach is obviously more compact and less prone to error (it's easy to forget a backslash in a long sequence of metacharacters). As you might surmise, the methods in the `re` module support metacharacters.

NOTE *The "^" character that is to the left (and outside) of a sequence in square brackets (such as `^[A-Z]`) "anchors" the RE to the beginning of a line, whereas the "^" character that is the first character inside a pair of square brackets negates the RE (such as `[^A-Z]`) inside the square brackets.*

The interpretation of the "^" character in an RE depends on its location in an RE, as shown here:

- "`^[a-z]`" means any string that starts with any lowercase letter
- "`[^a-z]`" means any string that does *not* contain any lowercase letters
- "`^[^a-z]`" means any string that starts with anything *except* a lowercase letter
- "`^[a-z]$`" means a single lowercase letter
- "`^[^a-z]$`" means a single character (including digits) that is *not* a lowercase letter

As a quick preview of the `re` module that is discussed later in this chapter, the `re.sub()` method enables you to remove characters (including metacharacters) from a text string. For example, the following code snippet removes all occurrences of a forward slash ("/") and the plus sign ("+") from the variable `str`:

```
>>> import re
>>> str  = "this string has a / and + in it"
>>> str2 = re.sub("[/]+","",str)
>>> print 'original:',str
original: this string has a / and + in it
>>> print 'replaced:',str2
replaced: this string has a  and + in it
```

We can easily remove occurrences of other metacharacters in a text string by listing them inside the square brackets, just as we have done in the preceding code snippet. Listing 3.1 displays the contents of `RemoveMetaChars1.py`, which illustrates how to remove other metacharacters from a line of text.

LISTING 3.1: RemoveMetaChars1.py

```
import re

text1 = "meta characters ? and / and + and ."
text2 = re.sub("[/\.*?=+]+","",text1)

print 'text1:',text1
print 'text2:',text2
```

The RE in Listing 3.1 might seem daunting if you are new to REs, but let's demystify its contents by examining the entire expression and then the meaning of each character. First of all, the term `[/\.*?=+]` matches a forward slash ("/"), a dot (".."), a question mark ("?"), an equals sign ("="), or a plus sign ("+"). Notice that the dot "." is preceded by a backslash character "\". Doing so "escapes" the meaning of the "." metacharacter (which matches any single non-whitespace character) and treats it as a literal character.

Thus the term `[/\.*?=+]+` means "one or more occurrences of any of the metacharacters—treated as literal characters—inside the square brackets".

Consequently, the expression `re.sub("[/\.*?=+]+","",text1)` means "search the text string `text1` and replace any of the metacharacters (treated as literal characters) found with an empty string (`""`)".

The output from Listing 3.1 is here:

```
text1: meta characters ? and / and + and .
text2: meta characters  and  and  and
```

Later in this chapter you will learn about other functions in the `re` module that enable you to modify and split text strings.

CHARACTER SETS IN PYTHON

A single digit in base 10 is a number between 0 and 9 inclusive, which is represented by the sequence `[0-9]`. Similarly, a lowercase letter can be any letter between `a` and `z`, which is represented by the sequence `[a-z]`. An uppercase letter can be any letter between `A` and `Z`, which is represented by the sequence `[A-Z]`.

The following code snippets illustrate how to specify sequences of digits and sequences of character strings using a short-hand notation that is much simpler than specifying every matching digit:

`[0-9]` matches a single digit
`[0-9][0-9]` matches two consecutive digits
`[0-9]{3}` matches three consecutive digits
`[0-9]{2,4}` matches two, three, or four consecutive digits
`[0-9]{5,}` matches five or more consecutive digits
`^[0-9]+$` matches a string consisting solely of digits

You can define similar patterns using uppercase or lowercase letters in a way that is much simpler than explicitly specifying every lowercase letter or every uppercase letter:

`[A-Z][a-z]` matches a single uppercase letter that is followed by one lowercase letter
`[a-zA-Z]` matches any upper- or lowercase letter

Working with "^" and "\" Metacharacters

The purpose of the "^" character depends on its context in an RE. For example, the following expression matches a text string that starts with a digit:

```
^[0-9].
```

However, the following expression matches a text string that does *not* start with a digit because of the "^" metacharacter that is at the beginning of the expression in square brackets as well as the "^" metacharacter that is to the left (and outside) the expression in square brackets (which you learned in a previous *note*):

```
^[^0-9]
```

Thus, the "^" character inside a pair of matching square brackets ("[]") negates the expression immediately to its right that is also located inside the square brackets.

The backslash ("\") allows you to "escape" the meaning of a metacharacter. Consequently, a dot "." matches a single character (except for whitespace characters), whereas the sequence "\." matches the dot "." character.

Other examples involving the backslash metacharacter are here:

`\.H.*` matches the string `.Hello`
`H.*` matches the string `Hello`
`H.*\.` matches the string `Hello.`
`.ell.` matches the string `Hello`
`.*` matches the string `Hello`
`\..*` matches the string `.Hello`

CHARACTER CLASSES IN PYTHON

Character classes are convenient expressions that are shorter and simpler than their "bare" counterparts that you saw in the previous section. Some convenient character sequences that express patterns of digits and letters are as follows:

`\d` matches a single digit
`\w` matches a single character (digit or letter)
`\s` matches a single whitespace (space, newline, return, or tab)

\b matches a boundary between a word and a non-word
\n, \r, \t represent a newline, a return, and a tab, respectively
\ "escapes" any character

Based on the preceding definitions, \d+ matches one or more digits and \w+ matches one or more characters, both of which are more compact expressions than using character sets. In addition, we can reformulate the expressions in the previous section:

\d is the same as [0-9] and \D is the same as [^0-9]
\s is the same as [\t\n\r\f\v] and it matches any non-whitespace character, whereas \S is the opposite (it matches [^ \t\n\r\f\v])
\w is the same as [a-zA-Z0-9_] and it matches any alphanumeric character, whereas \W is the opposite (it matches [^a-zA-Z0-9_])

Additional examples are here:

\d{2} is the same as [0-9][0-9]
\d{3} is the same as [0-9]{3}
\d{2,4} is the same as [0-9]{2,4}
\d{5,} is the same as [0-9]{5,}
^\d+$ is the same as ^[0-9]+$

The curly braces ("{}") are called quantifiers, and they specify the number (or range) of characters in the expressions that precede them.

MATCHING CHARACTER CLASSES WITH THE RE MODULE

The re module provides the following methods for matching and searching one or more occurrences of an RE in a text string:

match(): Determine if the RE matches at the *beginning* of the string.
search(): Scan through a string, looking for *any* location where the RE matches.
findall(): Find *all* substrings where the RE matches and return them as a list.
finditer(): Find all substrings where the RE matches and return them as an iterator.

NOTE *The match() function only matches patterns at the start of a string.*

The two methods match() and search() are discussed in this chapter, and you can read online documentation regarding the Python findall() and finditer() methods. The next section shows you how to use the match() function in the Python re module.

USING THE `re.match()` METHOD

The `re.match()` method attempts to match RE patterns in a text string (with optional flags), and it has the following syntax:

```
re.match(pattern, string, flags=0)
```

The `pattern` parameter is the RE that you want to match in the `string` parameter. The flags parameter allows you to specify multiple flags using the bitwise OR operator that is represented by the pipe "|" symbol.

NOTE *The `re.match()` method only matches patterns from the start of a text string, which is different from the `re.search()` method discussed later in this chapter. In addition, `re.match(RE)` is similar to `re.search(^RE)`, except that the `^` metacharacter applies to each line, where `re.match()` starts only with the start of the block of text.*

The `re.match()` method returns a match object on success and `None` on failure. Use the `group(num)` or `groups()` function of the match object to get a matched expression.

`group(num=0)`: This method returns the entire match (or specific subgroup `num`).
`groups()`: This method returns all matching subgroups in a tuple (empty if there weren't any).

The following code block illustrates how to use the `group()` function in REs:

```
>>> import re
>>> p = re.compile('(a(b)c)de')
>>> m = p.match('abcde')
>>> m.group(0)
'abcde'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

Notice that the higher numbers inside the `group()` method match more deeply nested expressions that are specified in the initial RE.

Listing 3.2 displays the contents of `MatchGroup1.py`, which illustrates how to use the `group()` function to match an alphanumeric text string and an alphabetic string.

LISTING 3.2: MatchGroup1.py

```
import re

line1 = 'abcd123'
line2 = 'abcdefg'
mixed = re.compile(r"^[a-z0-9]{5,7}$")
```

```
line3 = mixed.match(line1)
line4 = mixed.match(line2)

print 'line1:',line1
print 'line2:',line2
print 'line3:',line3.group(0)
print 'line4:',line4
print 'line5:',line4.group(0)

line6 = 'a1b2c3d4e5f6g7'
mixed2 = re.compile(r"^([a-z]+[0-9]+){5,7}$")
line7 = mixed2.match(line6)

print 'line6:',line6
print 'line7:',line7.group(0)
print 'line8:',line7.group(1)

line9 = 'abc123fgh4567'
mixed3 = re.compile(r"^([a-z]*[0-9]*){5,7}$")
line10 = mixed3.match(line9)
print 'line9:',line9
print 'line10:',line10.group(0)
```

The output from Listing 3.2 is here:

```
line1: abcd123
line2: abcdefg
line3: <_sre.SRE_Match object at 0x100485440>
line4: <_sre.SRE_Match object at 0x1004854a8>
line5: abcdefg
line6: a1b2c3d4e5f6g7
line7: a1b2c3d4e5f6g7
line8: g7
line9: abc123fgh4567
line10: abc123fgh4567
```

Notice that `line3` and `line7` involve two similar but different REs. The variable `mixed` specifies a sequence of lowercase letters followed by digits, where the length of the text string is also between 5 and 7. The string `'abcd123'` satisfies all of these conditions.

On the other hand, `mixed2` specifies a pattern consisting of one or more pairs, where each pair contains one or more lowercase letters followed by one or more digits, where the length of the matching pairs is also between 5 and 7. In this case, the string `'abcd123'` as well as the string `'a1b2c3d4e5f6g7'` both satisfy these criteria.

The third RE `mixed3` specifies a pair such that each pair consists of zero or more occurrences of lowercase letters and zero or more occurrences of a digit, and also that the number of such pairs is between 5 and 7. As you can see from the output, the RE in `mixed3` matches lowercase letters and digits in any order.

In the preceding example, the RE specified a range for the length of the string, which involves a lower limit of 5 and an upper limit of 7. However, you

can also specify a lower limit without an upper limit (or an upper limit without a lower limit).

The following RE `mixed4` specifies lowercase letters, and requires a match of five, six, or seven such characters:

```
mixed4 = re.compile(r"^[a-z]{5,7}$")
line11 = mixed4.match(line1)
print 'line11:',line11
```

Since `line1` only contains four lowercase letters, there is no match, and in this case the output is `None`, as shown here:

```
line11: None
```

Listing 3.3 displays the contents of `MatchGroup2.py`, which illustrates how to use an RE and the `group()` function to match an alphanumeric text string and an alphabetic string.

LISTING 3.3: MatchGroup2.py

```
import re

alphas = re.compile(r"^[abcde]{5,}")
line1 = alphas.match("abcde").group(0)
line2 = alphas.match("edcba").group(0)
line3 = alphas.match("acbedf").group(0)
line4 = alphas.match("abcdefghi").group(0)
line5 = alphas.match("abcdefghi abcdef")

print 'line1:',line1
print 'line2:',line2
print 'line3:',line3
print 'line4:',line4
print 'line5:',line5
```

Listing 3.3 initializes the variable `alphas` as an RE that matches any string that starts with one of the letters `a` through `e` and consists of at least five characters. The next portion of Listing 3.3 initializes the four variables `line1`, `line2`, `line3`, and `line4` by means of the `alphas` RE that is applied to various text strings. These four variables are set to the first matching group by means of the expression `group(0)`.

The output from Listing 3.3 is here:

```
line1: abcde
line2: edcba
line3: acbed
line4: abcde
line5: <_sre.SRE_Match object at 0x1004854a8>
```

Unlike the first four output lines, the output from `line5` fails the match simply because `.group(0)` was not specified in the definition of `line5`.

Listing 3.4 displays the contents of `MatchGroup3.py`, which illustrates how to use an RE with the `group()` function to match words in a text string.

LISTING 3.4: MatchGroup3.py

```
import re

line = "Giraffes are taller than elephants";

matchObj = re.match( r'(.*) are(\.*)', line, re.M|re.I)

if matchObj:
   print "matchObj.group()  : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "matchObj does not match line:", line
```

The code in Listing 3.4 produces the following output:

```
matchObj.group()  :  Giraffes are
matchObj.group(1) :  Giraffes
matchObj.group(2) :
```

Listing 3.4 contains a pair of delimiters separated by a pipe ("|") symbol. The first delimiter is `re.M` for "multi-line" (this example contains only a single line of text), and the second delimiter `re.I` means "ignore case" during the pattern matching operation.

Capture Groups

You have already seen examples of capture groups, such as `matchObj.group(1)` and `matchObj.group(2)`, in the preceding section. The groups contain the matched values, and the integer in the parentheses specifies different capture groups.

Specifically, `match.group(0)` returns the fully matched string, whereas `match.group(1)`, `match.group(2)`, and so forth will return the capture groups, from left to right, in the input string. In addition, `match.group()` is the same as `match.group(0)`.

Capture groups are powerful and can become quite complex, in part because a matching group can be a substring of an enclosing matching group, similar to the way that "back references" work with the `sed` utility (discussed in Chapter 5). If you want to learn more about capture groups, perform an Internet search where you can find some examples of highly complex capture groups.

OPTIONS FOR THE `RE.MATCH()` METHOD

The `match()` method supports various optional modifiers that affect the type of matching that will be performed. As you saw in the previous example, you can also specify multiple modifiers separated by the OR ("|") symbol. Additional modifiers that are available for RE are shown here:

`re.I` performs case-insensitive matches (see previous section)
`re.L` interprets words according to the current locale

`re.M` makes $ match the end of a line and makes ^ match the start of any line
`re.S` makes a period (".") match any character (including a newline)
`re.U` interprets letters according to the Unicode character set

Experiment with these modifiers by writing Python code that uses them in conjunction with different text strings.

MATCHING CHARACTER CLASSES WITH THE `RE.SEARCH()` METHOD

As you saw earlier in this chapter, the `re.match()` method only matches from the beginning of a string, whereas the `re.search()` method can successfully match a substring anywhere in a text string.

The `re.search()` method takes two arguments, an RE pattern and a string, and then searches for the specified pattern in the given string. The `search()` method returns a `match` object (if the search was successful) or `None`. As a simple example, the following searches for the pattern `tasty` followed by a five-letter word:

```
import re

str = 'I want a tasty pizza'
match = re.search(r'tasty \w\w\w\w\w', str)

if match:
  ## 'found tasty pizza'
  print 'found', match.group()
else:
  print 'Nothing tasty here'
```

The output of the preceding code block is here:

```
found tasty pizza
```

The following code block further illustrates the difference between the `match()` method and the `search()` methods:

```
>>> import re
>>> print re.search('this', 'this is the one').span()
(0, 4)
>>>
>>> print re.search('the', 'this is the one').span()
(8, 11)
>>> print re.match('this', 'this is the one').span()
(0, 4)
>>> print re.match('the', 'this is the one').span()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'span'
```

MATCHING CHARACTER CLASSES WITH THE `FINDALL()` METHOD

Listing 3.5 displays the contents of the Python script `RegEx1.py`, which illustrates how to define simple character classes that match various text strings.

LISTING 3.5: RegEx1.py

```
import re

str1 = "123456"
matches1 = re.findall("(\d+)", str1)
print 'matches1:',matches1

str1 = "123456"
matches1 = re.findall("(\d\d\d)", str1)
print 'matches1:',matches1

str1 = "123456"
matches1 = re.findall("(\d\d)", str1)
print 'matches1:',matches1

print
str2 = "1a2b3c456"
matches2 = re.findall("(\d)", str2)
print 'matches2:',matches2

print
str2 = "1a2b3c456"
matches2 = re.findall("\d", str2)
print 'matches2:',matches2

print
str3 = "1a2b3c456"
matches3 = re.findall("(\w)", str3)
print 'matches3:',matches3
```

Listing 3.5 contains simple REs (which you have seen already) for matching digits in the variables `str1` and `str2`. The final code block of Listing 3.5 matches every character in the string `str3`, effectively "splitting" `str3` into a list where each element consists of one character.

The output from Listing 3.5 is here (notice the blank lines after the first three output lines):

```
matches1: ['123456']
matches1: ['123', '456']
matches1: ['12', '34', '56']

matches2: ['1', '2', '3', '4', '5', '6']

matches2: ['1', '2', '3', '4', '5', '6']

matches3: ['1', 'a', '2', 'b', '3', 'c', '4', '5', '6']
```

Finding Capitalized Words in a String

Listing 3.6 displays the contents of the Python script `FindCapitalized.py`, which illustrates how to define simple character classes that match various text strings.

LISTING 3.6: FindCapitalized.py

```
import re

str = "This Sentence contains Capitalized words"
caps = re.findall(r'[A-Z][\w\.-]+', str)

print 'str: ',str
print 'caps:',caps
```

Listing 3.6 initializes the string variable `str` and the `RE` `caps` that matches any word that starts with a capital letter, because the first portion of `caps` is the pattern `[A-Z]` that matches any capital letter between `A` and `Z` inclusive.

The output of Listing 3.6 is here:

```
str:  This Sentence contains Capitalized words
caps: ['This', 'Sentence', 'Capitalized']
```

ADDITIONAL MATCHING FUNCTIONS FOR REs

After invoking any of the methods `match()`, `search()`, `findAll()`, `or finditer()`, you can invoke additional methods on the "matching object". An example of this functionality using the `match()` method is here:

```
import re

p1 = re.compile('[a-z]+')
m1 = p1.match("hello")
```

In the preceding code block, the `p1` object represents the compiled RE for one or more lowercase letters, and the "matching object" `m1` object supports the following methods:

`group()` return the string matched by the RE
`start()` return the starting position of the match
`end()` return the ending position of the match
`span()` return a tuple containing the (start, end) positions of the match

As a further illustration, Listing 3.7 displays the contents of the Python script `SearchFunction1.py`, which illustrates how to use the `search()` method and the `group()` method.

LISTING 3.7: SearchFunction1.py

```
import re

line = "Giraffes are taller than elephants";
```

```
searchObj = re.search( r'(.*) are(\.*)', line, re.M|re.I)
matchObj  = re.match( r'(.*) are(\.*)', line, re.M|re.I)

if searchObj:
   print "searchObj.group()  : ", searchObj.group()
   print "searchObj.group(1) : ", searchObj.group(1)
   print "searchObj.group(2) : ", searchObj.group(2)
else:
   print "searchObj does not match line:", line

if matchObj:
   print "matchObj.group() :   ", matchObj.group()
   print "matchObj.group(1):   ", matchObj.group(1)
   print "matchObj.group(2):   ", matchObj.group(2)
else:
   print "matchObj does not match line:", line
```

Listing 3.7 contains the variable line that represents a text string, and the variable `searchObj` is an RE involving the `search()` method and pair of pipe-delimited modifiers (discussed in more detail in the next section). If `searchObj` is not null, the `if/else` conditional code in Listing 3.7 displays the contents of the three groups resulting from the successful match with the contents of the variable `line`.

The same logic applies to `matchObj`, which is based on the `re.match()` function instead of the `re.search()` function (recall the distinction that was explained earlier in the chapter).

The output from Listing 3.7 is here:

```
searchObj.group()  :  Giraffes are
searchObj.group(1) :  Giraffes
searchObj.group(2) :
matchObj.group() :    Giraffes are
matchObj.group(1):    Giraffes
matchObj.group(2):
```

GROUPING WITH CHARACTER CLASSES IN REs

In addition to the character classes that you have seen earlier in this chapter, you can specify subexpressions of character classes. Listing 3.8 displays the contents of `Grouping1.py`, which illustrates how to use the `search()` method.

LISTING 3.8: Grouping1.py

```
import re

p1 = re.compile('(ab)*')
print 'match1:',p1.match('abababababab').group()
print 'span1: ',p1.match('abababababab').span()

p2 = re.compile('(a)b')
m2 = p2.match('ab')
```

```
print 'match2:',m2.group(0)
print 'match3:',m2.group(1)
```

Since the explanation is quite lengthy, let's look at the output and then delve into the explanation. The output from Listing 3.8 is here:

```
match1: abababababab
span1:  (0, 10)
match2: ab
match3: a
```

Listing 3.8 starts by defining the RE `p1` that matches zero or more occurrences of the string `ab`. The first `print` statement displays the result of using the `match()` function of `p1` (followed by the `group()` function) against a string, and the result is a string. This illustrates the use of "method chaining", which eliminates the need for an intermediate object (as shown in the second code block). The second `print` statement displays the result of using the `match()` function of `p1`, followed by applying the `span()` function, against a string. In this case the result is a numeric range (see output below).

The second part of Listing 3.8 defines the RE `p2` that matches an optional letter `a` followed by the letter `b`. The variable `m2` invokes the `match` method on `p2` using the string `ab`. The third `print` statement displays the result of invoking `group(0)` on `m2`, and the fourth `print` statement displays the result of involving `group(1)` on `m2`. Both results are substrings of the input string `ab`. Recall that `group(0)` returns the highest level match that occurred, and `group(1)` returns a more "specific" match that occurred, such as one that involves the parentheses in the definition of `p2`. The higher the value of the integer in the expression `group(n)`, the more specific the match.

USING CHARACTER CLASSES IN REs

This section contains some examples that illustrate how to use character classes to match various strings and also how to use delimiters in order to split a text string. For example, one common date string involves a date format of the form MM/DD/YY. Another common scenario involves records with a delimiter that separates multiple fields. Usually such records contain one delimiter, but as you will see, Python makes it very easy to split records using multiple delimiters.

Matching Strings with Multiple Consecutive Digits

Listing 3.9 displays the contents of the Python script `MatchPatterns1.py`, which illustrates how to define simple REs in order to split the contents of a text string based on the occurrence of one or more consecutive digits.

Although the REs `\d+/\d+/\d+` and `\d\d/\d\d/\d\d\d\d` both match the string `08/13/2014`, the first RE matches more patterns than the second RE which is an "exact match" with respect to the number of matching digits that are allowed.

LISTING 3.9: MatchPatterns1.py

```
import re

date1 = '02/28/2013'
date2 = 'February 28, 2013'

# Simple matching: \d+ means match one or more digits
if re.match(r'\d+/\d+/\d+', date1):
  print('date1 matches this pattern')
else:
  print('date1 does not match this pattern')

if re.match(r'\d+/\d+/\d+', date2):
  print('date2 matches this pattern')
else:
  print('date2 does not match this pattern')
```

The output from launching Listing 3.9 is here:

```
date1 matches this pattern
date2 does not match this pattern
```

Reversing Words in Strings

Listing 3.10 displays the contents of the Python script `ReverseWords1.py`, which illustrates how to reverse a pair of words in a string.

LISTING 3.10: ReverseWords1.py

```
import re

str1 = 'one two'
match = re.search('([\w.-]+) ([\w.-]+)', str1)

str2 = match.group(2) + ' ' + match.group(1)
print 'str1:',str1
print 'str2:',str2
```

The output from Listing 3.10 is here:

```
str1: one two
str2: two one
```

Now that you understand how to define REs for digits and letters, let's look at some more sophisticated REs. For example, the following expression matches a string that is any combination of digits, uppercase letters, or lowercase letters (i.e., no special characters):

```
^[a-zA-Z0-9]$
```

Here is the same expression rewritten using character classes:

```
^[\w\W\d]$
```

MODIFYING TEXT STRINGS WITH THE RE MODULE

The Python re module contains several methods for modifying strings. The split() method uses an RE to "split" a string into a list. The sub() method finds all substrings where the RE matches, and then replaces them with a different string. The subn() method performs the same functionality as sub(), and also returns the new string and the number of replacements. The following subsections contain examples that illustrate how to use the functions split(), sub(), and subn() in REs.

The subn() method returns a count of the number of matches of an RE in a given string, an example of which is here:

```
import re

pizzare = re.compile(r'pizza', re.IGNORECASE)
comment = 'hot pizza or cold pizza: both good'
ignoreme, count = pizzare.subn('', comment)
print 'Found', count, 'occurrences of "pizza"'
```

The variable count is populated with the number of occurrences of the string pizza in the string comment. Launch the preceding code block and you will see the following output:

```
Found 2 occurrences of "pizza"
```

SPLITTING TEXT STRINGS WITH THE RE.SPLIT() METHOD

Listing 3.11 displays the contents of the Python script RegEx2.py, which illustrates how to define simple REs in order to split the contents of a text string.

LISTING 3.11: RegEx2.py

```
import re

line1 = "abc def"
result1 = re.split(r'[\s]', line1)
print 'result1:',result1

line2 = "abc1,abc2:abc3;abc4"
result2 = re.split(r'[,:;]', line2)
print 'result2:',result2

line3 = "abc1,abc2:abc3;abc4 123 456"
result3 = re.split(r'[,:;\s]', line3)
print 'result3:',result3
```

Listing 3.11 contains three blocks of code, each of which uses the split() method in the re module in order to tokenize three different strings. The first RE specifies a whitespace, the second RE specifies three punctuation characters, and the third RE specifies the combination of the first two REs.

The output from launching `RegEx2.py` is here:

```
result1: ['abc', 'def']
result2: ['abc1', 'abc2', 'abc3', 'abc4']
result3: ['abc1', 'abc2', 'abc3', 'abc4', '123', '456']
```

SPLITTING TEXT STRINGS USING DIGITS AND DELIMITERS

Listing 3.12 displays the contents of `SplitCharClass1.py`, which illustrates how to use REs consisting of a character class, the "." character, and a whitespace to split the contents of two text strings.

LISTING 3.12: SplitCharClass1.py

```
import re

line1 = '1. Section one 2. Section two 3. Section three'
line2 = '11. Section eleven 12. Section twelve 13. Section
thirteen'

print re.split(r'\d+\. ', line1)
print re.split(r'\d+\. ', line2)
```

Listing 3.12 contains two text strings that can be split using the same RE `'\d+\. '`. *Note* that if you use the expression `'\d\. '` only the first text string will split correctly.

The result of launching Listing 3.12 is here:

```
['', 'Section one ', 'Section two ', 'Section three']
['', 'Section eleven ', 'Section twelve ', 'Section thirteen']
```

SUBSTITUTING TEXT STRINGS WITH THE `RE.SUB()` METHOD

Earlier in this chapter you saw a preview of using the `sub()` method to remove all the metacharacters in a text string. The following code block illustrates how to use the `re.sub()` method to substitute alphabetic characters in a text string.

```
>>> import re
>>> p = re.compile( '(one|two|three)')
>>> p.sub( 'some', 'one book two books three books')
'some book some books some books'
>>>
>>> p.sub( 'some', 'one book two books three books', count=1)
'some book two books three books'
```

The following code block uses the `re.sub()` method in order to insert a line feed after each alphabetic character in a text string:

```
>>> line = 'abcde'
>>> line2 = re.sub('', '\n', line)
>>> print 'line2:',line2
```

```
line2:
a
b
c
d
e
```

Now consider the following example that illustrates how to use the Python `subn()` function with a text string:

```
line = 'abcde'
linere = re.compile(r'', re.IGNORECASE)
line3 = linere.subn('', line)
print 'line3:',line3
```

The output from launching the preceding Python code block is here:

```
line3: ('abcde', 6)
```

MATCHING THE BEGINNING AND THE END OF TEXT STRINGS

Listing 3.13 displays the contents of the Python script `RegEx3.py`, which illustrates how to find substrings using the `startswith()` function and `endswith()` function.

LISTING 3.13: RegEx3.py

```
import re

line2 = "abc1,Abc2:def3;Def4"
result2 = re.split(r'[,:;]', line2)

for w in result2:
  if(w.startswith('Abc')):
    print 'Word starts with Abc:',w
  elif(w.endswith('4')):
    print 'Word ends with 4:',w
  else:
    print 'Word:',w
```

Listing 3.13 starts by initializing the string `line2` (with punctuation characters as word delimiters) and the RE `result2` that uses the `split()` function with a comma, colon, and semicolon as "split delimiters" in order to tokenize the string variable `line2`.

The output after launching Listing 3.13 is here:

```
Word: abc1
Word starts with Abc: Abc2
Word: def3
Word ends with 4: Def4
```

Listing 3.14 displays the contents of the Python script `MatchLines1.py`, which illustrates how to find substrings using character classes.

LISTING 3.14: MatchLines1.py

```
import re

line1 = "abcdef"
line2 = "123,abc1,abc2,abc3"
line3 = "abc1,abc2,123,456f"

if re.match("^[A-Za-z]*$", line1):
  print 'line1 contains only letters:',line1

# better than the preceding snippet:
if line1[:-1].isalpha():
  print 'line1 contains only letters:',line1

if re.match("^[\w]*$", line1):
  print 'line1 contains only letters:',line1

if re.match(r"^[^\W\d_]+$", line1, re.LOCALE):
  print 'line1 contains only letters:',line1
print

if re.match("^[0-9][0-9][0-9]", line2):
  print 'line2 starts with 3 digits:',line2

if re.match("^\d\d\d", line2):
  print 'line2 starts with 3 digits:',line2
print

# does not work: fixme
if re.match("[0-9][0-9][0-9][a-z]$", line3):
  print 'line3 ends with 3 digits and 1 char:',line3

# does not work: fixme
if re.match("[a-z]$", line3):
  print 'line3 ends with 1 char:',line3
```

Listing 3.14 starts by initializing three string variables `line1`, `line2`, and `line3`. The first RE contains an expression that matches any line containing uppercase or lowercase letters (or both):

```
if re.match("^[A-Za-z]*$", line1):
```

The following snippet also tests for the same thing:

```
line1[:-1].isalpha()
```

The preceding snippet starts from the rightmost position of the string and checks if each character is alphabetic.

The next snippet checks if `line1` can be tokenized into words (a word contains only alphabetic characters):

```
if re.match("^[\w]*$", line1):
```

The next portion of Listing 3.14 checks if a string contains three consecutive digits:

```
if re.match("^[0-9][0-9][0-9]", line2):
  print 'line2 starts with 3 digits:',line2

if re.match("^\d\d\d", line2):
```

The first snippet uses the pattern `[0-9]` to match a digit, whereas the second snippet uses the expression `\d` to match a digit.

The output from Listing 3.14 is here:

```
line1 contains only letters: abcdef
line1 contains only letters: abcdef
line1 contains only letters: abcdef
line1 contains only letters: abcdef

line2 starts with 3 digits: 123,abc1,abc2,abc3
line2 starts with 3 digits: 123,abc1,abc2,abc3
```

COMPILATION FLAGS

Compilation flags modify the manner in which REs work. Flags are available in the RE module as a long name (such as `IGNORECASE`) and a short, one-letter form (such as `I`). The short form is the same as the flags in pattern modifiers in Perl. You can specify multiple flags by using the "|" symbol. For example, `re.I | re.M` sets both the `I` and `M` flags.

You can check the online Python documentation regarding all the available compilation flags in Python.

COMPOUND REs

Listing 3.15 displays the contents of `MatchMixedCase1.py`, which illustrates how to use the pipe ("|") symbol to specify two REs in the same `match()` function.

LISTING 3.15: MatchMixedCase1.py

```
import re

line1 = "This is a line"
line2 = "That is a line"

if re.match("^[Tt]his", line1):
  print 'line1 starts with This or this:'
  print line1
else:
  print 'no match'

if re.match("^This|That", line2):
  print 'line2 starts with This or That:'
  print line2
```

```
else:
  print 'no match'
```

Listing 3.15 starts with two string variables `line1` and `line2`, followed by an if/else conditional code block that checks if `line1` starts with the RE `[Tt]his`, which matches the string `This` as well as the string `this`.

The second conditional code block checks if `line2` starts with the string `This` or the string `That`. Notice the "^" metacharacter, which in this context anchors the RE to the beginning of the string. The output from Listing 3.15 is here:

```
line1 starts with This or this:
This is a line
line2 starts with This or That:
That is a line
```

COUNTING CHARACTER TYPES IN A STRING

You can use an RE to check whether a character is a digit, a letter, or some other type of character. Listing 3.16 displays the contents of `CountDigitsAndChars.py`, which performs this task.

LISTING 3.16: CountDigitsAndChars.py

```
import re

charCount  = 0
digitCount = 0
otherCount = 0

line1 = "A line with numbers: 12 345"

for ch in line1:
   if(re.match(r'\d', ch)):
     digitCount = digitCount + 1
   elif(re.match(r'\w', ch)):
     charCount = charCount + 1
   else:
     otherCount = otherCount + 1

print 'charcount:',charCount
print 'digitcount:',digitCount
print 'othercount:',otherCount
```

Listing 3.16 initializes three numeric counter-related variables, followed by the string variable `line1`. The next part of Listing 3.16 contains a `for` loop that processes each character in the string `line1`. The body of the `for` loop contains a conditional code block that checks whether the current character is a digit, a letter, or some other non-alphanumeric character. Each time there is a successful match, the corresponding "counter" variable is incremented.

The output from Listing 3.16 is here:

```
charcount: 16
digitcount: 5
othercount: 6
```

REs AND GROUPING

You can also "group" subexpressions and even refer to them symbolically. For example, the following expression matches zero or one occurrences of three consecutive letters or digits:

```
^([a-zA-Z0-9]{3,3})?
```

The following expression matches a telephone number (such as `650-555-1212`) in the United States:

```
^\d{3,3}[-]\d{3,3}[-]\d{4,4}
```

The following expression matches a zip code (such as `67827` or `94343-04005`) in the United States:

```
^\d{5,5}([-]\d{5,5})?
```

The following code block partially matches an email address:

```
str = 'john.doe@google.com'
  match = re.search(r'\w+@\w+', str)
  if match:
    print match.group()  ## 'doe@google'
```

Exercise: use the preceding code block as a starting point in order to define an RE for email addresses.

As you saw in Chapter 2, most email checks are fairly simple in production code: at least one character, followed by an `@` symbol, at least one more character, followed by a period, and at least one character after the period. Such checks are obviously minimalistic, and they cannot prove that the email address is real.

SIMPLE STRING MATCHES

Listing 3.17 displays the contents of the Python script `RegEx4.py`, which illustrates how to define REs that match various text strings.

LISTING 3.17: RegEx4.py

```
import re

searchString = "Testing pattern matches"

expr1 = re.compile( r"Test" )
expr2 = re.compile( r"^Test" )
```

```
expr3 = re.compile( r"Test$" )
expr4 = re.compile( r"\b\w*es\b" )
expr5 = re.compile( r"t[aeiou]", re.I )

if expr1.search( searchString ):
   print '"Test" was found.'

if expr2.match( searchString ):
   print '"Test" was found at the beginning of the line.'

if expr3.match( searchString ):
   print '"Test" was found at the end of the line.'

result = expr4.findall( searchString )

if result:
   print 'There are %d words(s) ending in "es":' % \
      ( len( result ) ),

   for item in result:
      print " " + item,

print

result = expr5.findall( searchString )
if result:
   print 'The letter t, followed by a vowel, occurs %d times:' % \
      ( len( result ) ),

   for item in result:
      print " "+item,

print
```

Listing 3.17 starts with the variable `searchString` that specifies a text string, followed by the REs `expr1`, `expr2`, `expr3`. The RE `expr1` matches the string `Test` that occurs anywhere in `searchString`, whereas `expr2` matches `Test` if it occurs at the beginning of `searchString`, and `expr3` matches `Test` if it occurs at the end of `searchString`. The RE `expr` matches words that end in the letters `es`, and the RE `expr5` matches the letter `t` followed by a vowel.

The output from Listing 3.17 is here:

```
"Test" was found.
"Test" was found at the beginning of the line.
There are 1 words(s) ending in "es":  matches
The letter t, followed by a vowel, occurs 3 times:  Te  ti  te
```

Keep in mind that `re.match()` checks for a match at the beginning of the string, whereas `re.search()` checks for a match that occurs anywhere in the string. The code samples contain a mixture of both methods to show you how the results differ (and also to avoid favoring one method over the other method).

ADDITIONAL TOPICS FOR REs

In addition to the Python-based search/replace functionality that you have seen in this chapter, you can also perform a greedy search and substitution. Perform an Internet search to learn what these features are and how to use them in Python code.

CHAPTER SUMMARY

This chapter showed you how to create various types of REs. First you learned how to define primitive REs using sequences of digits, lowercase letters, and uppercase letters. Next you learned how to use character classes, which are more convenient and simpler expressions that can perform the same functionality. You also learned how to use the Python `re` library in order to compile REs and then use them to see if they match substrings of text strings.

CHAPTER 4

WORKING WITH REs IN R

This chapter introduces you to REs in R, which are used from a statistical viewpoint to solve tasks for data scientists. Keep in mind that basic familiarity with standard data types in R is required for this chapter, such as creating string vectors, vectors of sentences, and data frames. This chapter shows you how to use REs in some R-specific commands, thereby enhancing your knowledge of R. When you have finished this chapter, you will have enough knowledge to convert the code samples in the first two chapters to their R counterparts.

The first section of this chapter contains a summary of rules for metacharacters in R, an overview of search functions in R, as well an explanation of `grep`-related commands in R. The second section of this chapter contains basic examples of REs in R, which are similar to approximately 25% of Chapter 1. The final section of this chapter contains a collection of one-line REs in R that use some of the R commands that are discussed in the second section.

One recommendation: download and install RStudio for your platform and use RStudio to test the REs in this chapter. RStudio is an extremely powerful code development environment, and a must-learn tool if you plan to work extensively in R.

METACHARACTERS AND CHARACTER CLASSES IN R

In most cases, metacharacters in R have the same meaning as they do in bash, Python, or Perl, but there are situations where metacharacters in R behave differently. For instance, metacharacters are escaped via a double backslash "\\" in order to match them as regular characters. On the other hand, metacharacters are treated as normal characters when they are included in character classes if they also appear with the characters], ^, -, or \.

Here are the rules that specify how to match metacharacters as regular characters when they are included in a character class:

Matching a "," inside a character class: place it in first position
Matching a "-" inside a character class: place it in either first or last position
Matching a "^" inside a character class: place it anywhere (except first position)
Matching any other character or metacharacter (excluding \) inside a character class: place it anywhere.

The \ character is still a metacharacter inside a character class; however, a double backslash "\\" will match a \ character.

This section is a bit terse, some examples of precise syntax would help. At least one to give the general idea of what you are talking about to those who are new to the character class and metacharacter terms. It reinforces the earlier chapters. I recommend showing a "normal" and "r specific" syntax on one of the above cases to illustrate the point, then list the other specifics as you've done in the last couple paragraphs.

SEARCH FUNCTIONS IN R

R provides several functions for string searches, such as `grep()`, `gsub()`, and `strsplit()`, that are discussed later in this chapter. R interprets some of the arguments of these functions (usually the first one) as REs. We'll look at some of these R functions, which enable you to perform string searches (or matching) and modification tasks.

The Unix `grep` command and the `grep` command in R have a superficial resemblance. In general, Unix `grep` is used to match an RE against one or more text strings in a text file, and the output (if any) displays the lines that match the RE. Thus, Unix `grep` has a "line-oriented" philosophy.

On the other hand, `grep` in the R environment has an "element-oriented" philosophy, where an element refers to an element of a vector or a data frame. In particular, `grep` in R can check if an RE matches the strings in a vector or data frame and then returns the matching strings if the `value` argument equals `TRUE`. By contrast, if the `value` argument equals `FALSE`, then the index of strings that match the RE (if any) are displayed.

What happens if an RE does not match a string vector? Although you might expect a return value of -1, sometimes there is no index value returned (i.e., when the `value` argument equals `FALSE`), which means there is no convenient way to determine that no match occurred. If you need a list of which strings matched and which did not match, use the `grepl` function, which is discussed later in this chapter.

Perl RE Support in R

REs in R are usually restricted, and the inline help functionality does not provide extensive information about many topics. What an individual

command supports depends on who wrote it and what they chose to implement, which means behavior is more variable than something like Java, which was developed commercially by a single development team. On the plus side, R functions can correctly interpret Perl RE syntax when `perl=TRUE` is supported by the command and specified by the user. For your convenience, the Appendix contains examples of one-line Perl REs. In addition, the RE syntax in Python bears some resemblance to REs in R. All of the commands discussed in this chapter support `perl=TRUE` and share common RE behavior if `perl=FALSE`.

THE `GREP` COMMAND IN R

The previous section gave you a high-level description of the modus operandi of the Unix `grep command` versus the `grep` command in R. This section provides a deeper explanation and some examples that illustrate how to work with `grep` in R. In particular, this section briefly discusses the commands `grepl`, `regexpr`, and `gregexpr`, which also have `grep`-like functionality.

Here's a simple example of the `grep` command in R (an explanation is provided later):

```
>x<-c("abc","bcd","cde","def")
>grep("bc",x)
```

The preceding `grep` command matches the RE `bc` in position 1 and position 2 of the vector `x`, so the output is as follows:

```
[1] 1 2
```

The `grep` command in R requires an RE and the input vector as the first and second arguments, respectively. If you specify `value=FALSE` or omit the `value` parameter, then `grep` returns a new vector with the indexes of the elements in the input vector that partially or fully matched the RE. On the other hand, if you specify `value=TRUE`, then `grep` returns a vector with copies of the actual elements in the input vector that partially or fully matched.

For example, the following `grep` command in R matches the RE `a+` (one or more occurrences of the letter `a`) with the elements of a string vector:

```
grep("a+", c("abc", "def", "cba a", "aa"), perl=TRUE, value=FALSE)
```

The output displays the matching indexes (because `value=FALSE`):

```
[1] 1     3       4
```

The next version of the preceding `grep` command specifies `value=TRUE`:

```
grep("a+", c("abc", "def", "cba a", "aa"), perl=TRUE, value=TRUE)
```

The output displays the matching elements (because `value=TRUE`):

```
[1] "abc" "cba a" "aa"
```

The grepl Command in R

The grep command returns a list of indexes (discussed in the previous section), whereas the grepl command returns a list of Boolean values. As an illustration, here is the grepl counterpart to the first grep command in the previous section:

```
> grepl("bc",x)
[1]  TRUE  TRUE FALSE FALSE
```

The grepl function takes the same arguments as the grep function, except for the value argument, which is not supported. The grepl function returns a logical vector with the same length as the input vector. Each element in the returned vector is a Boolean value that indicates whether or not the RE found a match in the corresponding string element in the input vector.

As a simple illustration, the following grepl command specifies the same RE and vector of strings that you saw in the previous section, and also specifies perl=TRUE:

```
grepl("a+", c("abc", "def", "cba a", "aa"), perl=TRUE)
```

Here is the output:

```
[1] TRUE  FALSE TRUE  TRUE
```

The regexpr Command in R

The regexpr command in R returns the index of the first occurrence of a matched pattern in a string, as shown here:

```
>z<-"tomatoes"
>regexpr("o",z)
[1] 2
attr(,"match.length")
[1] 1
attr(,"useBytes")
[1] TRUE
```

The letter o appears in position 2 in the string tomatoes, which is the output of the preceding regexpr() command. The attr command displays the length of the matching string and whether or not useBytes was used. Too much in one gulp here. See below

The regexpr function in R also takes the same arguments as the grepl function in R. However, the regexpr function returns an integer vector with the same length as the input vector. Each element in the returned vector indicates the character position in each corresponding string element in the input vector at which the (first) RE match was found. In the preceding regexpr() example, the letter o appears in the second position in the string tomatoes, so the output is [1] 2.

A match at the beginning of the string is indicated with character position 1. If the RE did not find a match in a string, its corresponding element in the result vector is -1.

In addition, the returned vector also has a `match.length` attribute. This is another integer vector with the number of characters in the (first) RE match in each string, or -1 for strings that did not match the RE.

The `gregexpr` Command in R

The `gregexpr` function in R is similar to the `regexpr` function in R, except that it finds all matches in each string. The `gregexpr` function returns a vector with the same length as the input vector. Each element is another vector, with one element for each match found in the string indicating the character position where that match was found. Each vector element in the returned vector also has a `match.length` attribute with the lengths of all matches. If no matches could be found in a particular string, the element in the returned vector is still a vector, but with just one element -1.

For example, the following `regexpr` function matches the RE `a+` with a string vector (with `perl=TRUE`):

```
regexpr("a+", c("abc", "def", "cba a", "aa"), perl=TRUE)
```

The output is here:

```
[1]  1 -1  3  1
attr(,"match.length")
[1]  1 -1  1  2
attr(,"useBytes")
[1] TRUE
```

The following example of the `gregexpr` command in R shows how the command organizes the data output into a list structure instead of a `regexpr` vector structure:

```
gregexpr("a+", c("abc", "def", "cba a", "aa"), perl=TRUE)
```

The output from the preceding code snippet is here:

```
[[1]]
[1] 1
attr(,"match.length")
[1] 1
attr(,"useBytes")
[1] TRUE

[[2]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"useBytes")
[1] TRUE
```

```
[[3]]
[1] 3 5
attr(,"match.length")
[1] 1 1
attr(,"useBytes")
[1] TRUE

[[4]]
[1] 1
attr(,"match.length")
[1] 2
attr(,"useBytes")
[1] TRUE
```

The `regmatches` Command in R

The `regmatches` command in R returns the actual substrings that are matched by an RE, as shown here:

```
> x <- c("abc", "def", "cba a", "aa")
> m <- regexpr("a+", x, perl=TRUE)
> regmatches(x, m)
```

The output is here:

```
[1]  "a"  "a"  "aa"
```

Another example of the `regmatches` command:

```
> m <- gregexpr("a+", x, perl=TRUE)
> regmatches(x, m)
```

The output is here:

```
[[1]]
[1] "a"
[[2]]
character(0)
[[3]]
[1] "a" "a"
[[4]]
[1] "aa"
```

The first argument for the `regmatches` command is the same input that is supplied to the `regexpr` command or the `gregexpr` command. The second argument is the vector that is returned by the `regexpr` command or the list returned by the `gregexpr` command. If you pass the vector from the `regexpr` command, then `regmatches` returns a character vector with all the strings that were matched. *Note* that this vector may be shorter than the input vector if there is no match in some of the elements.

If you pass the list from the `gregexpr` command, then `regmatches` returns a vector with the same number of elements as the input list. Each output list is a character vector with all the matches of the corresponding element in

the input vector, or NULL if an element had no matches. The examples in the beginning of this section illustrate some of the preceding points.

Performing Multiple Text Substitutions on a Vector

If you combine `gregexpr` and `regmatches`, you can perform complex text substitutions on a text vector. This example shows one possibility: substituting "a" in the original vector with the values from a second list, where each element is not only different text, but sometimes a different type of object (vector with multiple elements, single element vector, a command that returns a single value). For this example the list vector (which specifies what will be substituted) must have the same number of elements as the input text vector.

```
x <- c("abc", "def", "cba a", "aa")
m <- gregexpr("a+", x, perl=TRUE)
regmatches(x, m) <- list(c("one"), character(0), c("two",
"three"), c("four"))
x
```

The result is here:

```
[1]  "onebc"       "def"          "cbtwo three" "four"
```

Notice that the second and fourth element didn't match and were unchanged. The first element replaced the "a" with the single vector value "one." In addition, the third element concatenated the vector into a single string (changing it to "two three") and then replaced the "a" with that string.

Other Useful String-Related Commands in R

While `gregexpr` and `regmatches` can be combined for complex substitutions, R has a number of specific commands to handle most commonly needed substitutions and string manipulation.

This section contains examples of the R commands `sub()`, `gsub()`, `substr()`, and `strsplit()`, which provide useful functionality when you are working with strings.

The `sub()` command substitutes the first occurrence of a pattern with a given string. If the input is a vector, it performs the substitution on each string within the vector. Here is a simple example of the `sub()` command:

```
sub("abc", "xyz", c("a","xabc","abcabc"))
```

The output is here:

```
[1] "a"   "x"   "xyz"
```

The following example uses the `sub()` command to remove everything in a string except the first instance of `bc`:

```
x<-c("abc","bcdbc","cde","def")
sub(".*(bc).*","\\1",grep("bc",x,value=TRUE))
```

The grep removes elements that don't match (the third and fourth strings), so the output vector has fewer elements than the input vector. If we remove everything except the first occurrence of bc in the first two strings, we get the following output:

```
[1] "bc" "bc"
```

Now let's look at the gsub() command that substitutes all occurrences of a pattern with a given string. By way of comparison, the sub() command is similar to find/replace, whereas gsub() is similar to find/replace all. Here is a simple example of the gsub() command:

```
gsub("abc", "xyz", c("a","xabc","abcabc"))
```

The output is here:

```
[1] "a"    "xxyz"    "xyzxyz"
```

The substr() command returns the start and stop positions of a substring in a given string:

```
> x<-"abcdefghijk"
> substr(x,5,8)
[1] "efgh"
```

The strsplit() command "splits" a string into substrings, based on another string. The following example splits a string using a "/" as a delimiter:

```
strsplit("11/03/2013","/")
[[1]]
[1] "11"    "03"    "2013"
```

Now that you have seen examples of how to use some useful string-related commands in R, let's look at how to use REs in R.

WORKING WITH REs IN R

Suppose that vect1 is a character vector (whose contents are shown as follows) and we want to check whether or not various words appear as elements in vect1:

```
vect1 = c("the","dog","is","grey","and","the","cat","is","gray")
```

As you can see, the word grey appears in the first and second lines, the word gray appears in the first and third lines, and all three lines contain either grey or gray.

Here are the tasks that we want to perform:

1. find the lines that contain grey
2. find the lines that contain gray
3. find the lines that contain either grey or gray

The following command performs the first task:

```
grep(pattern = "grey", vect1, value = TRUE)
```

The output is here:

```
"grey"
```

The following command displays the index of the occurrence of `grey` in `vect1`:

```
grep(pattern = "grey", vect1, value = FALSE)
```

The output is here:

```
4
```

The following pair of commands displays the occurrence of `grey`, `gray`, and `groy` in `vect1`, along with the index values of their positions in `vect1`:

```
grep(pattern = "grey|gray|groy", vect1, value = TRUE)
grep(pattern = "grey|gray|groy", vect1, value = FALSE)
```

The output is here:

```
[1] "grey" "gray"
[1] 4 9
```

Notice that the string `groy` is not displayed in the preceding output, nor is there a -1 (which you might have expected) as the index for the non-occurrence of the string `groy`.

The following pair of commands uses the pattern `[ae]` to combine `grey` and `gray`, and then displays the occurrence of `grey` and `gray` in `vect1`, along with the index values of their positions in `vect1`:

```
grep(pattern = "gr[ae]y ", vect1, value = TRUE)
grep(pattern = "gr[ae]y ", vect1, value = FALSE)
```

The output is here:

```
[1] "grey" "gray"
[1] 4 9
```

Specifying a Range of Letters

We can "expand" the RE in the preceding code snippet to include all the lowercase letters of the alphabet, which is represented by `[a-z]`. We can find all the lines that contain a string that is of the form `gr[a-z]y`, which matches any string that meets the following conditions:

1. starts with the letters gr
2. is followed by any single letter a, b, c, …, z
3. ends with the letter y

Just to confirm, launch the following commands:

```
grep(pattern = "gr[a-z]y", vect1, value = TRUE)
grep(pattern = "gr[a-z]y", vect1, value = FALSE)
```

The output is here:

```
[1] "grey" "gray"
[1] 4 9
```

The only matches are `grey` and `gray`, but if `vect1` included the "word" `grzy`, then this word would appear in the previous output.

We can also specify a single letter inside the square brackets. For example, the term `[a]` is an RE that matches the letter `a`. Launch this command:

```
grep(pattern = "[a]", vect1, value = TRUE)
grep(pattern = "[a]", vect1, value = FALSE)
```

The output is here:

```
[1] "and"  "cat"  "gray"
[1] 5 7 9
```

If we specify a vowel that does not appear in any word in `vect1`, then we see a message that indirectly hints at the absence of that vowel. An example is here:

```
grep(pattern = "[u]", vect1, value = TRUE)
grep(pattern = "[u]", vect1, value = FALSE)
```

The output is here:

```
character(0)
integer(0)
```

We can specify different subranges of letters. For example, suppose we want to find the words that contain any vowel except for the vowels `a` or `i`. This expression will do the job:

```
grep(pattern = "[eou]", vect1, value = TRUE)
grep(pattern = "[eou]", vect1, value = FALSE)
```

The output is here:

```
[1] "the"  "dog"  "grey" "the"
[1] 1 2 4 6
```

Once again, the order of the letters in the square brackets is irrelevant, which means that the following commands have the same output:

```
grep(pattern = "[eou]", vect1, value = TRUE)
grep(pattern = "[oeu]", vect1, value = TRUE)
grep(pattern = "[oue]", vect1, value = TRUE)
```

WORKING WITH ARRAYS OF STRINGS

In this section let's define an array of strings called `mytext1` whose contents are shown here:

```
mytext1 <- c("the dog is grey and the cat is gray.", "this dog is
grey", "that cat is gray")
```

Now let's apply the REs that we saw early in this chapter to the variable `mytext1`. For example, check for strings in `mytext1` with either of these two REs:

```
grep(pattern = "grey|gray", mytext1, value = TRUE)
grep(pattern = "grey|gray", mytext1, value = FALSE)
```

The result is here:

```
[1] "the dog is grey and the cat is gray." "this dog is grey"
[3] "that cat is gray"
```

Check for strings in `mytext1` with either of these two REs:

```
grep(pattern = "gr[ae]y", mytext1, value = TRUE)
grep(pattern = "gr[ae]y", mytext1, value = FALSE)
```

The result is here:

```
[1] "the dog is grey and the cat is gray." "this dog is grey"
[3] "that cat is gray"
```

Check for strings in `mytext1` with the following RE:

```
grep(pattern = "gr[a-z]y", mytext1, value = TRUE)
grep(pattern = "gr[a-z]y", mytext1, value = FALSE)
```

The result is here:

```
[1] "the dog is grey and the cat is gray." "this dog is grey"
[3] "that cat is gray"
```

Check for strings in `mytext1` with the following RE:

```
grep(pattern = "[eou]", mytext1, value = TRUE)
grep(pattern = "[eou]", mytext1, value = FALSE)
```

The result is here:

```
[1] "the dog is grey and the cat is gray." "this dog is grey"
[3] "that cat is gray"
```

The examples in this section use the `grep()` function, but you can also use the `sub()` and `gsub()` functions, described earlier in this chapter, in conjunction with REs.

ONE-LINE REs WITH METACHARACTERS IN R

This section contains examples of using the `grep` function (and related functions) in R in order to find matching strings in string vectors. If necessary, read the appropriate sections in Chapter 1 to refresh your memory regarding the REs in this section.

Initialize string vector strings:

```
(strings <- c("a", "ab", "acb", "accb", "acccb", "accccb"))
## [1] "a"      "ab"     "acb"    "accb"   "acccb"  "accccb"
```

Match the RE `ac*b:`

```
grep("ac*b", strings, value = TRUE)
## [1] "ab"     "acb"    "accb"   "acccb"  "accccb"
```

Match the RE `ac+b:`

```
grep("ac+b", strings, value = TRUE)
```

The output is here:

```
## [1] "acb"    "accb"   "acccb"  "accccb"
```

Match the RE `ac?b:`

```
grep("ac?b", strings, value = TRUE)
```

The output is here:

```
## [1] "ab"  "acb"
```

Match the RE `ac{2}b:`

```
grep("ac{2}b", strings, value = TRUE)
```

The output is here:

```
## [1] "accb"
grep("ac{2,}b", strings, value = TRUE)
```

The output is here:

```
## [1] "accb"   "acccb"  "accccb"
grep("ac{2,3}b", strings, value = TRUE)
```

The output is here:

```
## [1] "accb"  "acccb"
```

Assign a new set of strings to the vector strings:

```
(strings <- c("abcd", "cdab", "cabd", "c abd"))
## [1] "abcd"  "cdab"  "cabd"  "c abd"
```

The output is here:

```
grep("ab", strings, value = TRUE)
```

The output is here:

```
## [1] "abcd"  "cdab"  "cabd"  "c abd"
grep("^ab", strings, value = TRUE)
```

The output is here:

```
## [1] "abcd"
grep("ab$", strings, value = TRUE)
```

The output is here:

```
## [1] "cdab"
grep("\\bab", strings, value = TRUE)
```

The output is here:

```
## [1] "abcd"  "c abd"
```

Assign a new set of strings to the vector strings:

```
(strings <- c("^ab", "ab", "abc", "abd", "abe", "ab 12"))
## [1] "^ab"   "ab"    "abc"   "abd"   "abe"   "ab 12"
grep("ab.", strings, value = TRUE)
```

The output is here:

```
## [1] "abc"   "abd"   "abe"   "ab 12"
grep("ab[c-e]", strings, value = TRUE)
```

The output is here:

```
## [1] "abc" "abd" "abe"
grep("ab[^c]", strings, value = TRUE)
```

The output is here:

```
## [1] "abd"   "abe"   "ab 12"
grep("^ab", strings, value = TRUE)
```

The output is here:

```
## [1] "ab"    "abc"   "abd"   "abe"   "ab 12"
grep("\\^ab", strings, value = TRUE)
```

The output is here:

```
## [1] "^ab"
grep("abc|abd", strings, value = TRUE)
```

The output is here:

```
## [1] "abc" "abd"
gsub("(ab) 12", "\\1 34", strings)
```

The output is here:

```
## [1] "^ab"   "ab"    "abc"   "abd"   "abe"   "ab 34"
```

Assign a new set of strings to the vector strings:

```
(strings <- c("Axbc", "A.bc"))
## [1] "Axbc" "A.bc"
pattern <- "A.b"
grep(pattern, strings, value = TRUE)
```

The output is here:

```
## [1] "Axbc" "A.bc"
grep(pattern, strings, value = TRUE, fixed = TRUE)
```

The output is here:

```
## [1] "A.bc"
```

Case Sensitivity in R

Pattern matching is case sensitive in R. However, you can perform case insensitive pattern matching by specifying `ignore.case=TRUE` (in base R functions) or by "wrapping them" with `ignore.case()` for `stringr` functions.

Yet another way to specify case-insensitive pattern matching is to use the `tolower()` and `toupper()` functions to convert strings to lower- or upper-case and then perform pattern-matching operations. Consider the following example:

```
pattern <- "a.b"
grep(pattern, strings, value = TRUE)
```

The result is here:

```
## character(0)
grep(pattern, strings, value = TRUE, ignore.case = TRUE)
```

The result is here:

```
## [1] "Axbc" "A.bc"
```

Now let's look at an example involving a vector of sentences:

```
mytext <- c("This is the first line", "This is second", "This line
has 997 also")
```

This RE determines the index of the strings with "pure" numbers:

```
grep("[0-9]", mytext, ignore.case=T)
```

The result is here:

```
## [1] 3
```

This RE determines the index of the alphanumeric strings:

```
grep("[a-zA-Z0-9]", mytext, ignore.case=T)
```

The result is here:

```
## [1] 1 2 3
```

This RE determines the index of the strings containing only characters:

```
grep("(a-zA-Z)", mytext, ignore.case=T)
```

The result is here:

```
## [1] 1 2
```

Escaping Metacharacters in R Functions

In the first part of this chapter you learned that you can match a metacharacter as a "normal" character if you precede that character with a double backslash. In addition, you need to precede a backslash with *four* backslashes in order to treat it as a regular character.

As a simple example, the following code snippet uses the `gsub()` command to replace the metacharacter $ with the metacharacter * in the string `metastr`:

```
metastr <- "this has an asterisk * inside"
> gsub("\\*", "\\?", metastr)
[1] "this has an asterisk ? inside"
```

EXAMPLES OF R FUNCTIONS AND REs

The `sapply()` function in R enables you to apply a function (of your choosing) to a list or a vector, and the result is also a list or a vector of the same length. Hence, the following code snippet uses the `sapply()` function in order to invoke the R function `gregexpr()` to display the starting position of the first match (or -1 if there is none) in the string `str` (defined elsewhere):

```
sapply(gregexpr("\\S+", str), length)
```

Listing 4.1 displays the contents of `sapplyfunc.R`, which illustrates how to use the `sapply()` function with other R functions.

LISTING 4.1: sapplyfunc.R

```
str <- c("this is a short string",
         "two words",
         "pasta",
         "beer")

sapply(gregexpr("\\S+", str), length)
#[1] 5 2 1 1

sapply(strsplit(str, "\\s+"), length)
#[1] 5 2 1 1

require(stringi)
str <- c(
  "this is a string that is slightly longer",
  "nospacesinthisstring",
  "several    whitespaces",
  "   startswithspaces",
  "endswithgspaces    ",
  "    contains both leading and trailing    ",
  "just one space each")

stri_count(str,regex="\\S+")
#[1] 8 1 2 1 1 5 4
```

Listing 4.1 initializes `str` as a vector of strings, and then invokes the `sapply()` method twice. The first invocation invokes the `gregexpr()` method in order to find the position of the first occurrence of the RE in each of the four substrings of `str`, which yields the values 5, 2, 1, and 1, respectively.

The second invocation of the `sapply()` method invokes the `strsplit()` method that splits the four substrings of `str` into substrings based on the specified RE, which produces the values 5, 2, 1, and 1, respectively.

The third portion of Listing 4.1 initializes `str` as a new list of strings, and then invokes the `stri_count()` function in order to count the number of occurrences of non-whitespace character strings in each of the seven substrings of `str`, which is 8, 1, 2, 1, 1, 5, and 4, respectively.

ADVANCED STRING FUNCTIONS IN R

R supports several more advanced string functions that are somewhat related to REs, such as splitting a string, getting a subset of a string, pasting strings together, and so forth. These R functions are very useful for data cleaning, and here is a short introduction with above example.

The `strsplit()` function (which returns a list) splits its second argument into words, where the second argument split is an RE used for splitting strings.

The `unlist()` function converts a list into a character vector, and the function `str_split_fixed()` returns a data frame.

The `paste()` or `paste0()` functions put things together. The `paste0()` function is equivalent to `paste()` with `sep = ""`. We can use the `collapse = "-"` argument to concatenate a character vector into a string.

Another useful function is `substr()`, which extracts a part of a string with a start index and an end index.

An Internet search of any of these commands, of form "R <command> example" and "R <command> documentation", will provide both official documentation and many useful examples to show both syntax and scope of the commands beyond the brief description here.

The `stringr` Package in R

The R package `stringr` also provides several functions for RE operations. Specifically, the `stringr` package provides pattern-matching functions to detect, locate, extract, match, replace, and split strings.

All pattern-matching functions in `stringr` have the following general form, where the first argument is a string vector to be processed and the second argument is a single RE to match:

```
str_function(string, pattern)
```

For example, the `detect()` function checks whether or not a pattern appears in a string. The `extract()` and `extract_all()` functions extract the first occurrence and all occurrences, respectively, of a pattern in a string. The `match()` and `match_all()` functions extract the first matched group and all matched groups, respectively, from a string. Other functions in this package include: `locate()` and `locate_all()`, `replace()` and `replace_all()`, `split()` and `split_fixed()`. As with the previous R commands, an internet search for "R stringr package documentation" will provide more details regarding the functions in this package.

CHAPTER SUMMARY

In this chapter you got a summary of rules for metacharacters in R, an overview of search functions in R, as well as quick explanation of `grep`-related commands in R. Next, you learned about basic REs used in specific commands for matching and substitution in R, similar to some of the REs that you saw in Chapter 1. Then you learned how to use metacharacters in R, illustrated in the form of one-line REs, and finally a brief overview of other string-manipulation commands in R.

CHAPTER 5

WORKING WITH REs IN BASH

This section assumes a bit of basic familiarity with the Unix/Linux command line and how the commands accept input and generate output. While the examples use the bash "shell" environment for syntax, most of them will also work with other common shells such as `bourne` and `korn`.

This chapter shows you how to use REs in order to transform data using the Unix `sed` utility (an acronym for "stream editor"), followed by a short section that contains examples of REs with the Unix `awk` utility.

The first part of this chapter contains basic examples of the `sed` command, such as replacing and deleting strings, numbers, and letters. The second part of this chapter discusses various switches that are available for the `sed` command, along with an example of replacing multiple delimiters with a single delimiter in a dataset.

The third part of this chapter provides a very brief introduction of the `awk` command. You will learn about some built-in variables for `awk`, and also how to manipulate string variables using `awk`. *Note* that some of these string-related examples can also be handled using other `bash` commands.

The final section contains code samples that involve metacharacters (introduced in Chapter 1) and character sets in `awk` commands. You will also see how to use conditional logic in `awk` commands in order to determine whether or not to print specific lines of text.

WHAT IS THE `SED` COMMAND?

The `sed` command is the most common command line tool used in Unix/Linux environments to do find/replace-type functions using REs for pattern matching, although it has many other uses. As such it's worth a bit of explanation before diving into examples.

The name sed is an acronym for "stream editor", and the utility derives many of its commands from the ed line-editor (ed was the first UNIX text editor). The sed command is a "non-interactive" stream-oriented editor that can be used to automate editing via shell scripts. This ability to modify an entire stream of data (which can be the contents of multiple files, in a manner similar to how grep behaves) as if you were inside an editor is not common in modern programming languages. This behavior allows some capabilities not easily duplicated elsewhere, while behaving exactly like any other command (grep, cat, ls, find, and so forth) in how it can accept data, output data, and pattern match with REs.

Some of the more common uses for sed include: print matching lines, delete matching lines, and find/replace matching strings or REs.

The sed Execution Cycle

Whenever you invoke the sed command, an execution cycle refers to various options that are specified and executed until the end of the file/input is reached. Specifically, an execution cycle performs the following steps:

Read an entire line from stdin/file.
Remove any trailing newline.
Place the line in its pattern buffer.
Modify the pattern buffer according to the supplied commands.
Print the pattern buffer to stdout.

MATCHING STRING PATTERNS USING SED

The sed command requires you to specify a string in order to match the lines in a file. For example, suppose that the file numbers.txt contains the following lines:

```
1
2
123
3
five
4
```

The following sed command prints all the lines that contain the string 3:

```
cat numbers.txt |sed -n "/3/p"
```

Another way to produce the same result:

```
sed -n "/3/p" numbers.txt
```

In both cases the output of the preceding commands is as follows:

```
123
3
```

Keep in mind that it's always more efficient to just read in the file using the sed command than to pipe it in with a different command. You can "feed" it

data from another command if that other command adds value (such as adding line numbers, removing blank lines, or other similar helpful activities).

The `-n` option suppresses all output, and the `p` option prints the matching line. If you omit the `-n` option, then every line is printed, and the `p` option causes the matching line to be printed again. Hence, issue the following command:

```
sed "/3/p" numbers.txt
```

The output (the data to the right of the colon) is as follows. *Note* that the labels to the left of the colon show the source of the data, to illustrate the "one row at a time" behavior of sed.

```
Basic stream output :1
Basic stream output :2
Basic stream output :123
Pattern Matched text:123
Basic stream output :3
Pattern Matched text:3
Basic stream output :five
Basic stream output :4
```

It is also possible to match two patterns and print everything between the lines that match:

```
sed -n "/123/,/five/p" numbers.txt
```

The output of the preceding command (all lines between 123 and five, inclusive) is here:

```
123
3
five
```

SUBSTITUTING STRING PATTERNS USING `SED`

The examples in this section illustrate how to use `sed` to substitute new text for an existing text pattern.

```
x="abc"
echo $x |sed "s/abc/def/"
```

The output of the preceding code snippet is here:

```
def
```

In the prior command you have instructed `sed` to substitute (`"s`) the first text pattern (`/abc`) with the second pattern (`/def`) and no further instructions (`/"`).

Deleting a text pattern is simply a matter of leaving the second pattern empty:

```
echo "abcdefabc" |sed "s/abc//"
```

The result is here:

```
defabc
```

As you see, this only removes the first occurrence of the pattern. You can remove all the occurrences of the pattern by adding the "global" terminal instruction (/g"):

```
echo "abcdefabc" |sed "s/abc//g"
```

The result of the preceding command is here:

```
def
```

Note that we are operating directly on the main stream with this command, as we are not using the `-n` tag. You can also suppress the main stream with `-n` and print the substitution, achieving the same output if you use the terminal `p` (print) instruction:

```
echo "abcdefabc" |sed -n "s/abc//gp"
def
```

For substitutions either syntax will do, but that is not always true of other commands.

You can also remove digits instead of letters by using the numeric metacharacters as your regular expression match pattern (from Chapter 1):

```
ls svcc1234.txt |sed "s/[0-9]//g"
ls svcc1234.txt |sed –n "s/[0-9]//gp"
```

The result of either of the two preceding commands is here:

```
svcc.txt
```

Recall that the file `columns4.txt` contains the following text:

```
123 ONE TWO
456 three four
ONE TWO THREE FOUR
five 123 six
one two three
four five
```

The following `sed` command is instructed to identify the rows between 1 and 3, inclusive (`"1,3`), and delete (`d"`) them from the output:

```
cat columns4.txt  | sed "1,3d"
```

The output is here:

```
five 123 six
one two three
four five
```

The following `sed` command deletes a range of lines, starting from the line that matches `123` and continuing through the file until reaching the line that matches the string `five` (and also deleting all the intermediate lines). The syntax should be familiar from the earlier matching example:

```
sed "/123/,/five/d" columns4.txt
```

The output is here:

```
one two three
four five
```

Replacing Vowels from a String or a File

The following code snippet shows you how simple it is to replace multiple vowels from a string using the `sed` command:

```
echo "hello" | sed "s/[aeio]/u/g"
```

The output from the preceding code snippet is here:

```
Hullu
```

Deleting Multiple Digits and Letters from a String

Suppose that we have a variable x that is defined as follows:

```
x="a123zAB 10x b 20 c 300 d 40w00"
```

Recall that an integer consists of one or more digits, so it matches the regular expression `[0-9]+`, which matches one or more digits. However, you need to specify the regular expression [0-9]* in order to remove every number from the variable `x`:

```
echo $x | sed "s/[0-9]//g"
```

The output of the preceding command is here:

```
azAB x b  c  d w
```

The following command removes all lowercase letters from the variable `x`:

```
echo $x | sed "s/[a-z]*//g"
```

The output of the preceding command is here:

```
123AB 10  20  300  4000
```

The following command removes all lowercase and uppercase letters from the variable `x`:

```
echo $x | sed "s/[a-z][A-Z]*//g"
```

The output of the preceding command is here:

```
123 10  20  300  4000
```

SEARCH AND REPLACE WITH SED

The previous section showed you how to delete a range of rows of a text file, based on a start line and end line, using either a numeric range or a pair of strings. As deleting is just substituting an empty result for what you match, it should now be clear that a replace activity involves populating that part of the command with something that achieves your desired outcome. This section contains various examples that illustrate how to get the exact substitution you desire.

The following examples illustrate how to convert lowercase abc to uppercase ABC in sed:

```
echo "abc" |sed "s/abc/ABC/"
```

The output of the preceding command is here (which only works on one case of abc):

```
ABC
echo "abcdefabc" |sed "s/abc/ABC/g"
```

The output of the preceding command is here (/g" means works on every case of abc):

```
ABCdefABC
```

The following sed expression performs three consecutive substitutions, using -e to string them together. It changes exactly one (the first) a to A, one b to B, and one c to C:

```
echo "abcde" |sed -e "s/a/A/" -e "s/b/B/" -e "s/c/C/"
```

The output of the preceding command is here:

```
ABCde
```

Obviously you can use the following sed expression that combines the three substitutions into one substitution:

```
echo "abcde" |sed "s/abc/ABC/"
```

Nevertheless, the –e switch is useful when you need to perform more complex substitutions that cannot be combined into a single substitution.

The "/" character is not the only delimiter that sed supports, which is useful when strings contain the "/" character. For example, you can reverse the order of /aa/bb/cc/ with this command:

```
echo "/aa/bb/cc" |sed -n "s#/aa/bb/cc#/cc/bb/aa/#p"
```

The output of the preceding sed command is here:

```
/cc/bb/aa/
```

The following examples illustrate how to use the "w" terminal command instruction to write the `sed` output to both standard output and also to a named file `upper1` if the match succeeds:

```
echo "abcdefabc" |sed "s/abc/ABC/wupper1"
ABCdefabc
```

If you examine the contents of the text file `upper1` you will see that it contains the same string `ABCdefabc` that is displayed on the screen. This two-stream behavior that we noticed earlier with the print ("p") terminal command is unusual but sometimes useful. It is more common to simply send the standard output to a file using the ">" syntax, as shown in the following example (both syntaxes work for a replace operation), but in that case nothing is written to the terminal screen. The previous syntax allows both at the same time:

```
echo "abcdefabc" | sed "s/abc/ABC/" > upper1
echo "abcdefabc" | sed -n "s/abc/ABC/p" > upper1
```

Listing 5.1 displays the contents of `update2.sh` that replace the occurrence of the string `hello` with the string `goodbye` in the files with the suffix `txt` in the current directory.

LISTING 5.1: update2.sh

```
for f in `ls *txt`
do
  newfile="${f}_new"
  cat $f | sed -n "s/hello/goodbye/gp" > $newfile
  mv $newfile $f
done
```

Listing 5.1 contains a `for` loop that iterates over the list of text files with the txt suffix. For each such file, initialize the variable `newfile` that is created by appending the string `_new` to the first file (represented by the variable `f`). Next, replace the occurrences of hello with the string goodbye in each file `f`, and redirect the output to `$newfile`. Finally, rename `$newfile` to `$f` using the `mv` command.

If you want to perform the update in matching files in all subdirectories, replace the "for" statement with the following:

```
for f in `find . -print |grep "txt$"`
```

DATASETS WITH MULTIPLE DELIMITERS

Listing 5.2 displays the contents of the dataset `delim1.txt`, which contains multiple delimiters "|", ":", and "^". Listing 5.3 displays the contents of `delimiter1.sh`, which illustrates how to replace multiple delimiters with a single comma delimiter "," in `delimiter1.txt`.

LISTING 5.2: delimiter1.txt

```
1000|Jane:Edwards^Sales
2000|Tom:Smith^Development
3000|Dave:Del Ray^Marketing
```

LISTING 5.3: delimiter1.sh

```
inputfile="delimiter1.txt"
cat $inputfile | sed -e 's/:/,/' -e 's/|/,/' -e 's/\^/,/'
```

As you can see, the second line in Listing 5.3 is simple yet very powerful: you can extend the `sed` command with as many delimiters as you require in order to create a dataset with a single delimiter between values. The output from Listing 5.3 is shown here:

```
1000,Jane,Edwards,Sales
2000,Tom,Smith,Development
3000,Dave,Del Ray,Marketing
```

Do keep in mind that this kind of transformation can be a bit unsafe unless you have checked that your new delimiter is *not* already in use. For that a `grep` command is useful (you want the result to be zero, as `-c` counts the how many times the pattern matches in the input file):

```
grep -c ',' $inputfile
0
```

USEFUL SWITCHES IN `SED`

The three command line switches `-n`, `-e`, and `-i` are useful when you specify them with the `sed` command.

As a review, specify `-n` when you want to suppress the printing of the basic stream output:

```
sed -n 's/foo/bar/'
```

Specify `-n` and end with `/p'` when you want to match the result only:

```
sed -n 's/foo/bar/p'
```

We briefly touched on using `-e` to do multiple substitutions, but it can also be used to combine other commands. This syntax lets us separate the commands in the last example:

```
sed -n -e 's/foo/bar/' -e 'p'
```

A more advanced example that hints at the flexibility of `sed` involves the insertion of a character after a fixed number of positions. For example, consider the following code snippet:

```
echo "ABCDEFGHIJKLMNOPQRSTUVWXYZ" | sed "s/.\{3\}/&\n/g"
```

The output from the preceding command is here:

```
ABCnDEFnGHInJKLnMNOnPQRnSTUnVWXnYZ
```

While the previous example does not seem especially useful, consider a large text stream with no line breaks (everything on one line). You could use something like this to insert newline characters, or something else to break the data into easier-to-process chunks. It is possible to work through exactly what `sed` is doing by looking at each element of the command and comparing to the output, even if you don't know the syntax. (Tip: sometimes you will encounter very complex instructions for `sed` without any documentation in the code: try not to be that person when coding.)

The output is changing after every three characters and we know dot (`.`) matches any single character, so `.{3}` must be telling it to do that (with escape slashes \ because brackets are a special character for `sed`, and it won't interpret it properly if we just leave it as `.{3}`). The "`n`" is clear enough in the replacement column, so the "`&\`" must be somehow telling it to insert a character instead of replacing it. The terminal g command of course means to repeat. To clarify and confirm those guesses, take what you could infer and perform an Internet search.

WORKING WITH DATASETS

The `sed` utility is very useful for manipulating the contents of text files. For example, you can print ranges of lines or subsets of lines that match a regular expression. You can also perform search-and-replace on the lines in a text file. This section contains examples that illustrate how to perform such functionality.

Printing Lines

Listing 5.4 displays the contents of `test4.txt` (doubled-spaced lines) that is used for several examples in this section.

LISTING 5.4: test4.txt

```
abc

def

abc

abc
```

The following code snippet prints the first three lines in `test4.txt` (we used this syntax before when deleting rows, and it is equally useful for printing):

```
cat test4.txt  |sed -n "1,3p"
```

The output of the preceding code snippet is here (the second line is blank):

```
abc

def
```

The following code snippet prints lines 3 through 5 in `test4.txt`:

```
cat test4.txt  |sed -n "3,5p"
```

The output of the preceding code snippet is here:

```
def

abc
```

The following code snippet takes advantage of the basic output stream and the second match stream to duplicate every line (including blank lines) in `test4.txt`:

```
cat test4.txt  |sed "p"
```

The output of the preceding code snippet is here:

```
abc
abc

def
def

abc
abc

abc
abc
```

The following code snippet prints the first three lines and then capitalizes the string `abc`, duplicating `ABC` in the final output because we did not use `-n` and did end with `/p"` in the second `sed` command. Remember that `/p"` only prints the text that matched the `sed` command, where the basic output prints the whole file, which is why `def` does not get duplicated:

```
cat test4.txt  |sed -n "1,3p" |sed "s/abc/ABC/p"

ABC
ABC

def
```

Character Classes and `sed`

You can also use REs with `sed`. As a reminder, here are the contents of `columns4.txt`:

```
123 ONE TWO
456 three four
```

```
ONE TWO THREE FOUR
five 123 six
```

As our first example involving `sed` and character classes, the following code snippet illustrates how to match lines that contain lowercase letters:

```
cat columns4.txt | sed -n '/[0-9]/p'
```

The output from the preceding snippet is here:

```
one two three
one two
one two three four
one
one three
one four
```

The following code snippet illustrates how to match lines that contain lowercase letters:

```
cat columns4.txt | sed -n '/[a-z]/p'
```

The output from the preceding snippet is here:

```
123 ONE TWO
456 three four
five 123 six
```

The following code snippet illustrates how to match lines that contain the numbers 4, 5, or 6:

```
cat columns4.txt | sed -n '/[4-6]/p'
```

The output from the preceding snippet is here:

```
456 three four
```

The following code snippet illustrates how to match lines that start with any two characters followed by `EE`:

```
cat columns4.txt | sed -n '/^.\{2\}EE*/p'
```

The output from the preceding snippet is here:

```
ONE TWO THREE FOUR
```

Removing Control Characters

Listing 5.5 displays the contents of `controlchars.txt` that we used before in Chapter 2. Control characters of any kind can be removed by `sed` just like any other character.

LISTING 5.5: controlchars.txt

```
1 carriage return^M
2 carriage return^M
1 tab character^I
```

The following command removes the carriage return and the tab characters from the text file `ControlChars.txt`:

```
cat controlChars.txt | sed "s/^M//" |sed "s/   //"
```

You cannot see the tab character in the second `sed` command in the preceding code snippet; however, if you redirect the output to the file `nocontrol1.txt`, you can see that there are no embedded control characters in this new file by typing the following command:

```
cat -t nocontrol1.txt
```

COUNTING WORDS IN A DATASET

Listing 5.6 displays the contents of `WordCountInFile.sh`, which illustrates how to combine various `bash` commands in order to count the words (and their occurrences) in a file.

LISTING 5.6: wordcountinfile.sh

```
# The file is fed to the "tr" command, which changes uppercase to lowercase
# sed removes commas and periods, then changes whitespace to newlines
# uniq needs each word on its own line to count the words properly
# Uniq converts data to unique words and the number of times they appeared
# The final sort orders the data by the wordcount.

cat "$1" | xargs -n1 | tr A-Z a-z | \
sed -e 's/\.//g' -e 's/\,//g' -e 's/ /\ /g' | \
sort | uniq -c | sort -nr
```

The previous command performs the following operations:

* List each word in each line of the file.
* Shift characters to lowercase.
* Filter out periods and commas.
* Change the space between words to linefeed.
* Remove duplicates, prefix occurrence count, and sort numerically.

BACK REFERENCES AND FORWARD REFERENCES IN SED

In the chapter describing `grep` you learned about back references, and similar functionality is available with the `sed` command. The main difference

is that the back references can also be used in the replacement section of the command.

The following `sed` command matches two consecutive occurrences of the letter "a" and prints four of them:

```
echo "aa" |sed -n "s#\([a-z]\)\1#\1\1\1\1#p"
```

The output of the preceding code snippet is here:

```
aaaa
```

The following `sed` command replaces all duplicate pairs of letters with the letters `aa`:

```
echo "aa/bb/cc" |sed -n "s#\(aa\)/\(bb\)/\(cc\)#\1/\1/\1/#p"
```

The output of the previous `sed` command is here (*note* the trailing "/" character):

```
aa/aa/aa/
```

The following command inserts a comma in a four-digit number:

```
echo "1234" |sed -n "s@\([0-9]\)\([0-9]\)\([0-9]\)\
([0-9]\)@\1,\2\3\4@p"
```

The preceding `sed` command uses the `@` character as a delimiter. The character class `[0-9]` matches one single digit. Since there are four digits in the input string `1234`, the character class `[0-9]` is repeated four times, and the value of each digit is stored in `\1`, `\2`, `\3`, and `\4`. The output from the preceding `sed` command is here:

```
1,234
```

A more general `sed` expression that can insert a comma in five-digit numbers is here:

```
echo "12345" | sed 's/\([0-9]\{3\}\)$/,\1/g;s/^,//'
```

The output of the preceding command is here:

```
12,345
```

Working with Forward References

In programming languages, the term "forward reference" is a situation in which a function is invoked before that function has been defined. This brief section addresses RE forward references (i.e., REs that contain a forward reference).

A RE forward reference is essentially the opposite of an RE back reference: you need to look ahead (forward) to determine whether or not a captured

group appears later in the string instead of earlier in the string. Keep in mind that Perl supports RE forward references, whereas other languages (such as JavaScript) do not support RE forward references.

Use the symbol "=" to denote a forward reference in an RE. The following syntax shows you how to specify whether or not a forward reference contains a string, as shown here:

```
(?=abc): followed by abc
(?!abc): not followed by abc
```

Perform an online search to find examples of RE forward references.

DISPLAYING ONLY "PURE" WORDS IN A DATASET

In the previous chapter we solved this task using the egrep command, and this section shows you how to solve this task using the sed command.

For simplicity, let's work with a text string, and that way we can see the intermediate results as we work toward the solution. The approach will be similar to the code block shown earlier which counted unique words. Let's initialize the variable x as shown here:

```
x="ghi abc Ghi 123 #def5 123z"
```

The first step is to split x into one word per line by replacing space with newlines:

```
echo $x |tr -s ' ' '\n'
```

The output is here:

```
ghi
abc
Ghi
123
#def5
123z
```

The second step is to invoke sed with the regular expression ^[a-zA-Z]+, which matches any string consisting of one or more uppercase and/or lowercase letters (and nothing else). *Note* that the -E switch is needed to parse this kind of regular expression in sed, as it uses some of the newer/modern regular expression syntax not available when sed was new.

```
echo $x |tr -s ' ' '\n' |sed -nE "s/(^[a-zA-Z][a-zA-Z]*$)/\1/p"
```

The output is here:

```
ghi
abc
Ghi
```

If you also want to sort the output and print only the unique words, pipe the result to the `sort` and `uniq` commands:

```
echo $x |tr -s ' ' '\n' |sed -nE "s/(^[a-zA-Z][a-zA-Z]*$)/\1/
p"|sort|uniq
```

The output is here:

```
Ghi
abc
ghi
```

If you want to extract only the integers in the variable `x`, use this command:

```
echo $x |tr -s ' ' '\n' |sed -nE "s/(^[0-9][0-9]*$)/\1/p"
|sort|uniq
```

The output is here:

```
123
```

If you want to extract alphanumeric words from the variable `x`, use this command:

```
echo $x |tr -s ' ' '\n' |sed -nE "s/(^[0-9a-zA-Z][0-9a-zA-
Z]*$)/\1/p"|sort|uniq
```

The output is here:

```
123
123z
Ghi
abc
ghi
```

Now you can replace `echo $x` with a dataset in order to retrieve only alphabetic strings from that dataset.

This concludes the portion of the chapter pertaining to the `sed` command. The next portion of the chapter discusses the `awk` command, along with many simple code snippets that perform a variety of tasks.

THE `AWK` COMMAND

The `awk` (Aho, Weinberger, and Kernighan) command has a C-like syntax, and you can use this utility to perform very complex operations on numbers and text strings.

`Awk` has nearly the flexibility of an entire programming language contained in a command that Unix/Linux sees behaving as if it was any other command. As such it is the go-to command when `grep` and `sed` aren't enough to get the job done.

As a side comment, there is also the gawk command that is GNU awk, as well as the nawk command is "new" awk (neither command is discussed in this book). One advantage of nawk is that it allows you to set externally the value of an internal variable.

Built-In Variables That Control awk

The awk command provides variables that you can change from their default values in order to control how awk performs operations. Examples of such variables (and their default values) include: FS (" "), RS ("\n"), OFS (" "), ORS ("\n"), SUBSEP, and IGNORECASE. The variables FS and RS specify the field separator and record separator, whereas the variables OFS and ORS specify the output field separator and the output record separator, respectively.

You can think of the field separators as the delimiters/IFS we used in other commands earlier. The record separators behave in a way similar to how sed treats individual lines—for example sed can match or delete a range of lines instead of matching or deleting something that matches a regular expression (and the default awk record separator is the newline character, so by default awk and sed have the similar ability to manipulate and reference lines in a text file).

As a simple example, you can print a blank line after each line of a file by changing the ORS from the default of one newline to two newlines, as shown here:

```
cat columns.txt | awk 'BEGIN { ORS ="\n\n" } ; { print $0 }'
```

Other built-in variables include FILENAME (the name of the file that awk is currently reading), FNR (the current record number in the current file), NF (the number of fields in the current input record), and NR (the number of input records awk has processed since the beginning of the program's execution).

Consult the online documentation for additional information regarding these (and other) arguments for the awk command.

How Does the awk Command Work?

The awk command reads the input files one record at a time (by default, one record is one line). If a record matches a pattern, then an action is performed (otherwise no action is performed). If the search pattern is not given, then awk performs the given actions for each record of the input. The default behavior if no action is given is to print all the records that match the given pattern. Finally, empty braces without any action does nothing; that is, it will not perform the default printing operation. *Note* that each statement in actions should be delimited by a semicolon.

In other to make the preceding paragraph more concrete, here are some simple examples involving text strings and the awk command (the results are displayed after each code snippet). The -F switch sets the field separator to

whatever follows it, in this case a space. Switches will often provide a shortcut to an action that normally needs a command inside a 'BEGIN{ } block):

```
x="a b c d e"
echo $x |awk -F" " '{print $1}'
a
echo $x |awk -F" " '{print NF}'
5
echo $x |awk -F" " '{print $0}'
a b c d e
echo $x |awk -F" " '{print $3, $1}'
c a
```

Now let's change the FS (record separator) to an empty string to calculate the length of a string, this time using the BEGIN{ } syntax:

```
echo "abc" | awk 'BEGIN { FS = "" } ; { print NF }'
3
```

The following example illustrates several equivalent ways to specify test.txt as the input file for an awk command:

```
awk < test.txt '{ print $1 }'
awk '{ print $1 }' < test.txt
awk '{ print $1 }' test.txt
```

Yet another way is shown here (but as we've discussed earlier, it can be inefficient, so only do it if the cat command is adding value in some way):

```
cat test.txt | awk '{ print $1 }'
```

This simple example of four ways to do the same task should illustrate why commenting awk calls of any complexity is almost always a good idea. The next person to look at your code may not know/remember the syntax you are using.

ALIGNING TEXT WITH THE PRINTF COMMAND

Since awk is a programming language inside a single command, it also has its own way of producing formatted output via the printf command.

Listing 5.7 displays the contents of columns2.txt and Listing 5.8 displays the contents of the shell script AlignColumns1.sh, which show you how to align the columns in a text file.

LISTING 5.7: columns2.txt

```
one two
three four
one two three four
five six
one two three
four five
```

LISTING 5.8: AlignColumns1.sh

```
awk '
{
   # left-align  $1 on a 10-char column
   # right-align $2 on a 10-char column
   # right-align $3 on a 10-char column
   # right-align $4 on a 10-char column
   printf("%-10s*%10s*%10s*%10s*\n", $1, $2, $3, $4)
}
' columns2.txt
```

Listing 5.8 contains a `printf()` statement that displays the first four fields of each row in the file `columns2.txt`, where each field is 10 characters wide.

The output from launching the code in Listing 5.8 is here:

```
one       *       two*          *          *
three     *      four*          *          *
one       *       two*     three*      four*
five      *       six*          *          *
one       *       two*     three*          *
four      *      five*          *          *
```

Keep in mind that `printf` is reasonably powerful and as such has its own syntax, which is beyond the scope of this chapter. A search online can find the manual pages and also discussions of "how to do X with printf()."

MATCHING WITH METACHARACTERS AND CHARACTER SETS

If we can match a simple pattern, by now you probably expect that you can also match a regular expression, just as we did in `grep` and `sed`. Listing 5.9 displays the contents of `Patterns1.sh`, which uses metacharacters to match the beginning and the end of a line of text in the file `columns2.txt`.

LISTING 5.9: Patterns1.sh

```
awk '
   /^f/    { print $1 }
   /two $/ { print $1 }
' columns2.txt
```

The output from launching Listing 5.9 is here:

```
one
five
four
```

Listing 5.10 displays the contents of `RemoveColumns.txt` with lines that contain a different number of columns.

LISTING 5.10: columns3.txt

```
123 one two
456 three four
```

```
one two three four
five 123 six
one two three
four five
```

Listing 5.11 displays the contents of `MatchAlpha1.sh`, which matches text lines that start with alphabetic characters as well as lines that contain numeric strings in the second column.

LISTING 5.11: MatchAlpha1.sh

```
awk '
{
   if( $0 ~ /^[0-9]/) { print $0 }
   if( $0 ~ /^[a-z]+ [0-9]/) { print $0 }
}
' columns3.txt
```

The output from Listing 5.11 is here:

```
123 one two
456 three four
five 123 six
```

PRINTING LINES USING CONDITIONAL LOGIC

Listing 5.12 displays the contents of `products.txt`, which contains three columns of information.

LISTING 5.12: products.txt

```
MobilePhone 400  new
Tablet      300  new
Tablet      300  used
MobilePhone 200  used
MobilePhone 100  used
```

The following code snippet prints the lines of text in `products.txt` whose second column is greater than `300`:

```
awk '$2 > 300' products.txt
```

The output of the preceding code snippet is here:

```
MobilePhone 400  new
```

The following code snippet prints the lines of text in `products.txt` whose product is "new":

```
awk '($3 == "new")' products.txt
```

The output of the preceding code snippet is here:

```
MobilePhone 400  new
Tablet      300  new
```

The following code snippet prints the first and third columns of the lines of text in `products.txt` whose cost equals `300`:

```
awk ' $2 == 300 { print $1, $3 }' products.txt
```

The output of the preceding code snippet is here:

```
Tablet new
Tablet used
```

The following code snippet prints the first and third columns of the lines of text in `products.txt` that start with the string `Tablet`:

```
awk '/^Tablet/ { print $1, $3 }' products.txt
```

The output of the preceding code snippet is here:

```
Tablet new
Tablet used
```

SELECTING AND SWITCHING ANY TWO COLUMNS

The example in this section shows you how to switch any pairs of columns (and display them) in the rows of a text file. Listing 5.13 displays the contents of `switchcolumns.sh`, which performs this task. Notice that the code does not require any REs.

LISTING 5.13: switchanytwocolumns.sh

```
awk '
{
   if(NF >= 6) {
      printf("%s,%s\n", $6, $3)
   }
}
' manycolumns.txt
```

As you can see, the if statement in Listing 5.13 processes the rows that contain at least six columns and prints the sixth column and the third column. The output from Listing 5.13 is here:

```
four,one
two,three
three,three
```

If you want to switch the first two columns in `manycolumns.txt`, the code is even simpler:

```
awk –F" " '{print $2, $1}' < manycolumns.txt
```

REVERSING ALL ROWS WITH `AWK`

The example in this section shows you how to reverse the order of all the columns in each row in a text file. Listing 5.14 displays the contents of

manycolumns.txt and Listing 5.15 displays the contents of reversecolumns.sh, which perform this task. Notice that the code does not require any REs.

LISTING 5.14: manycolumns.txt

```
ten
one two three four
three four one two three four
one two three four one two three
five six seven
one two three four five
one two three one two three one two three
```

LISTING 5.15: reverserows.sh

```
awk '
{
  for(i=NF;i>0;i--) printf "%s ",$i;print ""
}
' manycolumns.txt
```

Listing 5.15 consists of a one-line for a loop that contains the logic required to reverse the fields in each row of manycolumns.txt. In fact, you could even replace the contents of Listing 5.14 with the following one-liner:

```
awk ' { for(i=NF;i>0;i--) printf "%s ",$i;print "" } '
manycolumns.txt
```

The output from Listing 5.15 is here:

```
ten
four three two one
four three two one four three
three two one four three two one
seven six five
five four three two one
three two one three two one three two one
```

REVERSING THE LINES IN A FILE

Listing 5.16 displays the contents of fliprows.sh, which illustrates how to "flip" the rows in a text file.

LISTING 5.16: fliprows.sh

```
awk '
{
   lines[i++]=$0
}
END {
   for(j=i-1;j>=0;j--)print lines[j];
}
' manycolumns.txt
```

Listing 5.16 initializes the array lines with all the rows of the input file, and the BEGIN block contains a loop that prints the contents of lines in reverse order. You could even replace the contents of Listing 5.16 with the following one-liner:

```
awk '{ lines[i++]=$0 } END { for(j=i-1;j>=0;j--)print lines[j]; }'
manycolumns.txt
```

The output from Listing 5.16 is here:

```
one two three one two three one two three
one two three four five
five six seven
one two three four one two three
three four one two three four
one two three four
ten
```

Incidentally, the BSD version of the Unix `tail` command can also reverse the order of the rows in a file, and it's much simpler than the `awk` script:

```
tail -r manycolumns.txt
```

SWITCHING TWO ADJACENT COLUMNS (1)

The example in this section shows you how to switch pairs of columns in a text file. For example, we can switch the first two columns, and also switch the third and fourth columns, after we verify that they exist. Listing 5.17 displays the contents of `switchcolumns.sh`, which performs this task. Notice that the code does not require any REs.

LISTING 5.17: switchcolumns.sh

```
awk '
   if(NF >= 2) { print $2, $1 }
   if(NF >= 4) { print $4, $3 }
' columns2.txt
```

The output from the awk script in Listing 5.17 is here:

```
two one
four three
two one
four three
six five
two one
five four
```

The first row of `columns2.txt` is not displayed because it contains only one field, which fails both `if` statements in Listing 5.17. If you look at the contents of `columns2.txt`, the third row contains four fields, but `switch-columns.sh` splits that row into two separate rows (after switching the columns). The next section provides a solution for this detail.

SWITCHING TWO ADJACENT COLUMNS (2)

Listing 5.18 displays the contents of `switchcolumns2.sh`, which prevents the row splitting by using the `printf()` function in `awk`.

LISTING 5.18: switchcolumns2.sh

```
awk '
{
   if(NF >= 2) {
      printf("%s,%s", $2, $1)
      if(NF >= 4) {
         printf(",%s,%s", $4, $3)
      }
      printf("\n")
   }
}
' columns2.txt
```

Once again, the first row of `columns2.txt` is not displayed because it contains only one field, which fails both if statements in Listing 5.18. The output from the `awk` script in Listing 5.18 is here:

```
two,one
four,three
two,one,four,three
six,five
two,one
five,four
```

SWITCHING CONSECUTIVE COLUMNS

The examples in the previous section work correctly for rows containing two or four columns, but they can become difficult to generalize in rows that have an arbitrarily large number of columns. However, the `awk`-based code example in this section does enable you to switch consecutive columns in a row, regardless of the number of columns in that row.

Listing 5.19 displays the contents of `switchcolumns3.sh`, which switches each pair of consecutive columns in `manycolumns.txt`.

LISTING 5.19: switchcolumns3.sh

```
awk '
{
  line = $0
  split(line, fields, " ")
  fieldCount = length(fields)
  fc2 = int(fieldCount/2)*2

  # switch consecutive columns
  for(idx=1; idx<=fc2; idx+= 2) {
     printf("%s,%s,", fields[idx+1], fields[idx])
  }
```

```
  # odd column count?
  if( fieldCount % 2 != 0) {
     printf("%s",fields[fieldCount])
  }

  # print linefeed
  printf("\n")
}
' manycolumns.txt
```

Listing 5.19 initializes the variable line as the current line and creates an array field whose contents are the columns of line. Next, the variable `fc2` is calculated as the largest even number that's no greater than the length of the array fields.

The next portion of Listing 5.19 contains a loop that switches consecutive columns of the current line. Notice that the subsequent `if` statement prints the rightmost field of the current line if the line has an odd number of fields. The last code snippet prints a linefeed (otherwise we would have a single line of output).

The output from the `awk` script is here:

```
ten
two,one,four,three,
four,three,two,one,four,three,
two,one,four,three,two,one,three
six,five,seven
two,one,four,three,five
two,one,one,three,three,two,two,one,three
```

There is one more detail to fix: remove the trailing "," that appears in rows with an even number of fields (can you explain why that happens?). One way to remove the trailing "," is with the `sed` command:

```
./switchcolumns3.sh | sed "s/,$//"
```

As you can see, the solution in Listing 5.19 is elegant in its simplicity (are you surprised?). In fact, there are even more simple solutions available, but the current solution demonstrates some of the other things that you can do in an `awk` script.

Although there are few situations where you need a shell script such as `columns3.sh` (possibly never), the point to keep in mind is that this task can be performed in a very simple manner, without the use of any REs. If you think that the latter is easy to do, see if you can create a suitable regular expression (hint: it's very difficult!).

Another point to keep in mind: the complexity of the solution to a particular task can vary among languages (or utilities), and it's worthwhile learning different languages—such as those discussed in this book—so that you can solve tasks more easily.

Finally, keep in mind that a short and simple solution is easier to debug and enhance, not only for you but also for the people who inherit your code.

A MORE COMPLEX EXAMPLE

The example in this section is admittedly more contrived than the other code samples, but it serves to illustrate the ease with which you can solve complex tasks with very simple `awk` scripts.

The `awk` script `rotaterows.sh` in this section does the following:

1. if the second field starts with the string six:
2. print the fourth, third, and first fields (if there are at least 4 fields), else
3. print the contents of the current row as-is
4. if the second field does not start with six, reverse the field order

Listing 5.20 displays the contents of `rotaterows.sh`, which performs the steps in the preceding list.

LISTING 5.20: rotaterows.sh

```
awk '
{
  if ($2 ~ /^six*/) {
     if(NF>=4) { print $4, $3, $1 }
     else      { print $0 }
  }
  else {
     for(i=NF;i>0;i--) printf "%s ",$i;print ""
  }
}
' manycolumns.txt
```

Listing 5.20 initializes the variable line as the current line and creates an array field whose contents are the columns of lines. Next, the `if` statement checks if the second fields starts with the string `six`, in which case it contains another code block that contains additional conditional logic. That logic prints the fourth, third, and first columns if the current row has at least four columns, otherwise it prints the contents of the current line.

The `else` portion of the code in Listing 5.20 is executed when the second column does not start with the string `six`, in which case a `for` loop is executed that reverses the order of the columns in the current row. The output from launching the code in Listing 5.20 is here:

```
ten
four three two one
four three two one four three
three two one four three two one
five six seven
five four three two one
three two one three two one three two one
```

Notice that Listing 5.20 is slightly shorter than Listing 5.19 from the previous use case, even though the current task is arguably more complex.

If you still aren't convinced of the power of `awk` scripts, suppose you need to do the following:

1. for rows with at least two and at most five columns:
2. if the second field starts with f and the fourth field starts with t: print the fourth, third, and first columns
3. otherwise reverse the order of the columns in the current row

Listing 5.21 displays the contents of `rotaterows2.sh`, which performs the steps in the preceding list.

LISTING 5.21: rotaterows2.sh

```
awk '
{
  if( NF >= 2 && NF <= 5) {
     if (($2 ~ /^f*/ && ($4 ~ /^t*/))) {
        print $4, $3, $1
     }
  }
  else {
     for(i=NF;i>0;i--) printf "%s ",$i;print ""
  }
}
' manycolumns.txt
```

If you have read the code in the previous two sections, the code in Listing 5.21 ought to be self-explanatory. Notice that Listing 5.21 has the same number of lines of code as Listing 5.20, despite having slightly greater complexity in terms of conditional logic.

Another point to notice is that Listing 5.21 is a straightforward implementation of the description of the task: if you read the code aloud, it's almost like English sentences, and the code contains only two simple REs.

CHAPTER SUMMARY

This chapter introduced you to the `sed` utility, illustrating the basic tasks of data transformation: allowing additions, removal, and mutation of data by matching individual patterns, or matching the position of the rows in a file, or a combination of the two.

Moreover, we showed that `sed` not only uses REs to match data, similar to the `grep` command, but can also use REs to describe how to transform the data.

Next you learned about the `awk` command, which is its own programming language that supports REs. A series of examples showed the versatility of the `awk` command, and hopefully communicated the sense that it is an even more flexible and powerful utility than we can show in a single chapter.

Now that you have finished this book, you might be interested in "next steps" to learn more about REs. The answer to this question varies widely,

mainly because the answer depends heavily on your objectives. The best answer is to try techniques from the book out on a problem or task you care about, professionally or personally. Precisely what that might be depends on who you are, as the needs of a data scientist, manager, student, or developer are all different. In addition, keep what you learned in mind as you tackle new challenges. Sometimes knowing a technique is possible makes finding a solution easier, even if you have to reread the section to remember exactly how the syntax works. In addition, there are various online resources and literature describing how to create complex and arcane regular expressions.

At this point there is one more thing to say: congratulations! You have completed a fast-paced yet dense book, and if you are an RE neophyte, the material will probably keep you busy for many hours. The examples in the chapters provide a solid foundation, and the Appendices contain additional examples of REs in Perl, Java, and Scala. The combined effect demonstrates that the universe of possibilities is larger than the examples in this book, and ultimately they will spark ideas in you. Good luck!

APPENDIX A

REs IN PERL

This Appendix contains an assortment of REs in Perl, with code snippets from earlier chapters that have been converted to Perl syntax. Please keep in mind that you will learn only rudimentary Perl functionality that pertains to REs, and that Perl has powerful features that are not discussed because they are beyond the scope of this Appendix.

The first section of this chapter is similar to the examples in Chapter 1, but without fully replicating the same details. Although the REs in this section are often the same as their counterparts in Chapter 1, there are some syntactic differences when you invoke Perl "one-liners" from the command line, versus doing so with the `grep` command.

The second section in this chapter contains a description of metacharacters and character classes, along with code snippets that illustrate how to use them. For example, you will see how to match alphabetic characters (uppercase, lowercase, or a combination of both types), pure digits, and regular expressions with combinations of digits and alphabetic characters.

The third section contains REs that match dates, phone numbers, and zip codes. This section also contains REs that match various types of numbers, such as integers, decimals, hexadecimals, octals, and binary numbers. You will also learn how to work with scientific numbers and REs.

The final section contains REs that match IP addresses and simple comment strings (in source code), as well as REs for matching ISBNs.

SIMPLE EXAMPLES OF REs

Recall that Chapter 1 uses the Unix `grep` utility and the Unix `egrep` utility to illustrate various REs, whereas this Appendix uses the Perl executable. If you work on a PC, please read the Preface for information about software to download to your PC so that you can run Perl commands.

Listing A.1 displays the contents of `lines1.txt`, which contains several lines of text that match various REs in this section.

LISTING A.1: lines1.txt

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

As you can see, the word `grey` appears in the first and second lines, the word `gray` appears in the first and third lines, and all three lines contain either `grey` or `gray`.

Here are the tasks that we want to perform:

1. Find the lines that contain grey.
2. Find the lines that contain gray.
3. Find the lines that contain either grey or gray.

The following command performs the first task:

```
perl -wln -e 'print if /gray/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
```

The preceding Perl command contains command-line options that specify the following:

-w : Use warnings.
-l : Remove ("chomp" in Perl parlance) the newline character from each line before processing and place it back during printing.
-n : Create an implicit while(<>) { ... } loop to perform an action on each line.
-e : Direct the Perl interpreter to execute the code that follows it.
Finally, print the entire line if the line contains the word `gray`.

The following command performs the second task:

```
perl -wln -e 'print if /grey/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
that cat is gray
```

Your First Character Class

The examples in the previous section show you how to search for hard-coded strings in a text file. You can combine two search expressions into one by using a character class. Specifically, suppose you want to search for either `gray`

or `grey` in a text file, which means matching with the vowel `a` or the vowel `e`. Square brackets provide this functionality: the term `[ae]` means "use either a or e" (and later you'll see other variations, such as a range of letters or numbers).

The following command performs the third task listed in the previous section:

```
perl -wln -e 'print if /gr[ae]y/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

The term `gr[ae]y` is a compact way of representing the two strings `gray` and `grey`. The order of the letters in the square brackets is irrelevant, which means that the third task can also be solved with this command:

```
perl -wln -e 'print if /gr[ae]y/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

Specifying a Range of Letters

We can "expand" the RE in the preceding code snippet to include all the lowercase letters of the alphabet, which is represented by `[a-z]`. We can find all the lines that contain a string that is of the form `gr[a-z]y`, which matches any string that meets the following conditions:

1. start with the letters `gr`
2. followed by any single letter a, b, c, …, z
3. end with the letter `y`

Just to confirm, launch the following command:

```
perl -wln -e 'print if /gr[az]y/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

The only matches are `grey` and `gray`, but if the text file included a line with the string `grzy`, then this line would appear in the previous output.

We can also specify a single letter inside the square brackets. For example, the term `[a]` is an RE that matches the letter `a`. Launch this command:

```
perl -wln -e 'print if /[a]/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
that cat is gray
```

If we specify a vowel that does not appear in any word in `lines1.txt`, then there is no output. An example is here:

```
perl -wln -e 'print if /[u]/' lines1.txt
```

We can specify different ranges of letters. For example, suppose we want to find the lines that contain words with any vowel except for the vowels `a` or `i`. This expression will do the job:

```
perl -wln -e 'print if /[eou]/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
```

Once again, the order of the letters in the square brackets is irrelevant, which means that the following commands have the same output:

```
perl -wln -e 'print if /[eou]/' lines1.txt
perl -wln -e 'print if /[oeu]/' lines1.txt
perl -wln -e 'print if /[oue]/' lines1.txt
```

WORKING WITH THE "^" AND "$" METACHARACTERS

The "`^`" metacharacter matches a pattern that starts from the beginning of a line. For example, the RE "`^the`" matches any lines that start with the string `the`, as shown here:

```
perl -wln -e 'print if /^the/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

On the other hand, the RE "`^[the]`" matches any lines that start with either a `t`, or an `h`, or an `e`, as shown here:

```
perl -wln -e 'print if /^[the]/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

Excluding Matches with the "^" Metacharacter

The interpretation of the "^" metacharacter is based on whether it's specified inside or outside (and precedes) a pair of square brackets. If it's inside the brackets, it means "do not use these letters", and if it precedes the brackets (as you saw in the previous section), it means "starting from the leftmost position of a string, match the pattern that follows the ^ character".

For example, the following RE matches any lines that start with the letter t:

```
perl -wln -e 'print if /^[t]/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

By contrast, the following RE matches any lines that do *not* start with the letter t, and in this case, there are no matching lines:

```
perl -wln -e 'print if /^[^t]/' lines1.txt
```

Since every line starts with the letter t, you can specify any other letter in the preceding code snippet and the result matches all the lines in the text file. For example, the following RE matches all lines:

```
perl -wln -e 'print if /^[^z]/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

Matches with the "$" Metacharacter

The "$" metacharacter enables you to match letters or words that appear at the end of a line. For example, the following expression matches any lines that end with the word gray:

```
perl -wln -e 'print if /gray$/' lines1.txt
```

The output is here:

```
that cat is gray
```

Notice that the first line is excluded: the next section explains why this happened, and also the type of RE that will match the first line.

WORKING WITH ".", "*", AND "\" METACHARACTERS

The "." metacharacter matches any single character (except a linefeed). At the other extreme is the "*" metacharacter that matches zero or more

occurrences of any character. The "*" metacharacter is useful when you want to match the intervening letters between a start character (or word) and an end character (or word).

For example, if you want to match the lines that start with the letter `t`, followed by an occurrence of the word `gray`, use this expression:

```
perl -wln -e 'print if /^t.*gray/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
that cat is gray
```

Notice how the "*" metacharacter enables you to "ignore" the intervening characters between the initial `t` and the occurrence of the word `gray` somewhere else in a line.

If you want to match the lines that start with the word `the`, followed by an occurrence of the word `gray`, use this expression:

```
perl -wln -e 'print if /^the.*gray/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

You can match the final "." character with the following expression:

```
perl -wln -e 'print if /^t.*gray.$/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

The following expression only matches one line:

```
perl -wln -e 'print if /^the.*gray.$/' lines1.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

Yet another variation of the preceding code snippet is here:

```
perl -wln -e 'print if /^t.*gra.$/' lines1.txt
```

The output is here:

```
that cat is gray
```

Checking for Whitespaces

Listing A.2 displays the contents of `spaces.txt`, which contains several lines of text consisting many of whitespaces.

LISTING A.2: spaces.txt

```
                        x
y
z                       w

a
```

Match all lines that contain a whitespace with this expression:

```
perl -wln -e 'print if / /' lines1.txt
```

The output is here:

```
                        x
y
z                       w
```

Match all lines that start with a whitespace with this expression:

```
perl -wln -e 'print if /^ /' lines1.txt
```

The output (of two lines) is here:

```
                        x
```

Match all lines that end with a whitespace with this expression:

```
perl -wln -e 'print if / $/' lines1.txt
```

The output (of two lines) is here:

```
y
```

Match lines that contain only whitespaces with this expression:

```
perl -wln -e 'print if /^[ ][ ]+$/' lines1.txt
```

The output consists of one blank line.

Note that the following REs will not match just the lines that contain only whitespaces:

```
perl -wln -e 'print if /[ ][ ]+.*$/'  spaces.txt
perl -wln -e 'print if /^[ ][ ].$/'   spaces.txt
perl -wln -e 'print if /[ ][ ].*$/'   spaces.txt
perl -wln -e 'print if /^[ ]*'        spaces.txt
perl -wln -e 'print if /^[ ].*$/'     spaces.txt
```

Test your understanding of the metacharacters in this section by determining why the preceding REs also match lines that contain characters other than whitespaces.

Match empty lines with this very simple expression:

```
perl -wln -e 'print if /^$/' spaces.txt
```

The output is a blank line, which you will see on the screen. *Note* that matching an empty line is different from matching a line containing only whitespaces.

ESCAPING A METACHARACTER

Use the backslash "\" character to "escape" the interpretation of metacharacters. For example, the term "\." escapes the "." and matches a "." character that appears inside a line.

Listing A.3 displays the contents of `lines2.txt`, which contains several lines of text and an embedded "." character.

LISTING A.3: lines2.txt

```
the dog is grey. the cat is gray.
this dog is called doc.
that cat is called .doc
```

If you want to match the lines that start with the letter `t` and also end with the word `gray`, use this expression:

```
perl -wln -e 'print if /^t.*gray\.$/' lines2.txt
```

The output is here:

```
the dog is grey and the cat is gray.
```

If you want to match the lines that contain a ".", use this expression:

```
perl -wln -e 'print if /\.$/' lines2.txt
```

The output is here:

```
the dog is grey. the cat is gray.
this dog is called doc.
that cat is called .doc
```

If you want to match the lines that match `.doc`, use this expression:

```
perl -wln -e 'print if /\.doc/' lines2.txt
```

The output is here:

```
that cat is called .doc
```

The following expression matches the lines that end with `.doc`:

```
perl -wln -e 'print if /\.doc$/' lines2.txt
```

The output is here:

```
that cat is called .doc
```

MIXING (AND ESCAPING) METACHARACTERS

Listing A.4 displays the contents of `lines3.txt`, which is used in some RE code snippets in this section.

LISTING A.4: lines3.txt

```
grey.
.gray
dog
doggy

cat
catty
 catfish
 small catfish
```

If you want to match the lines that contain `dog`, use this expression:

```
perl -wln -e 'print if /dog/' lines3.txt
```

The output is here:

```
dog
doggy
```

If you want to match the lines that start with the word `dog`, use this expression:

```
perl -wln -e 'print if /^dog/' lines3.txt
```

The output is here:

```
dog
doggy
```

If you want to match the lines that end with the word `dog`, use this expression:

```
perl -wln -e 'print if /dog$/' lines3.txt
```

The output is here:

```
dog
```

If you want to match the lines that start and also end with the word `dog`, use this expression:

```
perl -wln -e 'print if /^dog$/' lines3.txt
```

The output is here:

```
dog
```

If you want to match the lines that start with a blank space, use this expression:

```
perl -wln -e 'print if /^ /' lines3.txt
```

The output is here:

```
 catfish
```

If you want to match the lines that start with a period, use this expression:

```
perl -wln -e 'print if /^\./' lines3.txt
```

The output is here:

```
.gray
```

If you want to match the lines with any occurrence of a period, use this expression:

```
perl -wln -e 'print if /\./' lines3.txt
```

The output is here:

```
grey.
.gray
```

By contrast, the following expression matches all lines because the "." metacharacter has not been escaped:

```
perl -wln -e 'print if /^./' lines3.txt
```

The output is here:

```
grey.
.gray
dog
doggy
cat
catty
 catfish
 small catfish
```

The following expression matches lines that contain the string that ends with `cat`:

```
perl -wln -e 'print if /cat\b/' lines3.txt
```

The output is here:

```
cat
```

The following expression matches lines that start with a space, followed by any characters, and then followed by the string `cat`:

```
perl -wln -e 'print if /[ ].*cat/' lines3.txt
```

The output is here:

```
 catfish
 small catfish
```

The following expression matches lines that contain the letter `r` or the letter `e`:

```
perl -wln -e 'print if /[re]/' lines3.txt
```

The output is here:

```
grey.
.gray
```

The following expression matches lines that contain the letter `g`, followed by either the letter `r` or the letter `e`:

```
perl -wln -e 'print if /g[re]/' lines3.txt
```

The output is here:

```
grey.
```

The following three REs match the word `grey`:

```
perl -wln -e 'print if /^[.g][re]/'  lines3.txt
perl -wln -e 'print if /^[\.g][re]/' lines3.txt
perl -wln -e 'print if /^[^.][re]/'  lines3.txt
```

This RE matches the word `.gray`:

```
perl -wln -e 'print if /^.[g][re]/' lines3.txt
```

THE "?" METACHARACTER

Listing A.5 displays the contents of `lines4.txt`, which is used in some RE code snippets in this section.

LISTING A.5: lines4.txt

```
yes?
yes123? or no?
maybe?
maybe...? perhaps?
either/or? or yes?
```

The following expression matches lines that contain a period before a question mark "?":

```
perl -wln -e 'print if /\.[?]/' lines4.txt
```

The output is here:

```
maybe...? perhaps?
```

The following expression matches lines that contain the word `or`:

```
perl -wln -e 'print if /or/' lines4.txt
```

The output is here:

```
yes123? or no?
either/or? or yes?
```

The following expression matches lines that match the sequence of the word `or`, followed by a `?`, followed by a blank space, and then another occurrence of `or`:

```
perl -wln -e 'print if /or? or/' lines4.txt
```

The output is here:

```
either/or? or yes?
```

The following expression matches lines that contain three consecutive dot "." characters:

```
perl -wln -e 'print if /\.\.\./' lines4.txt
```

The output is here:

```
maybe...? perhaps?
```

The following expression matches all lines:

```
perl -wln -e 'print if /.../' lines4.txt
```

The output is here:

```
yes?
yes123? or no?
maybe?
maybe...? perhaps?
either/or? or yes?
```

The following expression matches lines that start with the word `yes`, and are optionally followed by the number `123`:

```
perl -wln -e 'print if /^yes([123])?/' lines4.txt
```

The output is here:

```
yes?
yes123? or no?
```

The "|" Metacharacter

The following expression matches lines that start with `maybe` or with `either`:

```
perl -wln -e 'print if /(maybe|either)/' lines4.txt
```

The output is here:

```
maybe?
maybe...? perhaps?
either/or? or yes?
```

DATES AND METACHARACTERS

Listing A.6 displays the contents of `lines5.txt`, which is used in some code snippets. As you can see, some of the lines contain valid dates and others contain invalid date formats.

LISTING A.6: lines5.txt

```
05/12/18
05/12/2018
05912918
05.12.18
05.12.2018
0591292018
```

The following expression matches lines that start with two digits (followed by anything):

```
perl -wln -e 'print if /^[0-9][0-9]\//' lines5.txt
```

The output is here:

```
05/12/18
05/12/2018
05912918
05.12.18
05.12.2018
0591292018
```

The following expression matches lines that start with two digits (followed by a forward slash):

```
perl -wln -e 'print if /^[0-9][0-9]/' lines5.txt
```

The output is here:

```
05/12/18
05/12/2018
```

The following expression matches lines that start with two digits (followed by a forward slash or period):

```
perl -wln -e 'print if /^[0-9][0-9][/ .]/' lines5.txt
```

The output is here:

```
05/12/18
05/12/2018
```

```
05.12.18
05.12.2018
```

The following expression matches lines that end with a forward slash or period, preceded by two digits:

```
perl -wln -e 'print if /[/ .][0-9][0-9]$/' lines5.txt
```

The output is here:

```
05/12/18
05/12/2018
05.12.18
05.12.2018
```

By contrast, the following expression matches lines that *contain* a forward slash or period, followed by two digits:

```
perl -wln -e 'print if /[/ .][0-9][0-9]/' lines5.txt
```

The output is here:

```
05/12/18
05.12.18
```

The following expression matches lines that contain four consecutive digits:

```
perl -wln -e 'print if /[0-9][0-9][0-9]/' lines5.txt
```

The output is here:

```
05/12/2018
05912918
05.12.2018
0591292018
```

The following expressions both match lines that end with four consecutive digits that are preceded by a forward slash or period:

```
perl -wln -e 'print if /[\/.][0-9][0-9][0-9]/' lines5.txt
perl -wln -e 'print if /[\/.][0-9][0-9][0-9]$/' lines5.txt
```

The output is here:

```
05/12/18
05/12/2018
05.12.18
05.12.2018
```

Working with \d and \D Metaclasses

A simpler (cleaner?) way to match a digit involves the `\d` character class (`d` is for digit). The following expression matches lines that contain three consecutive digits:

```
perl -wln -e 'print if /\d\d\d/' lines5.txt
```

The output is here:

```
05/12/2018
05912918
05.12.2018
0591292018
```

There is also a simpler way to match multiple consecutive digits via the `\d` character class. The following expression matches lines that contain three consecutive digits:

```
perl -wln -e 'print if /\d{3}/' lines5.txt
05/12/2018
05912918
05.12.2018
0591292018
```

The following expression matches lines that contain a pair of digits followed by a non-digit character:

```
perl -wln -e 'print if /\d{2}\D/' lines5.txt
05/12/18
05/12/2018
05.12.18
05.12.2018
```

The following expression matches lines that contain three pairs of digits that are separated by a non-digit character:

```
perl -wln -e 'print if /\d{2}\D\d{2}\D\d{2}/' lines5.txt
05/12/18
05/12/2018
05.12.18
05.12.2018
```

The following expression matches lines that contain three pairs of digits that are separated by a non-digit character, and also *excludes* four-digit sequences:

```
perl -wln -e 'print if /\d{2}\D\d{2}\D\d{2}$/' lines5.txt
05/12/18
05.12.18
```

REs AND ZIP CODES (U.S. AND CANADIAN)

Listing A.7 displays the contents of `lines6.txt`, which is used in some code snippets.

LISTING A.7: lines6.txt

```
94053
94053-06123
9405306123
V6K 8Z3
```

```
36K8Z3
123-45-6789
jsmith@acme.com
john.smith@acme.com
650 123-4567
650 123 4567
(650) 123 4567
1-650 123-4567
```

The following RE matches strings that contain five digits, which is a common U.S. zip code pattern:

```
perl -wln -e 'print if /\d{5}/' lines6.txt
```

The output is here:

```
94053
94053-06123s
9405306123
```

The following expression matches U.S. zip codes consisting of five digits:

```
perl -wln -e 'print if /^\d{5}$/' lines6.txt
```

The output is here:

```
94053
```

The following expression matches U.S. zip codes consisting of five digits followed by a hyphen, and then followed by another five digits:

```
perl -wln -e 'print if /^\d{5}-\d{5}$/' lines6.txt
```

The output is here:

```
94053-06123
```

Valid Canadian postal codes are of the form `A1A 1A1`, where A is a capital letter and 1 is a digit (with a space between the two triplets). The following RE matches Canadian zip codes:

```
egrep "^[A-Z][0-9][A-Z] [0-9][A-Z][0-9]" lines6.txt
```

The output is here:

```
V6K 8Z3
```

The following RE matches U.S. social security numbers (SSNs) consisting of three digits followed by a hyphen, two digits followed by a hyphen, and ending with three digits:

```
perl -wln -e 'print if /^\d{3}-\d{2}-\d{4}/' lines6.txt
```

The output is here:

```
123-45-6789
```

REs AND EMAIL ADDRESSES

Matching email addresses is a complex task. This section provides REs that match common (but not all) email addresses that have the following pattern:

1. an initial string having at least four characters and at most twelve characters (which can be any combination of lowercase letters, uppercase letters, or digits), then
2. followed by the "@" symbol, then
3. a string having at least four characters and at most twelve characters (which can be any combination of lowercase letters, uppercase letters, or digits), then
4. followed by the string ".com"

Here is the RE that has the structure described in the preceding list that matches an email address:

```
perl -wln -e 'print if /^[A-Za-z0-9]{4,12}\@[A-Za-z0-9]{4,8}\.
com$/' lines6.txt
```

The output is here:

```
jsmith@acme.com
```

There are some limitations regarding the preceding RE. First, it only matches email addresses with the suffix `.com`. Second, longer (yet still valid) email addresses are excluded, such as the one shown here:

```
myverylongemailaddress@acme.com
```

Third, it excludes special characters (such as `_`, `%`, and so forth) as part of the email address.

Consequently, you need to make decisions about the allowable set of email addresses that you want to match with your RE.

The following RE has the structure described in the preceding list, and also allows a dot "." as in the initial portion of the email address:

```
perl -wln -e 'print if /^[A-Za-z0-9]{4,12}\.[A-Za-z0-9]{4,12}\@
[A-Za-z0-9]{4,8}\.com$/' lines6.txt
```

The output is here:

```
john.smith@acme.com
```

The section shown in bold in the preceding RE shows you how to match the dot "." character, followed by an alphanumeric string that has at least four characters and at most twelve characters.

REs AND U.S. PHONE NUMBERS

The following RE matches U.S. phone numbers of the form `ddd ddd dddd`:

```
perl -wln -e 'print if /^\d{3} \d{3} \d{4}/' lines6.txt
```

The output is here:

```
650 123 4567
```

The following RE matches U.S. phone numbers of the form `ddd ddd-dddd`:

```
perl -wln -e 'print if /^\d{3} \d{3}-\d{4}/' lines6.txt
```

The output is here:

```
650 123-4567
```

The following RE matches U.S. phone numbers of the form `(ddd) ddd-dddd`:

```
perl -wln -e 'print if /^\(\d{3}\) \d{3}-\d{4}/' lines6.txt
```

The output is here:

```
(650) 123-4567
```

The following RE matches U.S. phone numbers of the form `1-ddd ddd-dddd`:

```
perl -wln -e 'print if /^1-\d{3} \d{3}-\d{4}/' lines6.txt
```

The output is here:

```
1-(650) 123-4567
```

WORKING WITH NUMBERS

This section contains examples of REs that match integers, floating point numbers, hexadecimal numbers, octal numbers, and binary numbers. The subsequent section discusses REs for scientific numbers, which are a "generalization" of decimal numbers: they are more complex, and so they merit their own section.

Listing A.8 displays the contents of `numbers.txt`, which is used in some RE code snippets in this section.

LISTING A.8: numbers.txt

```
#integers
1234
-123
```

```
#floating point numbers
1234.432
-123.528
0.458

#hexadecimal numbers
12345
FA4389
0xFA4389
0X4A3E5C

#octal numbers
1234
03434

#binary numbers
010101
110101
0b010101
```

REs, Integers, and Decimal Numbers

The following RE matches positive integers and negative integers:

```
perl -wln -e 'print if /^[-|+]?\d+$/' numbers.txt
```

The output is here:

```
1234
-123
12345
1234
03434
010101
110101
```

The following RE matches positive integers, negative integers, and decimal numbers:

```
perl -wln -e 'print if /^[-|+]?\d+([\.]?\d*)$/' numbers.txt
```

The output is here:

```
1234
-123
1234.432
-123.528
0.458
12345
1234
03434
010101
110101
```

The following RE matches only decimal numbers:

```
perl -wln -e 'print if /^[-|+]?\d+([\.]\d*)$/' numbers.txt
```

The output is here:

```
1234.432
-123.528
0.458
```

REs and Hexadecimal Numbers

Hexadecimal numbers can contain the digits 0 through 9 and also the letters `A` through `F`, and can also start with `0x` or `0X` (both of which are optional). The following RE matches hexadecimal numbers (and other patterns as well) without a `0x` or `0X` prefix:

```
perl -wln -e 'print if /^[a-fA-F0-9]+$/' numbers.txt
```

The output is here:

```
1234
12345
FA4389
1234
03434
010101
110101
0b010101
```

Notice that integers, octal numbers, and binary numbers also appear in the preceding list (because they are valid hexadecimal numbers).

The following RE matches hexadecimal numbers that start with either `0x` or `0X`:

```
perl -wln -e 'print if /^(0x|0X)[a-fA-F0-9]+$/' numbers.txt
```

The output is here:

```
0xFA4389
0X4A3E5C
```

REs and Octal Numbers

The following RE matches octal numbers without a 0 prefix:

```
perl -wln -e 'print if /^[1-7][0-7]+$/' numbers.txt
```

The output is here:

```
1234
110101
1234
110101
```

Notice that there are two occurrences of the number `1234`: the first one appears as an integer (and it's a valid octal number) and the second one appears in the section with octal numbers. Moreover, the number `110101` from the binary section is also a valid octal number.

The following RE matches octal numbers with a 0 prefix:

```
perl -wln -e 'print if /^0?[1-7]+$/' numbers.txt
```

The output is here:

```
1234
12345
1234
03434
```

Once again, there are two occurrences of the number `1234`: the first one appears as an integer (and it's a valid octal number) and the second one appears in the section with octal numbers.

REs and Binary Numbers

The following RE matches binary numbers without a `0b` prefix:

```
perl -wln -e 'print if /^[0-1]+$/' numbers.txt
```

The output is here:

```
010101
110101
```

The following RE matches binary numbers with or without a 0b prefix:

```
perl -wln -e 'print if /(^[0-1]+|0b[0-1]+)$/' numbers.txt
```

The output is here:

```
010101
110101
0b010101
```

REs AND SCIENTIFIC NUMBERS

This section contains examples of REs that match scientific numbers and hexadecimal numbers.

Listing A.9 displays the contents of `lines7.txt` and Listing A.10 displays the contents of `lines8.txt`, which are used in some code snippets.

LISTING A.9: lines7.txt

```
192.168.3.99
192.168.123.065
// this is a comment
v = 7; // this is also a comment
/* the third comment */
x = 7; /* the fourth comment */
y = \uFFEA;
MyPaSsW0rd
mypasss0rd
```

LISTING A.10: lines8.txt

```
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
```

Matching all scientific numbers (and nothing else) is rather complex, and this section contains some REs that *partially* succeed in this task. A useful exercise for you is to determine why these REs contain "false positives" (i.e., strings that you want to exclude).

Option #1: the following RE matches scientific numbers:

```
perl -wln -e 'print if /^[+-]?\d*(([,.]\d{3})+)?([,.]\d+)?([eE][+-
]?\d+)?$'/ lines8.txt
```

The output is here:

```
0.123
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
```

Unfortunately, the preceding RE also matches IP addresses (in `lines7.txt`):

```
perl -wln -e 'print if /^[+-]?\d*(([,.]\d{3})+)?([,.]\d+)?([eE][+-
]?\d+)?$'/ lines7.txt
```

The output is here:

```
192.168.123.065
```

Option #2: the following RE matches scientific numbers:

```
perl -wln -e 'print if /[-]?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?'
lines8.txt
```

The output is here:

```
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
```

For your convenience, Listing A.11 displays the contents of `scientific.sh`, which contains an assortment of REs that check for scientific numbers, tested against the file `lines8.txt`.

LISTING A.11: scientific.sh

```
echo "*** Option #1:"
echo "--------------"
perl -wln -e 'print if /^[+-]?\d*(([,.]\d{3})+)?([,.]\d+)?([eE]
[+-]?\d+)?$'/ lines8.txt

echo "*** Option #2:"
echo "--------------"
perl -wln -e 'print if /^[+-]?\d*(([,.]\d{3})+)?([,.]\d+)?([eE]
[+-]?\d+)?$'/ lines8.txt

echo "*** Option #3:"
echo "--------------"
perl -wln -e 'print if /[-]?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?'/
lines8.txt

echo "*** Option #4:"
echo "--------------"
perl -wln -e 'print if /[-]?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?'/
lines8.txt

echo "*** Option #5:"
echo "--------------"
perl -wln -e 'print if /[-]?(?:0|[1-9]\d*)(?:\.\d*)?(?:[eE][+\-]?\
d+)?'/ lines8.txt

echo "*** Option #6:"
echo "--------------"
perl -wln -e 'print if /[-]?(?:0|[1-9]\d*)(?:\.\d*)?(?:[eE][+\-]?\
d+)?'/ lines8.txt

echo "*** Option #7:"
echo "--------------"
perl -wln -e 'print if /[-]?(?:0|[1-9]\d*)(?:\.\d*)?(?:[eE][+\-]?\
d+)?'/ lines8.txt

echo "*** Option #8:"
echo "--------------"
perl -wln -e 'print if /[+\-]?(?:0|[1-9]\d*)(?:\.\d*)?(?:[eE]
[+\-]?\d+)?'/ lines8.txt
```

Launch the code in Listing A.9 with this command:

```
./scientific.sh > scientific.out
```

Listing A.12 displays the contents of `scientific.out`, which displays the result of launching the shell script `scientific.sh` displayed in Listing A.11.

LISTING A.12: scientific.out

```
*** Option #1:
--------------
0.123
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
*** Option #2:
--------------
192.168.123.065
*** Option #3:
--------------
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
*** Option #4:
--------------
192.168.3.99
192.168.123.065
v = 7; // this is also a comment
x = 7; /* the fourth comment */
*** Option #5:
--------------
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
*** Option #6:
--------------
192.168.3.99
192.168.123.065
v = 7; // this is also a comment
x = 7; /* the fourth comment */
*** Option #7:
--------------
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
```

```
+0005
125.e12
*** Option #8:
--------------
0.123
z = 0xFFFF00;
+13
423.2e32
-7.20e+19
-.4E-8
-27.6603
+0005
125.e12
```

As you can see, the REs in Listing A.10 have varying degrees of success in terms of matching scientific numbers. In general, they err by matching "false positives" (numbers that are *not* valid scientific numbers) instead of excluding "false negatives" (numbers that *are* valid scientific numbers).

DETECTING URL PATTERNS

Listing A.13 displays the contents of `urls.txt`, which contains examples of `ftp`, `http`, and `https` URLs.

LISTING A.13: urls.txt

```
ftp://www.acme.com
http://www.bdnf.com
https://www.ceog.com
a line with https://www.ceog.com embedded in it
```

The following snippet matches the lines that contain the strings `http` or `https`:

```
perl -wln -e 'print if /http/' urls.txt
```

The output is here:

```
ftp://www.acme.com
http://www.bdnf.com
https://www.ceog.com
a line with https://www.ceog.com embedded in it
```

The following snippet matches the lines that contain the strings `http` or `https`:

```
perl -wln -e 'print if /http?/' urls.txt
```

The output is here:

```
http://www.bdnf.com
https://www.ceog.com
a line with https://www.ceog.com embedded in it
```

The following snippet matches the lines that contain the strings `ftp`, `http`, or `https`:

```
ftp://www.acme.com
http://www.bdnf.com
https://www.ceog.com
a line with https://www.ceog.com embedded in it
```

The following snippet matches the lines that contain the string `http` embedded in the line of text:

```
perl -wln -e 'print if /^([a-z ]+)https?://[a-z\.]*/' urls.txt
```

The output is here:

```
a line with https://www.ceog.com embedded in it
```

One interesting point: the equivalent RE with the `egrep` command is here (the initial whitespace is specified in a *different* location):

```
egrep "^([ a-z]+)https?://[a-z\.]*" urls.txt
```

The preceding code snippet specifies a whitespace and any lowercase letter in this expression: `[ a-z]`. However, the corresponding section in the Perl expression must include the whitespace *after* the range of lowercase letters: `[a-z ]`. If you do not make this slight modification, you will see the following error message:

```
Unquoted string "a" may clash with future reserved word at -e line 1.
syntax error at -e line 1, near "a-z"
```

REs AND ISBNS

Valid IBSNs can start with the optional string `ISBN`, and also contain either ten-digit sequences or thirteen-digit sequences. Listing A.13 displays the contents of `ISBN.txt`, which contains examples of valid ISBN numbers.

LISTING A.14: ISBN.txt

```
ISBN 978-0-596-52068-7
ISBN-13: 978-0-596-52068-7
ISBN-10 0-596-52068-9
978 0 596 52068 7
9780596520687
0-596-52068-9
```

Notice that the first line in Listing A.14 contains the string `ISBN` followed by a blank space, and the next two lines contain the string `ISBN`, followed by a hyphen, and then two more digits, and then either a colon ":" or a blank space. Those two lines end with a hyphenated thirteen-digit number and a hyphenated ten-digit number, respectively.

The fourth line in Listing A.14 contains a thirteen-digit number with white spaces; the fifth line contains a "pure" thirteen-digit number; and the sixth line contains a hyphenated ten-digit number.

Now let's see how to match the numeric portion of the `ISBN`s in Listing A.14. The following RE matches the digits in the first and the second lines:

```
\d{3}-\d-\d{3}-\d{5}-\d
```

The following RE matches the digits in the third line as well as the sixth line:

```
\d-\d{3}-\d{5}-\d
```

The following RE matches the digits in the fourth line:

```
\d{3} \d \d{3} \d{5} \d
```

The following RE matches the digits in the fifth line:

```
\d{13}
```

Now let's create REs for the text prefix (when present) and combine them with the earlier list of REs to match entire lines in Listing A.14. The result involves four REs, as shown in the following examples:

1. the RE `(^([A-Z]{4}[ -]?)? \d{3}-\d-\d{3}-\d{5}-\d)` matches:

```
ISBN 978-0-596-52068-7
ISBN-13: 978-0-596-52068-7
ISBN-10 0-596-52068-9
```

2. the RE `(\d{3} \d \d{3} \d{5} \d)` matches:

```
978-0-596-52068-7
978 0 596 52068 7
```

3. the RE `(\d{13})` matches:

```
9780596520687
```

4. the RE `(\d-\d{3}-\d{5}-\d)` matches:

```
0-596-52068-9
```

Now we can combine the preceding four REs to create a single (and lengthy) RE that matches every valid `ISBN` in the text file `ISBN.txt`:

```
perl -wln -e 'print if /^((([A-Z]{4}-\d{2}: \d{3}-\d-)|([A-Z]{4}-\
d{2} \d-))?\d{3}-\d{5}-\d)|(^([A-Z]{4}[ -]?)? \d{3}-\d-\d{3}-\d{5}-\
d)|(\d{3} \d \d{3} \d{5} \d)|(\d{13})|(\d-\d{3}-\d{5}-\d)/' ISBN.txt
```

If you decide to use the preceding RE in a bash script, please include a comment block that explains how you derived the RE and any other assumptions that you made for the task at hand.

MISCELLANEOUS PATTERNS

This section contains examples of REs that match simple comment strings (in source code). Listing A.15 displays the contents of `lines7.txt`, which is used in some code snippets.

LISTING A.15: lines7.txt

```
192.168.3.99
192.168.123.065
// this is a comment
v = 7; // this is also a comment
/* the third comment */
x = 7; /* the fourth comment */
y = \uFFEA;
z = 0xFFFF00;
MyPaSsW0rd
mypasss0rd
```

The following RE matches lines that start with `//`:

```
perl -wln -e 'print if /^\/\//' lines7.txt
```

The output is here:

```
// this is a comment
```

The following RE matches lines that contain any occurrence of `//`:

```
perl -wln -e 'print if /\/\/' lines7.txt
```

The output is here:

```
// this is a comment
v = 7; // this is also a comment
```

The following RE matches lines that start with `/*`:

```
perl -wln -e 'print if /^\/\*/' lines7.txt
```

The output is here:

```
/* the third comment */
```

The following RE matches lines that contain an occurrence of `/*`:

```
perl -wln -e 'print if /^\/\*/' lines7.txt
```

The output is here:

```
/* the third comment */
x = 7; /* the fourth comment */
```

REs AND IP ADDRESSES

This section contains examples of REs that match IP addresses that are in Listing A.15 in the previous section.

The following RE matches arbitrary valid IP addresses:

```
perl -wln -e 'print if /^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'
lines7.txt
```

The output is here:

```
192.168.3.99
192.168.123.065
```

The following RE matches valid IP addresses that contain three digits in all four components:

```
perl -wln -e 'print if /^\d{3}\.\d{3}\.\d{3}\.\d{3}' lines7.txt
```

The output is here:

```
192.168.123.065
```

MIXED-CASE STRINGS AND REs

This section contains examples of REs that match mixed-case strings (typically user names). Listing A.15 displays the contents of `lines10.txt`, which is used in some code snippets.

LISTING A.16: lines9.txt

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
He is John smith. the cat is gray.
He is john smith. the cat is gray.
that cat is called .doc
```

The following RE matches mixed-case strings:

```
perl -wln -e 'print if /[A-Z][a-z]+'/ lines9.txt
```

The output is here:

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
```

The following RE matches mixed-case strings that end with a period ".":

```
perl -wln -e 'print if /[A-Z][a-z]+\.'/ lines9.txt
```

The output is here:

```
He is John Smith. the cat is gray.
```

The following RE matches mixed-case strings that start with an uppercase letter:

```
perl -wln -e 'print if /^[A-Z][a-z]+'/ lines9.txt
```

The output is here:

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
He is John smith. the cat is gray.
He is john smith. the cat is gray.
```

The following RE matches strings that start with an uppercase letter followed by a space, another lowercase string, and end in a period ".":

```
perl -wln -e 'print if /[A-Z][a-z]+ [a-z]+\.'/ lines9.txt
```

The output is here:

```
He is John smith. the cat is gray.
```

The following RE matches strings that start with an uppercase or lowercase `J`, followed by the letters `ohn`:

```
perl -wln -e 'print if /[Jj]ohn'/ lines9.txt
```

The output is here:

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
He is John smith. the cat is gray.
He is john smith. the cat is gray.
```

Another RE that uses the "|" metacharacter to match strings that contain either `John` or `john` is here:

```
perl -wln -e 'print if /(John|john)'/ lines9.txt
```

The output is here:

```
John Smith is grey. the cat is gray.
He is John Smith. the cat is gray.
He is John smith. the cat is gray.
He is john smith. the cat is gray.
```

The following RE matches strings that do *not* start with an uppercase or lowercase `J`, followed by the letters `ohn`:

```
perl -wln -e 'print if /[^Jj]ohn)/' lines9.txt
```

There is no output for the preceding RE because there are no matching lines.

USING \S AND \s WITH WHITESPACES

The expression \s matches a single whitespace. The following expression matches lines that start with one or more whitespaces, followed by the string `cat`:

```
perl -wln -e 'print if /\s+cat/' lines3.txt
```

The output is here:

```
catfish
```

The following expression matches lines that start with one or more whitespaces, any number of characters, then followed by the string `cat`:

```
perl -wln -e 'print if /\s+.*cat'/ lines3.txt
```

The output is here:

```
 catfish
 small catfish
```

Use `\S` when you want to match non-whitespace characters. For example, the following expression matches lines that do not start with a whitespace:

```
perl -wln -e 'print if /^\S+'/ lines3.txt
```

The output is here:

```
grey.
.gray
dog
doggy
cat
catty
```

USING \W AND \w WITH WORDS

The expression `\w` matches a single word. The following expression matches lines that start with a word:

```
perl -wln -e 'print if /^\w'/ lines3.txt
```

The output is here:

```
grey.
dog
doggy
cat
catty
```

The expression `\W` matches a non-word. The following expression matches lines that do not start with a word:

```
perl -wln -e 'print if /^\W'/ lines3.txt
```

The output is here:

```
.gray
 catfish
 small catfish
```

The following expression matches lines that do not start with a word, followed by the string `cat`:

```
perl -wln -e 'print if /^\Wcat'/ lines3.txt
```

The output is here:

```
 catfish
```

SEARCH AND REPLACE REs IN PERL

In Chapter 1 the "use case" section contains a search-and-replace example that uses a Perl-based RE. For your convenience, Listing A.17 displays the contents of `alphanums.txt`, which consists of two comma-separated fields in each row.

LISTING A.17: alphanums.txt

```
"AAA_1234_4",1XY
"BBB_5678_3",2YX
"CCC_9012_2",3YZ
"DDD_3456_1",4WX
```

A Perl-based RE for replacing everything except letters and digits with blank spaces is shown here:

```
perl -pln -e 's/[^a-zA-Z0-9]/ /g' alphanums.txt
```

The result of the preceding code snippet is here:

```
 AAA 1234 4  1XY
 BBB 5678 3  2YX
 CCC 9012 2  3YZ
 DDD 3456 1  4WX
```

If you read Chapter 5, you recognize that the RE (shown in bold) in the preceding code snippet bears an uncanny resemblance to a sed-based RE.

Now let's look at a Perl-based RE for replacing non-digits with blank spaces, as shown here:

```
perl -pln -e 's/[^0-9]/ /g' alphanums.txt
```

The result of the preceding code snippet is here:

```
    1234 4  1
    5678 3  2
    9012 2  3
    3456 1  4
```

As you can see, each of the four output lines starts with five blank spaces because the preceding Perl snippet replaces a non-digit with a blank. Since the lines in alphanums.txt start with a quote (“), followed by three capital letters, then another quote (“), those five characters are replaced by blanks. Similar comments apply to the other whitespaces that appear in the output.

TESTING REs: ARE THEY ALWAYS CORRECT?

The concepts in this section are applicable to other programming languages, and not just with Perl. The key point to remember: if slightly different REs generate the same output, is this due to the contents of the dataset? Phrased in a slightly different way: how do you know that your dataset contains a sufficient variety of text strings to ensure that differences in the output of slightly different REs is not due to your specific dataset?

This is an important question. Consider a production application that has worked correctly for months with user-based input. Suddenly the application fails, and after much effort you discover that an RE in the application does not handle a rarely encountered character sequence.

As an example, the following RE matches all the lines in `lines1.txt`:

```
perl -ne 'print if /\bg\w+/' lines1.txt
```

The preceding snippet produces this output:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

Now consider the following REs that match lines with words ending with ‘g’:

```
perl -ne 'print if /g\w+\b/'      lines1.txt
perl -ne 'print if /g\w+\b/ '     lines1.txt
perl -ne 'print if /g\w+\b/ '     lines1.txt
perl -ne 'print if / g\w+\b/'     lines1.txt
perl -ne 'print if /[^g ]g\w+\b/' lines1.txt
perl -ne 'print if /[ ]g\w+\b/'   lines1.txt
perl -ne 'print if /[ ]g\S+\b/'   lines1.txt
```

The preceding snippets produce the same output:

```
the dog is grey and the cat is gray.
this dog is grey
that cat is gray
```

However, the following snippet produces a different output:

```
perl -ne 'print if /[^ ]g\w*\b/' lines1.txt
```

The result of the preceding code snippet is here:

```
the dog is grey and the cat is gray.
this dog is grey
```

Next, the following REs match words that start with ' g ':

```
perl -ne 'print "$&\n" if /\bg\w+/'    lines1.txt
perl -ne 'print "$&\n" if /g\w+\b/ '   lines1.txt
perl -ne 'print "$&\n" if /g\w+\b/ '   lines1.txt
perl -ne 'print "$&\n" if / g\w+\b/'   lines1.txt
perl -ne 'print "$&\n" if /[ ]g\w+\b/' lines1.txt
perl -ne 'print "$&\n" if / g\w+\b/'   lines1.txt
```

The preceding snippets all produce the same output:

```
grey
grey
gray
```

However, the following snippets produce a different output:

```
perl -ne 'print "$&\n" if /[^g ]g\w+\b/' lines1.txt
```

[empty output]

```
perl -ne 'print "$&\n" if /[^ ]g\w*\b/' lines1.txt
og
og
```

The code snippets are grouped in blocks, and in each block the code snippets look very similar, but they have subtle differences that require a solid understanding of metacharacters in REs.

While it's virtually impossible to check all possible combinations of characters in a text string, you need to be vigilant and test your REs on a large variety of patterns to minimize the likelihood of matching (or not matching) an "outlier" RE.

SUMMARY

This Appendix started with an introduction to some basic REs in Perl, followed by examples that illustrate how to match (or how to not match) characters or words. Next you learned about the metacharacter "^" and how its interpretation depends on its location in an RE, followed by the "$" metacharacter for matching strings at the end of a line.

You then saw how to use the metacharacters ".", "*", and "\"to create REs that are combinations of metacharacters, along with "escaping" the meaning of metacharacters.

Moreover, you learned how to create REs for common strings, such as dates, U.S. phone numbers, zip codes (U.S. and Canadian), and some email addresses. Then you saw how to detect IP addresses and comments in source code, as well as create REs for matching ISBNs.

Although the REs in this Appendix are not exhaustive, they do provide you with enough information to help you define REs that are more comprehensive. In addition, you are now in a good position to convert the other REs in Chapter 2 (that are not covered in this Appendix) into Perl-based REs.

APPENDIX B

REs IN JAVA

This short Appendix introduces you to REs in Java, with code samples that illustrate how to work with REs in Java programs. Keep in mind that the Java code samples in this Appendix contain compiled code, which differs from the Perl Appendix and all the book chapters. However, even if you are new to Java, the REs in the code samples have already been discussed in the book chapters, so you can easily follow the Java code. In any case, this Appendix is optional.

The first section of this Appendix contains an eclectic mix of REs (all of which appear in Chapter 1 or Chapter 2) in complete Java code samples. Although there are fewer Java code samples in this Appendix (compared to the number of REs in Chapter 1), they contain a greater assortment of REs.

The second part of this Appendix contains some code snippets with Scala-based REs. This section is extremely short, and if you like the Java section, then this section will be a very simple transition. If you want more practice, feel free to convert the code samples in Chapter 1 into their Scala-based counterparts.

Please keep in mind the following points when you read this Appendix. First, the discussions of metacharacters and character classes in Chapter 1 are not repeated in this Appendix.

Second, if you want to launch the Java code samples from the command line, it means that you might need to perform an Internet search in order to download and install the necessary software for your platform. Alternatively, you can simply read the output that accompanies the code samples in this chapter (so you won't need to launch the code yourself).

Third, this Appendix does not provide any tutorial-style material that explains how to create, compile, and launch Java or Scala programs. Fortunately, the code samples are rudimentary, and they do not require any knowledge of OOP (Object Oriented Programming) to understand them. However,

familiarity with functional programming is definitely helpful if you decide to learn more about Scala.

Finally, this Appendix skips the details regarding the underlying classes and interfaces that provide support for REs in Java. If you're interested, you can perform an online search to learn about those details.

WORKING WITH STRINGS AND JAVA REs

Listing B.1 displays the contents of `SimpleStrings.java`, which illustrates how to work with strings and very simple REs in Java.

LISTING B.1: SimpleStrings.java

```
public class SimpleStrings
{
   public static void main(String[] args)
   {
      String line1 = "hello";
      System.out.println("===> LINE: "+line1);

      // true
      System.out.println("Pattern: hello");
      System.out.println("Match:   "+line1.matches("hello"));

      // true
      System.out.println("Pattern: [hH]ello");
      System.out.println("Match:   "+line1.matches("[hH]ello"));

      // false
      System.out.println("Pattern: he");
      System.out.println("Match:   "+line1.matches("he"));

      // false
      System.out.println("Pattern: goodbye");
      System.out.println("Match:   "+line1.matches("goodbye"));
   }
}
```

Listing B.1 contains four pairs of code snippets that compare REs with the string `hello`, and each code snippet is preceded by a comment line that indicates whether the result is true or false. All four pattern matches rely on the `matches()` method of the Java `String` class. The first result is obviously `true`, and the second result—which is also `true`—shows you how to define a simple RE with a character class.

The third result might surprise you: only a full match yields a `true` result, and since the string `he` is a proper substring of `hello`, the result is `false`. Finally, the fourth result is clearly `false`.

The output from launching the code in Listing B.1 is here:

```
===> LINE: hello
Pattern: hello
```

```
Match:   true
Pattern: [hH]ello
Match:   true
Pattern: he
Match:   false
Pattern: goodbye
Match:   false
```

WORKING WITH NUMBERS AND JAVA REs

Listing B.2 displays the contents of `SimpleNumbers.java`, which illustrates how to work with strings and familiar REs in Java.

LISTING B.2: SimpleNumbers.java

```
public class SimpleNumbers
{
   public static void main(String[] args)
   {
      String line1 = "123 456";
      System.out.println("===> LINE: "+line1);

      // true
      System.out.println("Pattern: ^[\\d].*");
      System.out.println("Match:   "+line1.matches("^[\\d].*"));

      // true
      System.out.println("Pattern: ^[\\d]+\\s+[\\d]+.*");
      System.out.println("Match:   "+
                         line1.matches("^[\\d]+\\s+[\\d]+.*"));
   }
}
```

Listing B.2 contains a string that consists of two integers that are separated by a space. The first RE matches a string that starts with a number, followed by zero or more arbitrary characters. The second RE matches a string that starts with a number, followed by one or more spaces, which is then followed by another number, and then zero or more arbitrary characters. As you can see, the string `line1` matches both REs.

The output from launching the code in Listing B.1 is here:

```
===> LINE: 123 456
Pattern: ^[\d].*
Match:   true
Pattern: ^[\d]+\s+[\d]+.*
Match:   true
```

WORKING WITH RANGES AND JAVA REs

Listing B.3 displays the contents of `SimpleRanges.java`, which illustrates how to work with the strings and REs that you saw in Chapter 1.

LISTING B.3: SimpleRanges.java

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class SimpleRanges
{
   public static void main(String[] args)
   {
    //String line1 = "hello world";
      String line1 = "hello";
      System.out.println("===> LINE: "+line1);

      // true
      System.out.println("Pattern: [hH]ello");
      System.out.println("Match:   "+line1.matches("[hH]ello"));

      // true (requires exact match)
      System.out.println("Pattern: hello|world");
      System.out.println("Match:   "+line1.matches("hello|world"));
   }
}
```

Listing B.3 contains two REs, where the first RE specifies a character class. The second RE contains a pipe "|" symbol, which you know is used for an either-or match. Based on your knowledge of REs, both pattern matches are clearly `true`.

The output from launching the code in Listing B.3 is here:

```
===> LINE: hello
Pattern: [hH]ello
Match:   true
Pattern: hello|world
Match:   true
```

WORKING WITH STRINGS AND NUMBERS AND JAVA REs

Listing B.4 displays the contents of `StringsNumbers.java`, which illustrates how to work with strings and REs that contain metacharacters.

LISTING B.4: StringsNumbers.java

```
public class StringsNumbers
{
   public static void main(String[] args)
   {
      String line1 = "123hello456";
      System.out.println("===> LINE: "+line1);

      // true
      System.out.println("Pattern: ^[\\d].*");
      System.out.println("Match:   "+line1.matches("^[\\d].*"));
```

```
      // true
      System.out.println("Pattern: .*\\d{3}$");
      System.out.println("Match:   "+line1.matches(".*\\d{3}$"));

      // true
      System.out.println("Pattern: ^\\d{3}[a-z]+\\d{3}$");
      System.out.println("Match:   "+
                         line1.matches("^\\d{3}[a-z]+\\d{3}$"));
   }
}
```

Listing B.4 contains a string consisting of three digits, followed by five characters, and then another three digits. The first RE matches strings that start with an integer. The second RE matches strings that end with three digits. The third RE matches strings that start with three digits, followed by one or more lowercase letters, and then end with three digits. Hence, the pattern match for all three REs is `true`.

The output from launching the code in Listing B.4 is here:

```
===> LINE: 123hello456
Pattern: ^[\d].*
Match:   true
Pattern: .*\d{3}$
Match:   true
Pattern: ^\d{3}[a-z]+\d{3}$
Match:   true
```

MIXED-CASE STRINGS AND REs

Listing B.5 displays the contents of `MixedCase.java`, which illustrates how to work with mixed-case strings and REs in Java.

LISTING B.5: MixedCase.java

```
public class MixedCase
{
   public static void main(String[] args)
   {
      String line1 = "Hello";
      System.out.println("===> LINE: "+line1);

      // true
      System.out.println("Pattern: [A-Za-z]+");
      System.out.println("Match:   "+line1.matches("[A-Za-z]+"));

      // true
      System.out.println("Pattern: ^[A-Za-z]+$");
      System.out.println("Match:   "+line1.matches("^[A-Za-z]+$"));

      // true
      System.out.println("Pattern: ^[A-Z][a-z]+");
      System.out.println("Match:   "+line1.matches("^[A-Z][a-z]+"));
```

```
      // true
      System.out.println("Pattern: ^[A-Z][a-z]{4,6}");
      System.out.println("Match:   "+
                         line1.matches("^[A-Z][a-z]{4,6}"));
   }
}
```

Listing B.5 contains a string consisting of lowercase letters. The first RE matches strings that contain one or more lowercase or uppercase letters, in any order. The second RE matches strings that *start* with one or more lowercase or uppercase letters (also in any order). The third RE matches strings that start with an uppercase letter, followed by a range of lowercase or uppercase letters (also in any order), where the range is between four and six inclusive. Hence, the pattern match for all three REs is `true`.

The output from launching the code in Listing B.5 is here:

```
===> LINE: Hello
Pattern: [A-Za-z]+
Match:   true
Pattern: ^[A-Za-z]+$
Match:   true
Pattern: ^[A-Z][a-z]+
Match:   true
Pattern: ^[A-Z][a-z]{4,6}
Match:   true
```

MIXING (AND ESCAPING) METACHARACTERS

Listing B.6 displays the contents of `EscapeMeta.java`, which illustrates how to "escape" and also match metacharacters in strings.

LISTING B.6: EscapeMeta.java

```
public class EscapeMeta
{
   public static void main(String[] args)
   {
      String line1 = ".hello$";
      System.out.println("===> LINE: "+line1);

      // true
      System.out.println("Pattern: .hello$");
      System.out.println("Match:   "+line1.matches(".hello$"));

      // true
      System.out.println("Pattern: \\.hello\\$");
      System.out.println("Match:   "+line1.matches("\\.hello\\$"));

      // true
      System.out.println("Pattern: \\.[A-Za-z]+\\$");
      System.out.println("Match:   "+line1.matches("\\.
[A-Za-z]+\\$"));
   }
}
```

Listing B.6 contains a string that starts with a period ".", followed by lowercase letters, and ending with a dollar sign "$". The first RE fails to match the string because the initial "." and final "$" are treated as metacharacters. By contrast, the second RE successfully matches because the initial "." and final "$" are both "escaped" via a pair of consecutive backslash "\" characters. The third RE is a modified version of the second RE: the hard-coded string hello is replaced with the RE `[A-Za-z]+`, which matches the initial string `hello`.

The output from launching the code in Listing B.6 is here:

```
===> LINE: .hello$
Pattern: .hello$
Match:   false
Pattern: \.hello\$
Match:   true
Pattern: \.[A-Za-z]+\$
Match:   true
```

WORKING WITH DATE-RELATED REs IN JAVA

The title of this section explicitly mentions date-related REs because Java provides extensive support for dates and calendars via date-related and calendar-related classes that match many different date formats. Hence, you do not need to use REs in Java if you need to work with dates. In fact, those classes provide many other date-related features that are unavailable in REs, and you ought to explore those Java classes if you need more sophisticated data-related functionality.

Keep in mind that this section does not use any date-related Java classes: we'll use simple REs to match patterns of strings that have two different date formats.

Listing B.7 displays the contents of `DateStrings.java`, which illustrates how to match some valid dates.

LISTING B.7: DateStrings.java

```
public class DateStrings
{
   public static void main(String[] args)
   {
      String line1 = "05/12/18";
      String line2 = "05.12.18";
      System.out.println("===> LINE: "+line1);

      // true
      System.out.println("Pattern: \\d{2}.\\d{2}.\\d{2}");
      System.out.println("Match:   "+
                         line1.matches("\\d{2}.\\d{2}.\\d{2}"));

      System.out.println("===> LINE: "+line2);

      // true
      System.out.println("Pattern: \\d{2}.\\d{2}.\\d{2}");
```

```
      System.out.println("Match:   "+
                         line2.matches("\\d{2}.\\d{2}.\\d{2}"));
   }
}
```

Listing B.7 contains two date-related strings: the first has the MM/DD/YY format, and the second has the MM.DD.YY format. However, we can use the same RE to match both date formats: \\d{2}.\\d{2}.\\d{2}. The preceding RE contains the "." metacharacter that matches the "." in the first date string and also the "/" in the second date string.

The output from launching the code in Listing B.7 is here:

```
===> LINE: 05/12/18
Pattern: \d{2}.\d{2}.\d{2}
Match:   true
===> LINE: 05.12.18
Pattern: \d{2}.\d{2}.\d{2}
Match:   true
```

WORKING WITH U.S. ZIP CODES

Listing B.8 displays the contents of USZipCodes.java, which illustrates how to match some U.S. zip codes.

LISTING B.8: USZipCodes.java

```
public class DateStrings
{
   public static void main(String[] args)
   {
      String line1 = "94043";
      String line2 = "94043-04123";
      System.out.println("===> LINE: "+line1);

      // true
      System.out.println("Pattern: \\d{5}");
      System.out.println("Match:   "+line1.matches("\\d{5}"));

      System.out.println("===> LINE: "+line2);

      // true
      System.out.println("Pattern: \\d{5}(-\\d{5})");
      System.out.println("Match:   "+line2.matches("\\d{5}(-\\
d{5})"));
   }
}
```

Listing B.8 contains two REs for U.S. zip codes. The first RE is \\d{5}, which matches many (most?) U.S. zip codes. The second RE is \\d{5}(-\\d{5})), which matches U.S. zip codes which are qualified by an extra five-digit sequence.

The output from launching the code in Listing B.8 is here:

```
===> LINE: 94043
Pattern: \d{5}
Match:   true
===> LINE: 94043-04123
Pattern: \d{5}(-\d{5})
Match:   true
```

This concludes the Java-related portion of this Appendix. The next section contains a few examples of working with REs in Scala.

WORKING WITH REs IN SCALA

Scala provides the `Regex` class for handling REs, which delegates to the `java.util.regex` package of the Java Platform. An instance of `Regex` represents a compiled RE pattern. For performance reasons, it's better to construct frequently used REs only (preferably outside of loops).

More information is available here:

http://www.scala-lang.org/api/current/scala/util/matching/Regex.html

As a simple example, the following RE in Scala matches an integer:

```
val num = raw"(\d+)".r
```

Even if you are unfamiliar with Scala, you can see that the preceding code snippet initializes the variable `num` as an RE that consists of one or more digits via the \d+ expression.

The following code snippet illustrates how to create an RE that matches dates that consist of four digits, a hyphen, a pair of digits, a hyphen, and another pair of digits:

```
val date = raw"(\d{4})-(\d{2})-(\d{2})".r
```

Since escape characters are not processed in multi-line string literals, three consecutive quotes (before and after) avoids the need to escape the backslash character. Hence, `\\d` can also be written as `"""\d"""`.

Extraction

To extract the capturing groups when an RE is matched, use it as an extractor in a pattern match:

REs in Scala have access to various methods, such as `start()` and `hasNext()`, as shown here:

```
val r = "(ab+c)".r
val s = "xxxabcyyyabbczzz"
r.findAllIn(s).start    // 3
val mi = r.findAllIn(s)
mi.hasNext  // true
mi.start    // 3
mi.next()   // "abc"
```

```
mi.start    // 3
mi.hasNext  // true
mi.start    // 9
mi.next()   // "abbc"
```

The method `findAllIn()` finds non-overlapping matches, as shown here:

```
val num = raw"(\d+)".r
val all = num.findAllIn("123").toList  // creates List("123")
```

SUMMARY

This Appendix started with an introduction to some basic REs in Java, followed by examples that illustrate how to match (or how to not match) characters or words. You saw examples of using metacharacters and character classes to match sequences of numbers and characters (uppercase and lowercase). Moreover, you learned how to create REs for common strings, such as dates, U.S. phone numbers, and U.S. zip codes.

Then you learned how to create REs in Scala, which provides a "wrapper" around a Java class for matching REs.

INDEX