Bash Command Line and Shell Scripts

Pocket Primer

Oswald Campesato

Bash Command Line and Shell Scripts

Pocket Primer

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and disc (the "Work"), you agree that this license grants permission to use the contents contained herein, including the disc, but does not give you the right of ownership to any of the textual content in the book / disc 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 disc, 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 author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and/or disc, 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.

(Companion files are also available for downloading from the publisher at info@merclearning.com.)

Bash Command Line and Shell Scripts

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. *Bash Command Line and Shell Scripts Pocket Primer.*
ISBN: 978-1-68392-504-0

Library of Congress Control Number: 2020935567

202122321 Printed on acid-free paper in the United States of America.

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc. For additional information, please contact the Customer Service Dept. at (800) 232-0223(toll free).

Digital versions of our titles are available at: www.academiccourseware.com and other electronic vendors. *Companion files are available from the publisher by writing to info@merclearning.com.*

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 an assortment of powerful command line utilities that can be combined to create simple, yet powerful shell scripts. While all examples and scripts use the "bash" command set, many of the concepts translate into other command shells (such as sh, ksh, zsh, and csh), including the concept of piping data between commands, regular expression substitution, and the sed and awk commands. Aimed at a reader relatively new to working in a bash environment, the book is comprehensive enough to be a good reference and teach a few new tricks to those who already have some experience with creating shells scripts.

This short book contains a variety of code fragments and shell scripts for data scientists, data analysts, and other people who want shell-based solutions to "clean" various types of text files. In addition, the concepts and code samples in this book are useful for people who want to simplify routine tasks.

This book takes introductory concepts and commands in `bash`, and then demonstrates their use in simple yet powerful shell scripts. This book does not cover "pure" system administration functionality for Unix or Linux.

Is This Book is For Me and What Will I Learn?

This book is intended for general users, data scientists, data analysts, and other people who perform a variety of tasks from the command line, and who also have a limited knowledge of shell programming.

You will acquire an understanding of how to use various `bash` commands, often as part of short shell scripts. The chapters also contain simple use cases that illustrate how to perform various tasks involving text files, such as switching the order of a two-column text file, removing control characters in a text file, find specific lines and merge them, reformatting a date field in a text file, and removing nested quotes.

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). The code samples are derived primarily from scripts prepared by 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 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 page length 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 desire to learn shell programming, 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 bash Commands are Excluded?

The commands that do not meet any of the criteria listed in the previous section are not included in this Primer. Consequently, there is no coverage of commands for system administration (e.g., shutting down a machine, scheduling backups, and so forth). The purpose of the material

in the chapters is to illustrate how to use bash commands for handling common data cleaning tasks with text files, 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 the installation processes.

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 that a particular technique is possible can make 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 on these commands, there is a wide variety of literature published and online resources describing the `bash` shell, Unix programming, and the `grep`, `sed`, and `awk` commands.

O. Campesato

April 2020

CHAPTER 1

INTRODUCTION

This chapter contains a fast-paced introduction to basic commands in the `bash` shell, such as navigating around the file system, listing files, and displaying the contents of files. As you will soon see, this chapter is dense and contains a very eclectic mix of topics in order to prepare you for later chapters. If you already have some knowledge of `bash` commands, you can probably skim quickly through this introductory chapter and then proceed to Chapter 2. Incidentally, sometimes you will "bash shell" instead of just `bash` (as in the first sentence of this paragraph), and although the former is actually redundant, there won't be any confusion about its intended meaning.

The first part of this chapter starts with a brief introduction to some Unix shells, followed by a discussion about files, file permissions, and directories. You will also learn how to create files and directories and how to change their access permissions.

The second part of this chapter introduces simple shell scripts, along with commands for making them executable. Since shell scripts involve various `bash` commands (and can optionally contain user-defined functions), it's a good idea to learn about `bash` commands before you create `bash` scripts.

The third portion of this chapter discusses two useful `bash` commands: the `cut` command (for cutting or extracting columns and/or fields from a dataset) and the `paste` command (for "pasting" text or datasets together vertically).

In addition, the final part of this chapter contains a use case involving the `cut` command and `paste` command that illustrates how to switch the order of two columns in a dataset. You can also perform this task using the `awk` command (discussed in Chapter 7 and Chapter 9).

There are a few points to keep in mind before delving into the details of shell scripts. First, shell scripts can be executed from the command line after

adding "execute" permissions to the text file containing the shell script. Second, you can use the `crontab` utility to schedule the execution of your shell scripts according to a schedule of your choice. Specifically, the `crontab` utility allows you to specify the execution of a shell script on an hourly, daily, weekly, or monthly basis. Tasks that are commonly scheduled via `crontab` include performing backups, removing unwanted files, and so forth. If you are completely new to Unix-based systems, just keep in mind that there is a way to run scripts both from the command line and in a "scheduled" manner. Setting file permissions to run the script from the command line will be discussed later.

Third, the contents of any shell script can be as simple as a single command or can comprise hundreds of lines of `bash` commands. In general, the more useful (and often more interesting) shell scripts involve a combination of several `bash` commands. A learning tip: since there are usually several ways to produce the desired result, it's helpful to read other people's shell scripts to learn how to combine commands in useful ways.

WHAT IS UNIX?

Unix is an operating system created by Ken Thompson in the early 1970s, which eventually led to a number of variations, such as HP/UX for HP machines and AIX for IBM machines. Linux Torvalds developed the Linux operating system during the 1990s, and many Linux commands are the same as their bash counterparts (but differences exist, often in the commands for system administrators). The Mac OS X operating system is based on AT&T Unix.

Unix has a rich and storied history, and if you are really interested in learning about its past, you can read online articles and also Wikipedia. This book foregoes those details and focuses on helping you quickly learn how to become productive with various commands.

Available Shell Types

The original Unix shell is the Bourne shell, which was written in the mid-1970s by Stephen R. Bourne. In addition, the Bourne shell was the first shell to appear on `bash` systems, and you will sometimes hear "the shell" as a reference to the Bourne shell. The Bourne shell is a `POSIX` standard shell, usually installed as `/bin/sh` on most versions of Unix, whose default prompt is the `$` character. Consequently, Bourne shell scripts will execute on almost every version of Unix. In essence, the AT&T branches of Unix support the Bourne shell (`sh`), `bash`, Korn shell (`ksh`), `tsh`, and `zsh`.

However, there is also the `BSD` branch of Unix that uses the "C" shell (`csh`), whose default prompt is the `%` character. In general, shell scripts written for `csh` will not execute on AT&T branches of Unix, unless the `csh` shell is also installed on those machines (and vice versa).

The Bourne shell is the most 'unadorned' in the sense that it lacks some commands that are available in the other shells, such as `history`,

noclobber, and so forth. Some well-known variants for Bourne Shell are listed as follows:

Korn shell (ksh)
Bourne Again shell (bash)
POSIX shell (sh)
zsh ("Zee shell")

The different C-type shells are as shown below:
C shell (csh)
TENEX/TOPS C shell (tcsh)

The commands and the shell scripts in this book are based on the bash shell, and many of the commands also work in other Bourne-related shells (and the remaining shells have a similar command to accomplish the same goal). When you are unable to perform a particular shell-related task, perform an Internet search for "how to use <bash command> in <shell name>" and you will often find an answer. Keep in mind that sometimes there are variations in syntax for a given command in a particular shell, and typing "man <command>" in a command shell can provide useful information.

WHAT IS BASH?

Bash is an acronym for "Bourne Again Shell", which has its roots in the Bourne shell created by Stephen R. Bourne. Shell scripts based on the Bourne shell will execute in bash, but the converse is not necessarily true. The bash shell provides additional features that are unavailable in the Bourne shell, such as support for arrays (discussed later in this chapter).

On Mac OS X, the /bin directory contains the following executable shells:

```
-r-xr-xr-x  1 root  wheel  1377872 Apr 28  2017 /bin/ksh

-r-xr-xr-x  1 root  wheel  630464 Apr 28  2017 /bin/sh

-rwxr-xr-x  1 root  wheel  375632 Apr 28  2017 /bin/csh

-rwxr-xr-x  1 root  wheel  592656 Apr 28  2017 /bin/zsh

-r-xr-xr-x  1 root  wheel  626272 Apr 28  2017 /bin/bash
```

In case you're interested, a nice comparison matrix of the support for various features among the preceding shells is here:

https://stackoverflow.com/questions/5725296/difference-between-sh-and-bash

Something else that might surprise you: in some environments the Bourne shell sh *is* the Bash shell, which you can check by typing the following command:

```
sh --version

GNU bash, version 3.2.57(1)-release (x86_64-apple-darwin16)

Copyright (C) 2007 Free Software Foundation, Inc.
```

If you are new to the command line (be it Mac, Linux, or PCs), please read the Preface that provides some useful guidelines for accessing command shells.

Getting help for `bash` *Commands*

If you want to see the options for a specific `bash` command, invoke the `man` command to see a description of that `bash` command and its options:

```
man cat
```

Keep in mind that the `man` command produces terse explanations, and if those explanations are not clear enough, you can search for online code samples that provide more details.

Navigating Around Directories

In a command shell, you will often perform some common operations, such as displaying (or changing) the current directory, listing the contents of a directory, displaying the contents of a file, and so forth. The following set of commands show you how to perform these operations, and you can execute a subset of these commands in the sequence that is relevant to you. Options for some of the commands in this section (such as the `ls` command) are described in greater detail later in this chapter.

A frequently used Bash command is `pwd` ("print working directory") that displays the current directory, as shown here:

```
pwd
```

The output of the preceding command might look something like this:

```
/Users/jsmith
```

Use the `cd` ("change directory") command to go to a specific directory. For example, type the command `cd /Users/jsmith/Mail` to navigate to this directory (or some other existing directory). If you are currently in the `/Users/jsmith` directory, just type `cd Mail`.

You can navigate to your home directory with either of these commands:

```
$ cd $HOME
```

```
$ cd
```

One convenient way to return to the previous directory is the command `cd -`. Keep in mind that the `cd` command on Windows merely displays the current directory and does not change the current directory (unlike the `cd` command in `bash`).

The `history` *Command*

The `history` command displays a list (i.e., the history) of commands that you executed in the current command shell, as shown here:

```
history
```

A sample output of the preceding command is given below:

```
1202  cat sample.txt > longfile2.txt
1203  vi longfile2.txt
```

```
1204  cat longfile2.txt |fold -40
1205  cat longfile2.txt |fold -30
1206  cat longfile2.txt |fold -50
1207  cat longfile2.txt |fold -45
1208  vi longfile2.txt
1209  history
1210  cd /Library/Developer/CommandLineTools/usr/include/
      c++/
1211  cd /tmp
1212  cd $HOME/Desktop
1213  history
```

If you want to navigate to the directory that is shown in line `1210`, you can do so simply by typing the following command:

```
!1210
```

The command `!cd` will search backwards through the history of commands to find the first command that matches the `cd` command, in this case, line `1212` is the first match. If there aren't any intervening `cd` commands between the current command and the command in line `1210`, then `!1210` and `!cd` will have the same effect.

NOTE *Be careful with the "!" option with bash commands because the command that matches the "!" might not be the one you intended, so it's safer to use the `history` command and then explicitly specify the correct number (in that history) when you invoke the "!" operator.*

LISTING FILENAMES WITH THE `ls` COMMAND

The `ls` command is for listing filenames, and there are many switches available that you can use, as shown in this section. For example, the `ls` command displays the following filenames (the actual display depends on the font size and the width of the command shell) on my Mac:

```
apple-care.txt      iphonemeetup.txt   outfile.txt
ssl-instructions.txt  checkin-commands.txt   kyrgyzstan.txt
output.txt
```

The command `ls  -1` (the digit "1") displays a vertical listing of filenames:

```
apple-care.txt
checkin-commands.txt
iphonemeetup.txt
kyrgyzstan.txt
outfile.txt
output.txt
ssl-instructions.txt
```

The command `ls -1` (the letter "l") displays a long listing of filenames:

```
total 56
-rwx------  1 ocampesato  staff   25 Apr 06 19:21
apple-care.txt
-rwx------  1 ocampesato  staff  146 Apr 06 19:21 checkin-
commands.txt
-rwx------  1 ocampesato  staff  478 Apr 06 19:21
iphonemeetup.txt
-rwx------  1 ocampesato  staff   12 Apr 06 19:21 kyrgyzstan.
txt
-rw-r--r--  1 ocampesato  staff   11 Apr 06 19:21 outfile.txt
-rw-r--r--  1 ocampesato  staff   12 Apr 06 19:21 output.txt
-rwx------  1 ocampesato  staff  176 Apr 06 19:21
ssl-instructions.txt
```

The command `ls -1t` (the letters "l" and "t") display a time-based long listing:

```
total 56
-rwx------  1 ocampesato  staff   25 Apr 06 19:21
apple-care.txt
-rwx------  1 ocampesato  staff  146 Apr 06 19:21 checkin-
commands.txt
-rwx------  1 ocampesato  staff  478 Apr 06 19:21
iphonemeetup.txt
-rwx------  1 ocampesato  staff   12 Apr 06 19:21 kyrgyzstan.
txt
-rw-r--r--  1 ocampesato  staff   11 Apr 06 19:21 outfile.txt
-rw-r--r--  1 ocampesato  staff   12 Apr 06 19:21 output.txt
-rwx------  1 ocampesato  staff  176 Apr 06 19:21
ssl-instructions.txt
```

The command `ls -ltr` (the letters "l", "t", and "r") display a reversed time-based long listing of filenames:

```
total 56
-rwx------  1 ocampesato  staff  176 Apr 06 19:21
ssl-instructions.txt
-rw-r--r--  1 ocampesato  staff   12 Apr 06 19:21 output.txt
-rw-r--r--  1 ocampesato  staff   11 Apr 06 19:21 outfile.txt
```

```
-rwx------  1 ocampesato  staff   12 Apr 06 19:21 kyrgyzstan.
txt

-rwx------  1 ocampesato  staff  478 Apr 06 19:21
iphonemeetup.txt

-rwx------  1 ocampesato  staff  146 Apr 06 19:21 checkin-
commands.txt

-rwx------  1 ocampesato  staff   25 Apr 06 19:21 apple-care.txt
```

Here is the description of all the listed columns in the preceding output:

Column #1: represents file type and permission given on the file (see below)
Column #2: the number of memory blocks taken by the file or directory
Column #3: the (Bash user) owner of the file
Column #4: represents a group of the owner
Column #5: represents the file size in bytes.
Column #6: the date and time when this file was created or last modified
Column #7: represents a file or directory name

In the `ls -l` listing example, every file line began with a d, -, or l. These characters indicate the type of file that is listed. These (and other) initial values are described below:

- Regular file (ASCII text file, binary executable, or hard link)
b Block special file (such as a physical hard drive)
c Character special file (such as a physical hard drive)
d Directory file that contains a listing of other files and directories.
l Symbolic link file
p Named pipe (a mechanism for interprocess communications)
s Socket (for interprocess communication)

If you look back at the long listing that is displayed earlier in this section, you will see that the leftmost character is a dash ("-"), which means that it's a long listing of regular files.

You can invoke the `wc` (word count) command to display the number of lines, words and characters in any text file, an example of which is shown here:

```
wc iphonemeetup.txt

10      5     478 iphonemeetup.txt
```

The preceding output shows that the file `iphonemeetup.txt` contains 10 lines, 5 words and 478 characters, which means that the file size is actually quite small.

Another point to keep in mind: this book works with files and directories, and occasionally with symbolic links; the other file types are primarily useful for programmers. Consult online documentation for more details regarding the `ls` command.

DISPLAYING CONTENTS OF FILES

This section introduces you to several commands for displaying different lines of text in a text file. The commands that you will learn about are cat, head, tail, fold, and also the pipe ("|") command.

The `cat` Command

Invoke the `cat` command to display the entire contents of `sample.txt`:

```
cat sample.txt
```

The preceding command displays the following text:

```
the contents
of this
long file
are too long
to see in a
single screen
and each line
contains
one or
more words
and if you
use the cat
command the
(other lines are omitted)
```

The `cat` command displays the entire contents of a file, which might be inconvenient when you want to see a small portion of a file. Fortunately, the `head` and `tail` commands are available, along with several commands that display only a portion of a file, such as `less` and `more` that are discussed later.

You can also display the contents of multiple files via the `cat` command and a *metacharacter* (discussed in more detail later), such as `?` or `*`. For example, suppose that the file `temp1` has the following contents:

```
this is line1 of temp1
this is line2 of temp1
this is line3 of temp1
```

Let's also suppose that the file `temp2` has these contents:

```
this is line1 of temp2
this is line2 of temp2
```

Now type the following command that contains the `?` metacharacter:

```
cat temp?
```

The output from the preceding command is shown here:

```
this is line1 of temp1
this is line2 of temp1
this is line3 of temp1
```

```
this is line1 of temp2
this is line2 of temp2
```

If you type the command `cat temp*` then the output will be the contents of all the files whose name starts with `temp` in the current directory. If you have a file – let's call it `temp2` – that contains binary data, then you will probably see some strange-looking output on your screen!

The `head` and `tail` Commands

The `head` command displays the first ten lines of a text file (by default), an example of which is here:

```
head sample.txt
```

The preceding command displays the following text:

```
the contents
of this
long file
are too long
to see in a
single screen
and each line
contains
one or
more words
```

The `head` command also provides an option to specify a different number of lines to display, as shown here:

```
head -4 sample.txt
```

The preceding command displays the following text:

```
the contents
of this
long file
are too long
```

The `tail` command displays the last 10 lines (by default) of a text file:

```
tail sample.txt
```

The preceding command displays the following text:

```
is available
in every shell
including the
bash shell
csh
zsh
ksh
and Bourne shell
```

NOTE *The last two lines in the preceding output are blank lines (not a typographical error in this page).*

Similarly, the `tail` command allows you to specify a different number of lines to display: `tail -4 sample.txt` displays the last 4 lines of `sample.txt`.

Use the `more` command to display a screenful of data, as shown here:

```
more sample.txt
```

Press the <spacebar> to view the next screenful of data, and press the <return> key to see the next line of text in a file. Incidentally, some people prefer the `less` command, which generates essentially the same output as the `more` command. (A geeky joke: "What's less? It's more.")

The Pipe Symbol

A very useful feature of `bash` is its support for the pipe symbol ("|") that enables you to "pipe" or redirect the output of one command to become the input of another command. The pipe command is very handy when you want to perform a sequence of operations involving various `bash` commands.

For example, the following code snippet combines the `head` command with the `cat` command and the pipe ("|") symbol:

```
cat sample.txt| head -2
```

A technical point: the preceding command creates two `bash` processes (more about processes later) whereas the command `head -2 sample.txt` only creates a single `bash` process.

You can use the `head` and `tail` commands in more interesting ways. For example, the following command sequence displays lines 11 through 15 of `sample.txt`:

```
head -15 sample.txt |tail -5
```

The preceding command displays the following text:

```
and if you
use the cat
command the
file contents
scroll
```

Display the line numbers for the preceding output as follows:

```
cat –n sample.txt | head -15 | tail -5
```

The preceding command displays the following text:

```
 11 and if you
 12 use the cat
 13 command the
 14 file contents
 15 scroll
```

You won't see the "tab" character from the output, but it's visible if you redirect the previous command sequence to a file and then use the "-t" option with the `cat` command:

```
cat -n sample.txt | head -15 | tail -5 > 1
cat -t 1
    11^Iand if you
    12^Iuse the cat
    13^Icommand the
    14^Ifile contents
    15^Iscroll
```

The `fold` Command

The `fold` command enables you to "fold" the lines in a text file, which is useful for text files that contain long lines of text that you want to split into shorter lines. For example, here are the contents of `longfile2.txt`:

```
the contents of this long file are too long to see in a
single screen and each line contains one or more words and
if you use the cat command the file contents scroll off the
screen so you can use other commands such as the head or
tail or more commands in conjunction with the pipe command
that is very useful in Bash and is available in every shell
including the bash shell csh zsh ksh and Bourne shell
```

You can "fold" the contents of `longfile2.txt` into lines whose length is `45` (just as an example) with this command:

```
cat longfile2.txt |fold -45
```

The output of the preceding command is here:

```
the contents of this long file are too long t
o see in a single screen and each line contai
ns one or more words and if you use the cat c
ommand the file contents scroll off the scree
n so you can use other commands such as the h
ead or tail or more commands in conjunction w
ith the pipe command that is very useful in U
nix and is available in every shell including
the bash shell csh zsh ksh and Bourne shell
```

Notice that some words in the preceding output are split based on the line width, and not "newspaper style."

In Chapter 4, you will learn how to display the lines in a text file that match a string or a pattern, and in Chapter 5 you will learn how to replace a string with another string in a text file.

FILE OWNERSHIP: OWNER, GROUP, AND WORLD

Bash files can have partial or full `rwx` privileges, where `r` = read privilege, `w` = write privilege, `x` = execute and can be executed from the command line, simply by typing the file name (or the full path to the file if the file is not in your current directory). Invoking an executable file from the command line

will cause the operating system to attempt to execute commands inside the text file (which must be valid shell commands or executable files with valid shell commands).

Use the `chmod` command to change permissions for files. For example, if you need to set the owner, group, and other permissions equal to `rwx rw- r--` for a file, use the following command:

```
chmod u=rwx g=rw o=r filename
```

In the preceding command the options `u`, `g`, and `o` represent user permissions, group permissions, and others permissions, respectively.

Modify permissions on a file by specifying + to add permission to a user, group or others and specify – to remove permissions. For example, given a file with the permissions `rwx rw- r--`, add the executable permission to "others" as follows:

```
chmod o+x filename
```

Add the executable permission to all permission categories, that is, for the user, group, and others as follows:

```
chmod a+x filename
```

As you can surmise, the letter `a` in the preceding code snippet means "all groups". Conversely, specify a – in order to remove permissions from all groups, as shown here:

```
chmod a-x filename
```

HIDDEN FILES

A so-called "hidden" file is a filename that starts with a period character (.). Bash programs (including the shell) use most of these files to store configuration information. Some common examples of hidden files include the files:

.profile: the Bourne shell (sh) initialization script
.bash_profile: the bash shell (bash) initialization script
.kshrc: the Korn shell (ksh) initialization script
.cshrc: the C shell (csh) initialization script
.rhosts: the remote shell configuration file

You can display a list of hidden files in a directory via the `ls` command and the `-a` option, as shown here:

```
ls -a
.        .profile       docs     lib     test_results
..       .rhosts         hosts    pub     users
.emacs   bin             hw1      res.01  work
.exrc    ch07            hw2      res.02
.kshrc   ch07.bak         hw3      res.03
```

Keep in mind that a single dot (".") represents the current directory and a double dot ("..") represents the parent directory of the current directory.

HANDLING PROBLEMATIC FILENAMES

A "problematic" filename is a filename that contains one or more whitespaces, hidden (non-printing) characters, or starts with a dash ("-") character.

You can use double quotes to list filenames that contain whitespaces, or you can precede each whitespace by a backslash "\") character. For example, if you have a file named `One Space.txt`, you can use the `ls` command as follows:

```
ls -1 "One Space.txt"
```

```
ls –l One\ Space.txt
```

Filenames that start with a dash ("-") character are difficult to handle because the dash character is the prefix that specifies options for bash commands. Consequently, if you have a file whose name `is –abc`, then the command `ls –abc` will not work correctly, because the "-a" is interpreted as a switch for the `ls` command (and there is no "a" option).

In most cases, the best solution to this type of filename is to rename the file. This can be done in your operating system if your client isn't a `bash` shell, or you can use the following special syntax for the `mv` ("move") command to rename the file. The preceding two dashes tell `mv` to ignore the dash in the filename. An example is here:

```
mv -- -abc.txt renamed-abc.txt
```

WORKING WITH ENVIRONMENT VARIABLES

There are many built-in environment variables available, and the following subsections discuss the `env` command that displays the variables that have values in the environment, along with some common variables that are available in the environment of a command shell.

The `env` Command

The `env` ("environment") command displays the variables that are in your `bash` environment. An example of the output of the `env` command is here:

```
SHELL=/bin/bash
TERM=xterm-256color
TMPDIR=/var/folders/73/39lngcln4dj_scmgvsv53g_w0000gn/T/
OLDPWD=/tmp
TERM_SESSION_ID=63101060-9DF0-405E-84E1-EC56282F4803
USER=ocampesato
COMMAND_MODE=bash2003PATH=/opt/local/bin:/Users/ocampesato/
android-sdk-mac_86/platform-tools:/Users/ocampesato/
android-sdk-mac_86/tools:/usr/local/bin:
PWD=/Users/ocampesato
JAVA_HOME=/System/Library/Java/
JavaVirtualMachines/1.6.0.jdk/Contents/Home
LANG=en_US.UTF-8
```

```
NODE_PATH=/usr/local/lib/node_modules
HOME=/Users/ocampesato
LOGNAME=ocampesato
DISPLAY=/tmp/launch-xnTgkE/org.macosforge.xquartz:0
SECURITYSESSIONID=186a4
_=/usr/bin/env
```

The common environment variables that are pre-defined for you include HOME, LOGNAME, PWD, SHELL, TERM, and TMPDIR. Use the echo command to see the value of a single environment variable. For example, if you want to see the value of the SHELL environment variable, type the following command (notice the "$" character):

```
echo $SHELL
```

Based on the output of the env command that you saw earlier in this section, the output of the preceding command is here:

```
SHELL=/bin/bash
```

One other point: if you do not specify the $ character, you will not see the value of the environment variable. For example, if you type:

```
echo SHELL
```

Then you will see the following output:

```
SHELL
```

Later you will learn how to change the value of a variable, and if you are feeling impatient, you can see some interesting examples of setting an environment variable:

https://stackoverflow.com/questions/13998075/setting-environment-variable-for-one-program-call-in-bash-using-env

Useful Environment Variables

This section discusses some important environment variables, most of which you probably will not need to modify, but it's useful to be aware of the existence of these variables and their purpose.

The HOME variable contains the absolute path of the user's home directory

The HOSTNAME variable specifies the Internet name of the host

The LOGNAME variable specifies the user's login name

The PATH variable specifies the search path (see next subsection)

The SHELL variable specifies the absolute path of the current shell

The USER specifies the user's current username. This value might be different than the login name if a superuser executes the su command to emulate another user's permissions.

Setting the PATH Environment Variable

Programs and other executable files can reside in many directories, so operating systems provide a search path that lists the directories that the OS searches for executable files. Tip: if a directory containing an executable file

is not included in your PATH environment variable, simply add that directory to your PATH environment variable so that you can invoke an executable file by specifying just the filename: you don't need to specify the full path to the executable file.

The search path is stored in an environment variable, which is a named string maintained by the operating system. Every environment variable contains information available to the command shell and other programs.

One detail to keep in mind: the path variable is named PATH in bash or Path in Windows (bash is case-sensitive; Windows is not). Set the PATH in bash/Linux as shown here:

```
export PATH=$HOME/anaconda:$PATH
```

Here is the command that adds the Python directory to the PATH variable for the current command shell when you are using the bash shell:

```
export PATH="$PATH:/usr/local/bin/python"
```

Another way to do the same thing as the preceding code snippet (when you are in the Bourne shell or ksh shell) is with this command:

```
PATH="$PATH:/usr/local/bin/python"
```

NOTE */usr/local/bin/python is the full path of the Python directory*

Specifying Aliases and Environment Variables

You can define an environment variable and its value in a straightforward manner. For example, the following command initializes an environment variable called h1:

```
h1=$HOME/test
```

Now if you enter the following command:

```
echo $h1
```

If the value of $HOME is /Users/jsmith, then you will see the following output on OS X:

```
/Users/jsmith/test
```

The next code snippet shows you how to set the alias ll so that it displays a long listing of a directory:

```
alias ll="ls -l"
```

The following three alias definitions involve the ls command and various switches:

```
alias ll="ls -l"

alias lt="ls -lt"

alias ltr="ls -ltr"
```

As an example, you can replace the command ls -ltr (the letters "l," "t," and "r") that you saw earlier in the chapter with the ltr alias and

you will see the same reversed time-based long listing of filenames (reproduced here):

```
total 56
-rwx------  1 ocampesato  staff  176 Apr 06 19:21 ssl-
instructions.txt
-rw-r--r--  1 ocampesato  staff   12 Apr 06 19:21 output.txt
-rw-r--r--  1 ocampesato  staff   11 Apr 06 19:21 outfile.txt
-rwx------  1 ocampesato  staff   12 Apr 06 19:21
kyrgyzstan.txt
-rwx------  1 ocampesato  staff  478 Apr 06 19:21
iphonemeetup.txt
-rwx------  1 ocampesato  staff  146 Apr 06 19:21 checkin-
commands.txt
-rwx------  1 ocampesato  staff   25 Apr 06 19:21 apple-
care.txt
```

The `bash` shell supports the pipe ("|") symbol that sends the output of one command to the input of another command, which is executed in a left-to-right fashion. For example, the following alias "pipes" the output of `ls -ltr` to the more command:

```
alias ltrm="ls -ltr|more"
```

In a similar manner, you can define aliases for directory-related commands:

```
alias ltd="ls -lt | grep '^d'"

alias ltdm="ls -lt | grep '^d'|more"
```

FINDING EXECUTABLE FILES

There are several commands available for finding executable files (binary files or shell scripts) by searching the directories in the `PATH` environment variable: `which`, `whence`, `whereis`, and `whatis`. The first pair of commands produce similar results as the `which` command, as discussed below.

The `which` command gives the full path to whatever executable that you specify or a blank line if the executable is not in any directory that is specified in the `PATH` environment variable." This is useful for finding out whether a particular command or utility is installed in the system.

```
which rm
```

The output of the preceding command is here:

```
/bin/rm
```

The `whereis` command provides the information that you get from the `where` command:

```
$ whereis rm

/bin/rm
```

The whatis command looks up the specified command in the whatis database, which is useful for identifying system commands and important configuration files:

```
git-rm(1)               - Remove files from the working tree
and from the index

grm(1), rm(1)           - remove files or directories

rm(1), unlink(1)        - remove directory entries
```

Consider it a simplified "man" command, which displays concise details about bash commands (e.g., type man ls and you will see several pages of explanation regarding the ls command).

THE PRINTF COMMAND AND THE ECHO COMMAND

In brief, use the printf command instead of the echo command if you need to control the output format. One key difference is that the echo command prints a newline character, whereas the printf statement does not print a newline character. Keep this point in mind when you see the printf statement in the awk code samples in Chapter 7.

As a simple example, place the following code snippet in a shell script:

```
printf "%-5s %-10s %-4s\n" ABC DEF GHI
printf "%-5s %-10s %-4.2f\n" ABC DEF 12.3456
```

Make the shell script executable and then launch the shell script, after which you will see the following output:

```
ABC   DEF        GHI
ABC   DEF        12.35
```

On the other hand, if you type the following pair of commands:

```
echo "ABC DEF GHI"
echo "ABC DEF 12.3456"
```

You will see the following output:

```
ABC DEF GHI
ABC DEF 12.3456
```

A detailed (and very lengthy) discussion regarding the printf statement and the echo command is here:

https://unix.stackexchange.com/questions/65803/why-is-printf-better-than-echo

THE CUT COMMAND

The cut command enables you to extract fields from a text file or an input stream (which itself might be part of a pipe command). When you read the documentation about bash commands that mention IFS ("Internal Field Separator"), this refers to the field separator. The typical value for IFS is a space character (" "), but you can change this value when you are processing files via the awk command (discussed in Chapter 7).

In addition, the `cut` command allows you to specify a range of columns from an input stream. Some examples are here:

```
x= "abc def ghi"
echo $x | cut -d" " -f2
```

The output (using space `" "` as `IFS`, and `-f2` to indicate the second column, whereas `-f1` is the first column) of the preceding code snippet is here:

```
def
```

The following code snippet specifies the range of columns 2 through 5 (and does not specify an IFS or the `-f` option):

```
x="abc def ghi"
echo $x | cut -c2-5
```

Column positions start with the value 1 (not 0), and the range of column positions is non-inclusive. Hence, the range in `-c2-5` is columns 2, 3, and 4 (but not column 5). The output of the preceding code snippet is here:

```
bc d
```

Listing 1.13 displays the contents of `SplitName1.sh` that illustrates how to split a filename containing the "." character as a delimiter/`IFS`.

Listing 1.13: SplitName1.sh

```
fileName="06.22.04p.vp.0.tgz"
f1=`echo $fileName | cut -d"." -f1`
f2=`echo $fileName | cut -d"." -f2`
f3=`echo $fileName | cut -d"." -f3`
f4=`echo $fileName | cut -d"." -f4`
f5=`echo $fileName | cut -d"." -f5`
f5=`expr $f5 + 12`
newFileName="${f1}.${f2}.${f3}.${f4}.${f5}"
echo "newFileName: $newFileName"
```

Listing 1.13 uses the `echo` command and the `cut` command in order to initialize the variables `f1, f2, f3, f4, and f5`, after which a new filename is constructed. The output of the preceding shell script is here:

```
newFileName: 06.22.04p.vp.12
```

THE ECHO COMMAND AND WHITESPACES

The `echo` command preserves whitespaces in variables when those variables are defined in a shell script. However, the echo command removes whitespaces from variables that are defined in a command shell.

For example, open a command shell and then type the following code snippet:

```
x= "   a     b      c   "
echo $x
```

The output of the preceding `echo` command is shown here:

```
a b c
```

As you can see, the `echo` command removed all leading whitespaces, duplicate whitespaces, and trailing whitespaces (even if you use single quotes instead of double quotes). However, if you define a variable with whitespaces inside a shell script, the `echo` command does not remove the extra whitespaces.

Listing 1.1 displays the contents of `EchoCut.sh` that illustrate the differences that can occur when the `echo` command is used with the `cut` command.

Listing 1.1: EchoCut.sh

```
x1="123   456   789"
x2="123 456 789"
echo "x1 = $x1"
echo "x2 = $x2"
x3=`echo $x1   | cut -c1-7`
x4=`echo "$x1" | cut -c1-7`
x5=`echo $x2   | cut -c1-7`
echo "x3 = $x3"
echo "x4 = $x4"
echo "x5 = $x5"
```

Launch the code in Listing 1.1 and you will see the following output:

```
x1 = 123   456   789
x2 = 123 456 789
x3 = 123 456
x4 = 123   4
x5 = 123 456
```

The value of x3 is probably different from what you expected: there is only one blank space between `123` and `456` instead of the three blank spaces that appear in the definition of the variable `x1`.

As another variant, suppose that you define a text file `tab1` with a single line of text that contains a leading "tab" character. Now define the variable x from the command line as follows:

```
x=`cat tab1`
echo "x = $x"
```

The output of the preceding code snippet removes all "tab" characters and multiple whitespaces. In case you're wondering, the purpose of the "backticks" in the preceding code snippet is explained in the next section.

Thus, you need to be careful when you write shell scripts that contain the echo command in a pipe command in order to determine the contents of specific columns of text files (such as payroll files and other files with financial data). The solution involves the use of double quotation marks (and sometimes the `IFS` variable that is discussed in Chapter 2) that you can see in the definition of `x4`.

COMMAND SUBSTITUTION ("BACKTICK")

The "backtick" (also called command substitution) feature of `bash` (and other shells) is very powerful and enables you to combine multiple `bash` commands. You can also write very compact and powerful (and complicated) shell scripts with command substitution. The syntax is to simply precede and follow your command with a "`" (backtick) character. In Listing 1.2 below, the backtick command is `` `ls *py` ``.

Listing 1.2 displays the contents of `CommandSubst.sh` displays a subset of the list of files in a directory.

Listing 1.2: CommandSubst.sh

```
for f in `ls *py`
do
  echo "file is: $f"
done
```

Listing 1.2 contains a `for` loop that displays the filenames (in the current directory) that have a `py` suffix.

The output of Listing 1.2 on my Macbook Pro is here:

```
file is: CapitalizeList.py
file is: CompareStrings.py
file is: FixedColumnCount1.py
file is: FixedColumnWidth1.py
file is: LongestShortest1.py
file is: My2DMatrix.pyß
file is: PythonBash.py
file is: PythonBash2.py
file is: StringChars1.py
file is: Triangular1.py
file is: Triangular2.py
file is: Zip1.py
```

NOTE *The output depends on whether or not you have any files with a `.py` suffix in the directory where you execute `CommandSubst.sh`.*

THE "PIPE" SYMBOL AND MULTIPLE COMMANDS

At this point, you've seen various combinations of `bash` commands that are connected with the "`|`" symbol. In addition, you can redirect the output to a file. The general form looks something like this:

```
cmd1 | cmd2 | cmd3 …. >mylist
```

What happens if there are intermediate errors? Fortunately, you can redirect error messages to a text file if you need to review them. In fact, it's also possible to redirect `stderr` ("standard error") to `stdout` ("standard out"), which is beyond the scope of this chapter.

Question: can an intermediate error cause the entire "pipeline" to fail? Yes, it's possible for this to happen, and unfortunately, it's usually a trial-and-error process to debug long and complex commands that involve multiple pipe symbols.

Now consider the case where you need to redirect the output of multiple commands to the same location. For example, the following commands display output on the screen:

```
ls | sort; echo "the contents of /tmp: "; ls /tmp
```

You can easily redirect the output to a file with this command:

```
(ls | sort; echo "the contents of /tmp:"; ls /tmp) > myfile1
```

However, each of the preceding commands inside the parentheses spawns a subshell (which is an extra process that consumes memory and cpu). You can avoid spawning subshells by using `{}` instead of `()`, as shown here (and the whitespace after `{` and before `}` are required):

```
{ ls | sort; echo "the contents of /tmp:"; ls /tmp } > myfile1
```

Suppose that you want to set a variable, execute a command, and invoke a second command via a pipe, as shown here:

```
name=SMITH cmd1 | cmd2
```

Unfortunately, `cmd2` in the preceding code snippet does not recognize the value of `name`, but there is a simple solution, as shown here:

```
(name=SMITH cmd1) | cmd2
```

Use the double ampersand `&&` symbol if you want to execute a command only if a prior command succeeds. For example, the `cd` command only works if the `mkdir` command succeeds in the following code snippet:

```
mkdir /tmp2/abc && cd /tmp2/abc
```

The preceding command will fail because (by default) the `/tmp2` does not exist. On the other hand, the following command succeeds because the `-p` option ensures that intermediate directories are created:

```
mkdir -p /tmp/abc/def && cd /tmp/abc && ls -l
```

USING A SEMICOLON TO SEPARATE COMMANDS

You can combine multiple commands with a semicolon (";"), as shown here:

```
cd /tmp; pwd; cd ~; pwd
```

The preceding code snippet navigates to the `/tmp` directory, prints the full path to the current directory, returns to the previous directory, and again prints the full path to the current directory. The output of the preceding command is here:

```
/tmp
/Users/jsmith
```

You can use command substitution (discussed in the next section) to assign the output to a variable, as shown here:

```
x=`cd /tmp; pwd; cd ~; pwd`
echo $x
```

The output of the preceding snippet is here:

```
/tmp /Users/jsmith
```

THE paste COMMAND

The `paste` command is useful when you need to combine two files in a "pairwise" fashion. For example, Listing 1.10 and Listing 1.11 display the contents of the text files `list1` and `list2`, respectively. You can think of `paste` as treating the contents of the second file as an additional column for the first file. In our first example, the first file has a list of files to copy, the second file also has a list of files that are the destination for the `cp` command. The paste command merges the two files into an output that could then be run to execute all the copy commands in one step.

Listing 1.10: list1

```
cp abc.sh
cp abc2.sh
cp abc3.sh
```

Listing 1.11: list2

```
def.sh
def2.sh
def3.sh
```

Listing 1.12 display the result of invoking the following command:

```
paste list1 list2 >list1.sh
```

Listing 1.12: list1.sh

```
cp abc.sh    def.sh
cp abc2.sh   def2.sh
cp abc3.sh   def3.sh
```

Listing 1.12 contains three `cp` commands that are the result of invoking the paste command. If you want to execute the commands in Listing 1.12, make this shell script executable and then launch the script, as shown here:

```
chmod +x list1.sh
./list1.sh
```

Inserting Blank Lines with the paste Command

Instead of merging two equal length files, `paste` can also be used to add the same thing to every line in a file.

Suppose that the text file names.txt contains the following lines:

```
Jane Smith
John Jones
Dave Edwards
```

The following command inserts a blank line after every line in `names.txt`:

```
paste -d'\n' - /dev/null < names.txt
```

The output from the preceding command is here:

```
Jane Smith

John Jones

Dave Edwards
```

Insert a blank line after every other line in `names.txt` with this command:

```
paste -d'\n' - - /dev/null < names.txt
```

The output is here:

```
Jane Smith
John Jones

Dave Edwards
```

Insert a blank line after every third line in `names.txt` with this command:

```
paste -d'\n' - - - /dev/null < names.txt
```

The output is here:

```
Jane Smith
John Jones
Dave Edwards

```

NOTE *That there is a blank line after the third line in the preceding output. The shell script `joinlines.sh` (later in this chapter) also contains examples of one-line paste commands for joining consecutive lines of a dataset or text file.*

A SIMPLE USE CASE WITH THE PASTE COMMAND

The code sample in this section shows you how to use the `paste` command in order to join consecutive rows in a dataset. Listing 1.14 displays the contents of `linepairs.csv` that contains letter and number pairs, and Listing 1.15 contains `reversecolumns.sh` that illustrates how to match the pairs even though the line breaks are in different places between numbers and letters.

Listing 1.14: linepairs.csv

```
a,b,c,d,e,f,g
h,i,j,k,l
1,2,3,4,5,6,7,8,9
10,11,12
```

Listing 1.15: linepairs.sh

```
inputfile="linepairs.csv"
outputfile="linepairsjoined.csv"
# join pairs of consecutive lines:
paste -d " "  - - < $inputfile > $outputfile
# join three consecutive lines:
#paste -d " "  - - - < $inputfile > $outputfile
# join four consecutive lines:
#paste -d " "  - - - - < $inputfile > $outputfile
```

The contents of the output file are shown here (note that the script is just joining pairs of lines, and the three and four line command examples are commented out):

```
a,b,c,d,e,f,g h,i,j,k,l
1,2,3,4,5,6,7,8,9 10,11,12
```

Notice that the preceding output is not completely correct: there is a space " "instead of a "," whenever a pair of lines are joined (between "g" and "h" and also between "9 and 10"). We can make the necessary revision using the `sed` command (discussed in Chapter 6):

```
cat $outputfile | sed "s/ /,/g" > $outputfile2
```

Examine the contents of `$outputfile` to see the result of the preceding code snippet.

A SIMPLE USE CASE WITH CUT AND PASTE COMMANDS

The code sample in this section shows you how to use the `cut` and `paste` commands in order to reverse the order of two columns in a dataset. Keep in mind that the purpose of the shell script in Listing 1.17 is to help you get some practice for writing `bash` scripts. The better solution involves a single line of code (shown at the end of this section).

Listing 1.16 displays the contents of `namepairs.csv` that contains the first name and last name of a set of people, and Listing 1.17 contains `reversecolumns.sh` that illustrates how to reverse these two columns.

Listing 1.16: namepairs.csv

```
Jane,Smith
Dave,Jones
Sara,Edwards
```

Listing 1.17: reversecolums.sh

```
inputfile="namepairs.csv"
outputfile="reversenames.csv"
fnames="fnames"
lnames="lnames"
cat $inputfile|cut -d"," -f1 > $fnames
```

```
cat $inputfile|cut -d"," -f2 > $lnames
paste -d"," $lnames $fnames > $outputfile
```

The contents of the output file `$outputfile` are shown here:

```
Smith,Jane
Jones,Dave
Edwards,Sara
```

The code in Listing 1.17 (after removing blank lines) consists of seven lines of code that involves creating two extra intermediate files. Unless you need those files, it's a good idea to remove those two files (which you can do with one `rm` command).

Although Listing 1.17 is straightforward, there is a simpler way to execute this task: use the `cat` command and the `awk` command (discussed in detail in Chapter 7).

Specifically, compare the contents of `reversecolumns.sh` with the following single line of code that combines the `cat` command and the `awk` command in order to generate the same output:

```
cat namepairs.csv |awk -F"," '{print $2 "," $1}'
```

The output from the preceding code snippet is here:

```
Smith,Jane
Jones,Dave
Edwards,Sara
```

An even simpler solution involves just the `awk` command in order to generate the same output, as shown here:

```
awk -F"," '{print $2 "," $1}' namepairs.csv
```

As you can see, there is a big difference between the preceding pair of one-line solutions and the initial solution that you saw at the beginning of this section. If you are unfamiliar with the `awk` command, then obviously you would not have thought of the second solution. However, the more you learn about `bash` commands and how to combine them, the more adept you will become in terms of writing better shell scripts to solve various tasks. Another important point: document the commands as they get more complex, as they can be hard to interpret later by others, or even by yourself if enough time has passed. A comment like the following can be extremely helpful in interpreting code:

```
# This command reverses first and last names in namepairs.txt
cat namepairs.txt |awk -F"," '{print $2 "," $1}'
```

WHAT ABOUT ZSH?

The latest version of OS X provides `zsh` (often pronounced ZEE-shell) as the default shell instead of the `bash` shell. You can find the directory that contains `zsh` by typing this command:

```
which zsh
```

and the result will be:

```
/bin/zsh
```

In case you didn't already know, `bash` and `zsh` have some highly useful features common, as shown below:

the z-command
auto-completion
auto-correction
color customization

Unlike `zsh`, the `bash` shell does not have inline wildcard expansion. Hence, tab completion in `bash` acts like a command output. On the other hand, tab completion in `zsh` resembles a "drop-down" list that disappears after you type additional characters.

In addition, the `bash` shell does not support prefix or postfix command aliases. A comparison of the `bash` shell and `zsh` is here:

https://sunlightmedia.org/bash-vs-zsh

Switching between bash and zsh

Type the following command to set `zsh` as the default shell in a command shell:

```
chsh -s /bin/zsh
```

Switch from `zsh` back to `bash` with this command:

```
chsh -s /bin/bash
```

NOTE *That both of the preceding commands only affect the command shell where you launched the commands.*

Configuring zsh

`Bash` stores user-related configuration settings in the hidden file (in your home directory) `.bashrc`, whereas `zsh` uses the file `.zshrc`. However, keep in mind that you need to create the latter file because it's not created for you.

`Bash` uses the login-related file `.bash_profile`, whereas `zsh` uses the file `.zprofile` (also in your home directory) that is invoked when you log into your system. Consider the use of configuration managers such as Prezto or Antigen to help you set values to variables. Perform an online search for more details regarding `zsh`.

SUMMARY

This chapter started with an introduction to some Unix shells, followed by a brief discussion of files, file permissions, and directories. You also learned how to create files and directories and how to change their permissions. Next, you learned about environment variables, how to display their values, and also how to use aliases.

Next, you learned about the `cut` command (for cutting columns and/or fields) and the `paste` command (for "pasting" test together vertically). Finally, you saw two use cases, the first of which involved the `paste` command to switch the order to two columns in a dataset, and the second showed you another way to perform the same task using a combination of the `cut` command and `paste` command.

CHAPTER 2

FILES AND DIRECTORIES

This chapter discusses files and directories and various useful `bash` commands for managing them. You will learn how to use simple commands that can simplify your tasks. In Chapter 8 and Chapter 9, you will learn how to create shell scripts involving some of the commands in this chapter, which will further reduce the amount of time that you spend performing routine tasks.

The first part of this chapter shows works with file-related commands, such as `touch`, `mv`, `cp`, `rm`, and so forth. The second part of this chapter contains shell commands for managing directories. The third part of this chapter discusses metacharacters and variables that you can use when working with files and shell scripts.

CREATE, COPY, REMOVE, AND MOVE FILES

This section discusses file-related commands in `bash`, such as `touch`, `cp`, `rm` that enable you to create, copy, and remove files, respectively. The following subsections illustrate how to use these convenient commands. Except for the section that discusses how to create text files, the other sections pertain to text files as well as binary files.

Creating Text Files

Use your favorite editor (such as vim, notepad++, emacs, and so forth) to create a text file. If you prefer, create an empty text file via the `touch` command. For example, the following command illustrates how to create an empty file called `abc`:

```
touch abc
```

If you issue the command `ls -l abc` you will see something like this:

```
-rw-r--r--  1 owner  staff  0 Jul  2 16:39 abc
```

Copying Files

You can copy the file `abc` (or any other file) to `abc2` with this command:

```
cp abc abc2
```

Be careful when you use the `cp` command: if the target file already exists, make sure that you really do want to overwrite its contents with the source file.

For example, both of the following commands copy the files `abc` and `abc2` to the `/tmp` directory:

```
cp abc abc2 /tmp
```

```
cp abc* /tmp
```

However, the following command replaces the file abc2 with the contents of the file `abc`:

```
cp abc*
```

The reason is simple: the bash shell expands the preceding regular expression *before* executing the `cp` command. Consequently, the preceding `cp` command is "expanded" to this `cp` command:

```
cp abc abc2
```

Fortunately, you can prevent an existing file from being overwritten by another file with this command:

```
set –o noclobber
```

Now if you invoke the earlier `cp` command, you will see the following error message:

```
bash: abc2: cannot overwrite existing file
```

The `cp` command provides several useful switches that enable you to control the set of files that you want to copy:

the `-a` archive flag (for copying an entire directory tree)

the `-u` update flag (which prevents overwriting identically-named newer files)

the `-r` and `-R` recursive flags.

The `-r` option is useful for copying the files and all the subdirectories of a directory to another directory. For example, the following command copies the files and sub-directories (if any) from `$HOME/abc` to the directory `$HOME/def`:

```
cd $HOME/abc
```

```
cp –r . ../def
```

Copy Files with Command Substitution

Listing 2.1 displays the contents of `CommandSubst.sh` that illustrates how to use "backtick" to copy a set of files to a directory.

Listing 2.1 CommandSubst.sh

```
mkdir textfiles
cp `ls *txt` textfiles
```

The preceding pair of commands creates a directory `textfiles` in the current directory and then copies all the files (located in the current directory) with the suffix `txt` into the `textfiles` sub-directory.

Keep in mind that this command will not copy any subdirectories that have the suffix `txt`. If you want to copy files that have the suffix `.txt`, use this command:

```
cp `ls *txt` textfiles
```

Another caveat: if you have the directory `abc.txt`, or some other directory with the `.txt` suffix, then the contents of that directory will not be copied (you will see an error message). The following commands will also fail to copy the contents of `abc.txt` to the sub-directory `textfiles`:

```
cp `ls -R *.txt` textfiles
cp -r `ls -R *.txt` textfiles
```

Invoke the following command to ensure that the subdirectory `abc.txt` is copied into the `textfiles` sub-directory:

```
cp -r abc.txt textfiles
```

If you want to copy the subdirectories of `abc.txt` but not the directory `abc.txt` into `textfiles`, use this command:

```
cp -r abc.txt/* textfiles
```

Deleting Files

The `rm` command removes files and when you specify the `-r` option, the `rm` command removes the contents of a directory (as well as the directory). For example, remove the file `abc` using the `rm` command:

```
rm abc
```

The `rm` command has some useful options, such as `-r`, `-f`, and `-i`, which represent "recursive," "force," and "interactive."

You can remove the contents of the current directory and all of its subdirectories with this command:

```
rm -rf *
```

NOTE *Be very careful when you use `rm -rf *` so that you do not inadvertently delete the wrong tree of files on your machine.*

The `-i` option is very handy when you want to be prompted before a file is deleted. Before deleting a set of files, use the `ls` command to preview the files that you intend to delete using the `rm` command. For example, run this command:

```
ls *.sh
```

The preceding command shows you the files that you will delete when you run this command:

```
rm *.sh
```

Earlier in this section you saw how to use command substitution, which redirects the output of one command as the input of another command, to copy a set of files. As another variation, the following command removes the files that are listed in the text file `remove_list.txt` instead of specifying a list of files from the command line:

```
rm `cat remove_list.txt`
```

If there is a possibility (at some point in the future) that you might need some of the files that you intend to delete, another option is to create a directory and move those files into that directory, as shown here:

```
mkdir $HOME/backup-shell-scripts
mv *sh $HOME/backup-shell-scripts
```

Moving Files

The `mv` command is equivalent to a combination of `cp` and `rm`. You can use this command to move multiple files to a directory or even to rename a directory. When used in a non-interactive script, `mv` supports the `-f` (force) option to bypass user input. When a directory is moved to a pre-existing directory, it becomes a sub-directory of the destination directory.

The `ln` Command

The `ln` command enables you to create a symbolic link to an existing file, which is advantageous when the existing file is large because the symbolic link involves minimal additional overhead. Moreover, changes to the existing file are automatically available in the symbolic link, which means that you can maintain one file as "the source of truth" instead of making the same update to multiple copies of a file.

As a simple example, suppose that you have a file called `document1.txt` in your `$HOME` directory. The following command creates a symbolic link called doc2 to the file `document1.txt`:

```
ln –s $HOME/document1.txt doc2
```

If you invoke `ls –l` on `doc2` you will see something like this:

```
lrwxr-xr-x  1 owner  staff  26 Feb  8 10:57 doc2 -> /
Users/owner/document1.txt
```

If you remove `doc2` it will not affect `$HOME/document1.txt`; however, if you remove the latter file without removing `doc2`, then `doc2` will still appear in a long listing, but when you attempt to view the contents of `doc2`, the file is empty (as you would expect).

THE BASENAME, DIRNAME, AND FILE COMMANDS

These commands enable you to find the base portion of a filename, the directory portion, and the type of file, respectively. Here are some examples:

```
$ x="/tmp/a.b.c.js"
$ basename $x .js
$ a.b.c
$ a="/tmp/a b.js"
$ basename $a .js
a
b.js
.js
$ basename "$a" .js
a b
$ dirname $x
/tmp
$ file /bin/ls
/bin/ls: Mach-O 64-bit executable x86_64
```

THE `wc` COMMAND

In Chapter 1, you learned how to use the `ls` command to obtain information about files in a given directory. You can view the number of lines, words, and characters in a set of files using the `wc` command. For example, if you execute the command `wc *` in a command shell you will see information for all the files in a directory, similar to the following output:

```
 3       6      25 apple-care.txt
12      28     146 checkin-commands.txt
27      55     478 iphonemeetup.txt
 2       1      12 kyrgyzstan.txt
 1       2      11 outfile.txt
 2       2      12 output.txt
 5      11     176 ssl-instructions.txt
52     105     860 total
```

As you can see from the last line in the previous output, there are a total of 52 lines, 105 words, and 860 characters in the text files in this directory. If you run the command "`wc o*`" you will see information about files that start with the letter `o`, as shown here:

```
1       2      11 outfile.txt
2       2      12 output.txt
3       4      23 total
```

You can count the number of files in a directory with `ls -1 |wc` as shown here:

```
7       7     112
```

The shell commands `cat`, `more`, `less`, `head`, and `tail` display different parts of a file. For short files, these commands can overlap in terms of their output. The next several sections provide more details about these commands, along with some examples.

THE CAT COMMAND

The `cat` command displays the entire contents of a file, which is convenient for small files. For longer files, you can use the more command that enables you to "page" through the contents of a file.

If you type this command:

```
cat iphonemeetup.txt
```

you will see the following output:

```
iPhone meetup
=============
* iPhone.WebDev.com
* iui.googlecode.com
* tinyurl.sqllite.com
* touchcode.googlecode.com: open source supports JSON

XCode: supports SVG-like functionality
blog site: iphoneinaction.manning.com
iPhone in Action: manning.com/callen
iPhone dev camp: probably summer 2021

openGL-ES how to:
* create arcs/circles/ellipses
* linear/radial gradients
* Bezier curves
* Urbanspoon vs Yelp
```

You can see size-related attributes with the command `wc iphonemeetup.txt`:

```
21      48     417 iphonemeetup.txt
```

THE MORE COMMAND AND THE LESS COMMAND

The `more` command enables you to view "pages" of content in a file. Press the space bar to advance to the next page, and press the return key to advance a

single line. The `less` command is similar to the `more` command. An example of the `more` command is here:

```
more abc.txt
```

Alternatively, you can use this form, but remember that it's less efficient because two processes are involved:

```
cat abc.txt |more
```

The `more` command contains some useful options. For instance, if you want the `more` command to start from the 15th line in a file instead of the first line, use this command:

```
more +15 abc.txt
```

If a file contains multiple consecutive blank lines, you can remove them from the output with this command:

```
more -s abc.txt
```

Search for the pattern `abc` in a text file via the following command:

```
more +/abc.txt
```

THE HEAD COMMAND

The `head` command enables you to display an initial set of lines, and the default number is 10. For example, the following command displays the first three lines of test.txt:

```
cat test.txt |head -3
```

The following command also displays the first three lines of `test.txt`:

```
head -3 test.txt
```

You can display the first three lines of multiple files. The following command displays the first three lines in the text file `columns2.txt`, `columns3.txt`, and `columns4.txt`:

```
head -3 columns[2-4].txt
```

The output of the preceding command is here:

```
==> columns2.txt <==
one two
three four
one two three four

==> columns3.txt <==
123 one two
456 three four
one two three four
```

```
==> columns4.txt <==
123 ONE TWO
456 three four
ONE TWO THREE FOUR
```

The following code snippet checks if the first line of `test.txt` contains the string `aa`:

```
x=`cat test.txt |head -1|grep aa`
if [ "$x" != "" ]
  echo "found aa in the first line"
fi
```

The `head` command displays the first 10 lines of a file, and the `tail` command displays the final 10 lines of a file. Instead of showing you the output (which you can see from the previous listing), let's combine the `head` and `tail` commands with the `wc` command.

The following command combines the `head` and `wc` commands:

```
head iphonemeetup.txt |wc
```

The preceding command displays the following output:

```
10      22     224
```

You can also display the contents of a file whereby each line is preceded by a line number. For example, the command combines the `cat` and `head` commands:

```
cat –n iphonemeetup.txt |head -4
```

The preceding command displays the following output:

```
1   iPhone meetup
2   =============
3   * iPhone.WebDev.com
4   * iui.googlecode.com
```

THE TAIL COMMAND

The `tail` command enables you to display a set of lines at the end of a file, and the default number is 10. For example, the following command displays the last three lines of `test.txt`:

```
cat test.txt |tail -3
```

The following command also displays the last three lines of `test.txt`:

```
tail -3 test.txt
```

You can display the last three lines of multiple files. The following command displays the last three lines in the text file `columns2.txt`, `columns3.txt`, and `columns4.txt`:

```
tail -3 columns[2-4].txt
```

The output of the preceding command is here:

```
==> columns2.txt <==
five six
one two three
four five

==> columns3.txt <==
five 123 six
one two three
four five

==> columns4.txt <==
five 123 six
one two three
four five
```

The following code snippet checks if the first line of `test.txt` contains the string `aa`:

```
x=`cat test.txt |tail -1|grep aa`
if [ "$x" != "" ]
  echo "found aa in the test.txt"
fi
```

The following command displays three values:

```
tail iphonemeetup.txt |wc
10      26     192
```

The first number in both output listings is 10, which confirms that only the first 10 lines or the final ten lines are displayed. Note that if a file contains 10 or fewer lines, then the output of `head`, `tail`, `cat`, and `more` is identical.

You can change the number of lines that you want to see in the output of the `head` command or the `tail` command. For example, the command displays three lines:

```
$ head -3 iphonemeetup.txt
iPhone meetup
=============
* iPhone.WebDev.com
```

The next command the last three lines:

```
$ tail -3 iphonemeetup.txt
* Bezier curves
* Urbanspoon vs Yelp
```

The `tail` command with the `-f` option is useful when you have a long-running process that is redirecting output to a file. For example, suppose that you invoke this command from your home directory:

```
find . -print |xargs grep -i abc >/tmp/abc &
```

Invoke the following command to see the contents of the file `/tmp/abc` whenever it is updated:

```
tail -f /tmp/abc
```

COMPARING FILE CONTENTS

There are several commands for comparing text files, such as the `cmp` command and the `diff` command.

The `cmp` command is a simpler version of the `diff` command: `diff` reports the differences between two files, whereas `cmp` only shows at what point they differ.

NOTE *Both `diff` and `cmp` return an exit status of 0 if the compared files are identical, and 1 if the files are different, so you can use both commands in a test construct within a shell script.*

The `comm` command is useful for comparing sorted files:

```
comm -options first-file second-file
```

The command `comm file1 file2` outputs three columns:

column 1 = lines unique to file1
column 2 = lines unique to file2
column 3 = lines common to both.

The options allow the suppressing of the output of one or more columns.

-1 suppresses column 1
-2 suppresses column 2
-3 suppresses column 3
-12 suppresses both columns 1 and 2, etc.

The `comm` command is useful for comparing "dictionaries" or word lists containing sorted text files with one word per line.

THE PARTS OF A FILENAME

The `basename` command "strips" the path information from a filename, printing only the filename. The construction `basename $0` is the name of the currently executing script. This functionality can be used for "usage" messages if, for example, a script is called with missing arguments:

```
echo "Usage: 'basename $0' arg1 arg2 ... argn"
```

The `dirname` command strips the `basename` from a filename, printing only the path information.

NOTE *`Basename` and `dirname` can operate on any arbitrary string. The argument does not need to refer to an existing file, or even be a filename.*

The `strings` command displays printable strings (if any) in a binary or data file. An example invocation is here:

```
strings /bin/ls
```

The first few lines of output from the preceding command are here:

```
$FreeBSD: src/bin/ls/cmp.c,v 1.12 2002/06/30 05:13:54
obrien Exp $
@(#) Copyright (c) 1989, 1993, 1994
The Regents of the University of California. All
rights reserved.
$FreeBSD: src/bin/ls/ls.c,v 1.66 2002/09/21 01:28:36
wollman Exp $
$FreeBSD: src/bin/ls/print.c,v 1.57 2002/08/29
14:29:09 keramida Exp $
$FreeBSD: src/bin/ls/util.c,v 1.38 2005/06/03
11:05:58 dd Exp $
\\""
@(#)PROGRAM:ls PROJECT:file_cmds-264.50.1
COLUMNS
1@ABCFGHLOPRSTUWabcdefghiklmnopqrstuvwx
bin/ls
Unix2003
```

WORKING WITH FILE PERMISSIONS

In a previous section, you used the `touch` command to create an empty file `abc` and then saw its long listing:

```
-rw-r--r-- 1 owner staff 0 Nov 2 17:12 abc
```

Each file in `bash` contains a set of permissions for three different user groups: the owner, the group, and the world. The set of permissions are read, write, and execute that they have value 4, 2, and 1, respectively, in base 8 (octal). Thus, the permissions for a file in each group can have the following values: 0 (none), 1 (execute), 2 (write), 4 (read), 5 (read and execute), 6 (read and write), and 7 (read, write, and execute).

For example, a file whose permissions are 755 indicate:

Owner has read/write/execute permissions

Group has write/execute permissions

World has write/execute permissions

You can use various options with the `chmod` command to change permissions for a file.

The chmod Command

The `chmod` command enables you to change permissions for files and directories. The octal representation 777 corresponds to the permissions `rwxrwxrwx`, which enables read, write, and execute for all three groups. The octal representation 644 corresponds to the permissions `rw-r—r—`.

The following command makes "filename" executable for all users:

```
chmod +x filename
```

Note that the following command makes a file executable only for the owner of the file:

```
chmod u+x filename
```

The following command sets "suid" bit on "filename" permissions, which allows an ordinary user to execute "filename" with same privileges as the owner of the file (but is not applicable to shell scripts):

```
chmod u+s filename
```

The following command makes filename readable/writable to the owner and only readable to group and others:

```
chmod 644 filename
```

The following command makes "filename" read-only for everyone:

```
chmod 444 filename
```

Provide everyone with read, write, and execute permission in the directory (and also sets the "sticky bit"):

```
chmod 1777 directory-name
```

Revoke all permissions for a directory (but no restrictions are enforced on the root user):

```
chmod 000 directory-name
```

Chapter 4 discusses how to use conditional logic to check (and also modify) file permissions via shell scripts.

Changing owner, permissions, and groups

The `chown` command enables you to change ownership of files and directories. For example, the following command assigns `dsmith` as the owner of the files with a suffix `txt` in the current directory:

```
chown dsmith *txt
```

The `chgrp` command changes the group of files and directories, as shown here:

```
chgrp internal *.txt
```

In addition, the `chown` and `chgrp` commands support recursion via the `–R` option, which means that the commands can affect files that are in a subdirectory of the current directory.

The umask and ulimit Commands

Whenever you create a file in `bash`, the environment variable `umask` contains the complement (base 8) of the default permissions that are "assigned" to that file. You can see the value of `umask` by typing this command at the command line, and its typical value is 0022. If you perform the (base 8) complement of this value, the result is 755.

The `ulimit` command specifies the maximum size of a file that you can create on your machine. When you invoke this command in a command shell, you will either see a numeric value or the word `unlimited`.

WORKING WITH DIRECTORIES

A directory is a file that stores filenames and related information. All files (i.e., ordinary, special, or directory) are contained in directories. UNIX-based file systems have a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree. The tree has a single root node, the slash character (/), and all other directories are contained below it.

The position of any file within the hierarchy is described by its pathname.

The elements of a pathname are separated by a slash ("/") character. A pathname is absolute if it is described in relation to root, so absolute pathnames always begin with a /. These are some examples of absolute filenames:

```
/etc/passwd
/users/oac/ml/class
/dev/rdsk/0s5
```

A pathname can also be relative to your current working directory. Relative pathnames begin with ./.

Absolute and Relative Directories

You can navigate to your home directory with any of these commands:

```
cd $HOME
cd ~
cd
```

Note that in Windows the `cd` command shows you the current directory (it does not navigate to the home directory).

The tilde "~" always indicates a home directory. If you want to go in any other user's home directory then use the following command:

```
cd ~username
```

You can navigate back to the location of the directory before navigating to the current directory with this command:

```
cd -
```

Absolute/Relative Pathnames

Directories are arranged in a hierarchy with root (/) at the top. The position of any file within the hierarchy is described by its full pathname. Elements of

a pathname are separated by a /. A pathname is absolute if it is described in relation to root, so absolute pathnames always begin with a /. Here are some examples of absolute filenames:

```
/etc/passwd
/users/oac/ml/class
/dev/rdsk/Os5
```

To determine your current directory, use the `pwd` command:

```
$ pwd
```

The output will be something like this:

```
/Users/owner/Downloads
```

Display the contents of a directory with the ls command:

```
$ ls /usr/bin
```

A tiny display from the preceding command is here:

```
fc
fddist
fdesetup
fg
fgrep
file
```

In fact, all the built-in executables that are discussed in this book reside in the `/usr/bin` directory.

Creating Directories

If you specify multiple directories on the command line, `mkdir` creates each of the directories. For example, the following command creates the directories `docs` and `pub` as siblings under the current directory:

```
mkdir docs pub
```

Compare the preceding command with the following command that creates the directory `test` and a sub-directory `data` under the current directory:

```
mkdir –p test/data
```

The `-p` option forces the creation of missing intermediate directories (if any). If an intermediate directory does not exist, the `mkdir` issues an error message. For example, suppose that the intermediate subdirectory account does not exist in the following code snippet:

```
$mkdir /tmp/accounting/test
mkdir: Failed to make directory "/tmp/accounting/
test";
No such file or directory
```

You can also use the `–p` option to create a sub-directory in your `$HOME` directory, as shown here:

```
mkdir –p $HOME/a/b/c/new-directory
```

The preceding command creates the following subdirectories in case any of them do not exist:

```
$HOME/a
$HOME/a/b
$HOME/a/b/c
$HOME/a/b/c/new-directory
```

Removing Directories

Earlier you learned how to use `rm -r` in order to remove a directory and its contents. You can also delete empty directories via the `rmdir` command as follows:

```
rmdir dirname1
```

You can delete multiple empty directories at a time as follows:

```
rmdir dirname1 dirname2 dirname3
```

Again, keep in mind that the preceding command removes the directories `dirname1`, `dirname2`, and `dirname2` if they are empty. The `rmdir` command produces no output if it is successful.

However, if there are files in the `dirname1` directory, you can either use this command:

```
rm –r dirname1
```

or, alternatively, you can first remove the files in the subdirectory and then remove the directory itself, as shown here:

```
cd dirname1
rm –rf *
cd ../
rmdir dirname1
```

Navigating to Directories

Use the `cd` command to change to any directory by specifying a valid absolute or relative path. The syntax is as follows:

```
cd dirname
```

The value of `dirname` is the name of the target directory. For example, the command:

```
cd /usr/local/bin
```

will change to the directory `/usr/local/bin`.

You can change directories using an absolute path (as in the previous example) or via a relative path. For example, you can switch from this directory to the directory `/usr/home/jsmith` via the following relative path:

```
cd ../../home/jsmith
```

Keep in mind that the file system is a hierarchical tree-like system; hence, you can navigate to a parent directory via the double dot ("../") syntax. You can

also navigate to the parent directory of the current directory (and even higher if there are additional ancestor directories), as shown in the preceding example.

Moving Directories

The `mv` (move) command can also be used to rename a directory as well as renaming a file. The syntax for renaming a directory is the same as renaming a file:

```
mv olddir newdir
```

However, if the target directory already exists and contains at least one file, then the `mv` command will fail, an example of which is here:

```
mkdir /tmp/abcd
touch /tmp/abcd/red
mkdir abcd
mv abcd /tmp
```

Since the directory `/tmp/abcd` is not empty, you will see the following error message:

```
mv: rename abcd to /tmp/abcd: Directory not empty
```

In essence, bash provides a noclobber feature for non-empty directories (which is actually a very good feature).

USING QUOTE CHARACTERS

There are three types of quotes characters: single quotes (‘), double quotes (“), and ‘backquotes’ (`) that occur in matching pairs (‘’, “”, or ``). Although these quote characters might seem interchangeable, there are some differences.

Single quotes prevent the shell from “globbing” the argument or substituting the argument with the value of a shell variable. Characters within a pair of single quotes are interpreted literally, which means that their metacharacter meanings (if any) are ignored. Similarly, the shell does not replace references to the shell or environment variables with the value of the referenced variable.

Characters within a pair of *double quotes* are interpreted literally: their metacharacter meanings (if any) are ignored. However, the shell does replace references to the shell or environment variables with the value of the referenced variable.

Text within a pair of *backquotes* is interpreted as a command, which the shell executes before executing the rest of the command line. The output of the command replaces the original back-quoted text.

In addition, a metacharacter is treated literally whenever it is preceded by the backslash (\) character.

The following examples illustrate the differences when you use different quote characters:

```
echo $PATH
```

The `echo` command will print the value of the `PATH` shell variable. However, by enclosing the argument within single quotes, you obtain the desired result:

```
echo '$PATH'
```

Double quotes have a similar effect. They prevent the shell from "globbing" a filename but permit the expansion of shell variables.

The power of backquotes is the ability to execute a command and use its output as an argument of another command. For example, the following command displays the number of files in a user's home directory:

```
echo My home directory contains `ls ~ | wc -l` files.
```

The preceding command first executes the command contained within backquotes:

```
ls ~ | wc -l
```

For the purpose of illustration, let's assume that the preceding command displays the value 22. Then the earlier `echo` command displays the following output:

```
My home directory contains 22 files.
```

STREAMS AND REDIRECTION COMMANDS

The numbers 0,1, and 2 have a special significance when 0 is preceded by the "<" symbol, and also when the numbers 1 and 2 are followed by the ">" symbol, with no spaces between the digit 0 and "<", and no spaces between the digits 1 and 2 and ">".

In this situation, 0 refers to standard input (stdin), 1 refers to standard output (stdout), and 2 refers to standard error (stderr). The pattern "`1>file1`" and "`2>file2`" enable you to redirect output from an executable file.

In addition, the "directory" `/dev/null` refers to the `null bit bucket`, and anything that you redirect to this directory is essentially discarded. This construct is useful when you want to redirect error message that can be safely ignored.

Here are some examples:

```
cat $abc 1>std1 2>std2
```

You can redirect standard error to standard output and then redirect the latter to a single file, as shown here:

```
cat $abc 2>&1 1>std1
```

You can redirect the output of `bash` commands in various ways, depending on what your needs are, as shown below:

```
cmd > myfile    redirects the output of cmd to myfile
cmd >> myfile   appends the output of cmd to myfile
cmd < myfile    sends the contents of myfile to cmd
```

```
cmd 1>out1 2>out2   sends stdout to out1 and stderr to
out2
cmd 2>&1 1>onefile1 sends stdout and stderr to onefile1
```

You can also redirect error messages to `/dev/null`, which means that you will never see those error messages.

For example, if you attempt to list the contents of the non-existent directory `/temp`, you will see this error message:

```
ls: /temp: No such file or directory
```

However, you can suppress the preceding error message with this command:

```
ls /temp 2>/dev/null
```

You can redirect standard output and standard error to the same file with this command:

```
ls /temp 1>output1.txt 2>&1
```

In the preceding code snippet the file `output1.txt` contains the same error message.

Note that the following command displays the error message on the screen (notice the different location of `2>&1`):

```
ls /temp 2>&1 1>output1.txt
```

NOTE *Redirect error messages to `/dev/null` only if you are certain that they can be safely ignored.*

Redirection can be convenient in conjunction with the `find` command as well as sequences of commands that connected with the pipe symbol.

WORKING WITH METACHARACTERS

Earlier in this chapter, you were introduced to the `?` and `*` metacharacters in conjunction with the `cat` command. If you are new to metacharacters, it might be helpful to think of them as a set of wildcards.

In addition, *regular expressions* are a combination of normal text, special characters (in some cases), metacharacters, and character classes (discussed in the next section). Conceptually, a regular expression is analogous to how to perform a search in a "find" tool (press `ctrl-f` on your search engine), but `bash` allows for much more complex pattern matching because of its rich metacharacter set. Although there are entire books devoted to regular expressions, this section contains a basic introduction to metacharacters.

The following three metacharacters are useful with regular expressions:

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

Note that "something" in the preceding descriptions can refer to a digit, letter, word, or more complex combinations.

Now that you have a general idea of regular expressions, let's look at some examples that contain metacharacters.

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`, and so forth (but abc and a12 do not match).

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

The pipe "|" metacharacter (which has a different context from the pipe symbol n the command line: regular expressions 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 "`$`" metacharacter refers to the end of a line of text, and in regular expressions inside the `vi` editor, the "`$`" metacharacter can also refer 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"
```

One other interesting detail: the "`^`" metacharacter means "does *not* match" when it is the first character inside a pair of square brackets. The next section includes some examples of the "`^`" metacharacter.

WORKING WITH CHARACTER CLASSES

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

The previous section introduced you to the "`^`" metacharacter and here are some example 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 can understand the purpose of the following regular expressions:

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

Notice that example #3 contains the term `[a-z]`, which matches any lowercase letter, whereas example #5 contains the term `[^a-z]`, which (as mentioned in the previous section) means that it does *not* match a lowercase letter.

METACHARACTERS AND CHARACTER CLASSES

As you saw in Chapter 1, `bash` supports metacharacters as well as regular expressions. If you have worked with a scripting language such as Perl, or languages such as JavaScript and Java, you have undoubtedly encountered metacharacters.

Here is an expanded list of metacharacters that the bash shell supports:

```
? (0 or 1):          a? matches the string a (but not ab)
* (0 or more):       a* matches the string aaa (but not
                     baa)
+ (1 or more):       a+ matches aaa (but not baa)
^ (start of line):  ^[a] matches the string abc (but
                     not bc)
$ (end of line):     [c]$ matches the string abc (but
                     not cab)
. (a single dot):    matches any character (except
                     newline)
```

The preceding metacharacters can be used in `bash` with commands such as `ls` and `sed`, in shell scripts, and in editors such as `vi` (but keep in mind that there are some subtle differences in interpretation of metacharacters).

Digits and Characters

The following code snippets illustrate how to specify sequences of digits and sequences of character strings:

```
[0-9]      matches a single digit
[0-9][0-9] matches 2 consecutive digits
^[0-9]+$   matches a string consisting solely of
digits
```

You can define similar patterns using uppercase or lowercase letters:

```
[a-z]      matches a single lowercase letter
[A-Z]      matches a single uppercase letter
[a-z][A-Z] matches a single lowercase letter that is
followed by 1 uppercase letter
[a-zA-Z]   matches any upper or lowercase letter
```

Working with "^" and "\" and "!"

The purpose of the "^" character depends on its context in a regular expression. 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:

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

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

The backslash ("\") allows you to "escape" the meaning of a metacharacter. Just to clarify a bit further: a dot "." matches a single character, whereas the sequence "\." matches the dot "." character. Other examples 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
```

The "!" metacharacter means negation, as shown here:

```
[! abc ...] Matches any character other than those
specified
[! a - z ] Matches any character not in the specified
range
```

FILENAMES AND METACHARACTERS

Before the shell passes arguments to an external command or interprets a built-in command, it scans the command line for certain special characters and performs an operation known as filename "globbing." You already saw some metacharacters in an early section, and this section shows you how to use them with the ls command.

```
ls -l file1 file2 file3 file04
```

However, the following command reports the same information and is much quicker to type:

```
ls -l file*
```

Suppose you issued the following command:

```
ls -l file?
```

The ? filename metacharacter can match only a single character. Therefore, file04 would not appear in the output of the command. Similarly, the command displays only file2 and file3:

```
$ ls -l file[2-3]
```

The files file2 and file3 are the only files whose names match the specified pattern, which requires that the last character of the filename be in the range 2-3.

You can use more than one metacharacter in a single argument. For example, consider the following command:

```
ls -l file??
```

Most commands let you specify multiple arguments. If no files match a given argument, the command ignores the argument. Here's another command that reports all four files:

```
ls -l file0* file[1-3]
```

Suppose that a command has one or more arguments that include one or more metacharacters. If none of the arguments match any filenames, the shell passes the arguments to the program with the metacharacters intact. When the program expects a valid filename, an unexpected error may result.

Another metacharacter lets you easily refer to your home directory. For example, the following command lists the files in the user's home directory:

```
ls ~
```

SUMMARY

This chapter started with an explanation of file permissions for files and directories, and also how to set them according to your requirements. Then you learned about an assortment of file-related commands, including the commands touch, mv, cp, and rm.

Then you saw some bash commands for managing directories and their contents. Finally, you learned more details about metacharacters and variables that you can use when working with files and shell scripts.

CHAPTER 3

USEFUL COMMANDS

This chapter discusses `bash` commands for manipulating the contents of text files, as well as searching for strings in text files using the `bash` "pipe" command that redirects the output of one `bash` command as the input of a second `bash` command.

The first part of this chapter shows you how to merge, fold, and split text files. This section also shows you how to sort files and find unique lines in files using the `sort` and `uniq` commands, respectively. The last part explains how to compare text files and binary files.

The second section introduces you to the `find` command, which is a powerful command that supports many options. For example, you can search for files in the current directory or in sub-directories; you can search for files based on their creation date and last modification date. One convenient combination is to "pipe" the output of the `find` command to the `xargs` command in order to search files for a particular pattern. Next, you will see how to use the `tr` command, a tool that handles a lot of commonly used text transformations such as capitalization or removal of whitespace. After the section that discusses the `tr` command, you will see a use case that shows you how to use the `tr` command in order to remove the `^M` control character from a dataset.

The third section contains compression-related commands, such as `cpio`, `tar`, and `bash` commands for managing files that are already compressed (such as `zdiff, zcmp, zmore,` and so forth).

The fourth section introduces you to the `IFS` option, which is useful when you need to specify a non-default field separator while extracting data from a range of columns in a dataset. You will also see how to use the `xargs` command in order to "line up" the columns of a dataset so that all rows have the same number of columns.

The fifth section shows you how to create shell scripts, which contain `bash` commands that are executed sequentially.

THE JOIN COMMAND

The `join` command allows you to merge two files in a meaningful fashion, which essentially creates a simple version of a relational database.

The `join` command operates on two files, and pastes together only those lines with a common tagged field (usually a numerical label), and writes the result to `stdout`. The files to be joined should be sorted according to the tagged field for the matchups to work properly. Listing 3.1 and Listing 3.2 displays the contents of `1.data` and `2.data`, respectively.

Listing 3.1: 1.data

```
100 Shoes
200 Laces
300 Socks
```

Listing 3.2: 2.data

```
100 $40.00
200 $1.00
300 $2.00
```

Now launch the following command:

```
join 1.data 2.data
```

File: 1.data 2.data

```
100 Shoes $40.00
200 Laces $1.00
300 Socks $2.00
```

THE FOLD COMMAND

As you know from Chapter 1, the `fold` command enables you to display a set of lines with fixed column width, and this section contains a few more examples. Note that this command does not take into account spaces between words: the output is displayed in columns that resemble a "newspaper" style.

The following command displays a set of lines with ten characters in each line:

```
x="aa bb cc d e f g h i j kk ll mm nn"
echo $x |fold -10
```

The output of the preceding code snippet is here:

```
aa bb cc d
e f g h i
j kk ll m
m nn
```

As another example, consider the following code snippet:

```
x="The quick brown fox jumps over the fat lazy dog. "
echo $x |fold -10
```

The output of the preceding code snippet is here:

```
The quick
brown fox
jumps over
the fat l
azy dog.
```

THE SPLIT COMMAND

The `split` command is useful when you want to create a set of sub-files of a given file. By default, the sub-files are named `xaa`, `xab`, …, `xaz`, `xba`, `xbb`, …, `xbz`, … `xza`, `xzb`, … , `xzz`. Thus, the `split` command creates a maximum of 676 files (=26x26). The default size for each of these files is 1,000 lines.

The following snippet illustrates how to invoke the `split` command in order to split the file `abc.txt` into files with 500 lines each:

```
split -l 500 one-dl-course-outline.txt
```

If the file `abc.txt` contains between 501 and 1,000 lines, then the preceding command will create the following pair of files:

```
xaa
xab
```

You can also specify a file prefix for the created files, as shown here:

```
split -l 500 one-dl-course-outline.txt shorter
```

The preceding command creates the following pair of files:

```
shorterxaa
shorterxab
```

THE SORT COMMAND

The `sort` command sorts the lines in a text file. For example, suppose you have a text file called `test2.txt` that contains the following lines:

```
aa
cc
bb
```

Both of the following commands will sort the lines in `test2.txt`:

```
cat test2.txt |sort
sort test2.txt
```

The output of the preceding commands is here:

```
aa
bb
cc
```

The `sort` command arranges lines of text alphabetically by default. Some options for the `sort` command are here:

```
-n Sort numerically (10 will sort after 2), ignore
blanks and tabs
-r Reverse the order of sort
-f Sort upper- and lowercase together
+x Ignore first x fields when sorting
```

You can use the `sort` command to display the files in a directory based on their file size, as shown here:

```
-rw-r--r--  1 ocampesato  staff   11 Apr 06 19:21
outfile.txt
-rw-r--r--  1 ocampesato  staff   12 Apr 06 19:21
output.txt
-rwx------  1 ocampesato  staff   12 Apr 06 19:21
kyrgyzstan.txt
-rwx------  1 ocampesato  staff   25 Apr 06 19:21
apple-care.txt
-rwx------  1 ocampesato  staff  146 Apr 06 19:21
checkin-commands.txt
-rwx------  1 ocampesato  staff  176 Apr 06 19:21 ssl-
instructions.txt
-rwx------  1 ocampesato  staff  417 Apr 06 19:43
iphonemeetup.txt
```

The `sort` command supports many options, some of which are summarized here.

The `sort -r` command sorts the list of files in reverse chronological order. The `sort -n` command sorts on numeric data and `sort -k` command sorts on a field. For example, the following command displays the long listing of the files in a directory that are sorted by their file size:

```
ls -l |sort -k 5
```

The output is here:

```
total 72
-rwx------  1 ocampesato  staff   12 Apr 06 20:46
kyrgyzstan.txt
-rw-r--r--  1 ocampesato  staff   12 Apr 06 20:46
output.txt
```

```
-rw-r--r--  1 ocampesato  staff   14 Apr 06 20:46
outfile.txt
-rwx------  1 ocampesato  staff   25 Apr 06 20:46
apple-care.txt
-rwxr-xr-x  1 ocampesato  staff   90 Apr 06 20:50
testvars.sh
-rwxr-xr-x  1 ocampesato  staff  100 Apr 06 20:50
testvars2.sh
-rwx------  1 ocampesato  staff  146 Apr 06 20:46
checkin-commands.txt
-rwx------  1 ocampesato  staff  176 Apr 06 20:46
ssl-instructions.txt
-rwx------  1 ocampesato  staff  417 Apr 06 20:46
iphonemeetup.txt
```

Notice that the file listing is sorted based on the fifth column, which displays the file size of each file. You can sort the files in a directory and display them from largest to smallest with this command:

```
ls –l |sort –n
```

In addition to sorting lists of files, you can use the `sort` command to sort the contents of a file. For example, suppose that the file `abc2.txt` contains the following:

```
This is line one
This is line two
This is line one
This is line three
Fourth line
Fifth line
The sixth line
The seventh line
```

The following command sorts the contents of `abc2.txt`:

```
sort abc2.txt
```

You can sort the contents of multiple files and redirect the output to another file:

```
sort outfile.txt output.txt > sortedfile.txt
```

An example of combining the commands `sort` and `tail` is shown here:

```
cat abc2.txt |sort |tail -5
```

The preceding command sorts the contents of the file `abc2.txt` and then displays the final five lines:

```
The seventh line
The sixth line
This is line one
```

```
This is line one
This is line three
This is line two
```

As you can see, the preceding output contains two duplicate lines. The next section shows you how to use the `uniq` command in order to remove duplicate lines.

THE `UNIQ` COMMAND

The `uniq` command prints only the unique lines in a sorted text file (i.e., it ignores duplicate lines). As a simple example, suppose the file `test3.txt` contains the following text:

```
abc
def
abc
abc
```

The following command sorts the contents of `test3.txt` and then displays the unique lines:

```
cat test3.txt |sort | uniq
```

The output of the preceding code snippet is here:

```
abc
def
```

HOW TO COMPARE FILES

The `diff` command enables you to compare two text files and the `cmp` command compares two binary files. For example, suppose that the file `output.txt` contains these two lines:

```
Hello
World
```

Suppose that the file `outfile.txt` contains these two lines:

```
goodbye
world
```

Then the output of this command:

```
diff output.txt outfile.txt
```

is shown here:

```
1,2c1,2
< Hello
< World
---
> goodbye
> world
```

Note that the `diff` command performs a case-sensitive text-based comparison, which means that the strings `Hello` and `hello` are treated as different strings.

THE OD COMMAND

The `od` command displays an octal dump of a file, which can be very helpful when you want to see embedded control characters (such as tab characters) that are not normally visible on the screen. This command contains many switches that you can see when you type `man od`.

As a simple example, suppose that the file `abc.txt` contains one line of text with the following three letters, separated by a tab character (which is not visible here) between each pair of letters:

```
a       b       c
```

The following command displays the tab and newline characters in the file `abc.txt`:

```
cat control1.txt |od -tc
```

The preceding command generates the following output:

```
0000000    a  \t   b  \t   c  \n
0000006
```

In the special case of tabs, another way to see them is to use the following `cat` command:

```
cat –t abc.txt
```

The output from the preceding command is here:

```
a^Ib^Ic
```

In Chapter 1, you learned that the `echo` command prints a newline character whereas the `printf` statement does not print a newline character (unless it is explicitly included). You can verify this fact for yourself with this code snippet:

```
echo abcde | od -c
0000000    a   b   c   d   e  \n
0000006
printf abcde | od -c
0000000    a   b   c   d   e
0000005
```

THE TR COMMAND

The `tr` command is a highly versatile command that supports some very useful options. For example, the `tr` command enables you to remove extraneous whitespaces in datasets, inserting blank lines, printing words on separate

lines, and also translating characters from one character set to another character set (i.e., from uppercase to lowercase, and vice versa).

The following command capitalizes every letter in the variable x:

```
x="abc def ghi"
echo $x | tr [a-z] [A-Z]
ABC DEF GHI
```

Here is another way to convert letters from lowercase to uppercase:

```
cat columns4.txt  |  tr '[:lower:]' '[:upper:]'
```

In addition to uppercase and lowercase, you can use the POSIX characters classes in the tr command:

- alnum: alphanumeric characters
- alpha: alphabetic characters
- cntrl: control (non-printable) characters
- digit: numeric characters
- graph: graphic characters
- lower: lower-case alphabetic characters
- print: printable characters
- punct: punctuation characters
- space: whitespace characters
- upper: upper-case characters
- xdigit: hexadecimal characters 0-9 A-F

The following example removes white spaces in the variable x (initialized above):

```
echo $x |tr -ds " " ""
abcdefghi
```

The following command prints each word on a separate line:

```
echo "a b c" | tr -s " " "\012"
a
b
c
```

The following command replaces every comma with a linefeed:

```
echo "a,b,c" | tr -s "," "\n"
a
b
c
```

The following example replaces the linefeed in each line with a blank space, which produces a single line of output:

```
cat test4.txt |tr '\n' ' '
```

The output of the preceding command is here:

```
abc  def  abc  abc
```

The following example removes the linefeed character at the end of each line of text in a text file. As an illustration, Listing 3.3 displays the contents of `abc2.txt`.

Listing 3.3: abc2.txt

```
This is line one
This is line two
This is line three
Fourth line
Fifth line
The sixth line
The seventh line
```

The following code snippet removes the linefeed character in the text file `abc2.txt`:

```
tr -d '\n' < abc2.txt
```

The output of the preceding `tr` code snippet is here:

```
This is line oneThis is line twoThis is line
threeFourth lineFifth lineThe sixth lineThe seventh
line
```

As you can see, the output is missing a blank space between consecutive lines, which we can insert with this command:

```
tr -s '\n' ' ' < abc2.txt
```

The output of the modified version of the `tr` code snippet is here:

```
This is line one This is line two This is line three
Fourth line Fifth line The sixth line The seventh line
```

You can replace the linefeed character with a period "." with this `tr` command:

```
tr -s '\n' '.' < abc2.txt
```

The output of the preceding version of the `tr` code snippet is here:

```
This is line one.This is line two.This is line three.
Fourth line.Fifth line.The sixth line.The seventh line.
```

The `tr` command with the `-s` option works on a one-for-one basis, which means that the sequence '.' has the same effect as the sequence '. '. As a sort of "preview," we can add a blank space after each period '.' by combining the `tr` command with the `sed` command (discussed in Chapter 6), as shown here:

```
tr -s '\n' '.' < abc2.txt | sed 's/\./\. /g'
```

The output of the preceding command is here:

```
This is line one. This is line two. This is line three.
Fourth line. Fifth line. The sixth line. The seventh
line.
```

Think of the preceding sed snippet as follows: "whenever a 'period' is encountered, replace it with a 'dot' followed by a space, and do this for every occurrence of a period."

You can also combine multiple commands using the bash "pipe" symbol. For example, the following command sorts the contents of Listing 3.3, retrieves the "bottom" five lines of text, retrieves the lines of text that are unique, and then converts the text to upper case letters,

```
cat abc2.txt  |sort |tail -5 | uniq | tr [a-z] [A-Z]
```

Here is the output from the preceding command:

```
THE SEVENTH LINE
THE SIXTH LINE
THIS IS LINE ONE
THIS IS LINE THREE
THIS IS LINE TWO
```

You can also convert the first letter of a word to uppercase (or to lowercase) with the tr command, as shown here:

```
x="pizza"
x=`echo ${x:0:1} | tr  '[a-z]' '[A-Z]'`${x:1}
echo $x
```

A slightly longer (one extra line of code) way to convert the first letter to uppercase is shown here:

```
x="pizza"
first=`echo $x|cut -c1|tr [a-z] [A-Z]`
second=`echo $x|cut -c2-`
echo $first$second
```

As you can see, it's possible to combine multiple commands using the bash pipe symbol "|" in order to produce the desired output.

A SIMPLE USE CASE

The code sample in this section shows you how to use the tr command in order to replace the control character "^M" with a linefeed. Listing 3.4 displays the contents of the dataset controlm.csv that contains embedded control characters.

Listing 3.4 controlm.csv

```
IDN,TEST,WEEK_MINUS1,WEEK0,WEEK1,WEEK2,WEEK3,WEEK4,
WEEK10,WEEK12,WEEK14,WEEK15,WEEK17,WEEK18,WEEK19,
WEEK21^M1,BASO,,1.4,,0.8,,1.2,,1.1,,,2.2,,,1.4^M1,B
ASOAB,,0.05,,0.04,,0.05,,0.04,,,0.07,,,0.05^M1,EOS,,
6.1,,6.2,,7.5,,6.6,,,7.0,,,6.2^M1,EOSAB,,0.22,,0.30
,,0.27,,0.25,,,0.22,,,0.21^M1,HCT,,35.0,,34.2,,
```

```
34.6,,34.3,,,36.2,,,34.1^M1,HGB,,11.8,,11.1,,11.6,,
11.5,,,12.1,,,11.3^M1,LYM,,36.7
```

Listing 3.5 displays the contents of the file `controlm.sh` that illustrates how to remove the control characters from `controlm.csv`.

Listing 3.5 controlm.sh

```
inputfile="controlm.csv"
removectrlmfile="removectrlmfile"
tr -s '\r' '\n' < $inputfile > $removectrlmfile
```

For convenience, Listing 3.5 defines the variable `inputfile` for the input file and the variable `removectrlmfile` for the output file.

The output from launching the shell script in Listing 3.5 is here:

```
IDN,TEST,WEEK_MINUS1,WEEK0,WEEK1,WEEK2,WEEK3,WEEK4,WE
EK10,WEEK12,WEEK14,WEEK15,WEEK17,WEEK18,WEEK19,WEEK21
1,BASO,,1.4,,0.8,,1.2,,1.1,,,2.2,,,1.4
1,BASOAB,,0.05,,0.04,,0.05,,0.04,,,0.07,,,0.05
1,EOS,,6.1,,6.2,,7.5,,6.6,,,7.0,,,6.2
1,EOSAB,,0.22,,0.30,,0.27,,0.25,,,0.22,,,0.21
```

As you can see, the task in this section is easily solved via the `tr` command. Note that there are empty fields in the preceding output, which means that additional processing is required.

You can also replace the current delimiter "," with a different delimiter, such as a "|" symbol that appears in the following command:

```
cat removectrlmfile |tr -s ',' '|' > pipedfile
```

The resulting output is shown here:

```
IDN|TEST|WEEK_MINUS1|WEEK0|WEEK1|WEEK2|WEEK3|WEEK4|WE
EK10|WEEK12|WEEK14|WEEK15|WEEK17|WEEK18|WEEK19|WEEK21
1|BASO|1.4|0.8|1.2|1.1|2.2|1.4
1|BASOAB|0.05|0.04|0.05|0.04|0.07|0.05
1|EOS|6.1|6.2|7.5|6.6|7.0|6.2
1|EOSAB|0.22|0.30|0.27|0.25|0.22|0.21
```

If you have a text file with multiple delimiters in arbitrary order in multiple files, you can replace those delimiters with a single delimiter via the `sed` command, which is discussed in Chapter 6.

THE `FIND` COMMAND

The `find` command supports many options, including one for printing (displaying) the files returned by the `find` command, and another one for removing the files returned by the `find` command.

In addition, you can specify logical operators such as `-a` (AND) as well as `-o` (OR) in a `find` command. You can also specify switches to find the files

(if any) that were created, accessed, or modified before (or after) a specific date.

Several examples are here:

`find . -print` displays all the files (including sub-directories)

`find . -print |grep "abc"` displays all the files whose names contain the string abc

`find . -print |grep "sh$"` displays all the files whose names have the suffix sh

`find . -depth 2 -print` displays all files of depth at most 2 (including sub-directories)

You can also specify access times pertaining to files. For example, `atime`, `ctime`, and `mtime` refer to the access time, creation time, and modification time of a file.

As another example, the following command finds all the files modified in less than 2 days and prints the record count of each:

```
$ find . -mtime -2 -exec wc -l {} ;
```

You can remove a set of files with the `find` command. For example, you can remove all the files in the current directory tree that have the suffix "m" as follows:

```
find . -name "*m$" -print -exec rm {}
```

NOTE *Be careful when you remove files: run the preceding command without "`exec rm {}`" to review the list of files before deleting them.*

THE `tee` COMMAND

The `tee` command enables you to display output to the screen and also redirect the output to a file at the same time. The `-a` option will append subsequent output to the named file instead of overwriting the file. Here is a simple example:

```
find . -print |xargs grep "sh$" | tee /tmp/blue
```

The preceding code snippet redirects the list of all files in the current directory (and those in any sub-directories) to the `xargs` command, which then searches – and prints – all the lines that end with the string "sh." The result is displayed on the screen and also redirected to the file `/tmp/blue`.

```
find . -print |xargs grep "^abc$" | tee -a /tmp/blue
```

The preceding code snippet also redirects the list of all files in the current directory (and those in any sub-directories) to the `xargs` command, which then searches – and prints – all the lines that contain only the string "`abc`." The result is displayed on the screen and also *appended* to the file `/tmp/blue`.

FILE COMPRESSION COMMANDS

Bash supports various commands for compressing sets of files, including the `tar`, `cpio`, `gzip`, and `gunzip` commands. The following subsections contain simple examples of how to use these commands.

The `tar` command

The `tar` command enables you to compress a set of files in a directory to create a new tar file, uncompress an existing tar file, and also display the contents of a tar file.

The "c" option specifies "create," the "f" option specifies "file" and the "v" option specifies "verbose." For example, the following command creates a compressed file called testing.tar and displays the files that are included in `testing.tar` during the creation of this file:

```
tar cvf testing.tar *.txt
```

The compressed file `testing.tar` contains the files with the suffix `txt` in the current directory and you will see the following output:

```
a apple-care.txt
a checkin-commands.txt
a iphonemeetup.txt
a kyrgyzstan.txt
a outfile.txt
a output.txt
a ssl-instructions.txt
```

The following command extracts the files that are in the `tar` file `testing.tar`:

```
tar xvf testing.tar
```

The following command displays the contents of a tar file without uncompressing its contents:

```
tar tvf testing.tar
```

The preceding command displays the same output as the "`ls -l`" command that displays a long listing.

The "z" option uses `gzip` compression. For example, the following command creates a compressed file called `testing.tar.gz`:

```
tar czvf testing.tar.gz *.txt
```

The `cpio` Command

The `cpio` command provides further compression after you create a `tar` file. For example, the following command creates the file `archive.cpio`:

```
ls file1 file2 file3 | cpio -ov > archive.cpio
```

The "-o" option specifies an output file and the "-v" option specifies verbose, which means that the files are displayed as they are placed in the archive file. The "-I" option specifies input, and the "-d" option specifies "display".

You can combine other commands (such as the `find` command) with the `cpio` command, an example of which is here:

```
find . -name ".sh" | cpio -ov > shell-scripts.cpio
```

You can display the contents of the file `archive.cpio` with the following command:

```
cpio -id < archive.cpio
```

The output of the preceding command is here:

```
file1
file2
file3
1 block
```

The gzip and gunzip Commands

The `gzip` command creates a compressed file. For example, the following command creates the compressed file `filename.gz`:

```
gzip filename
```

Extract the contents of the compressed file `filename.gz` with the `gunzip` command:

```
gunzip filename.gz
```

You can create gzipped tarballs using the following methods:

Method #1:

```
tar -czvvf archive.tar.gz [YOUR-LIST-OF-FILES]
```

Method #2:

```
tar -cavvf archive.tar.gz [YOUR-LIST-OF-FILES]
```

The `-a` option specifies that the compression format should automatically be detected from the extension.

The bunzip2 Command

The `bunzip2` utility uses a compression technique that is similar to `gunzip2`, except that `bunzip2` typically produces smaller (more compressed) files than `gzip`. It comes with all Linux distributions. In order to compress with `bzip2` use:

```
bzip2 filename
ls
filename.bz2
```

The zip Command

The `zip` command is another utility for creating zip files. For example, if you have the files called `file1`, `file2`, and `file3`, then the following command creates the file `file1.zip` that contains these three files:

```
zip file?
```

The `zip` command has useful options (such as `-x` for excluding files), and you can find more information in online tutorials.

COMMANDS FOR ZIP FILES AND BZ FILES

There are various commands for handling zip files, including `zdiff`, `zcmp`, `zmore`, `zless`, `zcat`, `zipgrep`, `zipsplit`, `zipinfo`, `zgrep`, `zfgrep`, and `zegrep`.

Remove the initial "z" or "zip" from these commands to obtain the corresponding "regular" `bash` command.

For example, the `zcat` command is the counterpart to the `cat` command, so you can display the contents of a file in a `.gz` file without manually extracting that file and also without modifying the contents of the `.gz` file. Here is an example:

```
ls test.gz
zcat test.gz
```

Another set of utilities for `bz` files includes `bzcat`, `bzcmp`, `bzdiff`, `bzegrep`, `bzfgrep`, `bzgrep`, `bzless`, and `bzmore`.

Read the online documentation to find out more about these commands.

INTERNAL FIELD SEPARATOR (IFS)

The Internal Field Separator is an important concept in shell scripting that is useful while manipulating text data. An Internal Field Separator (`IFS`) is an environment variable that stores delimiting characters. The `IFS` is the default delimiter string used by a running shell environment.

Consider the case where we need to iterate through words in a string or comma separated values (CSV). Specify `IFS=","` in order to display each substring on a separate line. For example, suppose that a shell script contains the following lines:

```
data="age,gender,street,state"
IFS=$','
for item in $data
do
  echo Item: $item
done
```

The output of the preceding code block is here:

```
Item: age
Item: gender
Item: street
Item: state
```

Note that you can also use the `awk` command (discussed in Chapter 7) to produce the same output.

The next section contains a code sample that relies on the value of `IFS` in order to extract data correctly from a dataset.

DATA FROM A RANGE OF COLUMNS IN A DATASET

Listing 3.6 displays the contents of the dataset `datacolumns1.txt` and Listing 3.7 displays the contents of the shell script `datacolumns1.sh` that illustrates how to extract data from a range of columns from the dataset in Listing 3.6.

Incidentally, this code sample contains a `while` loop, which is one of several types of loops that are available in `bash`, and are discussed in more detail in Chapter 3. As such, this code sample involves "forward referencing": using a bash construct before it's been discussed in detail. However, you are either already familiar with the concept of a `while` loop, or you have an intuitive grasp vis-à-vis its purpose, so you'll be able to understand the code in this code sample.

Listing 3.6 datacolumns1.txt

```
#2345678901234567890123456789O
  1000    Jane      Edwards
  2000    Tom       Smith
  3000    Dave      Del Ray
```

Listing 3.7 datacolumns1.sh

```
# empid: 03-09
# fname: 11-20
# lname: 21-30
IFS=''
inputfile="datacolumns1.txt"

while read line
do
  pound="`echo $line |grep '^#'`"

  if [ x"$pound" == x"" ]
  then
```

```
    echo "line: $line"
    empid=`echo "$line" |cut -c3-9`
    echo "empid: $empid"

    fname=`echo "$line" |cut -c11-19`
    echo "fname: $fname"

    lname=`echo "$line" |cut -c21-29`
    echo "lname: $lname"
    echo "--------------"
  fi
done < $inputfile
```

Listing 3.7 sets the value of `IFS` to an empty string, which is required for this shell script to work correctly (try running this script without setting `IFS` and see what happens). The body of this script contains a `while` loop that reads each line from the input file called `datacolumns1.txt` and sets the pound variable equal to "v" if a line does not start with the "#" character OR sets the pound variable equal to the entire line if it *does* start with the "#" character. This is a simple technique for "filtering" lines based on their initial character.

The `if` statement executes for lines that do not start with a "#" character, and the variables `empid`, `fname`, and `lname` are initialized to the characters in Columns 3 through 9, then 11 through 19, and then 21 through 29, respectively. The values of those three variables are printed each time they are initialized. As you can see, these variables are initialized by a combination of the echo command and the cut command, and the value of `IFS` is required in order to ensure that the `echo` command does not remove blank spaces.

The output from Listing 3.7 is shown below:

```
line:   1000    Jane      Edwards
empid: 1000
fname: Jane
lname: Edwards
--------------
line:   2000    Tom       Smith
empid: 2000
fname: Tom
lname: Smith
--------------
line:   3000    Dave      Del Ray
empid: 3000
fname: Dave
lname: Del Ray
--------------
```

WORKING WITH UNEVEN ROWS IN DATASETS

Listing 3.8 displays the contents of the dataset `uneven.txt` that contains rows with a different number of columns. Listing 3.9 displays the contents of the `bash` script `uneven.sh` that illustrates how to generate a dataset whose rows have the same number of columns.

Listing 3.8: uneven.txt

```
abc1 abc2 abc3 abc4
abc5 abc6
abc1 abc2 abc3 abc4
abc5 abc6
```

Listing 3.9: uneven.sh

```
inputfile="uneven.txt"
outputfile="even2.txt"

# ==> four fields per line

#method #1: four fields per line
cat $inputfile | xargs -n 4 >$outputfile

#method #2: two equal rows
#xargs -L 2 <$inputfile > $outputfile

echo "input file:"
cat $inputfile

echo "output file:"
cat $outputfile
```

Listing 3.9 contains two techniques for realigning the text in the input file so that the output appears with four columns in each row. As you can see, both techniques involve the `xargs` command (which is an interesting use of the `xargs` command).

Launch the code in Listing 3.9 and you will see the following output:

```
abc1 abc2 abc3 abc4
abc5 abc6 abc1 abc2
abc3 abc4 abc5 abc6
```

SUMMARY

This chapter showed you examples of how to use some useful and versatile `bash` commands. First, you learned about the bash commands `join`, `fold`, `split`, `sort`, and `uniq`. Next, you learned about the `find` command and

the `xargs` command. You also learned about various ways to use the `tr` command, which is also in the use case in this chapter.

Then you saw some compression-related commands, such as `cpio` and `tar`, which help you create new compressed files and also help you examine the contents of compressed files.

In addition, you learned how to extract column ranges of data, as well as the usefulness of the `IFS` option.

CHAPTER 4

CONDITIONAL LOGIC AND LOOPS

This chapter introduces you to operators (for numeric data and string variables), conditional logic (`if/else/fi`), and several types of loops (`for`, `while`, and `until`) in `bash`.

The first part of this chapter shows you how to perform arithmetic operations and the operators that are available for doing so. You will also see how to assign values to variables, and then how to read user input in a shell script.

The second portion of this chapter shows you how to use the `test` command for variables, files, and directories (such as determining if two variables are equal). You will learn how to use various relational, Boolean, and string operators.

The third section introduces conditional logic (`if/else/fi`), the `case/esac` switch statement, along with `for` loops, nested `for` loops, `while` loops, and `until` loops. You will also learn how to define your own custom functions in shell scripts.

The final section shows you how to work with arrays in `bash`, which includes examples of iterating through array elements and updating their contents.

QUICK OVERVIEW OF OPERATORS IN BASH

The `bash` shell supports the following operators, each of which is discussed in greater detail in this chapter:

Arithmetic Operators
String Operators
File Test Operators
Boolean Operators

The `expr` command is often used to perform arithmetic operations (add, subtract, multiply, or divide) on numeric values.

Arithmetic operators enable you to compare pairs of numbers and pairs of strings. The operators for comparing numbers include `-eq`, `-lt`, and `-gt` for comparing equality, less than and greater than, respectively. On the other hand, string operators for comparing strings include `==`, `<`, and `>` for comparing equality, less than and greater than, respectively.

Some file-related operators that are available in `bash` include `-f`, `-d`, `-e` for checking if a filename is a file, a directory, or checking for the existence of a file, respectively.

Boolean operators (also called logical operators) in `bash` are `-a`, `-o`, `!` for the AND operation, OR operation, and negation, respectively.

ARITHMETIC OPERATIONS AND OPERATORS

Arithmetic operators include `+`, `-`, `*`, and `/`, in order to add, subtract, divide and multiply two numbers, respectively. The % operator is the modulus operator, which returns the remainder of the division of two numbers. Use the `=` operator to assign a value to an operand, `==` to test if two operands are equal, and `!=` to test if two operands are unequal.

Arithmetic in POSIX shells is performed with `$` and double parentheses, as shown here:

```
echo "$(($num1+$num2))"
```

In addition, you can use command substitution to assign the result of an arithmetic operation to a variable:

```
num1=3
num2=5
x='echo "$(($num1+$num2))"'
```

A simpler alternative to the preceding code snippet involves the `expr` command, which is discussed in the next section.

The expr Command

The previous section shows you how to add two numbers using double parentheses; another technique uses the `expr` command, as shown here:

```
expr $num1 + $num2
```

As you probably expect, the `expr` command supports arithmetic operations involving hard-coded numbers, as shown in the following example that adds two numbers:

```
sum=`expr 2 + 2`
echo "The sum: $sum"
```

This would produce the following result:

```
The sum: 4
```

Keep in mind that spaces are required between operators and expressions (so `2+2` is incorrect), and expressions must be inside "backtick" characters (also called inverted commas).

An interesting use of the `expr` command is for finding the length of a string, as shown here:

```
x="abc"
echo `expr "$x" : '.*'`
3
echo ${#x}
3
echo `expr "$x" : '.*'`
3
```

Arithmetic Operators

The `bash` shell supports the arithmetic operations addition, subtraction, multiplication, and division via the operators `+`, `-`, `*`, and `/`, respectively. The following example illustrates these operations.

```
x=15
y=4

sum=`expr $x + $y`
diff=`expr $x - $y`
prod=`expr $x \* $y`
div=`expr $x / $y`
mod=`expr $x % $y`

echo "sum = $sum"
echo "difference = $diff"
echo "product    = $prod"
echo "quotient   = $div"
echo "modulus    = $mod"
```

Here are some examples (assume that `x` and `y` have numeric values) of the equality ("==") and inequality ("!=") operators:

`[ $x == $y ]` returns false
`[ $x != $y ]` returns true

Note the required spaces in the preceding expressions. All arithmetic calculations are done using long integers.

Boolean and Numeric Operators

Bash supports relational operators that are specific to numeric values: they will not work correctly for string values unless their value is numeric. Here is a list of some common operators:

```
$a -eq $b checks if $a and $b are equal
$a -ne $b checks if $a and $b are unequal
$a -gt $b checks if $a is larger than $b
$a -lt $b checks if $a is smaller than $b
```

```
$a -ge $b checks if $a is larger or equal to $b
$a -ge $b checks if $a is smaller than or equal to $b
```

Note that the preceding expressions are written inside a pair of square brackets with spaces on both sides: `[ $a -eq $b ]`, `[ $a -ne $b ]`, and so forth.

Compound Operators and Numeric Operators

Suppose that `a` equals `5` and variable `b` equals `15` in the following examples:

`[ ! false ]` is true

-o If one operand is true then the condition is true:

`[ $a -lt 20 -o $b -gt 100 ]` is true

-a If both operands are true then the condition is true (otherwise it is false):

`[ $a -lt 20 -a $b -gt 100 ]` is false

`!` The not operator reverses the value of the condition

`\( ... \)` Group expressions by enclosing them within `\(` and `\)`

Logical conditions and other tests are usually enclosed in square brackets `[]`. Note that there is a space between square brackets and operands. It will show an error if no space is provided. An example of a valid syntax is as follows:

```
[ $var -eq 0 ]
```

Performing arithmetic conditions on variables or values can be done as follows:

```
[ $var -eq 0 ]  # true when $var equal to 0
[ $var -ne 0 ]  # true when $var differs from 0
```

You can also combine the preceding operators with `-a` ("`AND`") or `-o` ("`OR`") to specify compound test conditions, as shown here:

```
[ $var1 -ne 0 -a $var2 -gt 2 ]
[ $var1 -ne 0 -o $var2 -gt 2 ]
```

The `test` command performs condition checks and also reduces the number of brackets. The same set of test conditions enclosed within `[]` can be used for the `test` command, as shown here:

```
if  [ $var -eq 0 ]; then echo "True"; fi
```

can be written as

```
if  test $var -eq 0 ; then echo "True"; fi
```

WORKING WITH VARIABLES

You already saw some example of variables in `bash`, and this section provides information about how to assign values to variables. You will also see how to use conditional logic to test the values of variables.

Always remember that `bash` variables do not have any type-related information, which means that no distinction is made between a number and a

string (similar to JavaScript). However, you will get an error message if you attempt to perform arithmetic operations on non-numeric values in `bash` (which is not always the case in JavaScript).

Assigning Values to Variables

This section contains some simple examples of assigning values to variables with double quotes and single quotes:

```
x="abc"
y="123"
echo "x = $x and y = ${y}"
echo "xy = $x$y"
echo "double and single quotes: $x" '$x'
```

The preceding code block results in the following output:

```
x = abc and y = 123
xy = abc123
double and single quotes: abc $x
```

Make sure that you do not insert any whitespace between a variable and its value. For example, if you type the following command:

```
z = "abc"
```

You will see the following output:

```
-bash: z: command not found
```

On the other hand, you can insert whitespace between text strings and variables in the `echo` command, as you saw in the previous code block.

One more thing to keep in mind: the following syntax is invalid because the variable `y` is preceded by the `$` symbol:

```
$y=3
-bash: =3: command not found
```

Listing 4.1 displays the contents of `variable-operations.sh` that illustrates how to assign variables with different values and how to update them.

Listing 4.1: variable-operations.sh

```
#length of myvar:
myvar=123456789101112
echo ${#myvar}

#print last 5 characters of myvar:
echo ${myvar: -5}

#10 if myvar was not assigned
echo ${myvar:-10}

#last 10 symbols of myvar
```

```
echo ${myvar: -10}

#substitute part of string with echo:
echo ${myvar//123/999}

#add integers a to b and assign to c:
a=5
b=7
c=$((a+b))
echo "a: $a b: $b c: $c"

# other ways to calculate c:
c='expr $a + $b'
echo "c: $c"
c='echo "$a+$b"|bc'
echo "c: $c"
Launch the code in Listing 4.1 and you will see the
following output:
15
01112
123456789101112
6789101112
999456789101112
a: 5 b: 7 c: 12
```

The `read` Command for User Input

The following statement is the syntax for reading characters from input into the variable `myvar`:

```
read -n number_of_chars myvar
```

For example, the following code snippet reads two characters from the command line (in the form of user input) and then displays those two characters:

```
$ read -n 2 var
echo "var: $var"
```

Various other options are possible with the `read` command. For example, the following command reads a password in non-echoed mode:

```
read -s var  Display a message with read using:
read -p "Enter input:"  var
```

BOOLEAN OPERATORS AND STRING OPERATORS

There are various operators in `bash` for testing string variables and combining those operators with Boolean operators. Suppose that the variables `x` and `y` have the values "`abc`" and `"efg"`, respectively:

`[ $x = $y ]` is false
`[$x != $y ]` is true
`[ -z $x ]` is false because $a has non-zero length
`[ -n $x ]` is true because $a has non-zero length
`[ $x ]` is false because $x is a non-empty string

You can also combine the preceding operators to form compound statements, similar to the compound statements in the previous section.

Keep in mind that the "==" operator is for string comparisons, whereas "`-eq`" is for numeric tests and numeric comparisons. You can also determine whether or not a string has non-zero length, as shown here:

```
-n s1 String s1 has nonzero length
-z s1 String s1 has zero length
```

When you perform string comparison, use double square brackets because single brackets can sometimes lead to errors.

Two strings can be compared to determine whether they are the same as follows:

`[[ $str1 = $str2 ]]`: true when str1 equals str2
`[[ $str1 == $str2 ]]`: alternative method for string equality check

We can check whether two strings are not the same as follows:

`[[ $str1 != $str2 ]]`: true when str1 and str2 mismatches

We can find out the alphabetically smaller or larger string as follows:

`[[ $str1 > $str2 ]]`: true when str1 is alphabetically greater than str2
`[[ $str1 < $str2 ]]`: true when str1 is alphabetically lesser than str2
`[[ -z $str1 ]]`: true if str1 holds an empty string
`[[ -n $str1 ]]`: true if str1 holds a non-empty string

Compound Operators and String Operators

Combine multiple string-related conditions using the logical operators `&&` and `||` as follows:

```
if [[ -n $str1 ]] && [[ -z $str2 ]] ;
then
   commands;
fi
```

For example:

```
str1="Not empty "
str2=""
if [[ -n $str1 ]] && [[ -z $str2 ]];
then
   echo str1 is non-empty and str2 is empty string.
fi
```

The output is as follows:

```
str1 is non-empty and str2 is empty string.
```

Sometimes you will see `bash` scripts (such as installation-related shell scripts) that contain compound expressions to perform multiple operations. In particular, the `&&` operator is used to "connect" multiple commands that executed sequentially (in a left-to-right fashion). Each command in the sequence is executed only if all the preceding commands in the sequence executed successfully. If the current command (in a sequence) does not execute successfully, the remaining commands (if any) that appear on the right-side of the failed command will not be executed.

For example, the following code block uses the `&&` operator to first create a directory, then `cd` into that directory, and then display a message:

```
OLDDIR='pwd'
cd /tmp
CURRDIR='pwd'
echo "current directory: $CURRDIR"
mydir="/tmp/abc/def"
mkdir -p $mydir && cd $mydir && echo "now inside
$mydir"
newdir='pwd'
echo "new directory: $newdir"
echo "new directory: 'pwd'"
cd $OLDDIR
echo "current directory: 'pwd'"
```

At this point you are familiar with all the `bash` commands in the preceding code block: try to predict the output before you launch the preceding code block (were you correct)?

FILE TEST OPERATORS

Bash shell supports numerous operators to test various properties of files. Suppose that the variable `file` is a non-empty text file that has read, write, and execute permissions:

```
-b file Checks if file is a block special file
-c file Checks if file is a character special file
-d file Checks if file is a directory
-e file Checks if file exists
-f file Checks if file is an ordinary file
-g file Checks if file has its set group ID (SGID) bit set
-k file Checks if file has its sticky bit set
-p file Checks if file is a named pipe
-t file Checks if file descriptor is open and associated
with a terminal
```

```
-u file Checks if file has its set user id (SUID) bit set
-r file Checks if file is readable
-w file Checks if file is writeable
-x file Checks if file is executable
-s file Checks if file has size greater than 0
-e file Checks if file exists
f1 -nt f2 File f1 is newer than file f2
f1 -ot f2 File f1 is older than file f2
```

An example of testing for the existence of a file with the `-e` option is shown here:

```
fpath="/etc/passwd"
if [ -e $fpath ]; then
   echo File exists;
else
   echo Does not exist;
fi
```

Compound Operators and File Operators

Combine Boolean operators and file-related operators with the `&&` ("AND") operator or the `||` ("OR") operators. The following example checks if a file exists and also if it has write permissions:

```
fpath="/tmp/somedata"

if [ -e $fpath ] && [ -w $fpath ]
then
   echo "File $fpath exists and is writable"
else
  if [ ! -e $fpath ]
  then
    echo "File $fpath does not exist "
  else
    echo "File $fpath exists but is not writable"
  fi
fi
```

Notice the use of the `&&` operator in the first `if` statement in the preceding code block. The following syntax is incorrect because there are two consecutive operators and the `bash` shell will not interpret the syntax correctly:

```
if [ -e $fpath -a -w $fpath ]
```

Notice that compound operators with string operators also use the `&&` or the || operators:

```
if [[ -n $str1 ]] && [[ -z $str2 ]] ;
```

However, compound operators with numeric operators do not require the && or the || operators, as shown here:

```
[ $a -lt 20 -a $b -gt 100 ]
```

CONDITIONAL LOGIC WITH IF/ELSE/FI STATEMENTS

Bash supports conditional logic, but with a slightly different syntax from other programming languages. The following example shows you how to use an if/else/if statement in bash that prints one message if the variable x (which is initialized with the value 25) is less than 30 and a different message if the value of x is not less than 30:

```
x=25
if [ $x -lt 30 ]
then
  echo "x is less than 30"
else
  echo "x is at least 30"
fi
```

Listing 4.2 displays the contents of the shell script testvars.sh that checks if the variable x is defined.

Listing 4.2: testvars.sh

```
x="abc"

if [ -n "$x" ]
then
  echo "x is defined: $x"
else
  echo "x is not defined"
fi
```

Listing 4.2 initializes the variable x with the value abc, and then uses the if/else/fi construct to determine whether or not x is initialized, and also print an appropriate message. Launch the shell script in Listing 4.2 and you will see the following output:

```
x is defined: abc
```

Listing 4.3 displays the contents of the shell script testvars2.sh that checks if the variable y is undefined.

Listing 4.3: testvars2.sh

```
if [ -z "$y" ]
then
  y="def"
```

```
  echo "y is defined: $y"
else
  echo "y is defined: $y"
fi
```

Listing 4.3 first checks whether or not the variable y defined, and since it is not defined, the following two statements are executed in order to initialize y and then print a message:

```
y="def"
echo "y is defined: $y"
```

Launch the shell script in Listing 4.3 and you will see the following output:

```
y is defined: def
```

THE CASE/ESAC STATEMENT

The case/esac statement is the counterpart to a switch statement in other programming languages. This statement allows you to test various conditions that can include metacharacters. A common scenario involves testing user input: you can check if users entered a string that starts with an upper case or lower case "n" (for no) as well as "y" (for yes).

Listing 4.4 displays the contents of case1.sh that checks various conditions in a case/esac statement.

Listing 4.4: case1.sh

```
x="abc"

case $x in
  a) echo "x is an a" ;;
  c) echo "x is a c" ;;
  a*) echo "x starts with a" ;;
  *) echo "no matches occurred" ;;
esac
```

Listing 4.4 starts by initializing the variable x with the value abc, followed by the case keyword which checks various conditions. As you can see, x matches the third condition, which is true because the value of x starts with the letter a. Now launch the shell script in Listing 4.4 and you will see the following output:

```
x starts with a
```

Listing 4.5 shows you how to prompt users for an input string and then process that input via a case/esac statement.

Listing 4.5: UserInfo.sh

```
echo -n "Please enter your first name: "
read fname
echo -n "Please enter your last name: "
read lname
echo -n "Please enter your city: "
read city

fullname="$fname $lname"
echo "$fullname lives in $city"

case $city in
  San*) echo "$fullname lives in California " ;;
  Chicago) echo "$fullname lives in the Windy City "
;;
  *) echo "$fname lives in la-la land " ;;
esac
```

Listing 4.5 starts by prompting users for their first name, last name and city and then assigning those values to the variables `fname`, `lname` and `city`, respectively. Next, the variable `fullname` is defined as the concatenation of the values of `fname` and `lname`.

The next portion of Listing 4.5 is the `case` keyword that checks if the `city` variable starts with the string `San` or if the `city` variable equals `Chicago`. The third option is the default option, which is true if both of the preceding conditions are false.

Listing 4.6 displays the contents of `StartChar.sh` that checks the type of the first character of a user-provided string.

Listing 4.6: StartChar.sh

```
while (true)
do
  echo -n "Enter a string: "
  read var

  case ${var:0:1} in
  [0-9]*) echo "$var starts with a digit" ;;
  [A-Z]*) echo "$var starts with an uppercase letter"
;;
  [a-z]*) echo "$var starts with a lowercase letter"
;;
       *) echo "$var starts with another symbol" ;;
  esac
done
```

Listing 4.6 starts by prompting users for a string and then initializes the variable `var` with that input string.

The next portion of Listing 4.6 is the `case` keyword that checks if the variable `var` starts with 0 or more digits, upper case letters, or lower case letters, and then displays an appropriate message. The default condition is executed if none of the preceding conditions is true.

Listing 4.7 displays the contents of `StartChar2.sh` that checks the type of the first pair of characters of a user-provided string.

Listing 4.7: StartChar2.sh

```
while (true)
do
  echo -n "Enter a string: "
  read var

  case ${var:0:2} in
  [0-9][0-9]) echo "$var starts with to digit" ;;
  [A-Z][A-Z]) echo "$var starts with two uppercase
letters" ;;
  [a-z][a-z]) echo "$var starts with two lowercase
letters" ;;
       *) echo "$var starts with another pattern" ;;
  esac
done
```

Listing 4.7 starts with a `while` loop whose contents are identical to the contents of Listing 4.6, and you can read the preceding section for an explanation of the code. The only difference is that this code sample repeats indefinitely, and you can press `ctrl-c` to terminate the execution of the shell script.

Listing 4.8 displays the contents of `StartChar3.sh` that checks the type of the first character of a user-provided string.

Listing 4.8: StartChar3.sh

```
while (true)
do
  echo -n "Enter a string: "
  read var

   case ${var:0:1} in
     [0-9]*) echo "$var starts with a digit" ;;
     [[:upper:]]) echo "$var starts with a uppercase
letter" ;;
     [[:lower:]]) echo "$var starts with a lowercase
letter" ;;
```

```
    *)          echo "$var starts with another
symbol" ;;
   esac
done
```

Listing 4.8 also starts with a `while` loop that contains a `case/esac` statement. However, in this example, the conditions involve zero or more digits, upper case letters, and lower case letters. In addition, this code sample also repeats indefinitely, and you can press `ctrl-c` to terminate the execution of the shell script.

WORKING WITH STRINGS IN SHELL SCRIPTS

Notice the "curly brackets" syntax in the second code snippet. Listing 4.9 displays the contents of `substrings.sh` that illustrates examples of the "curly brackets" syntax in order to find substrings of a given string.

Listing 4.9: substrings.sh

```
x="abcdefghij"
echo ${x:0}
echo ${x:1}
echo ${x:5}
echo ${x:7:3}

echo ${x:-4}
echo ${x:(-4)}
echo ${x: -4}
```

The output from the preceding echo statements is here:

```
abcdefghij
bcdefghij
fghij
hij
abcdefghij
ghij
ghij
```

Listing 4.9 initializes the variable `x` as `abcdefghij`, followed by three `echo` statements that display substrings of the variable `x`, starting from index 0, 1, and 5, respectively. The fourth `echo` statement displays the substring of `x` that starts from column 7 and has length 3.

The next portion of Listing 4.9 contains three `echo` statements that specify negative values for column positions, which means that the index position is calculated in a right-to-left fashion. For example, the expression `${x: -4}` refers to the right-most 4 characters in the variable x, which is equal to `ghij`. However, the expression `${x:-4}`

is the entire string value of x because there is a missing whitespace. Finally, the expression `${x:(-4)}` is interpreted as we expect, which is to say that it is also equal to `ghij`.

The next portion of this chapter discusses constructs such as `for, while, and until` statements that you can use in shell scripts.

WORKING WITH LOOPS

The `bash` shell supports various constructs that enable you to iterate through a set of values, such as an array, a list of file names, and so forth. Several loop constructs are available in bash, including a `for` loop construct, a `while` loop, and an `until` loop. The following subsections illustrate how you can use each of these constructs.

Using a `for` loop

The bash shell supports the `for` loop whose syntax is slightly different from other languages (such as JavaScript and Java).

The following code block shows you how to use a `for` loop in order to iterate through a set of files in the current directory.

```
for f in `ls *txt`
do
  echo "file: $f"
done
```

The output of the preceding `for` loop depends on the files with the suffix `txt` that are in the directory where you launch the `for` loop (if there aren't any such files, then the `for` loop does nothing).

Listing 4.10 displays the contents of `renamefiles.sh2` that illustrates how to rename a set of files in a directory.

Listing 4.10: renamefiles.sh2

```
newsuffix="txt"

for f in `ls *sh`
do
  base='echo $f |cut -d"." -f1'
  suffix='echo $f |cut -d"." -f2'

  newfile="${base}.${newsuffix}"
  echo "file: $f new: $newfile"

 #either 'cp' or 'mv' the file
 #mv $f $newfile
 #cp $f $newfile
done
```

Listing 4.10 initializes the variable `newsuffix` with the value `txt`, followed by a `for` loop that iterates through the files whose suffix is `sh`. The variable `f` is the loop variable that is assigned each of those filenames.

The first part of the `for` loop assigns the variables `base` and `suffix` with the initial portion and remaining portion, respectively, of each filename, using a period (".") as a delimiter.

Next, the variable `newfile` is the concatenation of the value of the variable `base` with the value of the variable `newsuffix`, with a period (".") as a delimiter.

The next portion of Listing 4.10 is an `echo` statement that displays the value of the variable `newfile`. You can uncomment either of the last two commands, depending on whether you want to make copies of the existing files or you want to rename the files.

CHECKING FILES IN A DIRECTORY

The code sample in this section contains a `for` loop that checks for various properties of the files in the current directory.

Listing 4.11 displays the contents of `checkdir.sh` that counts the number of directories, executable files, readable files, and ASCII files in the current directory.

Listing 4.11: checkdir.sh

```
#!/bin/bash
# initialize 'counter' variables
TOTAL_FILES=0
ASCII_FILES=0
NONASCII_FILES=0
READABLE_FILES=0
EXEC_FILES=0
DIRECTORIES=0

for f in `ls`
do
  TOTAL_FILES=`expr $TOTAL_FILES + 1`

  if [ -d $f ]
  then
    DIRECTORIES=`expr $DIRECTORIES + 1`
  fi

  if [ -x $f ]
  then
    EXEC_FILES=`expr $EXEC_FILES + 1`
  fi
```

```
  if [ -r $f ]
  then
    READABLE_FILES=`expr $READABLE_FILES + 1`
  fi

  readable=`file $f`
  ascii=`echo $readable |grep ASCII`
  if [ "$ascii" != "" ]
  then
    ASCII_FILES=`expr $ASCII_FILES + 1`
  else
    #echo "readable: $readable"
    NONASCII_FILES=`expr $NONASCII_FILES + 1`
  fi
done

# results:
echo "TOTAL_FILES:     $TOTAL_FILES"
echo "DIRECTORIES:     $DIRECTORIES"
echo "EXEC_FILES:      $EXEC_FILES"
echo "ASCII_FILES:     $ASCII_FILES"
echo "NON-ASCII_FILES: $NONASCII_FILES"
```

Listing 4.11 initializes some count-related variables, followed by a `for` loop that iterates through the files in the current directory. The body of the `for` loop contains multiple conditional code blocks that determine whether or not the current file is a directory, is executable, is readable, and whether or not the current file is an ASCII file.

The last portion of Listing 4.11 displays the values of the count-related variables. Launch the code in Listing 4.11, and you will see something like the following output (the results shown here are for one of my sub-directories):

```
TOTAL_FILES:     33
DIRECTORIES:     1
EXEC_FILES:      26
ASCII_FILES:     29
NON-ASCII_FILES: 4
```

WORKING WITH NESTED LOOPS

This section is mainly for fun: you will see a nested loop to display a "triangular" output. This code sample uses an array with a pair of values: arrays in `bash` are discussed in the final section of this chapter. Listing 4.12 displays the contents of `nestedloops.sh` that illustrates how to display an alternating set of symbols in a triangular fashion.

Listing 4.12: nestedloops2.sh

```
#!/bin/bash

outermax=10
symbols[0]="#"
symbols[1]="@"

for (( i=1; i<${outermax}; i++ ));
do
  for (( j=1; j<${i}; j++ ));
  do
    printf "%-2s" ${symbols[($i+$j)%2]}
  done
  printf "\n"
done

for (( i=1; i<${outermax}; i++ ));
do
  for (( j=${i}+1; j<${outermax}; j++ ));
  do
    printf "%-2s" ${symbols[($i+$j)%2]}
  done
  printf "\n"
done
```

Listing 4.12 initializes some variables, followed by a nested loop. The outer loop is "controlled" by the loop variable `i`, whereas the inner loop (which depends on the value of `i`) is "controlled" by the loop variable `j`. The key point to notice is how the following code snippet prints alternating symbols in the `symbols` array, depending on whether or not the value of `$i + $j` is even or odd:

```
printf "%-2s" ${symbols[($i+$j)%2]}
```

You can easily generalize this code: if the `symbols` array contains `arrlength` elements, then replace the preceding code snippet with the following:

```
printf "%-2s" ${symbols[($i+$j)% $arrlength]}
```

Launch the code in Listing 4.12 and you will see the following output:

```
@
# @
@ # @
# @ # @
@ # @ # @
```

```
# @ # @ # @
@ # @ # @ # @
# @ # @ # @ # @
@ # @ # @ # @ #
@ # @ # @ # @
@ # @ # @ #
@ # @ # @
@ # @ #
@ # @
@ #
@
```

USING A `WHILE` LOOP

Listing 4.13 displays the contents of `while1.sh` that illustrates how to use a `while` loop to iterate through a set of numbers.

Listing 4.13: while1.sh

```
x=0
x=`expr $x + 1`
echo "new x: $x"

while (true)
do
  echo "x = $x"
  x=`expr $x + 1`

  if [ $x -gt 4 ]
  then
    break
  fi
done
```

Listing 4.13 initializes the variable `x` with the value 0 and then increments its value and prints its new value (which is 1).

The next portion of Listing 4.13 is a while loop that displays the value of `x` and then increments its value. Next, an `if` statement checks if the value of `x` is greater than 4; if this is true, then the code "breaks out" of the `while` loop.

Launch the code in Listing 4.13 and you will see the following output:

```
new x: 1
x = 1
x = 2
x = 3
x = 4
```

Listing 4.14 illustrates how to use a `while` loop to iterate through a text file and convert the lines with an even number of words to upper case and the lines with an odd number of words to lower case.

Listing 4.14: upperlowercase.sh

```
infile="wordfile.txt"
outfile="converted.txt"
rm -f $outfile 2>/dev/null

while read line
do
  # word count of current line
  wordcount=`echo "$line" |wc -w`

  modvalue=`expr $wordcount % 2`
  if [ $modvalue = 0 ]
  then
    # even: convert to uppercase
    echo "$line" | tr '[a-z]' '[A-Z]' >> $outfile
  else
    # odd: convert to lowercase
    echo "$line" | tr '[A-Z]' '[a-z]' >> $outfile
  fi
done < $infile
```

Listing 4.14 initializes the variables `infile` and `outfile` with the name of an input file and an output file, respectively, and then invokes the `rm` command in order to unconditionally remove the file `outfile`.

The next portion of Listing 4.14 is a `while` loop that processes one line of input at a time in the input file. Next, the variable `wordcount` is initialized as the number of words in the current line, and the variable `modvalue` is initialized to half the value of `wordcount`.

The next portion of Listing 4.14 is a conditional block that checks whether `modvalue` is even or odd. If `modvalue` is even, the current line is converted to upper case letters and then appended to the output file. If `modvalue` is odd, the current line is converted to lower case and appended to the output file.

Listing 4.15 displays the contents of `wordfile.txt` that is the input file for Listing 4.14.

Listing 4.15: wordfile.txt

```
abc def
def
abc ghi
abc
```

Launch the code in Listing 4.14 and the output creates the file converted.txt whose contents are shown here:

```
ABC DEF
def
ABC GHI
abc
```

THE WHILE, CASE, AND IF/ELIF/ELSE/FI STATEMENTS

Listing 4.16 displays the contents of yesno.sh that illustrates how to combine the while, case/esac, and if/elif/else/fi statements in the same shell script.

Listing 4.16: yesno.sh

```
while(true)
do
  echo -n "Proceed with backup (Y/y/N/n): "
  read response

  case $response in
    n*|N*) proceed="false" ;;
    y*|Y*) proceed="true"  ;;
    *) proceed="unknown"   ;;
  esac

  if [ "$proceed" = "true" ]
  then
    echo "proceeding with backup"
    break
  elif [ "$proceed" = "false" ]
  then
    echo "cancelling backup"
  else
    echo "Invalid response"
  fi
done
```

Listing 4.16 contains a while loop that prompts users to enter an upper case or lower case Y, or an upper case or lower case n. The input value is assigned to the variable response.

The next portion of Listing 4.16 is a case/esac statement that contains regular expressions that are used to assign the value false, true, or unknown to the variable proceed.

The next portion of Listing 4.16 contains an `if/elif/else/fi` code block that prints an appropriate message that depends on the value of the variable `proceed`.

```
Proceed with backup (Y/y/N/n): abc
Invalid response
Proceed with backup (Y/y/N/n):
Invalid response
Proceed with backup (Y/y/N/n): YES
proceeding with backup
```

USING AN `UNTIL` LOOP

The `until` command allows you to execute a series of commands as long as a condition tests false:

```
until command
do
  commands
done
```

Listing 4.17 illustrates how to use an `until` loop to iterate through a set of numbers.

Listing 4.17: until1.sh

```
x="0"
until [ "$x" = "5" ]
do
  x=`expr $x + 1`
  echo "x: $x"
done
```

Listing 4.17 initializes the variable `x` with the value 0 and then enters an `until` loop. The body of the loop increments `x` and prints its value until `x` equals 5, at which point the loop terminates execution. Launch the code in Listing 4.17 and you will see the following output:

```
x: 1
x: 2
x: 3
x: 4
x: 5
```

USER-DEFINED FUNCTIONS

The `bash` shell provides built-in functions and also enables you to define your own functions, which means that you can define custom functions that

are specific to your needs. Here are simple rules to define a function in a shell script:

- Function blocks begin with the keyword function followed by the function name and parentheses (( )).
- The code block in a function starts with the curly brackets "{"and ends with the curly brace brackets"}".

The following code block defines a very simple custom function that contains the `echo` command:

```
Hello()
{
  echo "Hello from a function"
}
```

You can define the `Hello()` function directly from the command line or place the function in a shell script. Execute the function by invoking its name:

```
Hello
```

The output is exactly what you expect:

```
Hello from a function
```

The preceding function definition does nothing interesting. In order to make the function more useful, modify the body of the function as shown here:

```
Hello()
{
  echo "Hello $1 how are you"
}
```

Execute the modified function and also specify a string, as shown here:

```
Hello Bob
```

The output is exactly what you expect:

```
Hello Bob how are you
```

The next example uses conditional logic to check for the existence of a parameter and then prints the appropriate message:

```
Hello()
{
  if [ "$1" != "" ]
  then
    echo "Hello $1"
  else
    echo "Please specify a name"
  fi
}
```

Execute the modified function without specifying a string, as shown here:

```
Hello
```

The output is exactly what you expect:

```
Please specify a name
```

The following example illustrates how to define a function that iterates through all the parameters that are passed to the function:

```
Hello()
{
  while [ "$1" != "" ]
  do
    echo "hello $1"
    shift
  done
}

Hello a b c
```

Place the preceding code in a shell script, and after making the shell script executable, launch the code and you will see the following output:

```
hello a
hello b
hello c
```

CREATING A SIMPLE MENU FROM SHELL COMMANDS

Listing 4.18 displays the contents of `AppendRow.sh` that illustrates how to update a set of users in a text file.

Listing 4.18: AppendRow.sh

```
DataFile="users.txt"

addUser()
{
  echo -n "First Name: "
  read fname

  echo -n "Last Name: "
  read lname

  if [ -n $fname -a -n $lname ]
  then
    # append new line to the file
    echo "$fname $lname" >> $DataFile
  else
```

```
    echo "Please enter non-empty values"
  fi
}

while (true)
do
  echo ""
  echo "List of Users"
  echo "============="
  cat users.txt 2>/dev/null

  echo "----------------------------"
  echo "Enter 'a' to add a new user"
  echo "Enter 'd' to delete all users"
  echo "Enter 'x' to exit this menu"
  echo "----------------------------"
  echo

  read answer
  case $answer in
    a|A) addUser ;;
    d|D) rm $DataFile 2>/dev/null ;;
    x|X) break ;;
  esac
done
```

Listing 4.18 starts by initializing the variable `DataFile` with the value `users.txt`, which will be updated with string values that are supplied by users. Next, the `addUser()` function is defined: keep in mind that this function is executed in the first option of the `case/esac` code block.

The next section is a `while` loop that displays the contents of the file `users.txt` (which is initially empty). Next, a set of `echo` statements prompts users to enter the character `x` to stop the execution of the shell script, or the letter `d` to delete all the names.

If users enter the string `x`, the shell script exits the loop via the `break` statement, which in turn terminates the execution of the shell script. If users enter the string `d`, the contents of the file `users.txt` are deleted. If users enter the string `a`, then the `addUser` function is invoked in order to add a new user. This function prompts for a first name and last name: if both strings are non-empty, the new user is appended to the file `users.txt`; otherwise, an appropriate message is displayed (i.e., a prompt to enter non-empty values for both the first name and the last name).

A sample invocation of Listing 4.18 is here, which has already added three users:

```
List of Users
=============
abc def
123 456
888 777
----------------------------
Enter 'a' to add a new user
Enter 'd' to delete all users
Enter 'x' to exit this menu
----------------------------
```

ARRAYS IN BASH

Arrays are available in many (all?) programming languages, and they are also available in `bash`. Note that a one-dimensional array is known as a vector in mathematics and a two-dimensional array is called a matrix; however, most online code samples use the word array in shell scripts.

Listing 4.19 displays the contents of `Array1.sh` that illustrates how to define an array and access elements in an array

Listing 4.19: Array1.sh

```
# initialize the names array
names[0]="john"
names[1]="nancy"
names[2]="jane"
names[3]="steve"
names[4]="bob"

# display the first and second entries
echo "First Index: ${names[0]}"
echo "Second Index: ${names[1]}"
```

Listing 4.19 defines the `names` array that is initialized with five strings, starting from index 0 through index 4. The two `echo` statements display the first and second elements in the `names` array, which are at index 0 and 1, respectively. The output from Listing 4.19 is here:

```
First Index: john
Second Index: nancy
```

If you need to access all the items in an array, you can do so with either of the following code snippets:

```
${array_name[*]}
${array_name[@]}
```

Listing 4.20 displays the contents of the shell script `loadarray.sh` that initializes an array and then prints its contents.

Listing 4.20: loadarray.sh

```
#!/bin/bash
numbers="1 2 3 4 5 6 7 8 9 10"
array1=( 'echo "$numbers" ')
total1=0
total2=0

for num in "${array1[@]}"
do
 #echo "array item: $num"
  total1+=$num
  let total2+=$num
done

echo "Total1: $total1"
echo "Total2: $total2"
```

Listing 4.20 defines a string variable `numbers` that contains the digits from 1 to 10 inclusive. The `array1` variable is initialized with all the values of the `numbers` array by the `echo` statement that is inside a pair of backticks.

Next, the two numeric variables `total1` and `total2` are initialized to 0, followed by a `for` loop that finds the sum of all the numbers in the `array1` variable. The last pair of `echo` statements display the results. Launch the shell script in Listing 4.20 and the output is as follows:

```
Total1: 012345678910
Total2: 55
```

As you can see, `total1` is the result of appending the elements of the numbers array into a single string, whereas `total2` is the numeric sum of the elements of the `numbers` array. The difference is due to the `let` keyword in the loop.

Listing 4.21 displays the contents of the shell script `update-array.sh` that shows you some operations you can perform on an initialized array.

Listing 4.21: updated-array.sh

```
array=("I" "love" "deep" "dish" "pizza")

#the first array element:
echo ${array[0]}

#all array elements:
echo ${array[@]}

#all array indexes:
echo ${!array[@]}
```

```
#Remove array element at index 3:
unset array[3]

#add new array element with index 1234:
array[1234]="in Chicago"

#all array elements:
echo ${array[@]}
```

Launch the code in Listing 4.21 and you will see the following output:

```
I
I love deep dish pizza
0 1 2 3 4
I love deep pizza in Chicago
```

WORKING WITH ARRAYS

Arrays enable you to "group together" related data elements as rows, and then each row contains logically related data values. As a simple example, the following array defines three fields for a customer (obviously not a complete set of fields):

```
cust[0] = name
cust[1] = Address
cust[2] = phone number
```

Customer records can be saved in a text file that can be read later by a shell script. If you are unfamiliar with text files, there are `CSV` (comma-separated-values) files and `TSV` (tab-separated-values) files, as well as files that have different delimiters, such as a colon ":", a pipe "|" symbol, and so forth. The delimiter is called the `IFS` (Internal Field Separator).

This section contains several shell scripts that illustrate some useful features of arrays in `bash`. Listing 4.22 displays the contents of `fruits-array1.sh` that illustrates how to use an array and some operations that you can perform on arrays.

The syntax in `bash` is different enough from other programming languages that it's worthwhile to see several examples to explore its behavior.

Listing 4.22: fruits-array1.sh

```
#!/bin/bash
# method #1:
fruits[0]="apple"
fruits[1]="banana"
fruits[2]="cherry"
fruits[3]="orange"
```

```
fruits[4]="pear"
echo "first fruit: ${fruits[0]}"

# method #2:
declare -a fruits2=(apple banana cherry orange pear)
echo "first fruit: ${fruits2[0]}"

# range of elements:
echo "last two: ${fruits[@]:3:2}"

# substring of element:
echo "substring: ${fruits[1]:0:3}"

arrlength=${#fruits[@]}
echo "length: ${#fruits[@]}"
```

Launch the code in Listing 4.22 and you will see the following output:

```
first fruit: apple
first fruit: apple
last two: orange pear
substring: ban
length: 5
```

Listing 4.23 displays the contents of `names.txt` and Listing 4.24 displays the contents of `array-from-file.sh` that contains a `for` loop to iterate through the elements of an array whose initial values are based on the contents of `names.txt`.

Listing 4.23: names.txt

```
Jane Smith
John Jones
Dave Edwards
```

Listing 4.24: array-from-file.sh

```
#!/bin/bash

names="names.txt"
contents1=(`cat "$names"`)

echo "First loop:"
for w in "${contents1[@]}"
do
  echo "$w"
done
IFS=""
names="names.txt"
```

```
contents1=(`cat "$names"`)

echo "Second loop:"
for w in "${contents1[@]}"
do
  echo "$w"
done
```

Listing 4.24 initializes the array variable `contents1` by using command substitution with the `cat` command, followed by a `for` loop that displays elements of the array `contents1`. The second `for` loop is the same code as the first `for` loop, but this time with the value of `IFS` equal to "", which has the effect of specifying the newline as a separator. Consequently, the second loop displays two data values per row, which reflects the contents and the same layout `names.txt`.

Launch the code in Listing 4.24 and you will see the following output:

```
First loop:
Jane
Smith
John
Jones
Dave
Edwards
Second loop:
Jane Smith
John Jones
Dave Edwards
```

Listing 4.25 displays the contents of `array-function.sh` that illustrates how to initialize an array and then display its contents in a user-defined function.

Listing 4.25: array-function.sh

```
#!/bin/bash

# compact version of the code later in this script:
#items() { for line in "${@}" ; do printf "%s\n"
"${line}" ; done ; }
#aa=( 7 -4 -e ) ; items "${aa[@]}"

items() {
  for line in "${@}"
  do
    printf "%s\n" "${line}"
  done
}
```

```
arr=( 123 -abc 'my data' )
items "${arr[@]}"
```

Listing 4.25 contains the `items()` function that displays the contents of the `arr` array that has been initialized prior to invoking this function. The output is shown here:

```
123
-abc
my data
```

Listing 4.26 displays the contents of `array-loops1.sh` that illustrates how to determine the length of an initialized array and then display its contents via a `for` loop.

Listing 4.26: array-loops1.sh

```
#!/bin/bash

fruits[0]="apple"
fruits[1]="banana"
fruits[2]="cherry"
fruits[3]="orange"
fruits[4]="pear"


# array length:
arrlength=${#fruits[@]}
echo "length: ${#fruits[@]}"


# print each element via a loop:
for (( i=1; i<${arrlength}+1; i++ ));
do
  echo "element $i of ${arrlength} : " ${fruits[$i-1]}
done
```

Listing 4.26 contains straightforward code for initializing an array and displaying its values. Launch the code in Listing 4.26 and you will see the following output:

```
length: 5
element 1 of 5 :  apple
element 2 of 5 :  banana
element 3 of 5 :  cherry
element 4 of 5 :  orange
element 5 of 5 :  pear
```

SUMMARY

This chapter started with examples of arithmetic operations and the operators that are available for doing so. You then learned how to assign values to variables, and then how to read user input in a shell script.

Next, you saw how to use the `test` command for variables, files, and directories (such as determining if two variables are equal). In addition, you saw examples of various relational, Boolean, and string operators that are available in bash.

Then you learned about conditional logic (`if/elif/fi`), the `case/esac` statement, along with `for` loops, nested loops, and `while` loops. Moreover, you saw how to create your own functions in shell scripts. Finally, you learned how to work with arrays in `bash`, such as initializing arrays and updating their contents in `for` loops.

CHAPTER 5

FILTERING DATA WITH GREP

This chapter introduces you to the versatile `grep` command, whose purpose is to take a stream of text data and reduce it to only the parts that you care about. The `grep` command is useful not only by itself but also in conjunction with other commands, especially the `find` command. This chapter contains many short code samples that illustrate various options of the `grep` command. Some code samples illustrate how to combine the `grep` command with commands from previous chapters.

The first part of this chapter introduces the `grep` command used in isolation, combined with the regular expression metacharacters (from Chapter 2) and also with code snippets that illustrate how to use some of the options of the `grep` command. Next, you will learn how to match ranges of lines, how to use the so-called "back references" in `grep`, and how to "escape" metacharacters in `grep`.

The second part of this chapter shows you how to use the `grep` command in order to find empty lines and common lines in datasets, as well as the use of keys to match rows in datasets. Next, you will learn how to use character classes with the `grep` command, as well as the backslash "\" character, and how to specify multiple matching patterns. Next, you will learn how to combine the `grep` command with the `find` command and the `xargs` command, which is useful for matching a pattern in files that reside in different directories. This section also contains some examples of common mistakes that people make with the `grep` command.

The third section briefly discusses the `egrep` command and the `fgrep` command, which are related commands that provide additional functionality that is unavailable in the standard `grep` utility. The final section contains a use case that illustrates how to use the `grep` command in order to find matching lines that are then merged in order to create a new dataset.

WHAT IS THE GREP COMMAND?

The grep ("Global Regular Expression Print") command is useful for finding substrings in one or more files. Several examples are here:

grep abc *sh displays all the *lines* of abc in files with suffix sh

grep –i abc *sh is the same as the preceding query, but case-insensitive

grep –l abc *sh displays all the *filenames* with suffix sh that contain abc

grep –n abc *sh displays all the *line numbers* of the occurrences of the string abc in files with suffix sh

You can perform logical AND and logical OR operations with this syntax:

grep abc *sh |grep def matches lines containing abc AND def

grep "abc\|def" *sh matches lines containing abc OR def

You can combine switches as well: the following command displays the names of the files that contain the string abc (case insensitive):

```
grep –il abc *sh
```

In other words, the preceding command matches filenames that contain abc, Abc, ABc, ABC, abC, and so forth.

Another (less efficient way) to display the lines containing abc (case insensitive) is here:

```
cat file1 |grep –i abc
```

The preceding command involves two processes, whereas the "grep using –l switch instead of cat to input the files you want" approach involves a single process. The execution time is roughly the same for small text files, but the execution time can become more significant if you are working with multiple large text files.

You can combine the sort command, the pipe symbol, and the grep command. For example, the following command displays the files with a "Jan" date in increasing size:

```
ls -l |grep " Jan " | sort -n
```

A sample output from the preceding command is here:

```
-rw-r--r--  1 oswaldcampesato2  staff      3 Sep 27
2013 abc.txt
-rw-r--r--  1 oswaldcampesato2  staff      6 Sep 21
2013 control1.txt
-rw-r--r--  1 oswaldcampesato2  staff     27 Sep 28
2013 fiblist.txt
-rw-r--r--  1 oswaldcampesato2  staff     28 Sep 14
2013 dest
-rw-r--r--  1 oswaldcampesato2  staff     36 Sep 14
2013 source
```

```
-rw-r--r--  1 oswaldcampesato2  staff    195 Sep 28
2013 Divisors.py
-rw-r--r--  1 oswaldcampesato2  staff    267 Sep 28
2013 Divisors2.py
```

METACHARACTERS AND THE GREP COMMAND

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash.

A regular expression may be followed by one of several repetition operators, as shown below.

"." matches any single character:

"?" indicates that the preceding item is optional and will be matched at most once: Z? matches Z or ZZ.

"*" indicates that the preceding item will be matched zero or more times: Z* matches Z, ZZ, ZZZ, and so forth.

"+" indicates that the preceding item will be matched one or more times: Z+ matches ZZ, ZZZ, and so forth.

"{n}" indicates that the preceding item is matched exactly n times: Z{3} matches ZZZ.

"{n,}" indicates that the preceding item is matched n or more times: Z{3} matches ZZZ, ZZZZ, and so forth.

"{,m}" indicates that the preceding item is matched at most m times: Z{,3} matches Z, ZZ, and ZZZ.

"{n,m}" indicates that the preceding item is matched at least n times, but not more than m times: Z{2,4} matches ZZ, ZZZ, and ZZZZ.

The empty regular expression matches the empty string (i.e., a line in the input stream with no data). Two regular expressions may be joined by the infix operator "|". When used in this manner, the infix operator behaves exactly like a logical "OR" statement, which directs the `grep` command to return any line that matches either regular expression.

ESCAPING METACHARACTERS WITH THE GREP COMMAND

Listing 5.1 displays the contents of `lines.txt` that contains lines with characters and some lines with metacharacters.

Listing 5.1: lines.txt

```
abcd
ab
abc
cd
defg
.*.
..
```

The following `grep` command lists the lines of length 2 (using the `^` begin with and `$` end with operators to restrict length) in `lines.txt`:

```
grep '^..$' lines.txt
```

The following command lists the lines of length two in `lines.txt` that contain two dots (the backslash tells `grep` to interpret the dots as actual dots, not as metacharacters):

```
grep '^\.\.$' lines.txt
```

The result is shown here:

```
ab
cd
..
```

The following command also displays lines of length two that begin and end with a dot (the * matches any text of any length, including no text at all and is used as a metacharacter because it is not preceded with a backslash):

```
grep '^\.*\.$' lines.txt
```

The following command lists the lines that contain a period, followed by an asterisk, and then another period (the * is now a character that must be matched because it is preceded by a backslash):

```
grep '^\.\*\.$' lines.txt
```

USEFUL OPTIONS FOR THE GREP COMMAND

There are many types of pattern matching possibilities with the `grep` command, and this section contains an eclectic mix of such commands that handle common scenarios.

In the following examples, we have four text files (two .sh files and two .txt files) and two Word documents in a directory. The string `abc` is found on one line in `abc1.txt` and three lines in `abc3.sh`. The string `ABC` is found on 2 lines in `ABC2.txt` and 4 lines in `ABC4.sh`. Notice that `abc` is not found in `ABC` files, and `ABC` is not found in `abc` files.

```
ls *

ABC.doc         ABC4.sh         abc1.txt
ABC2.txt        abc.doc         abc3.sh
```

The following code snippet searches for occurrences of the string `abc` in all the files in the current directory that have `sh` as a suffix:

```
grep abc *sh
abc3.sh:abc at start
abc3.sh:ends with -abc
abc3.sh:the abc is in the middle
```

The "`-c`" option counts the number of occurrences of a string (note that even though `ABC4.sh` has no matches, it still counts them and returns zero):

```
grep –c abc *sh
```

The output of the preceding command is here:

```
ABC4.sh:0
abc3.sh:3
```

The "`-e`" option lets you match patterns that would otherwise cause syntax problems (the "–" character normally is interpreted as an argument for `grep`):

```
grep –e "-abc" *sh
abc3.sh:ends with -abc
```

The "`-e`" option also lets you match multiple patterns.

```
grep –e "-abc" -e "comment" *sh

ABC4.sh:# ABC in a comment
abc3.sh:ends with -abc

The "-i" option is to perform a case insensitive
match:
grep –i abc *sh
ABC4.sh:ABC at start
ABC4.sh:ends with ABC
ABC4.sh:the ABC is in the middle
ABC4.sh:# ABC in a comment
abc3.sh:abc at start
abc3.sh:ends with -abc
abc3.sh:the abc is in the middle
```

The "`-v`" option "inverts" the matching string, which means that the output consists of the lines that do not contain the specified string (`ABC` doesn't match because `-i` is not used, and `ABC4.sh` has an entirely empty line):

```
grep –v abc *sh
```

Use the "-iv" options to display the lines that do not contain a specified string using a case insensitive match:

```
grep -iv abc *sh
ABC4.sh:
abc3.sh:this line won't match
```

The "-l" option is to list only the filenames that contain a successful match (note this matches the contents of the files, not the filenames). The Word document matches because the actual text is still visible to grep, it is just surrounded by proprietary formatting gibberish. You can do similar things with other formats that contain text, such as XML, HTML, .csv, and so forth:

```
grep -l abc *

abc1.txt
abc3.sh
abc.doc
```

The "-l" option is to list only the filenames that contain a successful match:

```
grep -l abc *sh
```

Use the "-il" options to display the filenames that contain a specified string using a case insensitive match:

```
grep -il abc *doc
```

The preceding command is very useful when you want to check for the occurrence of a string in Word documents.

The "-n" option specifies line numbers of any matching file:

```
grep -n abc *sh
abc3.sh:1:abc at start
abc3.sh:2:ends with -abc
abc3.sh:3:the abc is in the middle
```

The "-h" option suppresses the display of the filename for a successful match:

```
grep -h abc *sh
abc at start
ends with -abc
the abc is in the middle
```

For the next series of examples, we will use columns4.txt as shown in Listing 5.2.

Listing 5.2: columns4.txt

```
123 ONE TWO
456 three four
ONE TWO THREE FOUR
five 123 six
```

```
one two three
four five
```

The "-o" option shows only the matched string (this is how you avoid returning the entire line that matches):

```
grep –o one columns4.txt
```

The "-o" option followed by the "-b" option shows the position of the matched string (returns character position, not the line number. The "o" in "one" is the 59th character of the file):

```
grep –o –b one columns4.txt
```

You can specify a recursive search as shown here (output not shown because it will be different on every client or account. This searches not only every file in directory `/etc`, but every file in every subdirectory of `etc`):

```
grep –r abc /etc
```

The preceding commands match lines where the specified string is a substring of a longer string in the file. For instance, the preceding commands will match occurrences of `abc` as well as `abcd`, `dabc`, `abcde`, and so forth.

```
grep ABC *txt

ABC2.txt:ABC at start or ABC in middle or end in ABC
ABC2.txt:ABCD DABC
```

If you want to exclude everything except for an exact match, you can use the `–w` option, as shown here:

```
grep –w ABC *txt

ABC2.txt:ABC at start or ABC in middle or end in ABC
The --color switch displays the matching string in
color:
grep --color abc *sh
abc3.sh:abc at start
abc3.sh:ends with -abc
abc3.sh:the abc is in the middle
```

You can use the pair of metacharacters `.*` to find the occurrences of two words that are separated by an arbitrary number of intermediate characters.

The following command finds all lines that contain the strings `one` and `three` with any number of intermediate characters:

```
grep "one.*three" columns4.txt
one two three
```

You can "invert" the preceding result by using the `–v` switch, as shown here:

```
grep –v "one.*three" columns4.txt
123 ONE TWO
456 three four
```

```
ONE TWO THREE FOUR
five 123 six
four five
```

The following command finds all lines that contain the strings `one` and `three` with any number of intermediate characters, where the match involves a case-insensitive comparison:

```
grep -i "one.*three" columns4.txt
ONE TWO THREE FOUR
one two three
```

You can "invert" the preceding result by using the `-v` switch, as shown here:

```
grep -iv "one.*three" columns4.txt
123 ONE TWO
456 three four
five 123 six
four five
```

Sometimes you need to search a file for the presence of either of two strings. For example, the following command finds the files that contain "`start`" or "`end`":

```
grep -l 'start\|end' *
ABC2.txt
ABC4.sh
abc3.sh
```

Later in the chapter, you will see how to find files that contain a pair of strings via the `grep` and `xargs` commands.

Character Classes and the `grep` Command

This section contains some simple one-line commands that combine the `grep` command with character classes.

```
echo "abc" | grep '[:alpha:]'
abc
echo "123" | grep '[:alpha:]'
(returns nothing, no match)
echo "abc123" | grep '[:alpha:]'
abc123
echo "abc" | grep '[:alnum:]'
abc
echo "123" | grep '[:alnum:]'
(returns nothing, no match)
echo "abc123" | grep '[:alnum:]'
abc123
echo "123" | grep '[:alnum:]'
(returns nothing, no match)
```

```
echo "abc123" | grep '[:alnum:]'
abc123
echo "abc" | grep '[0-9]'
(returns nothing, no match)
echo "123" | grep '[0-9]'
123
echo "abc123" | grep '[0-9]'
abc123
echo „abc123" | grep -w ,[0-9]'
(returns nothing, no match)
```

WORKING WITH THE –C OPTION IN GREP

Consider a scenario in which a directory (such as a log directory) has files created by an outside program. Your task is to write a shell script that determines which (if any) of the files contain two occurrences of a string, after which additional processing is performed on the matching files (e.g., use email to send log files containing two or more errors messages to a system administrator, for investigation).

One solution involves the –c option for `grep`, followed by additional invocations of the `grep` command.

The command snippets in this section assume the following data files whose contents are shown below.

The file `hello1.txt` contains the following:

```
hello world1
```

The file `hello2.txt` contains the following:

```
hello world2
hello world2 second time
```

The file `hello3.txt` contains the following:

```
hello world3
hello world3 two
hello world3 three
```

Now launch the following commands: (`2>/dev/null` suppresses warnings and errors caused by empty directories so they do not appear in the output):

```
grep -c hello hello*txt 2>/dev/null
hello1.txt:1
hello2.txt:2
hello3.txt:3
grep -l hello hello*txt 2>/dev/null
hello1.txt
hello2.txt
```

```
hello3.txt
grep -c hello hello*txt 2>/dev/null |grep ":2$"
hello2.txt:2
```

Note how we use the "ends with" `"$"` metacharacter to grab just the files that have exactly two matches. We also use the colon `":2$"` rather than just `"2$"` to prevent grabbing files that have 12, 32 or 142 matches. (which would end in :12, :32 and :142).

What if we wanted to show "two or more" (as in the "2 or more errors in a log")? You would instead use the invert (`-v`) command to exclude counts of exactly 0 or exactly 1.

```
grep -c hello hello*txt 2>/dev/null |grep -v ':[0-1]$'
hello2.txt:2
hello3.txt:3
```

In a real-world application, you would want to strip off everything after the colon to return only the filenames. There are many ways to do so, but we'll use the `cut` command we learned in Chapter 1, which involves defining `:` as a delimiter with `-d":"` and using `-f1` to return the first column (i.e., the part before the colon in the return text):

```
grep -c hello hello*txt 2>/dev/null | grep -v ':
[0-1]$'| cut -d":" -f1
hello2.txt
hello3.txt
```

MATCHING A RANGE OF LINES

In Chapter 1, you saw how to use the `head` and `tail` commands to display a range of lines in a text file. Now suppose that you want to search a range of lines for a string. For instance, the following command displays lines 9 through 15 of `longfile.txt`:

```
cat -n longfile.txt |head -15|tail -9
```

The output is here:

```
   7   and each line
   8   contains
   9   one or
  10   more words
  11   and if you
  12   use the cat
  13   command the
  14   file contents
  15   scroll
```

This command displays the subset of lines 9 through 15 of longfile.txt that contains the string and:

```
cat -n longfile.txt |head -15|tail -9 | grep and
```

The output is here:

```
   7   and each line
  11   and if you
  13   command the
```

This command includes a whitespace after the word and, thereby excluding the line with the word "command":

```
cat -n longfile.txt |head -15|tail -9 | grep "and "
```

The output is here:

```
   7   and each line
  11   and if you
```

Note that the preceding command excludes lines that end in "and" because those lines do not have the whitespace after "and" at the end of the line. You could remedy this situation with an "OR" operator including both cases:

```
cat -n longfile.txt |head -15|tail -9 | grep "
and\|and "
   7   and each line
  11   and if you
  13   command the
```

However, the preceding allows "command' back into the mix. Hence, if you really want to match a specific word it's best to use the -w tag, which is smart enough to handle the variations:

```
cat -n longfile.txt |head -15|tail -9 | grep -w "and"
   7   and each line
  11   and if you
```

The use of whitespace is safer if you are looking for something at the beginning or end of a line. This is a common approach when reading contents of log files or other structured text where the first word is often important (a tag like ERROR or Warning, a numeric code or a date). This command displays the lines that start with the word and:

```
cat longfile.txt |head -15|tail -9 | grep "^and "
```

The output is here (without the line number because we are not using "cat -n"):

```
and each line
and if you
```

Recall that the "use the file name(s) in the command, instead of using `cat` to display the file first" style is more efficient:

```
head -15 longfile.txt |tail -9 | grep "^and "
and each line
and if you
```

However, the `head` command does not display the line numbers of a text file, so the "cat first" (`cat -n` adds line numbers) style is used in the earlier examples when you want to see the line numbers, even though this style is less efficient. Basically, you only want to add an extra command to a pipe if it is adding value; otherwise, it's better to start with a direct call to the files you are trying to process with the first command in the pipe, assuming the command syntax is capable of reading in filenames.

USING BACK REFERENCES IN THE GREP COMMAND

The `grep` command allows you to reference a set of characters that match a regular expression placed inside a pair of parentheses. For `grep` to parse the parentheses correctly, each has to be preceded with the escape character "\".

For example, `grep 'a(\.\)'` uses the "." regular expression to match `ab` or "`a3`" but not "`3a`" or "`ba`".

The back reference '\n', where `n` is a single digit, matches the substring previously matched by the nth parenthesized sub-expression of the regular expression. For example, `grep '\(a\)\1'` matches 'aa' and `grep '\(a\)\2'` matches "`aaa`".

When used with alternation, if the group does not participate in the match then the back reference makes the whole match fail. For example, `grep 'a\(.\)|b\1'` will not match `ba` or `ab` or `bb` (or anything else really).

If you have more than one regular expression inside a pair of parentheses, they are referenced (from left to right) by `\1`, `\2`, …, `\9`:

```
grep -e '\([a-z]\)\([0-9]\)\1' is the same as this
command:
grep -e '\([a-z]\)\([0-9]\)\([a-z]\)'
grep -e '\([a-z]\)\([0-9]\)\2' is the same as this
command:
grep -e '\([a-z]\)\([0-9]\)\([0-9]\)'
```

The easiest way to think of it is that the number (for example, `\2`) is a placeholder or variable that saves you from typing the longer regular expression it references. As regular expressions can get extremely complex, this often helps code clarity.

You can match consecutive digits or characters using the pattern `\([0-9]\)\1`. For example, the following command is a successful match because the string "`1223`" contains a pair of consecutive identical digits:

```
echo "1223" | grep -e '\([0-9]\)\1'
```

Similarly, the following command is a successful match because the string "12223" contains three consecutive occurrences of the digit 2:

```
echo "12223" | grep -e '\([0-9]\)\1\1'
```

You can check for the occurrence of two identical digits separated by any character with this expression:

```
echo "12z23" | grep -e '\([0-9]\).\1'
```

In an analogous manner, you can test for the occurrence of duplicate letters, as shown here:

```
echo "abbc" | grep -e '\([a-z]\)\1'
```

The following example matches an IP address, and does not use back references, just the "\d" and "\." Regular expressions to match digits and periods:

```
echo "192.168.125.103" | grep -e '\(\d\d\d\)\.\
(\d\d\d\)\.\(\d\d\d\)\.\(\d\d\d\)'
```

If you want to allow for fewer than three digits, you can use the expression {1,3}, which matches 1, 2, or 3 digits on the third block. In a situation where any of the four blocks might have fewer than three characters, you must use the following type of syntax in all four blocks:

```
echo "192.168.5.103" | grep -e '\(\d\d\d\)\.\
(\d\d\d\)\.\(\d\)\{1,3\}\.\(\d\d\d\)'
```

You can perform more complex matches using back references. Listing 5.3 displays the contents of columns5.txt that contains several lines that are palindromes (the same spelling from left-to-right as right-to-left). Note that the third line is an empty line.

Listing 5.3: columns5.txt

```
one eno
ONE ENO

ONE TWO OWT ENO
four five
```

The following command finds all lines that are palindromes:

```
grep -w -e '\(.\)\(.\).*\2\1' columns5.txt
```

The output of the preceding command is here:

```
one eno
ONE ENO
ONE TWO OWT ENO
```

The idea is as follows: the first \(.\) matches a set of letters, followed by a second \(.\) that matches a set of letters, followed by any number of intermediate characters. The sequence \2\1 reverses the order of the matching sets of letters specified by the two consecutive occurrences of \(.\).

FINDING EMPTY LINES IN DATASETS

Recall that the metacharacter "^" refers to the beginning of a line and the metacharacter "$" refers to the end of a line. Thus, an empty line consists of the sequence `^$`. You can find the single empty in `columns5.txt` with this command:

```
grep -n "^$" columns5.txt
```

The output of the preceding `grep` command is here (use the `-n` switch to display line numbers, as blank lines will not otherwise show in the output):

3:

More commonly the goal is to simply strip the empty lines from a file. We can do that just by inverting the prior query (and not showing the line numbers)

```
grep -v "^$" columns5.txt
```

```
one eno
ONE ENO
ONE TWO OWT ENO
four five
```

As you can see, the preceding output displays four non-empty lines, and as we saw in the previous `grep` command, line #3 is an empty line.

USING KEYS TO SEARCH DATASETS

Data is often organized around unique values (typically numbers) in order to distinguish otherwise similar things: for example, `John Smith` the *manager* must not be confused with `John Smith` the *programmer* in an employee data set. Hence, each record is assigned a unique number that will be used for all queries related to employees. Moreover, their names are merely data elements of a given record, rather than a means of identifying a record that contains a particular person.

With the preceding points in mind, suppose that you have a text file in which each line contains a single key value. In addition, another text file consists of one or more lines, where each line contains a key-value followed by a quantity value.

As an illustration, Listing 5.4 displays the contents of `skuvalues.txt` and Listing 5.5 displays the contents of `skusold.txt`. Note that an `SKU` is a term often used to refer to an individual product configuration, including its packaging, labeling, and so forth.

Listing 5.4: skuvalues.txt

```
4520
5530
6550
7200
8000
```

Listing 5.5: skusold.txt

```
4520 12
4520 15
5530 5
5530 12
6550 0
6550 8
7200 50
7200 10
7200 30
8000 25
8000 45
8000 90
```

THE BACKSLASH CHARACTER AND THE GREP COMMAND

The "\" character has a special interpretation when it's followed by the following characters:

"\b" = Match the empty string at the edge of a word
"\B" = Match the empty string provided it's not at the edge of a word, so:
"\brat\b" matches the separate word "rat" but not "crate", and
"\Brat\B" matches "crate" but not "furry rat"
"\<" = Match the empty string at the beginning of the word.
"\>" = Match the empty string at the end of the word.
"\w" = Match word constituent, it is a synonym for "[_[:alnum:]]".
"\W" = Match non-word constituent, it is a synonym for "[^_[:alnum:]]".
"\s" = Match whitespace, it is a synonym for "[[:space:]]".
"\S" = Match non-whitespace, it is a synonym for "[^[:space:]]".

MULTIPLE MATCHES IN THE GREP COMMAND

In an earlier example, you saw how to use the `-i` option to perform a case insensitive match. However, you can also use the pipe "|" symbol to specify more than one sequence of regular expressions.

For example, the following `grep` expression matches any line that contains one as well as any line that contains `ONE TWO`:

```
grep "one\|ONE TWO" columns5.txt
```

The output of the preceding `grep` command is here:

```
one eno
ONE TWO OWT ENO
```

Although the preceding `grep` command specifies a pair of character strings, you can specify an arbitrary number of character sequences or regular expressions, as long as you put `"\|"` between each thing you want to match.

THE GREP COMMAND AND THE XARGS COMMAND

The xargs command is often used in conjunction with the find command in bash. For example, you can search for the files under the current directory (including sub-directories) that have the sh suffix and then check which one of those files contains the string abc, as shown here:

```
find . -print |grep "sh$" | xargs grep -l abc
```

A more useful combination of the find and xargs command is shown here:

```
find . -mtime -7 -name "*.sh" -print | xargs grep -l abc
```

The preceding command searches for all the files (including sub-directories) with suffix "sh" that have not been modified in at least seven days, and pipes that list to the xargs command, which displays the files that contain the string abc (case insensitive).

The find command supports many options, which can be combined via AND as well as OR in order to create very complex expressions.

Note that grep -R hello . also performs a search for the string hello in all files, including sub-directories, and follows the "one process" recommendation. On the other hand, the find . -print command search for all files in all sub-directories, and you can pipe the output to xargs grep hello in order to find the occurrences of the word hello in all files (which involves two processes instead of one process).

You can use the output of the preceding code snippet in order to copy the matching files to another directory, as shown here:

```
cp `find . -print |grep "sh$" | xargs grep -l abc` /tmp
```

Alternatively, you can copy the matching files in the current directory (without matching files in any sub-directories) to another directory with the grep command:

```
cp `grep -l abc *sh` /tmp
```

Yet another approach is to use "backtick" so that you can obtain additional information:

```
for file in `find . -print`
do
   echo "Processing the file: $file"
   # now do something here
done
```

Keep in mind that if you pass too many filenames to the xargs command you will see a "too many files" error message. In this situation, try to insert additional grep commands prior to the xargs command in order to reduce the number of files that are piped into the xargs command.

If you work with NodeJS, you know that the node_modules directory contains a large number of files. In most cases, you probably want to exclude the files in that directory when you are searching for a string, and the "-v" option is ideal for this situation. The following command excludes the files in the node_modules directory while searching for the names of the HTML files that contain the string src and redirecting the list of filenames to the file src_list.txt (and also redirecting error messages to /dev/null):

```
find . –print |grep –v node |xargs grep –il src > src_
list.txt 2>/dev/null
```

You can extend the preceding command to search for the HTML files that contain the string src and the string angular with the following command:

```
find . –print |grep –v node |xargs grep –il src |xargs
grep –il angular >angular_list.txt 2>/dev/null
```

You can use the following combination of grep and xargs to find the files that contain both xml and defs:

```
grep -l xml *svg |xargs grep -l def
```

A variation of the preceding command redirects error messages to /dev/null, as shown here:

```
grep -l hello *txt 2>/dev/null | xargs grep -c hello
```

Searching zip Files for a String

There are at least three ways to search for a string in one or more zip files. As an example, suppose that you want to determine which zip files contain SVG documents.

The first way is shown here:

```
for f in `ls *zip`
do
   echo "Searching $f"
   jar tvf $f |grep "svg$"
done
```

When there are many zip files in a directory, the output of the preceding loop can be very verbose, in which case you need to scroll backward and probably copy/paste the names of the files that actually contain SVG documents into a separate file. A better solution is to put the preceding loop in a shell and redirect its output. For instance, create the file findsvg.sh whose contents are the preceding loop, and then invoking this command:

```
./findsvg.sh 1>1 2>2
```

Notice that the preceding command redirects error messages (2>) to file 2 and the results of the jar/grep command (1>) to file 1. See the Appendix for another example of searching zip files for SVG documents.

CHECKING FOR A UNIQUE KEY VALUE

Sometimes you need to check for the existence of a string (such as a key) in a text file, and then perform additional processing based on its existence. However, do not assume that the existence of a string means that that string only occurs once. As a simple example, suppose the file `mykeys.txt` has the following content:

```
2000
22000
10000
3000
```

Suppose that you search for the string 2000, which you can do with `findkey.sh` whose contents are displayed in Listing 5.6.

Listing 5.6: findkey.sh

```
key="2000"

if [ "`grep $key mykeys.txt`" != "" ]
then
 foundkey=true
else
  foundkey=false
fi

echo "current key = $key"
echo "found key   = $foundkey"
```

Listing 5.6 contains if/else conditional logic to determine whether or not the file `mykeys.txt` contains the value of `$key` (which is initialized as 2000). Launch the code in Listing 5.6 and you will see the following output:

```
current key = 2000
found key   = true
linecount   = 2
```

While the key value of 2000 does exist in `mykeys.txt`, you can see that it matches *two* lines in `mykeys.txt`. However, if `mykeys.txt` were part of a file with 100,000 (or more) lines, it's not obvious that the value of 2000 matches more than one line. In this dataset, 2000 and 22000 both match, and you can prevent the extra matching line with this code snippet:

```
grep –w $key
```

Thus, in files that have duplicate lines, you can count the number of lines that match the key via the preceding code snippet. Another way to do so involves the use of `wc –l` that displays the line count.

Redirecting Error Messages

Another scenario involves the use of the xargs command with the grep command, which can result in "no such … " error messages:

```
find . –print |xargs grep –il abc
```

Make sure to redirect errors using the following variant:

```
find . –print |xargs grep –il abc 2>/dev/null
```

THE EGREP COMMAND AND FGREP COMMAND

The egrep command is ("extended grep") that supports added grep features like "+" (1 or more occurrence of the previous character), "?" (0 or 1 occurrence of the previous character) and "|" (alternate matching). The egrep command is almost identical to the grep -E, along with some caveats that are described here:

https://www.gnu.org/software/grep/manual/html_node/Basic-vs-Extended.html

One advantage of using the egrep command is that it's easier to understand the regular expressions than the corresponding expressions in grep (when it's combined with backward references).

The egrep ("extended grep") command supports extended regular expressions, as well as the pipe "|" in order to specify multiple words in a search pattern. A match is successful if any of the words in the search pattern appears, so you can think of the search pattern as "any" match. Thus, the pattern "abc|def" matches lines that contain either abc or def (or both).

For example, the following code snippet enables you to search for occurrences of the string abc as well as occurrences of the string def in all files with the suffix sh:

```
egrep -w 'abc|def' *sh
```

The preceding egrep command is an "or" operation: a line matches if it contains either abc *or* def.

You can also use metacharacters in egrep expressions. For example, the following code snippet matches lines that start with abc or end with four and a whitespace:

```
egrep '^123|four $' columns3.txt
```

A more detailed explanation of grep, egrep, and frep is here:

https://superuser.com/questions/508881/what-is-the-difference-between-grep-pgrep-egrep-fgrep

Displaying "Pure" Words in a Dataset with egrep

For simplicity, let's work with a text string and that way we can see the intermediate results as we work toward the solution. Let's initialize the variable x as shown here:

```
x="ghi abc Ghi 123 #def5 123z"
```

The first step is to split x into words:

```
echo $x |tr -s ' ' '\n'
```

The output is here:

```
ghi
abc
Ghi
123
#def5
123z
```

The second step is to invoke egrep with the regular expression ^[a-zA-Z]+, which matches any string consisting of one or more uppercase and/or lowercase letters (and nothing else):

```
echo $x |tr -s , , ,\n' |egrep „^[a-zA-Z]+$"
```

The output is here:

```
ghi
abc
Ghi
```

If you also want to sort the output and print only the unique words, use this command:

```
echo $x |tr -s , , ,\n' |egrep „^[a-zA-Z]+$" |sort |
uniq
```

The output is here:

```
123
123z
Ghi
abc
ghi
```

If you want to extract only the integers in the variable x, use this command:

```
echo $x |tr -s ' ' '\n' |egrep "^[0-9]+$" |sort | uniq
```

The output is here:

```
123
```

If you want to extract alphanumeric words from the variable x, use this command:

```
echo $x |tr -s , , ,\n' |egrep „^[a-zA-Z0-9]+$" |sort |
uniq
```

The output is here:

```
123
123z
Ghi
```

```
abc
ghi
```

Note that the ASCII collating sequences places digits before uppercase letters, and the latter are before lowercase letters for the following reason: 0 through 9 are hexadecimal values 0x30 through 0x39, and the uppercase letters in A-Z are hexadecimal 0x41 through 0x5a, and the lowercase letters in a-z are hexadecimal 0x61 through 0x7a.

Now you can replace echo $x with a dataset in order to retrieve only alphabetic strings from that dataset.

The fgrep Command

The fgrep ("fast grep") is the same as grep -F and although fgrep is deprecated, it's still supported in order to allow historical applications that rely on them to run unmodified. In addition, some older systems might not support the -F option for the grep command, so they use the fgrep command. If you really want to learn more about the fgrep command, perform an Internet search for tutorials.

A SIMPLE USE CASE

The code sample in this section shows you how to use the grep command in order to find specific lines in a dataset and then "merge" pairs of lines to create a new dataset. This is very much like what a "join" command does in a relational database. Listing 5.7 displays the contents of the file test1.csv that contains the initial dataset.

Listing 5.7: test1.csv

```
F1,F2,F3,M0,M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12
1,KLM,,1.4,,0.8,,1.2,,1.1,,,2.2,,,1.4
1,KLMAB,,0.05,,0.04,,0.05,,0.04,,,0.07,,,0.05
1,TP,,7.4,,7.7,,7.6,,7.6,,,8.0,,,7.3
1,XYZ,,4.03,3.96,,3.99,,3.84,4.12,,,,4.04,,
2,KLM,,0.9,0.7,,0.6,,0.8,0.5,,,,0.5,,
2,KLMAB,,0.04,0.04,,0.03,,0.04,0.03,,,,0.03,,
2,EGFR,,99,99,,99,,99,99,,,,99,,
2,TP,,6.6,6.7,,6.9,,6.6,7.1,,,,7.0,,
3,KLM,,0.9,0.1,,0.5,,0.7,,0.7,,,0.9,,
3,KLMAB,,0.04,0.01,,0.02,,0.03,,0.03,,,0.03,,
3,PLT,,224,248,,228,,251,,273,,,206,,
3,XYZ,,4.36,4.28,,4.58,,4.39,,4.85,,,4.47,,
3,RDW,,13.6,13.7,,13.8,,14.1,,14.0,,,13.4,,
3,WBC,,3.9,6.5,,5.0,,4.7,,3.7,,,3.9,,
3,A1C,,5.5,5.6,,5.7,,5.6,,5.5,,,5.3,,
4,KLM,,1.2,,0.6,,0.8,0.7,,,0.9,,,1.0,
```

```
4,TP,,7.6,,7.8,,7.6,7.3,,,7.7,,,7.7,
5,KLM,,0.7,,0.8,,1.0,0.8,,0.5,,,1.1,,
5,KLM,,0.03,,0.03,,0.04,0.04,,0.02,,,0.04,,
5,TP,,7.0,,7.4,,7.3,7.6,,7.3,,,7.5,,
5,XYZ,,4.73,,4.48,,4.49,4.40,,,4.59,,,4.63,
```

Listing 5.8 displays the contents of the file `joinlines.sh` that illustrates how to merge the pairs of matching lines in `joinlines.csv`.

Listing 5.8 joinlines.sh

```
inputfile="test1.csv"
outputfile="joinedlines.csv"
tmpfile2="tmpfile2"

# patterns to match:
klm1="1,KLM,"
klm5="5,KLM,"
xyz1="1,XYZ,"
xyz5="5,XYZ,"

#output:
#klm1,xyz1
#klm5,xyz5

# step 1: match patterns with CSV file:
klm1line="`grep $klm1 $inputfile`"
klm5line="`grep $klm5 $inputfile`"
xyz1line="`grep $xyz1  $inputfile`"
# $xyz5 matches 2 lines (we want first line):
grep $xyz5 $inputfile > $tmpfile2
xyz5line="`head -1 $tmpfile2`"
echo "klm1line: $klm1line"
echo "klm5line: $klm5line"
echo "xyz1line: $xyz1line"
echo "xyz5line: $xyz5line"

# step 3: create summary file:
echo "$klm1line" | tr -d '\n' >  $outputfile
echo "$xyz1line"              >> $outputfile
echo "$klm5line" | tr -d '\n' >> $outputfile
echo "$xyz5line"              >> $outputfile
echo; echo
```

The output from launching the shell script in Listing 5.8 is here:

```
1,KLM,,1.4,,0.8,,1.2,,1.1,,,2.2,,,1.41,X
YZ,,4.03,3.96,,3.99,,3.84,4.12,,,,4.04,,
5,KLM,,0.7,,0.8,,1.0,0.8,,0.5,,,1.1,,5,KLM
,,0.03,,0.03,,0.04,0.04,,0.02,,,0.04,,5,X
YZ,,4.73,,4.48,,4.49,4.40,,,4.59,,,4.63,
```

As you can see, the task in this section is very easily solved via the `grep` command. Note that additional data cleaning is required in order to handle the empty fields in the output.

SUMMARY

This chapter showed you how to work with the `grep` utility, which is a very powerful `bash` command for searching text fields for strings. You saw various options for the `grep` command and examples of how to use those options to find string patterns in text files.

Next, you learned about `egrep`, which is a variant of the `grep` command, which can simplify and also expand on the basic functionality of `grep`, indicating when you might choose one option over another.

Finally, you learned how to use key values in one text file to search for matching lines of text in another file, and perform join-like operations using the `grep` command.

CHAPTER 6

TRANSFORMING DATA WITH SED

In the previous chapter, we learned how to reduce a stream of data to only the contents that interested us. In this chapter, we will learn how to transform that data using the `sed` utility, which is an acronym for "stream editor."

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.

In the final section, you will see a number of examples of how to perform stream-oriented processing on datasets, bringing the capabilities of `sed` together with the commands and regular expressions from prior chapters to accomplish difficult tasks with relatively simple code.

WHAT IS THE SED COMMAND?

The name `sed` is an abbreviation for "stream editor," and the utility derives many of its features 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 regular expressions.

Some of the more common uses for `sed` include: print matching lines, delete matching lines, and find/replace matching strings or regular expressions.

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
Removes any trailing newline
Places 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
```

As we saw earlier with other commands, it is always more efficient to just read the file using the `sed` command than to pipe it in with a different command. You can "feed" 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, if you 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 preceding 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 mainstream with this command, as we are not using the `–n` tag. You can also suppress the mainstream 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
```

The preceding `sed` command is admittedly contrived (when was the last time that you needed to replace vowels in a string?), yet it illustrates the fact that you can perform highly context-specific text transformations with the `sed` command.

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 lower case 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 lower case and upper case 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 replacement 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 lower case `abc` to upper case `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, 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 below (both syntaxes work for a replacement operation), but in that case nothing is written to the terminal screen. The above syntax allows both at the same time:

```
echo "abcdefabc" | sed "s/abc/ABC/" > upper1
echo "abcdefabc" | sed -n "s/abc/ABC/p" > upper1
```

Listing 6.1 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 6.1: 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 6.1 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 6.2 displays the contents of the dataset `delim1.txt` that contains multiple delimiters "|", ":", and "^". Listing 6.3 displays the contents of `delimiter1.sh` that replaces the various delimiters in `delimiter1.txt` with a single comma delimiter ",".

Listing 6.2: delimiter1.txt

```
1000|Jane:Edwards^Sales
2000|Tom:Smith^Development
3000|Dave:Del Ray^Marketing
```

Listing 6.3: delimiter1.sh

```
inputfile="delimiter1.txt"
cat $inputfile | sed -e 's/:/,/' -e 's/|/,/' -e
's/\^/,/'
```

As you can see, the second line in Listing 6.3 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 6.3 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 result to be zero):

```
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` involve 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 above 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. Even if you are unfamiliar with the syntax, you can understand exactly what `sed` is doing by looking at each element of the command and comparing to the output. (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 for all occurrences that match the pattern. 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, 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 6.4 displays the contents of test4.txt (doubled-spaced lines) that is used for several examples in this section.

Listing 6.4: test4.txt

```
abc

def

abc

abc
```

The following code snippet prints the first 3 lines in test4.txt (we used this syntax before when deleting rows, 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 duplicates 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, whereas 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 regular expressions 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
one two three
four five
```

As our first example involving `sed` and character classes, the following code snippet illustrates how to match lines that contain lower case 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 lower case 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 6.5 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 6.5: 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 6.6 displays the contents of wordcountinfile.sh that illustrates how to combine various bash commands in order to count the words (and their occurrences) in a file.

Listing 6.6: wordcountinfile.sh

```
# The file is fed to the "tr" command, which changes upper case to lower case
# 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 word count.

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 lower case
* Filter out periods and commas
* Change space between words to linefeed,
* Remove duplicates, prefix occurrence count and sort numerically

Launch the code in Listing 6.6 by specifying `columns4.txt` that is displayed earlier in this chapter (you can specify a different file):

```
wordcountinfile.sh columns4.txt
```

The preceding command displays the following output:

```
3 two
3 three
3 one
3 four
2 five
2 123
1 six
1 456
```

BACK 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 backreferences can also be used in the replacement section of the command.

The following `sed` command matches the consecutive "a" letters 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 4 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
```

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 old with the regular expression ^[a-zA-Z]+, which matches any string consisting of one or more upper case and/or lower case 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. Incidentally, it's worth while for you to compare the code snippets in this section with the corresponding section in Chapter 5 that has essentially the same title as this section.

ONE LINE SED COMMANDS

This section is intended to show a lot of the more useful problems you can solve with a single line of `sed`, and expose you to yet more switches and arguments that they can mix and match to solve related tasks.

Moreover, `sed` supports other options (which are beyond the scope of this book) to perform many other tasks, some of which are sophisticated and correspondingly complex. If you encounter something that none of the examples in this chapter cover, but seems like it is the sort of thing

sed might do, the odds are decent that it does: an Internet search along the lines of "how to do <xxx> in sed" will likely either point you in the right direction or at least to an alternative bash command that will be helpful.

Listing 6.7 displays the contents of data4.txt that is referenced in some of the sed commands in this section. Note that some examples contain options that have not been discussed earlier in this chapter: they are included in case you need the desired functionality (and you can find more details by reading online tutorials).

Listing 6.7: data4.txt

```
  hello world4
    hello world5 two
hello world6 three
               hello world4 four
line five
line six
line seven
```

Print the first line of data4.txt with this command:

```
sed q < data4.txt
```

The output is here:

```
hello world3
```

Print the first three lines of data4.txt with this command:

```
sed 3q < data4.txt
```

The output is here:

```
  hello world4
    hello world5 two
hello world6 three
```

Print the last line of data4.txt with this command:

```
sed '$!d' < data4.txt
```

The output is here:

```
line seven
```

You can also use this snippet to print the last line:

```
sed -n '$p' < data4.txt
```

Print the last two lines of data4.txt with this command:

```
sed '$!N;$!D' <data4.txt
```

The output is here:

```
line six
line seven
```

Suppress the lines in `data4.txt` that contain `world` with this command:

```
sed '/world/d' < data4.txt
```

The output is here:

```
line five
line six
line seven
```

Print duplicates of the lines in `data4.txt` that contain the word `world` with this command:

```
sed '/world/p' < data4.txt
```

The output is here:

```
  hello world4
  hello world4
    hello world5 two
    hello world5 two
hello world6 three
hello world6 three
         hello world4 four
         hello world4 four
line five
line six
line seven
```

Print the fifth line of `data4.txt` with this command:

```
sed -n '5p' < data4.txt
```

The output is here:

```
line five
```

Print the contents of `data4.txt` and duplicate line five with this command:

```
sed '5p' < data4.txt
```

The output is here:

```
  hello world4
    hello world5 two
hello world6 three
         hello world4 four
line five
line five
line six
line seven
```

Print lines four through six of `data4.txt` with this command:

```
sed -n '4,6p' < data4.txt
```

The output is here:

```
  hello world4 four
line five
line six
```

Delete lines four through six of `data4.txt` with this command:

```
sed '4,6d' < data4.txt
```

The output is here:

```
  hello world4
    hello world5 two
hello world6 three
line seven
```

Delete the section of lines between `world6` and `six` in `data4.txt` with this command:

```
sed '/world6/,/six/d' < data4.txt
```

The output is here:

```
  hello world4
    hello world5 two
line seven
```

Print the section of lines between `world6` and `six` of `data4.txt` with this command:

```
sed -n '/world6/,/six/p' < data4.txt
```

The output is here:

```
hello world6 three
        hello world4 four
line five
line six
```

Print the contents of `data4.txt` *and* duplicate the section of lines between `world6` and `six` with this command:

```
sed '/world6/,/six/p' < data4.txt
```

The output is here:

```
  hello world4
    hello world5 two
hello world6 three
hello world6 three
        hello world4 four
        hello world4 four
line five
line five
line six
```

```
line six
line seven
```

Delete the even-numbered lines in `data4.txt` with this command:

```
sed 'n;d;' <data4.txt
```

The output is here:

```
 hello world4
hello world6 three
line five
line seven
```

Replace letters `a` through `m` with a "`,`" with this command:

```
sed "s/[a-m]/,/g" <data4.txt
```

The output is here:

```
 ,,,,o wor,,4
 ,,,,o wor,,5 two
,,,,o wor,,6 t,r,,
       ,,,,o wor,,4 ,our
,,n, ,,v,
,,n, s,x
,,n, s,v,n
```

Replace letters `a` through `m` with the characters "`,@#`" with this command:

```
sed "s/[a-m]/,@#/g" <data4.txt
```

The output is here:

```
 ,@#,@#,@#,@#o wor,@#,@#4
 ,@#,@#,@#,@#o wor,@#,@#5 two
,@#,@#,@#,@#o wor,@#,@#6 t,@#r,@#,@#
       ,@#,@#,@#,@#o wor,@#,@#4 ,@#our
,@#,@#n,@# ,@#,@#v,@#
,@#,@#n,@# s,@#x
,@#,@#n,@# s,@#v,@#n
```

The `sed` command does not recognize escape sequences such as `\t`, which means that you must literally insert a tab on your console. In the case of the `bash` shell, enter the control character `^V` and then press the `<TAB>` key in order to insert a `<TAB>` character.

Delete the tab characters in `data4.txt` with this command:

```
sed 's/  //g' <data4.txt
```

The output is here:

```
 hello world4
hello world5 two
 hello world6 three
```

```
hello world4 four
line five
line six
line seven
```

Delete the tab characters and blank spaces in `data4.txt` with this command:

```
sed 's/  //g' <data4.txt
```

The output is here:

```
helloworld4
helloworld5two
helloworld6three
helloworld4four
linefive
linesix
lineseven
```

Replace every line of `data4.txt` with the word `pasta` with this command:

```
sed 's/.*/\pasta/' < data4.txt
```

The output is here:

```
pasta
pasta
pasta
pasta
pasta
pasta
pasta
```

Insert two blank lines after the third line and one blank line after the fifth line in `data4.txt` with this command:

```
sed '3G;3G;5G' < data4.txt
```

The output is here:

```
   hello world4
  hello world5 two
 hello world6 three


         hello world4 four
line five

line six
line seven
```

Insert a blank line after every line of `data4.txt` with this command:

```
sed G < data4.txt
```

The output is here:

```
 hello world4

  hello world5 two

 hello world6 three

        hello world4 four

line five

line six

line seven
```

Insert a blank line after every other line of `data4.txt` with this command:

```
sed n\;G < data4.txt
```

The output is here:

```
 hello world4
  hello world5 two

 hello world6 three
        hello world4 four

line five
line six

line seven
```

Reverse the lines in `data4.txt` with this command:

```
sed '1! G; h;$!d' < data4.txt
```

The output of the preceding `sed` command is here:

```
line seven
line six
line five
        hello world4 four
 hello world6 three
  hello world5 two
   hello world4
```

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 regular expressions to match data, similar to the `grep` command but can also use regular expressions to describe how to transform the data. Finally, there was a list of examples showing both the versatility of the `sed` command and hopefully communicating the sense that it is an even more flexible and powerful utility than we can show in a single chapter.

CHAPTER 7

WORKING WITH `AWK`

This chapter introduces you to the `awk` command, which is a highly versatile utility for manipulating data and restructuring datasets. In fact, this utility is so versatile that entire books have been written about the `awk` utility. `Awk` is essentially an entire programming language in a single command, which accepts standard input, gives standard output and uses regular expressions and metacharacters in the same way other `bash` commands do. This lets you combine `awk` with other expressions and do almost anything, at the cost of adding complexity to a command string that may already be doing quite a lot already. It is almost always worthwhile to add a comment when using `awk`, it is so versatile that it won't be clear which of the many features you are using at a glance.

The first 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 second part of this chapter shows you conditional logic, `while` loops, and `for` loops in `awk` in order to manipulate the rows and columns in datasets. This section also shows you how to delete lines and merge lines in datasets, and also how to print the contents of a file as a single line of text. You will see how to "join" lines and groups of lines in datasets.

The third 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 a line of text.

The fourth section illustrates how to "split" a text string that contains multiple "." characters as a delimiter, followed by examples of `awk` to perform numeric calculations (such as addition, subtraction, multiplication, and division) in files containing numeric data. This section also shows you various

numeric functions that are available in awk, and also how to print text in a fixed set of columns.

The fifth section explains how to align columns in a dataset and also how to align and merge columns in a dataset. You will see how to delete columns, how to select a subset of columns from a dataset, and how to work with multiline records in datasets. This section contains some one-line awk commands that can be useful for manipulating the contents of datasets.

The final section of this chapter has a pair of use cases involving nested quotes and date formats in structured data sets.

THE AWK COMMAND

The awk (Aho, Weinberger, and Kernighan) command has C-like syntax and you can use this utility to perform very complex operations on numbers and text strings.

As a side comment, there is also the gawk command that is GNU awk, as well as the nawk command, which is the "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 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 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 brackets without any action does nothing; i.e., it will not perform the default printing operation. Note that each statement in actions should be delimited by a semicolon.

In order 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 is adding value in some way):

```
cat test.txt | awk '{ print $1 }'
```

This simple example of four ways to do the same task illustrates 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 7.1 displays the contents of columns2.txt and Listing 7.2 displays the contents of the shell script AlignColumns1.sh that shows you how to align the columns in a text file.

Listing 7.1: columns2.txt

```
one two
three four
one two three four
five six
one two three
four five
```

Listing 7.2: 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 7.2 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 7.2 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()."

CONDITIONAL LOGIC AND CONTROL STATEMENTS

Like other programming languages, awk provides support for conditional logic (if/else) and control statements (for/while loops). awk is the only way to

put conditional logic inside a piped command stream without creating, installing and adding to the path a custom executable shell script. The following code block shows you how to use if/else logic:

```
echo "" | awk '
BEGIN { x = 10 }
{
  if (x % 2 == 0) }
      print "x is even"
  }
  else }
      print "x is odd"
  }
}
'
```

The preceding code block initializes the variable `x` with the value 10 and prints "`x is even`" if x is divisible by 2, otherwise it prints "`x is odd`".

The* `while` *Statement

The following code block illustrates how to use a `while` loop in `awk`:

```
echo "" | awk '
{
  x = 0
  while(x < 4) {
    print "x:",x
    x = x + 1
  }
}
'
```

The preceding code block generates the following output:

```
x:0
x:1
x:2
x:3
```

The following code block illustrates how to use a `do while` loop in `awk`:

```
echo "" | awk '
{
  x = 0

  do {
    print "x:",x
```

```
    x = x + 1
  } while(x < 4)
}
'
```

The preceding code block generates the following output:

```
x:0
x:1
x:2
x:3
```

A for loop in `awk`

Listing 7.3 displays the contents of `Loop.sh` that illustrates how to print a list of numbers in a loop. Note that "`i++`" is another way of writing "`i=i+1`" in `awk`.

Listing 7.3: Loop.sh

```
awk '
BEGIN {
  for(i=0; i<5; i++) {
    printf("%3d", i)
  }
}
'
```

Listing 7.3 contains a `for` loop that prints numbers on the same line via the `printf()` statement. The output from Listing 7.3 is here:

```
0 1 2 3 4
```

A `for` *loop with a* `break` *Statement*

The following code block illustrates how to use a `break` statement in a `for` loop in `awk`:

```
echo "" | awk '
{
  for(x=1; x<4; x++) {
     print "x:",x
     if(x == 2) {
        break;
     }
  }
}
'
```

The preceding code block prints output only until the variable x has the value 2, after which the loop exits (because of the break inside the conditional logic). The following output is displayed:

```
x:1
```

The `next` *and* `continue` *Statements*

The following code snippet illustrates how to use `next` and `continue` in a `for` loop in `awk`:

```
awk '
{
   /expression1/ { var1 = 5; next }
   /expression2/ { var2 = 7; next }
   /expression3/ { continue }
   // some other code block here
' somefile
```

When the current line matches `expression1`, then `var1` is assigned the value 5 and `awk` reads the next input line: hence, `expression2` and `expression3` will not be tested. If `expression1` does not match and `expression2` *does* match, then `var2` is assigned the value 7 and then `awk` will read the next input line. If only `expression3` results in a positive match, then `awk` skips the remaining block of code and processes the next input line.

DELETING ALTERNATE LINES IN DATASETS

Listing 7.4 displays the contents of `linepairs.csv` and Listing 7.5 displays the contents of `deletelines.sh` that illustrates how to print alternating lines from the dataset `linepairs.csv` that have exactly two columns.

Listing 7.4: linepairs.csv

```
a,b,c,d
e,f,g,h
1,2,3,4
5,6,7,8
```

Listing 7.5: deletelines.sh

```
inputfile="linepairs.csv"
outputfile="linepairsdeleted.csv"
awk ' NR%2 {printf "%s", $0; print ""; next}' <
$inputfile > $outputfile
```

Listing 7.5 checks if the current record number `NR` is divisible by 2, in which case it prints the current line and skips the next line in the dataset.

The output is redirected to the specified output file, the contents of which are here:

```
a,b,c,d
1,2,3,4
```

A slightly more common task involves merging consecutive lines, which is the topic of the next section.

MERGING LINES IN DATASETS

Listing 7.6 displays the contents of `columns.txt` and Listing 7.7 displays the contents of `ColumnCount1.sh` that illustrates how to print the lines from the text file `columns.txt` that have exactly two columns.

Listing 7.6: columns.txt

```
one two three
one two
one two three four
one
one three
one four
```

Listing 7.7: ColumnCount1.sh

```
awk '
{
   if( NF == 2 ) { print $0 }
}
' columns.txt
```

Listing 7.7 is straightforward: if the current record number is even, then the current line is printed (i.e., odd-numbered rows are skipped). The output from launching the code in Listing 7.7 is here:

```
one two
one three
one four
```

If you want to display the lines that do *not* contain 2 columns, use the following code snippet:

```
if( NF != 2 ) { print $0 }
```

Printing File Contents as a Single Line

The contents of `test4.txt` are here (note the blank lines):

```
abc


def
```

```
abc

abc
```

The following code snippet illustrates how to print the contents of `test4.txt` as a single line:

```
awk '{printf("%s", $0)}' test4.txt
```

The output of the preceding code snippet is shown below. See if you can tell what is happening before reading the explanation in the next paragraph:

```
Abcdefabcabc
```

Explanation: `%s` is the record separator syntax for `printf`, and the end quote (") that immediately follows `%s` means the record separator is the empty field "". Our default record separator for awk is /n (newline), what the `printf` is doing is stripping out all the newlines. The blank rows will vanish entirely because they only consist of a new line. Hence, all the lines of text are merged together without a newline character between them. Replace `%s` with `"%s"` and you will still see the output as a single line, but with a space between consecutive lines of the text file. Now replace `"%s"` with `"%s\n"` and the output will be identical (in terms of content and layout) as the text file.

Notice how the following comment helps the comprehension of the code snippet:

```
# Merging all text into a single line by removing the newlines
awk '{printf("%s", $0)}' test4.txt
```

Joining Groups of Lines in a Text File

Listing 7.8 displays the contents of `digits.txt` and Listing 7.9 displays the contents of `digits.sh` that "joins" three consecutive lines of text in the file `digits.txt`.

Listing 7.8: digits.txt

```
1
2
3
4
5
6
7
8
9
```

Listing 7.9: digits.sh

```
awk -F" " '{
  printf("%d",$0)
```

```
  if(NR % 3 == 0) { printf("\n") }
}' digits.txt
```

Listing 7.9 prints three consecutive lines of text on the same line, after which a linefeed is printed. This has the effect of "joining" every three consecutive lines of text. The output from launching `digits.sh` is here:

```
123
456
789
```

Joining Alternate Lines in a Text File

Listing 7.10 displays the contents of `columns2.txt` and Listing 7.11 displays the contents of `JoinLines.sh` that "joins" two consecutive lines of text in the file `columns2.txt`.

Listing 7.10: columns2.txt

```
one two
three four
one two three four
five six
one two three
four five
```

Listing 7.11: JoinLines.sh

```
awk '
{
   printf("%s",$0)
   if( $1 !~ /one/) { print " " }
}
' columns2.txt
```

The output from launching Listing 7.11 is here:

```
one two three four
one two three four five six
one two three four five
```

Notice that the code in Listing 7.11 depends on the presence of the string "one" as the first field, which appears in alternating lines of text in `columns2.txt`. Hence, the fact that every line in the output starts with the string `one` is just a coincidence.

Listing 7.12 illustrates how to merge consecutive pairs of lines without specifying a particular string.

Listing 7.12: JoinLines2.sh

```
awk '
BEGIN { count = 0 }
```

```
{
   printf("%s",$0)
   if( ++count % 2 == 0) { print " " }
} columns2.txt
```

Yet another way to "join" consecutive lines is shown in Listing 7.13, where the input file and output file refer to files that you can populate with data. This is another example of an `awk` command that might be puzzling if encountered in a program without a comment. It is doing exactly the same thing as Listing 7.12, but its purpose is less obvious because of the more compact syntax.

Listing 7.13: JoinLines2.sh

```
inputfile="linepairs.csv"
outputfile="linepairsjoined.csv"
awk ' NR%2 {printf "%s,", $0; next;}1' < $inputfile>
$outputfile
```

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 7.14 displays the contents of `Patterns1.sh` that uses metacharacters to match the beginning and the end of a line of text in the file `columns2.txt`.

Listing 7.14: Patterns1.sh

```
awk '
   /^f/    { print $1 }
   /two $/ { print $1 }
' columns2.txt
```

The output from launching Listing 7.14 is here:

```
one
five
four
```

Listing 7.15 displays the contents of `RemoveColumns.txt` with lines that contain a different number of columns.

Listing 7.15: columns3.txt

```
123 one two
456 three four
one two three four
five 123 six
one two three
four five
```

Listing 7.16 displays the contents of `MatchAlpha1.sh` that matches text lines that start with alphabetic characters as well as lines that contain numeric strings in the second column.

Listing 7.16: MatchAlpha1.sh

```
awk '
{
   if( $0 ~ /^[0-9]/) { print $0 }
   if( $0 ~ /^[a-z]+ [0-9]/) { print $0 }
}
' columns3.txt
```

The output from Listing 7.16 is here:

```
123 one two
456 three four
five 123 six
```

PRINTING LINES USING CONDITIONAL LOGIC

Listing 7.17 displays the contents of `products.txt` that contains three columns of information.

Listing 7.17: 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
```

SPLITTING FILENAMES WITH AWK

Listing 7.18 displays the contents of `SplitFilename2.sh` that illustrates how to split a filename containing the "." character in order to increment the numeric value of one of the components of the filename. Note that this code only works for a filename with exactly the expected syntax. It is possible to write more complex code to count the number of segments, or alternately to just say "change the field right before the .zip," which would only require that the filename had a format matching the final two sections (<anystructure>.number.zip).

Listing 7.18: SplitFilename2.sh

```
echo "05.20.144q.az.1.zip" | awk -F"." '
{
  f5=$5 + 1
  printf("%s.%s.%s.%s.%s.%s",$1,$2,$3,$4,f5,$6)
}'
```

The output from Listing 7.18 is here:

```
05.20.144q.az.2.zip
```

WORKING WITH POSTFIX ARITHMETIC OPERATORS

Listing 7.19 displays the contents of `mixednumbers.txt` that contains postfix operators, which means numbers where the negative (and/or positive) sign appears at the end of a column value instead of the beginning of the number.

Listing 7.19: mixednumbers.txt

```
324.000-|10|983.000-
453.000-|30|298.000-
783.000-|20|347.000-
```

Listing 7.20 displays the contents of `AddSubtract1.sh` that illustrates how to add the rows of numbers in Listing 7.19.

Listing 7.20: AddSubtract1.sh

```
myFile="mixednumbers.txt"

awk -F"|" '
BEGIN { line = 0; total = 0 }
{
   split($1, arr, "-")
   f1 = arr[1]
   if($1 ~ /-/) { f1 = -f1 }
   line += f1

   split($2, arr, "-")
   f2 = arr[1]
   if($2 ~ /-/) { f2 = -f2 }
   line += f2

   split($3, arr, "-")
   f3 = arr[1]
   if($3 ~ /-/) { f3 = -f3 }
   line += f3

   printf("f1: %d f2: %d f3: %d line: %d\n",f1,f2,f3,
line)
   total += line
   line = 0
}
END { print "Total: ",total }
' $myfile
```

The output from Listing 7.20 is shown below. See if you can work out what the code is doing before reading the explanation that follows:

```
f1: -324 f2: 10 f3: -983 line: -1297
f1: -453 f2: 30 f3: -298 line: -721
f1: -783 f2: 20 f3: -347 line: -1110
Total:  -3128
```

The code assumes we know the format of the file. The split function turns each field record into a length two vector, where the first position is a number, and the second position either an empty value or a dash: *then* capture the first position number into a variable. The `if` statement just sees if the original field has a dash in it. If the field has a dash, then the numeric variable is made negative, otherwise, it is left alone. Then it adds the line up.

NUMERIC FUNCTIONS IN AWK

The `int(x)` function returns the integer portion of a number. If the number is not already an integer, it falls between two integers. Of the two possible

integers, the function will return the one closest to zero. This is different from a rounding function, which chooses the closer integer.

For example, `int(3)` is 3, `int(3.9)` is 3, `int(-3.9)` is -3, and `int (-3)` is -3 as well. An example of the `int(x)` function in an `awk` command is here:

```
awk 'BEGIN {
   print int(3.534);
   print int(4);
   print int(-5.223);
   print int(-5);
}'
```

The output is here:

```
3
4
-5
-5
```

The `exp(x)` function gives you the exponential of `x`, or reports an error if x is out of range. The range of values `x` can have depends on your machine's floating-point representation.

```
awk 'BEGIN{
   print exp(123434346);
   print exp(0);
   print exp(-12);
}'
```

The output is here:

```
inf
1
6.14421e-06
```

The `log(x)` function gives you the natural logarithm of `x`, if `x` is positive; otherwise, it reports an error (`inf` means infinity and `nan` in output means "not a number").

```
awk 'BEGIN{
  print log(12);
  print log(0);
  print log(1);
  print log(-1);
}'
```

The output is here:

```
2.48491
-inf
0
nan
```

The `sin(x)` function gives you the sine of `x` and `cos(x)` gives you the cosine of `x`, with `x` in radians:

```
awk 'BEGIN {
   print cos(90);
   print cos(45);
}'
```

The output is here:

```
-0.448074
0.525322
```

The `rand()` function gives you a random number. The values of `rand()` are uniformly-distributed between 0 and 1: the value is never 0 and never 1. However, if you want to generate random integers, this user-defined function obtains a random non-negative integer less than n:

```
function randint(n) {
     return int(n * rand())
}
```

The product produces a random real number greater than 0 and less than n. We then make it an integer (using `int`) between 0 and n - 1.

Here is an example where a similar function is used to produce random integers between 1 and n:

```
awk '
# Function to roll a simulated die.
function roll(n) { return 1 + int(rand() * n) }
# Roll 3 six-sided dice and print total number of
points.
{
      printf("%d points\n", roll(6)+roll(6)+roll(6))
}'
```

Note that `rand` starts generating numbers from the same point (or "seed") each time `awk` is invoked. Hence, a program will produce the same results each time it is launched. If you want a program to do different things each time it is used, you must change the seed to a value that will be different in each run.

Use the `srand(x)` function to set the starting point, or seed, for generating random numbers to the value `x`. Each seed value leads to a particular sequence of "random" numbers. Thus, if you set the seed to the same value a second time, you will get the same sequence of "random" numbers again. If you omit the argument `x`, as in `srand()`, then the current date and time of day are used for a seed. This is how to obtain random numbers that are truly unpredictable. The return value of `srand()` is the previous seed. This makes it easy to keep track of the seeds for use in consistently reproducing sequences of random numbers.

The time() function (not in all versions of awk) returns the current time in seconds since January 1, 1970. The function ctime (not in all versions of awk) takes a numeric argument in seconds and returns a string representing the corresponding date, suitable for printing or further processing.

The sqrt(x) function gives you the positive square root of x. It reports an error if x is negative. Thus, sqrt(4) is 2.

```
awk 'BEGIN{
   print sqrt(16);
   print sqrt(0);
   print sqrt(-12);
}'
```

The output is here:

```
4
0
nan
```

ONE LINE AWK COMMANDS

The code snippets in this section reference the text file short1.txt, which you can populate with any data of your choice.

The following code snippet prints each line preceded by the number of fields in each line:

```
awk '{print NF ":" $0}' short1.txt
```

Print the right-most field in each line:

```
awk '{print $NF}' short1.txt
```

Print the lines that contain more than 2 fields:

```
awk '{if(NF > 2) print }' short1.txt
```

Print the value of the right-most field if the current line contains more than 2 fields:

```
awk '{if(NF > 2) print $NF }' short1.txt
```

Remove leading and trailing whitespaces:

```
echo « a b c « | awk '{gsub(/^[ \t]+|[
\t]+$/,»»);print}'
```

Print the first and third fields in reverse order for the lines that contain at least 3 fields:

```
awk '{if(NF > 2) print $3, $1}' short1.txt
```

Print the lines that contain the string one:

```
awk '{if(/one/) print }' *txt
```

As you can see from the preceding code snippets, it's easy to extract information or subsets of rows and columns from text files using simple conditional logic and built-in variables in the awk command.

USEFUL SHORT AWK SCRIPTS

This section contains a set of short `awk` -based scripts for performing various operations. Some of these scripts can also be used in other shell scripts to perform more complex operations. Listing 7.21 displays the contents of the file `data.txt` that is used in various code samples in this section.

Listing 7.21: data.txt

```
this is line one that contains more than 40 characters
this is line two
this is line three that also contains more than 40
characters
four

this is line six and the preceding line is empty

line eight and the preceding line is also empty
```

The following code snippet prints every line that is longer than 40 characters:

```
awk 'length($0) > 40' data.txt
```

Now print the length of the longest line in data.txt:

```
awk '{ if (x < length()) x = length() }
END { print "maximum line length is " x }' < data.txt
```

The input is processed to change tabs into spaces, so the widths compared are actually the right-margin columns.

Print every line that has at least one field:

```
awk 'NF > 0' data.txt
```

The preceding code snippet illustrates an easy way to delete blank lines from a file (or rather, to create a new file similar to the old file but from which the blank lines have been removed).

Print seven random numbers from 0 to 100, inclusive:

```
awk 'BEGIN { for (i = 1; i <= 7; i++)
print int(101 * rand()) }'
```

Count the lines in a file:

```
awk 'END { print NR }' < data.txt
```

Print the even-numbered lines in the data file:

```
awk 'NR % 2 == 0' data.txt
```

If you use the expression `'NR % 2 == 1'` in the previous code snippet, the program would print the odd-numbered lines.

Insert a duplicate of every line in a text file:

```
awk '{print $0, '\n', $0}' < data.txt
```

Insert a duplicate of every line in a text file and also remove blank lines:

```
awk '{print $0, "\n", $0}' < data.txt | awk 'NF > 0'
```

Insert a blank line after every line in a text file:

```
awk '{print $0, "\n"}' < data.txt
```

PRINTING THE WORDS IN A TEXT STRING IN AWK

Listing 7.22 displays the contents of Fields2.sh that illustrates how to print the words in a text string using the awk command.

Listing 7.22: Fields2.sh

```
echo "a b c d e"| awk '
{
  for(i=1; i<=NF; i++) {
     print "Field",i,":",$i
  }
}
'
```

The output from Listing 7.22 is here:

```
Field 1 : a
Field 2 : b
Field 3 : c
Field 4 : d
Field 5 : e
```

COUNT OCCURRENCES OF A STRING IN SPECIFIC ROWS

Listing 7.23 and Listing 7.24 display the contents data1.csv and data2.csv, respectively, and Listing 7.25 displays the contents of checkrows.sh that illustrates how to count the number of occurrences of the string "past" in column 3 in rows 2, 5, and 7.

Listing 7.23: data1.csv

```
in,the,past,or,the,present
for,the,past,or,the,present
in,the,past,or,the,present
for,the,paste,or,the,future
in,the,past,or,the,present
completely,unrelated,line1
in,the,past,or,the,present
completely,unrelated,line2
```

Listing 7.24: data2.csv

```
in,the,past,or,the,present
completely,unrelated,line1
for,the,past,or,the,present
completely,unrelated,line2
for,the,paste,or,the,future
in,the,past,or,the,present
in,the,past,or,the,present
completely,unrelated,line3
```

Listing 7.25: checkrows.sh

```
files="`ls data*.csv| tr '\n' ' '`"
echo "List of files: $files"

awk -F"," '
( FNR==2 || FNR==5 || FNR==7 ) {
    if ( $3 ~ "past" ) { count++ }
}
END {
    printf "past: matched %d times (INEXACT) ", count
    printf "in field 3 in lines 2/5/7\n"
}' data*.csv
```

Listing 7.25 looks for occurrences in the string `past` in columns 2, 5, and 7 because of the following code snippet:

```
( FNR==2 || FNR==5 || FNR==7 ) {
    if ( $3 ~ "past" ) { count++ }
}
```

If a match occurs, then the value of count is incremented. The `END` block reports the number of times that the string past was found in columns 2, 5, and 7. Note that strings such as `paste` and `pasted` will match the string `past`. The output from Listing 7.25 is here:

```
List of files: data1.csv data2.csv
past: matched 5 times (INEXACT) in field 3 in lines
2/5/7
```

The shell script `checkrows2.sh` replaces the term `$3 ~ "past"` with the term `$3 == "past"` in `checkrows.sh` in order to check for exact matches, which produces the following output:

```
List of files: data1.csv data2.csv
past: matched 4 times (EXACT) in field 3 in lines
2/5/7
```

PRINTING A STRING IN A FIXED NUMBER OF COLUMNS

Listing 7.26 displays the contents of FixedFieldCount1.sh that illustrates how to print the words in a text string using the awk command.

Listing 7.26: FixedFieldCount1.sh

```
echo "aa bb cc dd ee ff gg hh"| awk '
BEGIN { colCount = 3 }
{

  for(i=1; i<=NF; i++) {
     printf("%s ", $i)
     if(i % colCount == 0) {
        print " "
     }
  }
}
'
```

The output from Listing 7.26 is here:

```
aa bb cc
dd ee ff
gg hh
```

PRINTING A DATASET IN A FIXED NUMBER OF COLUMNS

Listing 7.27 displays the contents of VariableColumns.txt with lines of text that contain a different number of columns.

Listing 7.27: VariableColumns.txt

```
this is line one
this is line number one
this is the third and final line
```

Listing 7.28 displays the contents of Fields3.sh that illustrates how to print the words in a text string using the awk command.

Listing 7.28: Fields3.sh

```
awk '{printf("%s ", $0)}' | awk '
BEGIN { columnCount = 3 }
{
  for(i=1; i<=NF; i++) {
     printf("%s ", $i)
     if( i % columnCount == 0 )
```

```
      print " "
  }
}
' VariableColumns.txt
```

The output from Listing 7.28 is here:

```
this is line
one this is
line number one
this is the
third and final
line
```

ALIGNING COLUMNS IN DATASETS

If you have read the preceding two examples, the code sample in this section is easy to understand: you will see how to realign columns of data that are correct in terms of their content, but have been placed in different rows (and therefore are misaligned). Listing 7.29 displays the contents of `mixed-data.csv` with misaligned data values. In addition, the first line and final line in Listing 7.29 are empty lines, which will be removed by the shell script in this section.

Listing 7.29: mixed-data.csv

```
Sara, Jones, 1000, CA, Sally, Smith, 2000, IL,
Dave, Jones, 3000, FL, John, Jones,
4000, CA,
Dave, Jones, 5000, NY, Mike,
Jones, 6000, NY, Tony, Jones, 7000, WA
```

Listing 7.30 displays the contents of `mixed-data.sh` that illustrates how to realign the dataset in Listing 7.29.

Listing 7.30: mixed-data.sh

```
#----------------------------------------
# 1) remove blank lines
# 2) remove line feeds
# 3) print a LF after every fourth field
# 4) remove trailing ',' from each row
#----------------------------------------

inputfile="mixed-data.csv"

grep -v "^$" $inputfile |awk -F"," '{printf("%s",$0)}'
| awk '
BEGIN { columnCount = 4 }
```

```
{
   for(i=1; i<=NF; i++) {
     printf("%s ", $i)
     if( i % columnCount  == 0) { print "" }
   }
}' > temp-columns

# 4) remove trailing ',' from output:
cat temp-columns | sed 's/, $//' | sed 's/ $//' >
 $outputfile
```

Listing 7.30 starts with a `grep` command that removes blank lines, followed by an `awk` command that prints the rows of the dataset as a single line of text. The second `awk` command initializes the `columnCount` variable with the value 4 in the `BEGIN` block, followed by a `for` loop that iterates through the input fields. After four fields are printed on the same output line, a linefeed is printed, which has the effect of realigning the input dataset as an output dataset consisting of rows that have four fields. The output from Listing 7.30 is here:

```
Sara, Jones, 1000, CA
Sally, Smith, 2000, IL
Dave, Jones, 3000, FL
John, Jones, 4000, CA
Dave, Jones, 5000, NY
Mike, Jones, 6000, NY
Tony, Jones, 7000, WA
```

ALIGNING COLUMNS AND MULTIPLE ROWS IN DATASETS

The preceding section showed you how to re-align a dataset so that each row contains the same number of columns and also represents a single data record. The code sample in this section illustrates how to realign columns of data that are correct in terms of their content, and also place two records in each line of the new dataset. Listing 7.31 displays the contents of `mixed-data2.csv` with misaligned data values, followed by Listing 7.32 that displays the contents of `aligned-data2.csv` with the correctly formatted dataset.

Listing 7.31: mixed-data2.csv

```
Sara, Jones, 1000, CA, Sally, Smith, 2000, IL,
Dave, Jones, 3000, FL, John, Jones,
4000, CA,
Dave, Jones, 5000, NY, Mike,
Jones, 6000, NY, Tony, Jones, 7000, WA
```

Listing 7.32: aligned-data2.csv

```
Sara, Jones, 1000, CA, Sally, Smith, 2000, IL
Dave, Jones, 3000, FL, John, Jones, 4000, CA
Dave, Jones, 5000, NY, Mike, Jones, 6000, NY
Tony, Jones, 7000, WA
```

Listing 7.33 displays the contents of `mixed-data2.sh` that illustrates how to realign the dataset in Listing 7.31.

Listing 7.33: mixed-data2.sh

```
#----------------------------------------
# 1) remove blank lines
# 2) remove line feeds
# 3) print a LF after every 8 fields
# 4) remove trailing ',' from each row
#----------------------------------------
inputfile="mixed-data2.txt"
outputfile="aligned-data2.txt"

grep -v "^$" $inputfile |awk -F"," '{printf("%s",$0)}'
| awk '
BEGIN { columnCount = 4; rowCount = 2; currRow = 0 }
{
   for(i=1; i<=NF; i++) {
     printf("%s ", $i)

          if( i % columnCount == 0) { ++currRow }
          if(currRow > 0 && currRow % rowCount == 0) {currRow = 0; print
""}
   }
}' > temp-columns

# 4) remove trailing ',' from output:
cat temp-columns | sed 's/, $//' | sed 's/ $//' >
$outputfile
```

Listing 7.33 is very similar to Listing 7.30. The key idea is to print a linefeed character after a pair of "normal" records have been processed, which is implemented via the code that is shown in bold in Listing 7.33.

Now you can generalize Listing 7.33 very easily by changing the initial value of the `rowCount` variable to any other positive integer, and the code will work correctly without any further modification. For example, if you initialize `rowCount` to the value `5`, then every row in the new dataset (with the possible exception of the final output row) will contain 5 "normal" data records.

REMOVING A COLUMN FROM A TEXT FILE

Listing 7.34 displays the contents of VariableColumns.txt with lines of text that contain a different number of columns.

Listing 7.34: VariableColumns.txt

```
this is line one
this is line number one
this is the third and final line
```

Listing 7.35 displays the contents of RemoveColumn.sh that removes the first column from a text file.

Listing 7.35: RemoveColumn.sh

```
awk '{ for (i=2; i<=NF; i++) printf "%s ", $i; printf
"\n"; }' products.txt
```

The loop is between 2 and NF, which iterates over all the fields except for the first field. In addition, printf explicitly adds newlines. The output of the preceding code snippet is here:

```
400 new
300 new
300 used
200 used
100 used
```

SUBSETS OF COLUMNS ALIGNED ROWS IN DATASETS

Listing 7.35 showed you how to align the rows of a dataset, and the code sample in this section illustrates how to extract a subset of the existing columns and a subset of the rows. Listing 7.36 displays the contents of sub-rows-cols.txt of the desired dataset that contains two columns from every even row of the file aligned-data.txt.

Listing 7.36: sub-rows-cols.txt

```
Sara, 1000
Dave, 3000
Dave, 5000
Tony, 7000
```

Listing 7.37 displays the contents of sub-rows-cols.sh that illustrates how to generate the dataset in Listing 7.36. Most of the code is the same as Listing 7.33, with the new code shown in bold.

Listing 7.37: sub-rows-cols.sh

```
#----------------------------------------
# 1) remove blank lines
# 2) remove line feeds
# 3) print a LF after every fourth field
# 4) remove trailing ',' from each row
#----------------------------------------

inputfile="mixed-data.csv"
outputfile="sub-rows-cols.csv"

grep -v "^$" $inputfile |awk -F"," '{printf("%s",$0)}'
| awk '
BEGIN { columnCount = 4 }
{
   for(i=1; i<=NF; i++) {
     printf("%s ", $i)
     if( i % columnCount  == 0) { print "" }
   }
}' > temp-columns

# 4) remove trailing ',' from output:
cat temp-columns | sed 's/, $//' | sed 's/$//' > temp-
columns2

cat temp-columns2 | awk `
BEGIN { rowCount = 2; currRow = 0 }
{
   if(currRow % rowCount == 0) { print $1, $3 }
   ++currRow
}' > temp-columns3

cat temp-columns3 | sed 's/,$//' | sed 's/ $//' > $outputfile
```

Listing 7.37 contains a new block of code that redirects the output of step #4 to a temporary file `temp-columns2` whose contents are processed by another `awk` command in the last section of Listing 7.37. Notice that that `awk` command contains a `BEGIN` block that initializes the variables `rowCount` and `currRow` with the values 2 and 0, respectively.

The main block prints columns 1 and 3 of the current line if the current row number is even, and then the value of `currRow` is then incremented. The output of this `awk` command is redirected to yet another temporary file that is the input to the final code snippet, which uses the `cat` command and

two occurrences of the sed command in order to remove a trailing "," and a trailing space, as shown here:

```
cat temp-columns3 | sed 's/,$//' | sed 's/ $//' > $outputfile
```

Keep in mind that there are other ways to perform the functionality in Listing 7.37, and the main purpose is to show you different techniques for combining various bash commands.

COUNTING WORD FREQUENCY IN DATASETS

Listing 7.38 displays the contents of WordCounts1.sh that illustrates how to count the frequency of words in a file.

Listing 7.38: WordCounts1.sh

```
awk '
# Print list of word frequencies
{
    for (i = 1; i <= NF; i++)
        freq[$i]++
}
END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}
' columns2.txt
```

Listing 7.38 contains a block of code that processes the lines in columns2.txt. Each time that a word (of a line) is encountered, the code increments the number of occurrences of that word in the hash table freq. The END block contains a for loop that displays the number of occurrences of each word in columns2.txt.

The output from Listing 7.38 is here:

```
two     3
one     3
three   3
six     1
four    3
five    2
```

Listing 7.39 displays the contents of WordCounts2.sh that perform a case insensitive word count.

Listing 7.39: WordCounts2.sh

```
awk '
{
    # convert everything to lower case
    $0 = tolower($0)
```

```
    # remove punctuation
   #gsub(/[^[:alnum:]_[:blank:]]/, "", $0)

    for(i=1; i<=NF; i++) {
       freq[$i]++
    }
}
END {
    for(word in freq) {
       printf "%s\t%d\n", word, freq[word]
    }
}
' columns4.txt
```

Listing 7.39 is almost identical to Listing 7.38, with the addition of the following code snippet that converts the text in each input line to lowercase letters, as shown here:

```
$0 = tolower($0)
```

Listing 7.40 displays the contents of `columns4.txt`.

Listing 7.40: columns4.txt

```
123 ONE TWO
456 three four
ONE TWO THREE FOUR
five 123 six
one two three
four five
```

The output from launching Listing 7.39 with `columns4.txt` is here:

```
456     1
two     3
one     3
three   3
six     1
123     2
four    3
five    2
```

DISPLAYING ONLY "PURE" WORDS IN A DATASET

For simplicity, let's work with a text string and that way we can see the intermediate results as we work toward the solution. This example will be familiar from prior chapters, but now we see how `awk` does it.

Listing 7.41 displays the contents of onlywords.sh that contains three awk commands for displaying the words, integers, and alphanumeric strings, respectively, in a text string.

Listing 7.41: onlywords.sh

```
x="ghi abc Ghi 123 #def5 123z"

echo "Only words:"
echo $x |tr -s ' ' '\n' | awk -F" " '
{
  if($0 ~ /^[a-zA-Z]+$/) { print $0 }
}
' | sort | uniq
echo

echo "Only integers:"
echo $x |tr -s ' ' '\n' | awk -F" " '
{
  if($0 ~ /^[0-9]+$/) { print $0 }
}
' | sort | uniq
echo

echo "Only alphanumeric words:"
echo $x |tr -s ' ' '\n' | awk -F" " '
{
  if($0 ~ /^[0-9a-zA-Z]+$/) { print $0 }
}
' | sort | uniq
echo
```

Listing 7.41 starts by initializing the variable x:

```
x="ghi abc Ghi 123 #def5 123z"
```

The next step is to split x into words:

```
echo $x |tr -s , , ,\n'
```

The output is here:

```
ghi
abc
Ghi
123
#def5
123z
```

The third step is to invoke `awk` and check for words that match the regular expression `^[a-zA-Z]+`, which matches any string consisting of one or more uppercase and/or lowercase letters (and nothing else):

```
if($0 ~ /^[a-zA-Z]+$/) { print $0 }
```

The output is here:

```
ghi
abc
Ghi
```

Finally, if you also want to sort the output and print only the unique words, redirect the output from the `awk` command to the `sort` command and the `uniq` command.

The second `awk` command uses the regular expression `^[0-9]+` to check for integers and the third `awk` command uses the regular expression `^[0-9a-zA-Z]+` to check for alphanumeric words. The output from launching Listing 7.41 is here:

```
Only words:
Ghi
abc
ghi

Only integers:
123

Only alphanumeric words:
123
123z
Ghi
abc
ghi
```

Now you can replace the variable `x` with a dataset in order to retrieve only alphabetic strings from that dataset.

WORKING WITH MULTILINE RECORDS IN AWK

Listing 7.42 displays the contents of `employee.txt` and Listing 7.43 displays the contents of `Employees.sh` that illustrates how to concatenate text lines in a file.

Listing 7.42: employees.txt

```
Name:  Jane Edwards
EmpId: 12345
Address: 123 Main Street Chicago Illinois
```

```
Name:  John Smith
EmpId: 23456
Address: 432 Lombard Avenue SF California
```

Listing 7.43: employees.sh

```
inputfile="employees.txt"
outputfile="employees2.txt"

awk '
{
  if($0 ~ /^Name:/) {
    x = substr($0,8) ","
    next
  }

  if( $0 ~ /^Empid:/) {
   #skip the Empid data row
   #x = x substr($0,7) ","
    next
  }

  if($0 ~ /^Address:/) {
    x = x substr($0,9)
    print x
  }
}
' < $inputfile > $outputfile
```

The output from launching the code in Listing 7.43 is here:

```
Jane Edwards, 123 Main Street Chicago Illinois
John Smith, 432 Lombard Avenue SF California
```

Now that you have seen a plethora of `awk` code snippets and shell scripts containing the `awk` command that illustrate the various types of tasks that you can perform on files and datasets you are ready for some uses cases. The next section (which is the first use case) shows you how to replace multiple field delimiters with a single delimiter, and the second use case shows you how to manipulate date strings.

A SIMPLE USE CASE

The code sample in this section shows you how to use the `awk` command in order to split the comma-separated fields in the rows of a dataset, where fields can contain nested quotes of arbitrary depth.

Listing 7.44 displays the contents of the file `quotes3.csv` that contains a "," delimiter and multiple quoted fields.

Listing 7.44: quotes3.csv

```
field5,field4,field3,"field2,foo,bar",field1,field6,field7,"
fieldZ"
fname1,"fname2,other,stuff",fname3,"fname4,foo,bar",f
name5
"lname1,a,b","lname2,c,d","lname3,e,f","lname4,foo,ba
r",lname5
```

Listing 7.45 displays the contents of the file `delim1.sh` that illustrates how to replace the delimiters in `delim1.csv` with a "," character.

Listing 7.45 delim1.sh

```
#inputfile="quotes1.csv"
#inputfile="quotes2.csv"
inputfile="quotes3.csv"

grep -v "^$" $inputfile |  awk '
{
   print "LINE #" NR ": " $0
   printf ("-------------------------\n")
   for (i = 0; ++i <= NF;)
     printf "field #%d : %s\n", i, $i
   printf ("\n")
}' FPAT='([^,]+)|("[^"]+")' < $inputfile
```

The output from launching the shell script in Listing 7.45 is here:

```
LINE #1: field5,field4,field3,"field2,foo,bar",field1,field
6,field7,"fieldZ"
-------------------------
field #1 : field5
field #2 : field4
field #3 : field3
field #4 : "field2,foo,bar"
field #5 : field1
field #6 : field6
field #7 : field7
field #8 : "fieldZ"

LINE #2: fname1,"fname2,other,stuff",fname3,"fname4,f
oo,bar",fname5
-------------------------
```

```
field #1 : fname1
field #2 : "fname2,other,stuff"
field #3 : fname3
field #4 : "fname4,foo,bar"
field #5 : fname5

LINE #3: "lname1,a,b","lname2,c,d","lname3,e,f","lnam
e4,foo,bar",lname5
-------------------------
field #1 : "lname1,a,b"
field #2 : "lname2,c,d"
field #3 : "lname3,e,f"
field #4 : "lname4,foo,bar"
field #5 : lname5

LINE #4: "Outer1 "Inner "Inner "Inner C" B" A" Outer1"
,"XYZ1,c,d","XYZ2lname3,e,f"
-------------------------
field #1 : "Outer1 "Inner "Inner "Inner C" B" A"
Outer1"
field #2 : "XYZ1,c,d"
field #3 : "XYZ2lname3,e,f"

LINE #5:
-------------------------
```

As you can see, the task in this section is very easily solved via the `awk` command, probably simpler than a solution involving other command line utilities.

ANOTHER USE CASE

The code sample in this section shows you how to use the `awk` command in order to reformat the date field in a dataset and change the order of the fields in the new dataset. For example, given the following input line in the original dataset:

```
Jane,Smith,20140805234658
```

The reformatted line in the output dataset has this format:

```
2014-08-05 23:46:58,Jane,Smith
```

Listing 7.46 displays the contents of the file `dates2.csv` that contains a "," delimiter and three fields.

Listing 7.46: dates2.csv

```
Jane,Smith,20140805234658
Jack,Jones,20170805234652
Dave,Stone,20160805234655
John,Smith,20130805234646
Jean,Davis,20140805234649
Thad,Smith,20150805234637
Jack,Pruit,20160805234638
```

Listing 7.47 displays the contents of `string2date2.sh` that converts the date field to a new format and shifts the new date to the first field.

Listing 7.47: string2date2.sh

```
inputfile="dates2.csv"
outputfile="formatteddates2.csv"

rm -f $outputfile; touch $outputfile

for line in `cat $inputfile`
do
  fname='echo $line |cut -d"," -f1'
  lname='echo $line |cut -d"," -f2'
  date1='echo $line |cut -d"," -f3'

  # convert to new date format
  newdate='echo $date1 | awk '{ print substr($0,1,4)"-
"substr($0,5,2)"-"substr($0,7,2)" "substr($0,9,2)":"s
ubstr($0,11,2)":"substr($0,13,2)}'’

  # append newly formatted row to output file
  echo "${newdate},${fname},${lname}" >> $outputfile
done
```

The contents of the new dataset is here:

```
2014-08-05 23:46:58,Jane,Smith
2017-08-05 23:46:52,Jack,Jones
2016-08-05 23:46:55,Dave,Stone
2013-08-05 23:46:46,John,Smith
2014-08-05 23:46:49,Jean,Davis
2015-08-05 23:46:37,Thad,Smith
2016-08-05 23:46:38,Jack,Pruit
```

SUMMARY

This chapter introduced the `awk` command, which is essentially an entire programming language packaged into a single `bash` command.

We explored some of its built-in variables as well as conditional logic, `while` loops, and `for` loops in `awk` in order to manipulate the rows and columns in datasets. You then saw how to delete lines and merge lines in datasets, and also how to print the contents of a file as a single line of text. Next, you learned how to use metacharacters and character sets in `awk` commands. You learned how to perform numeric calculations (such as addition, subtraction, multiplication, and division) in files containing numeric data, and also some numeric functions that are available in `awk`.

In addition, you saw how to align columns in a dataset, how to delete columns, how to select a subset of columns from a dataset, and how to work with multiline records in datasets. Finally, you saw a couple of simple use cases involving nested quotes and date formats in a structured dataset.

At this point, you have all the tools necessary to do quite sophisticated data cleansing and processing, and you are encouraged to try to apply them on some task or problem of interest.

CHAPTER 8

INTRO TO SHELL SCRIPTS

This chapter introduces you to shell scripts that illustrate how to solve some well-known tasks. Although Chapter 4 showed you how to define a custom function in the `AppendRow.sh` shell script, this chapter is devoted to shell scripts. Some examples rely on the `grep` command, so this would be a good time to review the material in the chapter that contains `grep`-related information. Later in this chapter, you will see shell scripts that involve recursion-based algorithms for well-known tasks such as the GCD and LCM of two positive integers.

The first part of this chapter starts with examples of very simple shell scripts and also how to make those shell scripts executable. This section also shows you how to "source" or "dot" a shell script, and also describes situations when it's necessary to do so.

The second part of this chapter shows you how to use pass parameters to shell functions that are defined in shell scripts, how to determine the number of values passed to a function, and how to display their values. This section also contains an example of an interactive shell script (i.e., prompts users for their input).

The third part of this chapter shows you how to use recursion in order to compute the factorial value of a positive integer. In addition, this section shows you shell scripts for calculating Fibonacci numbers, the GCD and LCM of two positive integers, and the divisors of a positive integer.

One detail to keep in mind regarding the shell scripts in this book: although some of them might not have immediate value for you, it's still worth your time to read them to see if they contain techniques that you can use in your own shell scripts.

WHAT ARE SHELL SCRIPTS?

Shell scripts contain a collection of `bash` commands that are executed in order to complete a task defined by you. If the shell script does not contain any functions, then the commands are executed sequentially from top to bottom (i.e., in the sequence that they appear in a shell script). As you will see later in this chapter, you can define functions and use conditional logic in order to alter the sequence commands are executed in.

Shell scripts can contain whatever `bash` commands are available on your system, but keep in mind that some commands require the `sudo` command, which in turn requires a password. Simple examples of shell scripts include file-related commands that create files, read data from files, and update the contents of files. Regardless of the contents of your shell scripts, they are interpreted "on the fly," so there are no compilation steps that create a binary executable.

Shell scripts automate the process of executing a set of `bash` commands so that you don't need to execute them manually from the command line. If you need to execute a simple command from the command line, then it's unlikely that you need to do so via a shell script: just type the command and press the `<RETURN>` key. Note that the `crontab` utility enables you to schedule the execution of shell scripts on a regular basis (hourly, daily, weekly, and so forth). Chapter 10 provides some additional information regarding the `crontab` utility.

A Simple Shell Script

This section shows you how to create a shell script that contains an assortment of very simple commands that are executed sequentially. Specifically, create the text file `test.sh` (using your favorite text editor) with the following contents:

```
#!/bin/bash
pwd
ls
cd /tmp
ls
mkdir /tmp/abc
touch /tmp/abc/emptyfile
ls /tmp/abc/
```

The second step involves making this shell script executable, which involves the `chmod` command, as shown here:

```
chmod +x test.sh
```

Now your shell script is ready for execution, and simply type the following command in the directory that contains `test.sh`:

```
./test.sh
```

NOTE *The output from launching* `test.sh` *depends on the contents of the* `/tmp` *directory.*

The first line in `test.sh` is called the "shebang" line, which directs the system to launch the bash shell in order to invoke the commands in `test.sh`. The term shebang is sort of a contraction of "hash" (for the "#" character) and "bang" (for the "!" character). Note that the initial "`./`" of `./test.sh` specifies the file `test.sh` in the current directory: if the file `test.sh` is in your home directory, specify `$HOME/test.sh`. In addition, if "." is included in the `PATH` environment variable, then you can simply type `test.sh` without the "`./`" prefix.

One point regarding the `mkdir` command: if you specify a path in which intermediate directories do not exist, then you need to use the `-p` switch. For example, if the directory `/tmp/abc` does not exist, then the following command requires the `-p` switch:

```
mkdir -p /tmp/abc/def
```

SETTING ENVIRONMENT VARIABLES VIA SHELL SCRIPTS

A very important concept when using shell scripts is that any variables set inside the script are no longer set when the script finishes its execution. The rules are shown below:

If a variable isn't set in a script but is already defined before the script is executed, that variable will also be available inside the script.

If a variable is set in a script, it will override any existing variable with the same name after the variable is set, but once the script ends, the variable will revert to its old value (or to no value, if it did not exist outside the shell script)

For example, if your `$HOME` directory is `/Users/jsmith` but inside a script on line 10 you define `$HOME` to be `/Users/common/bin`, then the value of `$HOME` is initially `/Users/jsmith` for lines 1-9, then becomes `/Users/common/bin` on line 10, and maintains that value until the last command in the shell script is executed. When the shell script has finished its execution, the value of `$HOME` reverts to `/Users/jsmith`.

The reason for this behavior is related to how Unix structures its processes (known as "shells," hence the term "shell script"). That discussion is beyond the scope of this book, but you can perform an online search to find articles with a detailed explanation.

Thus, the default behavior is that if you set the value of a variable in a shell script, then that variable (and its value) exists only for the duration of the execution of the shell script. There is a simple "workaround" whereby variables "hold" their values after a shell script has completed, and you'll learn how to do so in a subsequent section.

Just to make sure that the distinction is clear, consider Listing 8.1 that displays the contents of the shell script `abc.sh`.

Listing 8.1: abc.sh

```
export x="123"
echo "inside abc.sh"
echo "x = $x"
```

Make sure that `abc.sh` is an executable shell script with the `chmod` command (as shown earlier in this chapter) and then launch the following sequence of commands from the command line:

```
export x="tom"
echo "x = $x"
```

./abc.sh

```
echo "x = $x"
```

The output from the preceding commands is here:

```
x = tom
inside abc.sh
x = 123
x = tom
```

As you can see, the value that is assigned to the variable `x` is only for the duration of the process associated with the shell script `abc.sh`. After execution has completed, the process terminates and the value of `x` reverts to its original value. Fortunately, there is a way to ensure that the values of variables in a shell script can be "set" for the current shell, a technique called "sourcing" the shell script, as described in the next section.

Sourcing or "Dotting" a Shell Script

Now execute the following sequence of commands:

```
export x="tom smith"
echo "x = $x"
```

. abc.sh

```
echo "x = $x"
```

The output from the preceding commands is here:

```
x = "tom smith"
inside abc.sh
x = 123
x = 123
```

In the preceding code block, the value assigned to the variable `x` inside the shell script `abc.sh` overrides its previously defined value because "sourcing" (also called "dotting") a shell script does not create a new process. Consequently, if a shell script assigns a new value to an existing variable, that new value is placed in the current environment and the previously defined value is lost.

As you probably know, comments are important in source code. A good shell script contains meaningful comments, which are preceded by a pound sign "#", that explain the purpose of different sections in the shell script. The exception is when the "#" symbol appears in the first line of a shell script, as you will see in the next section.

WORKING WITH FUNCTIONS IN SHELL SCRIPTS

A shell function can be defined by using the keyword `function`, followed by the name of the function (specified by you) and a pair of round parentheses, followed by a pair of curly brackets that contain shell commands. The general form is shown here:

```
function fname()
{
   statements;
}
```

An alternate method of defining a shell function involves placing the left curly bracket on a separate line, as shown here:

```
fname()
{
   statements;
}
```

A shell function can be invoked by its name, as shown here:

```
fname ; # executes the function
```

Arguments can be passed to functions and can be accessed by the shell script:

```
fname arg1 arg2 ; # passing args
```

Listing 8.2 displays the contents of `simple-shell.sh` that illustrates how to define a function in a shell script.

Listing 8.2: simple-shell.sh

```
#!/bin/sh

function1 ()
{
   echo "inside function 1"
}

function2 ()
{
   echo "you entered $1 in function 2"
}
```

```
# invoke function1 here:
function1

echo "Enter a string: "
read str

# invoke function2 here:
function2 str
```

Listing 8.2 defines `function1` that displays a text message and `function2` that displays the string that you entered at the command line. Launch Listing 8.2 and enter `abc` at the prompt and you will see the following output:

```
inside function 1
Enter a string:
abc
you entered str in function 2
```

PASSING VALUES TO FUNCTIONS IN A SHELL SCRIPT (1)

Positional parameters are built-in variables that contain the values of command-line arguments to scripts and functions in shell scripts. The positional parameters are named 1, 2, 3, etc., and their values are denoted by $1, $2, $3, etc. The positional parameter 0 has the value equal to the name of the script.

Listing 8.3 displays the contents of `parameters-function.sh` that illustrates how to pass values to a function in a shell script.

Listing 8.3: parameters-function.sh

```
#!/bin/sh

function1 ()
{
   echo "top of function 1"
   echo "param 1: $1"
   echo "param 2: $2"
   echo "param 3: $3"
}

# invoke function1 here:
function1 a
function1 a b
function1 a b c
```

Listing 8.3 defines function1 that displays the values of the first three parameters that it receives. Launch Listing 8.3 and you will see the following output:

```
top of function 1
param 1: a
param 2:
param 3:
top of function 1
param 1: a
param 2: b
param 3:
top of function 1
param 1: a
param 2: b
param 3: c
```

There is an obvious problem with the function in Listing 8.3: this function assumes that there are exactly three parameters. Hence, it displays parameters that do not have values and do not display the values beyond the first three parameters.

The solution involves determining the number of parameters that are supplied to a function, which is the topic of the next section.

PASSING VALUES TO FUNCTIONS IN A SHELL SCRIPT (2)

Two special variables contain all of the positional parameters (except for positional parameter 0): `*` and `@`. The difference between them is subtle but important, and it's apparent only when they are within double quotes.

Listing 8.4 displays the contents of `parameters-function2.sh` that illustrates how to determine the number of values that are passed to a function in a shell script.

Listing 8.4: parameters-function2.sh

```
#!/bin/sh

function1 ()
{
   echo "param count: $#"
   echo "all params:  $@"
   echo ""
}

# invoke function1 here:
function1 a
```

```
function1 a b
function1 a b c
function1 a b c d
function1 1 2 3 4 5

# display the command-line values:
echo "param count: $#"
echo "all params:  $@"
```

As you can see, the function in Listing 8.4 uses `$3` to display the number of parameters and `$@` in order to display their values each time that the function is invoked.

Launch Listing 8.4 and you will see the following output:

```
param count: 1
all params:  a

param count: 2
all params:  a b

param count: 3
all params:  a b c

param count: 4
all params:  a b c d

param count: 5
all params:  1 2 3 4 5
param 3: c

param count: 2
all params:  good pasta
```

ITERATE THROUGH VALUES PASSED TO A FUNCTION

As another example, this shell function and invocation of the shell function shows you how to list parameter values in a convenient block:

```
show_args ()
{
   echo "Argument count:  $#"
   echo "Name of script:  $0"
   echo "First argument:  $1"
   echo "Second argument: $2"
   echo "Third argument:  $3"
   echo "All arguments:   $@"
}

show_args new york chicago pizza
```

The output from the previous code block is here:

```
Argument count:  4
Name of script:  ./arguments.sh
First argument:  new
Second argument: york
Third argument:  chicago
All arguments:   new york chicago pizza
```

Listing 8.5 displays the contents of `iterate-args1.sh` that illustrates how to iterate through a set of values in a shell script.

Listing 8.5: iterate-args1.sh

```
#!/bin/sh

for i in {1..5}
do
  echo "Value of i: ${i}"
done
```

As you can see, the function in Listing 8.5 iterate through the values from 1 through 5 inclusive. Launch Listing 8.5 and you will see the following output:

```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

Listing 8.6 displays the contents of `iterate-args2.sh` that illustrates how to iterate through a set of values in a shell script.

Listing 8.6: iterate-args2.sh

```
#!/bin/sh

iterate()
{
   for arg
   do
     echo "value: $arg";
   done
}

iterate a b c d e
```

As you can see, the function in Listing 8.6 iterate through the values that are passed into the function iterate. Launch Listing 8.6 and you will see the following output:

```
value: a
value: b
value: c
value: d
value: e
```

Listing 8.7 displays the contents of `iterate-args3.sh` that illustrates how to iterate through a set of values in a shell script.

Listing 8.7: iterate-args3.sh

```
#!/bin/sh

iterate()
{
  echo "this will be skipped ... why?"
}

iterate()
{
   arg1=”$1”; shift;

   for arg
   do
     echo "value: $arg";
   done
}

iterate a b c d e
```

As you can see, the function in Listing 8.7 iterate through the values that are passed into the function iterate. However, there are two definitions of the `iterate()` function, and the rule is simple: the last (bottom-most) definition is executed, and all other preceding definitions are ignored (there might be more than two definitions of the same function).

Another detail to notice is the first code snippet that's shown in bold in the (second) definition of the iterate function, and reproduced here:

```
arg1="$1"; shift;
```

The preceding code snippet saves the value of $1 in the variable `arg1`, just in case you want to process this value elsewhere in the code (which we simply ignore in this code sample). Next, the shift keyword performs a “left shift” on

the set of arguments that were passed into the iterate function. As a result, $1 is replaced with $2, and $2 is replaced with $3, and so on, until all the arguments have been shifted leftward.

Launch Listing 8.7 and you will see the following output:

```
value: b
value: c
value: d
value: e
```

Listing 8.8 displays the contents of `iterate-args3.sh` that illustrates how to use a `for` loop in order to iterate through a set of values in a shell script.

Listing 8.8: iterate-args4.sh

```
#!/bin/sh

iterate()
{
   for (( i=2; i <= "$#"; i++ ))
   do
     echo "arg position: ${i}"
     echo "arg value:    ${!i}"
   done
}

iterate a b c d e
```

As you can see, the function in Listing 8.8 iterate through the values that are passed into the function `iterate()`. However, the `for` loop starts from the value 2, which skips the first argument. Launch Listing 8.8 and you will see the following output:

```
arg position: 2
arg value:    b
arg position: 3
arg value:    c
arg position: 4
arg value:    d
arg position: 5
arg value:    e
```

Listing 8.9 displays the contents of `iterate-args5.sh` that illustrates how to use a `for` loop in order to iterate through a set of values, some of which are inside quotation marks, in a shell script.

Listing 8.9: iterate-args5.sh

```
#!/bin/sh

iterate()
{
   echo "Argument count: $#"
   echo "Argument list:  $@"
   echo ""

   for i in "${@}"
   do
     echo "argument: $i"
   done
}

iterate a "b c" d "e f"
```

As you can see, the function in Listing 8.9 iterate through the values that are passed into the function `iterate()`. If two or more strings are inside a pair of quotes, then they are treated as a single argument. Launch Listing 8.9 and you will see the following output:

```
Argument count: 4
Argument list:  a b c d e f

argument: a
argument: b c
argument: d
argument: e f
```

POSITIONAL PARAMETERS IN USER-DEFINED FUNCTIONS

Earlier in this chapter, you saw various examples of passing values to shell functions. The following list of positional parameters are useful when you write shell functions:

```
$# contains the number of arguments
$0 contains the command name
$1, $2, ... , $9 contain the individual arguments of
the command
$* contains the entire list of arguments, treated as a
single word
$@ contains the entire list of arguments, treated as a
series of words
```

```
$? contains the exit status of the previous command,
and the value 0 denotes successful completion
$$ contains the process id of the current process
```

Listing 8.10 displays the contents of `PositionalParameters1.sh` that displays the values of the preceding positional parameters.

Listing 8.10 PositionalParameters1.sh

```
echo "number of arguments: $#"
echo "command name: $0"
echo "all params: $1 $2 $3 $4 $5 $6 $7 $8 $9"
echo "all params: $*"
echo "all params: $@"
echo "exit status: $?"
echo "process id: $$"

if [ x"$1" != "x" ]
then
  echo "Position parameter #1 = $1"
else
  echo "Position parameter #1 is null"
fi

if [ "$5" == "" ]
then
  echo "Position parameter #5 is nul1"
fi

case $1 in
  n|N) echo "#1 is an n or N" ;;
  y*|Y*) echo "#1 starts with a y or Y" ;;
  *) echo "no matches occurred" ;;
esac
```

Launch Listing 8.10 with the following command:

```
./PositionalParameters1.sh yes 2 3 4
```

The output is shown here:

```
./PositionalParameters1.sh 2 3 4
number of arguments: 3
command name: ./PositionalParameters1.sh
all params: 2 3 4
all params: 2 3 4
all params: 2 3 4
exit status: 0
process id: 58003
```

```
Position parameter #1 = 2
Position parameter #5 is nul1
no matches occurred
```

SHELL SCRIPTS, FUNCTIONS, AND USER INPUT

Listing 8.11 displays the contents of `checkuser.sh` that illustrates how to prompt users for two input strings and then pass those two strings as parameters to a custom function `checkNewUser()`.

Listing 8.11: checkuser.sh

```
#!/bin/bash

function checkNewUser()
{
  echo "argument #1 = $1"
  echo "argument #2 = $2"
  echo "arg count   = $#"

  if test "$1" = "John" && test "$2" = "Smith"
  then
    return 1
  else
    return 0
  fi
}

/bin/echo -n "First name: "
read fname
/bin/echo -n "Last name: "
read lname

checkNewUser $fname $lname
echo "result = $?"
```

Listing 8.11 contains the function `checkNewUser()` that displays the value of the first argument, the second argument, and the total number of arguments, respectively. This function returns the value 1 if the first argument is `John` and the second argument is `Smith`; otherwise, the function returns 0.

The remaining portion of Listing 8.11 invokes the echo command twice in order to prompt users to enter a first name and the last name and then invokes the function `checkNewUser()` with these two input values. A sample output from launching Listing 8.11 is shown here:

```
First name: John
Last name: Smith
argument #1 = John
```

```
argument #2 = Smith
arg count   = 2
result = 1
```

What about using command substitution in order to invoke the function `checkNewUser`? In order to find out what would happen, let's add the following code snippet to the bottom of Listing 8.11:

```
result=`checkNewUser $fname $lname`
echo "result = $result"
```

Launch the modified version of Listing 8.11, provide the same input values of John and Smith, and compare the following result with the previous result:

```
First name: John
Last name: Smith
argument #1 = John
argument #2 = Smith
arg count   = 2
result = 1
result = argument #1 = John
argument #2 = Smith
arg count   = 2
```

As another example of a simple shell script, the following script uses the `read` command which takes the input from the keyboard and assigns that input value as the value of the variable `PERSON`. The `echo` command prints the input value on `STDOUT`, which is the screen (by default).

```
#!/bin/sh
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
Here is sample invocation of this script:
$./test.sh
What is your name?
John Smith
Hello, John Smith
```

RECURSION AND SHELL SCRIPTS

This section contains several examples of shell scripts with recursion, which is a topic that occurs in many programming languages. Although you probably won't need to write many scripts that use recursion, it's worthwhile to learn this concept, especially if you plan to study other languages.

If you already understand recursion, then the scripts in this section will be straightforward. In particular, you will learn how to calculate the factorial value of a positive integer.

Listing 8.12 displays the contents of `Factorial.sh` that computes the factorial value of a positive integer.

Listing 8.12: Factorial.sh

```
#!/bin/sh

factorial()
{
   if [ "$1" -gt 1 ]
   then
      decr=`expr $1 - 1`
      result=`factorial $decr`
      product=`expr $1 \* $result`
      echo $product
   else
      # we have reached 1:
      echo 1
   fi
}

echo "Enter a number: "
read num

# add code to ensure it's a positive integer

echo "$num! = `factorial $num`"
```

Listing 8.12 contains the `factorial()` function with conditional logic: if the first parameter is greater than 1, then the variable `decr` is initialized as 1 less than the value of $1, followed by initializing result with the recursive invocation of the `factorial()` function with the argument `decr`. Finally, this block of code initializes the product as the value of `$1` multiplied by the value of the result. Note that if the first parameter is not greater than 1, then the value 1 is returned.

The last portion of Listing 8.12 prompts users for a number and then the factorial value of that number is computed and displayed. For simplicity, non-integer values are not checked (you can try to add that functionality yourself).

Launch the code in Listing 8.12 and you will see the following output:

```
Enter a number:
7
7! = 5040
```

ITERATIVE SOLUTIONS FOR FACTORIAL VALUES

Listing 8.13 displays the contents of `Factorial2.sh` that computes the factorial value of a positive integer using a `for` loop.

Listing 8.13: Factorial2.sh

```
#!/bin/bash

factorial()
{
   num=$1
   result=1
   for (( i=2; i<=${num}; i++ ));
   do
     result=$((${result}*$i))
   done

   echo $result
}

printf "Enter a number: "
read num

echo "$num! = `factorial $num`"
```

Listing 8.13 contains a function called `factorial()` that initializes the variable `num` to the first argument passed into the function `factorial()`, followed by the variable `result` whose initial value is 1. The next portion of Listing 8.13 is a `for` loop that iteratively multiples the value of result by the numbers between 2 and `num` inclusive, and then returns the value of the variable `result`.

The final portion of Listing 8.13 prompts users for a number and then uses command substitution to invoke the function `factorial()` with the user-supplied value. Note that no validation is performed in order to ensure that the input value is a non-negative integer. The `echo` statement displays the calculated factorial value.

Launch the code in Listing 8.13 and you will see the following output:

```
Enter a number: 8
8! = 40320
```

Listing 8.14 displays the contents of `Factorial3.sh` that computes the factorial value of a positive integer using a `for` loop and an array that keeps track of intermediate factorial values.

Listing 8.14: Factorial3.sh

```
#!/bin/bash

factorial()
{
   num=$1
   result=1
   for (( i=2; i<=${num}; i++ ));
```

```
   do
     result=$((${result}*$i))
     factvalues[$i]=$result
   done
}

printf "Enter a number: "
read num

for (( i=1; i<=${num}; i++ ));
do
  factvalues[$i]=1
done

factorial $num

# print each element via a loop:
for (( i=1; i<=${num}; i++ ));
do
  echo "Factorial of $i : " ${factvalues[$i]}
done
```

Listing 8.14 is very similar to the code in Listing 8.13: the key difference is that intermediate factorial values are stored in the array `factvalues`. Notice that the initial loop that initializes the values in `factvalues`: doing so makes the values global, so we don't need to return anything from the `factorial()` function.

The last portion of Listing 8.14 contains a `for` loop that displays the intermediate factorial values as well as the factorial of the user-provided input.

Launch the code in Listing 8.14 and you will see the following output:

```
Enter a number: 9
Factorial of 1 :  1
Factorial of 2 :  2
Factorial of 3 :  6
Factorial of 4 :  24
Factorial of 5 :  120
Factorial of 6 :  720
Factorial of 7 :  5040
Factorial of 8 :  40320
Factorial of 9 :  362880
```

CALCULATING FIBONACCI NUMBERS

In case you don't already know, the Fibonacci sequence of positive integers is defined as follows:

`F(1) = 1; F(2) = 2; and F(n) = F(n-1) + F(n-2) for n >= 2.`

Listing 8.15 displays the contents of `Fibonacci.sh` that computes the Fibonacci value of a positive integer.

Listing 8.15: Fibonacci.sh

```
#!/bin/sh
LOGFILE="/tmp/a1"
rm -f $LOGFILE 2>/dev/null

fib()
{
   if [ "$1" -gt 3 ]
   then
echo "1 = $1 2 = $2 3 = $3" >> $LOGFILE

      decr1=`expr $2 - 1`
      decr2=`expr $3 - 1`
      decr3=`expr $3 - 2`
echo "d1 = $decr1 d2 = $decr2 d3 = $decr3" >> $LOGFILE

      fib1=`fib $2 $3 $decr2`
      fib2=`fib $3 $decr2 $decr3`
      fib=`expr $fib1 + $fib2`
      echo $fib
   else
      if [ "$1" -eq 3 ]
      then
        echo 2
      else
        echo 1
      fi
   fi
}

echo "Enter a number: "
read num

# add code to ensure it's a positive integer
```

```
if [ "$num" -lt 3 ]
then
  echo "fibonacci $num = 1"
else
  decr1=`expr $num - 1`
  decr2=`expr $num - 2`x
  echo "fibonacci $num = `fib $num $decr1 $decr2`"
fi
```

Listing 8.15 contains code that decrements two variables, called `decr1` and `decr2` in order to make recursive invocations of the `fib()` function, whereas the calculation of factorial values involves decrementing only a single variable.

Moreover, Listing 8.15 contains a code snippet that writes intermediate calculations to a text file, which you can then examine to trace the execution path of the code. Launch the code in Listing 8.15 and you will see the following output:

```
Enter a number:
10
fibonacci 10 = 55
```

CALCULATING THE GCD OF TWO POSITIVE INTEGERS

Listing 8.16 displays the contents of the shell script `gcd.sh` that computes the greatest common divisor of two positive integers.

Listing 8.16: gcd.sh

```
#!/bin/sh

function gcd()
{
  if [ $1 -lt $2 ]
  then
    result=`gcd $2 $1`
    echo $result
  else
    remainder=`expr $1 % $2`

    if [ $remainder == 0 ]
    then
      echo $2
    else
      echo `gcd $2 $remainder`
    fi
```

```
  fi
}

a="4"
b="20"
result=`gcd $a $b`
echo "GCD of $a and $b = $result"

a="4"
b="22"
result=`gcd $a $b`
echo "GCD of $b and $a = $result"

a="20"
b="3"
result=`gcd $a $b`
echo "GCD of $b and $a = $result"

a="10"
b="10"
result=`gcd $a $b`
echo "GCD of $b and $a = $result"
```

Listing 8.16 is a straightforward implementation of the Euclidean algorithm (check Wikipedia for details) for finding the GCD of two positive integers. The output from Listing 8.16 shows the GCD of 4 and 20, as shown here:

```
GCD of 4 and 20 = 4
GCD of 22 and 4 = 2
GCD of 3 and 20 = 1
GCD of 10 and 10 = 10
```

CALCULATING THE LCM OF TWO POSITIVE INTEGERS

Listing 8.17 displays the contents of the shell script `lcm.sh` that computes the lowest common multiple (LCM) of two positive integers. This script contains the code in the shell script `gcd.sh` in order to compute the LCM of two positive integers.

Listing 8.17: lcm.sh

```
#!/bin/sh

function gcd()
{
  if [ $1 -lt $2 ]
```

```
  then
    result=`gcd $2 $1`
    echo $result
  else
    remainder=`expr $1 % $2`

    if [ $remainder == 0 ]
    then
      echo $2
    else
      result=`gcd $2 $remainder`
      echo $result
    fi
  fi
}

function lcm()
{
   gcd1=`gcd $1 $2`
   lcm1=`expr $1 / $gcd1`
   lcm2=`expr $lcm1 \* $2`
   echo $lcm2
}

a="24"
b="10"
result=`lcm $a $b`
echo "The LCM of $a and $b = $result"

a="10"
b="30"
result=`lcm $a  $b`
echo "The LCM of $a and $b = $result"
```

Notice that Listing 8.17 contains the `gcd()` function to compute the GCD of two positive integers. This function is necessary because the next portion of Listing 8.17 contains the `lcm()` function that invokes the `gcd()` function, followed by some multiplication steps in order to calculate the LCM of two numbers. The output from Listing 8.17 displays the LCM of 10 and 24, as shown here:

```
The LCM of 24 and 10 = 120
The LCM of 10 and 30 = 30
```

CALCULATING PRIME DIVISORS

Listing 8.18 displays the contents of the shell script `Divisors2.sh` that calculates the prime factors of a positive integer.

Listing 8.18: Divisors2.sh

```
#!/bin/sh

function divisors()
{
   div="2"
   num="$1"
   primes=""

   while (true)
   do
     remainder=`expr $num % $div`

     if [ $remainder == 0 ]
     then
      #echo "divisor: $div"
       primes="${primes} $div"
       num=`expr $num / $div`
     else
       div=`expr $div + 1`
     fi

     if [ $num -eq  1 ]
     then
       break
     fi
   done

   # use 'echo' instead of 'return'
   echo $primes
}

num="12"
primes=`divisors $num`
echo "The prime divisors of $num: $primes"

num="768"
primes=`divisors $num`
echo "The prime divisors of $num: $primes"
```

```
num="12345"
primes=`divisors $num`
echo "The prime divisors of $num: $primes"

num="23768"
primes=`divisors $num`
echo "The prime divisors of $num: $primes"
```

Listing 8.18 contains the `divisors()` function that consists primarily of a `while` loop that checks for the divisors of `num` (which is initialized as the value of `$1`). The initial value of `div` is 2, and each time `div` divides `num`, the value of `div` is appended to the primes string, and `num` is replaced by `num/div`. If `div` does not divide `num`, `div` is incremented by 1. Note that the `while` loop in Listing 8.18 terminates when `num` reaches the value of 1.

The output from Listing 8.18 displays the prime divisors of 12, 768, 12345, and 23768, as shown here:

```
The prime divisors of 12: 2 2 3
The prime divisors of 768: 2 2 2 2 2 2 2 2 3
The prime divisors of 12345: 3 5 823
The prime divisors of 23768: 2 2 2 2971
```

The prime factors of 12 and 678 are computed in under 1 second, but the calculation of the prime factors of 12345 and 23768 is significantly slower.

SUMMARY

In this chapter, you have some examples of how to use some useful and versatile `bash` commands. First, you saw examples of shell scripts for various tasks involving recursions, such as computing the GCD (greatest common divisor) and the LCM (lowest common multiple) of two positive integers, the Fibonacci value of a positive integer, and also the prime divisors of a positive integer.

CHAPTER 9

SHELL SCRIPTS WITH GREP *AND* AWK *COMMAND*

This chapter contains an assortment of bash scripts that illustrate how to solve some well-known tasks. Please make sure that you have read the earlier chapter pertaining to the grep command if you have not already done so.

The first part of this chapter shows you an assortment of bash scripts that use awk in order to perform various tasks, such as converting multiline records into single-line records. You will also learn how to compute the total of each row in a dataset.

The second part of this chapter shows you how to display the main diagonal and off-diagonal values, as well as the sum of those values.

One detail to keep in mind is that although some of the shell scripts in this chapter might not have immediate value for you, it's still worth your time to read them to see if they contain techniques that you can use in your own shell scripts.

THE GREP COMMAND WITH ZIP FILES

The first example in this section illustrates how to determine which zip files contain SVG documents. The second example in this section shows you how to check the entries in a log file (with simulated values). The third code sample shows you how to use the grep command in order to simulate a relational database consisting of three "tables", each of which is represented by a dataset.

Listing 9.1 displays the contents of myzip.sh that produces two lists of files: the first list contains the names of the zip files that contain SVG documents, and the second list contains the names of the zip files that do not contain SVG documents.

Listing 9.1: myzip.sh

```
foundlist=""
notfoundlist=""

for f in `ls *zip`
do
  found=`unzip -v $f |grep "svg$"`
  if [ "$found" != "" ]
  then
   #echo "$f contains SVG documents:"
   #echo "$found"
    foundlist="$f ${foundlist}"
  else
    notfoundlist="$f ${notfoundlist}"
  fi
done

echo "Files containing SVG documents:"
echo $foundlist| tr ' ' '\n'

echo "Files not containing SVG documents:"
echo $notfoundlist |tr ' ' '\n'
```

Listing 9.1 searches zip files for the hard-coded string `svg`: manually replace this hard-coded string with a different string of your choiee whenever you want to search a set of zip files for your specific string. Alternatively, you can prompt users for a search string so you don't need to make manual modifications to the shell script.

For your convenience, Listing 9.2 displays the contents of `searchstrings.sh` that illustrates how to enter one or more strings on the command line in order to search for those strings in the zip files in the current directory.

Listing 9.2: searchstrings.sh

```
foundlist=""
notfoundlist=""

if [ "$#" == 0 ]
then
   echo "Usage: $0 <string-list>"
   exit
fi

zipfiles=`ls *zip 2>/dev/null`
```

```
if [ "$zipfiles" = "" ]
then
   echo "*** No zip files in 'pwd' ***"
   exit
fi

for str in "$@"
do
  echo "Checking zip files for $str:"
  for f in `ls *zip`
  do
    found=`unzip -v $f |grep "$str"`
    if [ "$found" != "" ]
    then
      foundlist="$f ${foundlist}"
    else
      notfoundlist="$f ${notfoundlist}"
    fi
  done

  echo "Files containing $str:"
  echo $foundlist| tr ' ' '\n'

  echo "Files not containing $str:"
  echo $notfoundlist |tr ' ' '\n'
  foundlist=""
  notfoundlist=""
done
```

Listing 9.2 first checks that at least one file is specified on the command line, and then initializes the `zipfiles` variable with the list of zip files in the current directory. If `zipfiles` is null, an appropriate message is displayed.

The next section of Listing 9.2 contains a `for` loop that processes each argument that was specified at the command line. For each such argument, another `for` loop checks for the names of the zip files that contain that argument. If there is a match, then the variable `$foundlist` is updated, otherwise, the `$notfoundlist` variable is updated. When the inner loop has completed, the names of the matching files and the non-matching files are displayed, and then the outer loop is executed with the next command-line argument.

Although the preceding explanation might seem complicated, a sample output from launching Listing 9.2 will clarify how the code works:

```
./searchstrings.sh svg abc
Checking zip files for svg:
Files containing svg:
```

```
Files not containing svg:
shell-programming-manuscript.zip
shell-progr-manuscript-0930-2013.zip
shell-progr-manuscript-0207-2015.zip
shell-prog-manuscript.zip
Checking zip files for abc:
Files containing abc:

Files not containing abc:
shell-programming-manuscript.zip
shell-progr-manuscript-0930-2013.zip
shell-progr-manuscript-0207-2015.zip
shell-prog-manuscript.zip
```

```
  If you want to perform the search for zip files in sub-
directories, modify the loop as shown here:
for f in `find . -print |grep "zip$"`
do
   echo "Searching $f…"
   unzip -v $f |grep "svg$"
done
```

If you have Java SDK on your machine, you can also use the `jar` command instead of the `unzip` command, as shown here:

```
jar tvf $f |grep "svg$"
```

THE GREP COMMAND WITH MULTIPLE FILES

This section contains shell scripts that process simulated "products" that have SKU values, along with the price per unit and the number of units sold in order to calculate the revenue from selling those products. The text files with the product-related information are very small, which makes it easier to validate the accuracy of the shell scripts.

Listing 9.3, Listing 9.4, and Listing 9.5 display the contents of `skuvalues.txt`, `skuprices.txt`, and `skusold.txt`, respectively. These text files contain the SKU values, the prices for each product, and the number of units sold for each product.

Listing 9.3: skuvalues.txt

```
4520
5530
6550
7200
8000
```

Listing 9.4: skuprices.txt

```
4520 3.50
5530 5.00
6550 2.75
7200 6.25
8000 3.50
```

Listing 9.5: skusold.txt

```
4520 3.50
4520 50
5530 80
6550 115
7200 125
8000 150
```

Listing 9.6 displays the contents of `skutotals.sh` that calculates the number of units sold for each `SKU` in `skuvalues.txt`.

Listing 9.6: skutotals.sh

```
SKUVALUES="skuvalues.txt"
SKUSOLD="skusold.txt"

for sku in `cat $SKUVALUES`
do
  total=`cat $SKUSOLD |grep $sku | awk '{total += $2}
END {print total}'`
  echo "UNITS SOLD FOR SKU $sku: $total"
done
```

Listing 9.6 contains a `for` loop that iterates through the rows of the file `skuvalues.txt`, and passes those `SKU` values – one at a time – to a command that involves the `cat`, `grep`, and `awk` commands. The purpose of the latter combination of commands is three-fold:

1. find the matching lines in `skusold.txt`
2. compute the sum of the values of the numbers in the second column
3. print the subtotal for the current `SKU`.

In essence, this shell script prints the subtotals for each `SKU` value. Launch `skutotals.sh` and you will see the following output:

```
UNITS SOLD FOR SKU 4520: 50
UNITS SOLD FOR SKU 5530: 80
UNITS SOLD FOR SKU 6550: 115
UNITS SOLD FOR SKU 7200: 125
UNITS SOLD FOR SKU 8000: 150
```

We can generalize the previous shell script to take into account different prices for each SKU.

Listing 9.7 displays the contents of skutotals2.sh that extends the code in Listing 9.6 in order to calculate the revenue for each SKU.

Listing 9.7: skutotals2.sh

```
SKUVALUES="skuvalues.txt"
SKUSOLD="skusold.txt"
SKUPRICES="skuprices.txt"

forsku in `cat $SKUVALUES`
do
  skuprice=`grep $sku $SKUPRICES | cut -d" " -f2`
  subtotal=`cat $SKUSOLD |grep $sku | awk '{total +=
$2} END {print total}'`
  total=`echo "$subtotal * $skuprice" |bc`
  echo "AMOUNT SOLD FOR SKU $sku: $total"
done
```

Listing 9.7 contains a slight enhancement: instead of computing the subtotals of the number of units for each SKU, the *revenue* for each SKU is computed, where the revenue for each item equals the price of the SKU multiplied by the number of units sold for the given SKU. Launch skutotals2.sh and you will see the following output:

```
AMOUNT SOLD FOR SKU 4520: 175.00
AMOUNT SOLD FOR SKU 5530: 400.00
AMOUNT SOLD FOR SKU 6550: 316.25
AMOUNT SOLD FOR SKU 7200: 781.25
AMOUNT SOLD FOR SKU 8000: 525.00
```

Listing 9.8 displays the contents of skutotals3.sh that calculates the minimum, maximum, average, and the total number of units sold for each SKU in skuvalues.txt.

Listing 9.8: skutotals3.sh

```
SKUVALUES="skuvalues.txt"
SKUSOLD="skusold.txt"
TOTALS="totalspersku.txt"
rm -f $TOTALS 2>/dev/null

##############################
#calculate totals for each sku
##############################
```

```
for sku in `cat $SKUVALUES`
do
  total=`cat $SKUSOLD |grep $sku | awk '{total += $2}
END {print total}'`
  echo "UNITS SOLD FOR SKU $sku: $total"
  echo "$sku|$total" >> $TOTALS
done

##########################
#calculate max/min/average
##########################
awk -F"|" '
  BEGIN {first = 1;}
  {if(first) { min = max= avg = sum = $2; first=0;
next}}

  { if($2 < min) { min = $2 }
    if($2 > max) { max = $2 }
    sum += $2
  }
  END {print "Minimum = ",min
       print "Maximum = ",max
       print "Average = ",avg
       print "Total   = ",sum
  }
' $TOTALS
```

Listing 9.8 initializes some variables, followed by a `for` loop that invokes an `awk` command in order to compute subtotals (i.e., number of units sold) for each `SKU` value. The next portion of Listing 9.8 contains an `awk` command that calculates the maximum, minimum, average, and sum for the `SKU` units in the files `$TOTALS`.

Launch the script file in Listing 9.8 and you will see the following output:

```
UNITS SOLD FOR SKU 4520: 50
UNITS SOLD FOR SKU 5530: 80
UNITS SOLD FOR SKU 6550: 115
UNITS SOLD FOR SKU 7200: 125
UNITS SOLD FOR SKU 8000: 150
Minimum =  50
Maximum =  150
Average =  50
Total   =  520
```

SIMULATING RELATIONAL DATA WITH THE GREP COMMAND

This section shows you how to combine the `grep` and `cut` commands in order to keep track of a small database of customers, their purchases, and the details of their purchases that are stored in three text files.

Keep in mind that there are many open-source toolkits available that can greatly facilitate working with relational data and non-relational data. These toolkits can be very robust and also minimize the amount of coding that is required.

Moreover, you can use the `join` command (discussed in Chapter 2) to perform SQL-like operations on datasets. Nevertheless, the real purpose of this section is to illustrate some techniques with `grep` that might be useful in your own shell scripts.

Listing 9.9, Listing 9.10, and Listing 9.11 display the contents of the text files `MasterOrders.txt`, `Customers.txt`, and `PurchaseOrders.txt`, respectively.

Listing 9.9: MasterOrders.txt

```
M10000 C1000 12/15/2012
M11000 C2000 12/15/2012
M12000 C3000 12/15/2012
```

Listing 9.10: Customers.txt

```
C1000 John Smith LosAltos California 94002
C2000 Jane Davis MountainView California 94043
C3000 Billy Jones HalfMoonBay California 94040
```

Listing 9.11: PurchaseOrders.txt

```
C1000,"Radio",54.99,2,"01/22/2013"
C1000,"DVD",15.99,5,"01/25/2013"
C2000,"Laptop",650.00,1,"01/24/2013"
C3000,"CellPhone",150.00,2,"01/28/2013"
```

Listing 9.12 displays the contents of the `MasterOrders.shbash` script that performs various operations that involve the three preceding text files.

Listing 9.12: MasterOrders.sh

```
# initialize variables for the three main files
MasterOrders="MasterOrders.txt"
CustomerDetails="Customers.txt"
PurchaseOrders="PurchaseOrders.txt"

# iterate through the "master table"
for mastCustId in `cat $MasterOrders | cut -d" " -f2`
do
```

```
  # get the customer information
  custDetails=`grep $mastCustId $CustomerDetails`

  # get the id from the previous line
  custDetailsId=`echo $custDetails | cut -d" " -f1`

  # get the customer PO from the PO file
  custPO=`grep $custDetailsId $PurchaseOrders`

  # print the details of the customer
  echo "Customer $mastCustId:"
  echo "Customer Details: $custDetails"
  echo "Purchase Orders: $custPO"
  echo "----------------------"
  echo
done
```

Listing 9.12 initializes some variables for orders, details, and purchase-related datasets. The next portion of Listing 9.12 contains a `for` loop that iterates through the id values in the `MasterOrders.txt` file and uses each `id` to find the corresponding row in the `Customers.txt` file as well as the corresponding row in the `PurchaseOrders.txt` file. Finally, the bottom of the loop displays the details of the information that was retrieved from the initial portion of the `for` loop. The output from Listing 9.12 is here:

```
Customer C1000:
Customer Details: C1000 John Smith LosAltos California
94002
Purchase Orders: C1000,"Radio",54.99,2,"01/22/2013"
C1000,"DVD",15.99,5,"01/25/2013"
----------------------

Customer C2000:
Customer Details: C2000 Jane Davis MountainView
California 94043
Purchase Orders: C2000,"Laptop",650.00,1,"01/24/2013"
----------------------

Customer C3000:
Customer Details: C3000 Billy Jones HalfMoonBay
California 94040
Purchase Orders: C3000,"CellPho
ne",150.00,2,"01/28/2013"
----------------------
```

CHECKING UPDATES IN A LOGFILE

Listing 9.13 displays the contents of `CheckLogUpdates.sh` that illustrates how to periodically check the last line in a log file to determine the status of a system. This shell script simulates the status of a system by appending a new row that is based on the current timestamp. The shell script sleeps for a specified number of seconds, and on the third iteration, the script appends a row with an error status in order to simulate an error. In the case of a shell script that is monitoring a live system, the error code is obviously generated outside the shell script.

Listing 9.13: CheckLogUpdates.sh

```
DataFile="mylogfile.txt"
OK="okay"
ERROR="error"
sleeptime="2"
loopcount=0

rm -f $DataFile 2>/dev/null; touch $DataFile
newline="`date` SYSTEM IS OKAY"
echo $newline >> $DataFile

while (true)
do
  loopcount=`expr $loopcount + 1`

  echo "sleeping $sleeptime seconds..."
  sleep $sleeptime
  echo "awake again..."

  lastline=`tail -1 $DataFile`

  if [ "$lastline" == "" ]
  then
    continue
  fi

  okstatus=`echo $lastline |grep -i $OK`
  badstatus=`echo $lastline |grep -i $ERROR`
  if [ "$okstatus" != "" ]
  then
    echo "system is normal"
    if [ $loopcount -lt 5 ]
    then
       newline="`date` SYSTEM IS OKAY"
    else
       newline="`date` SYSTEM ERROR"
```

```
    fi
    echo $newline >> $DataFile
  elif [ "$badstatus" != "" ]
  then
    echo "Error in logfile: $lastline"
    break
  fi
done
```

Listing 9.13 initializes some variables and then ensures that the log file `mylogfile.txt` is empty. After an initial line is added to this log file, a `while` loop sleeps periodically and then examines the contents of the final line of text in the log file. New text lines are appended to this log file, and when an error message is detected, the code exits the `while` loop. A sample invocation of Listing 9.13 is here:

```
sleeping 2 seconds...
awake again...
system is normal
sleeping 2 seconds...
awake again...
system is normal
sleeping 2 seconds...
awake again...
system is normal
sleeping 2 seconds...
awake again...
system is normal
sleeping 2 seconds...
awake again...
system is normal
sleeping 2 seconds...
awake again...
Error in logfile: Thu Nov 23 18:22:22 PST 2017 SYSTEM
ERROR
```

The contents of the log file are shown here:

```
Thu Nov 23 18:22:12 PST 2017 SYSTEM IS OKAY
Thu Nov 23 18:22:14 PST 2017 SYSTEM IS OKAY
Thu Nov 23 18:22:16 PST 2017 SYSTEM IS OKAY
Thu Nov 23 18:22:18 PST 2017 SYSTEM IS OKAY
Thu Nov 23 18:22:20 PST 2017 SYSTEM IS OKAY
Thu Nov 23 18:22:22 PST 2017 SYSTEM ERROR
```

PROCESSING MULTILINE RECORDS

Listing 9.14 displays the contents of the dataset `multiline.txt` and Listing 9.15 displays the contents of the shell script `multiline.sh` that combines multiple lines into a single record.

Listing 9.14: multiline.txt

```
  Mary Smith
999 Appian Way
Roman Town, SF 94234

       Jane Adams
123 Main Street
Chicago, IL 67840

John Jones
321 Pine Road
Anywhere, MN 94949
```

Note that each record spans multiple lines that can contain whitespaces, and records are separated by a blank line.

Listing 9.15: multiline.sh

```
# Records are separated by blank lines
awk '
BEGIN { RS = "" ; FS = "\n" }
{
   gsub(/[ \t]+$/, "", $1)
   gsub(/[ \t]+$/, "", $2)
   gsub(/[ \t]+$/, "", $3)

   gsub(/^[ \t]+/, "", $1)
   gsub(/^[ \t]+/, "", $2)
   gsub(/^[ \t]+/, "", $3)

   print $1 ":" $2 ":" $3 ""
  #printf("%s:%s:%s\n",$1,$2,$3)
}
' multiline.txt
```

Listing 9.15 contains a `BEGIN` block that sets `RS` ("record separator") as an empty string and `FS` ("field separator") as a linefeed. Doing so enables us to "slurp" multiple lines into the same record, using a blank line as a separator for different records. The `gsub()` function removes leading and trailing

whitespaces and tabs for three fields in the datasets. The output from launching Listing 9.15 is here:

```
Mary Smith:999 Appian Way:Roman Town, SF 94234
Jane Adams:123 Main Street:Chicago, IL 67840
John Jones:321 Pine Road:Anywhere, MN 94949
```

ADDING THE CONTENTS OF RECORDS

Listing 9.16 displays the contents of the dataset `numbers.txt` and Listing 9.17 displays the contents of the shell script `sumrows.sh` that computes the row-size sum of each line in `numbers.txt`.

Listing 9.16: numbers.txt

```
1 2 3 4 5
6 7 8 9 10
5 5 5 5 5
```

Listing 9.17: sumrows.sh

```
awk '{ for(i=1; i<=NF;i++) j+=$i; print j; j=0 }'
numbers.txt
```

Listing 9.17 contains a simple invocation of the `awk` command that contains a `for` loop that uses the variable `j` to hold the sum of the values of the fields in each record, after which the sum is printed and `j` is re-initialized to 0. The output from Listing 9.17 is here:

```
15
40
25
```

USING THE `split` FUNCTION IN `awk`

Listing 9.18 displays the contents of the dataset `genetics.txt` (some rows wrap across more than one line) and Listing 9.19 displays the contents of the shell script `genetics.sh` that uses the `split()` function in order to parse the contents of a field in a record.

Listing 9.18: genetics.txt

```
#extract rows with 'gene' and print rows and 'key'
value
xyz3    GTF2GFF chro    55555   44444
key=chr1;Name=chr1
xyz3    GTF2GFF gene    77774   11111   key=XYZ123;NB=
standard;Name=extra
xyz3    GTF2GFF exon    71874   12227   Super=NR_55555
xyz3    GTF2GFF exon    72613   12721   Super=NR_55555
```

```
xyz3    GTF2GFF exon    83221   14408   Super=NR_55555
xyz3    GTF2GFF gene    84362   29370   key=WASH7P;Not
e=extra;Name=ALPHA
xyz3    GTF2GFF exon    84362   14829   Super=NR_222222
```

Listing 9.19: genetics.sh

```
# required output:
#xyz3:77774:XYZ123
#xyz3:84362:WASH7P

awk -F" " '
{
  if( $3 == "gene" ) {
    split($6, triplet, /[;=]/)
    printf("%s:%s:%s\n", $1, $4, triplet)
  }
}
' genetics.txt
```

Listing 9.19 matches input lines whose third field equals `gene`, after which the array triplet is populated with the components of the sixth field, using the characters ";" and "=" as delimiters in the sixth field. The output consists of the first field, the fourth field, and the second element in the array `triplet`. The output from launching Listing 9.19 is here:

```
xyz3:77774:XYZ123
xyz3:84362:WASH7P
```

SCANNING DIAGONAL ELEMENTS IN DATASETS

Listing 9.20 displays the contents of the dataset `diagonal.csv` and Listing 9.21 displays the contents of the shell script `diagonal.sh` that displays the elements in the main diagonal and off-diagonal, and also computes the sum of the elements in the main diagonal and off-diagonal. Although you re unlikely to need to perform such a task, the code might contain some techniques that are useful for your own shell scripts.

Listing 9.20: diagonal.csv

```
1,1,1,1,1
5,4,3,2,1
8,8,1,8,8
5,4,3,2,1
1,6,6,7,7
```

Listing 9.21: diagonal.sh

```
# NF is the number of fields in the current record.
# NR is the number of the current record/line
# (not the number of records in the file).
# In the END block (or the last line of the file)
# it's the number of lines in the file.
# Solution in R: https://gist.github.com/
dsparks/3693115

echo "Main diagonal:"
awk -F"," '{ for (i=0; i<=NF; i++) if (NR >= 1 && NR
== i) print $(i) }' diagonal.csv

echo "Off diagonal:"
awk -F"," '{print $(NF+1-NR)}' diagonal.csv

echo "Main diagonal sum:"
awk -F"," '
BEGIN { sum = 0 }
{
  for (i=0; i<=NF; i++) { if (NR >= 1 && NR == i) { sum
+= $i } }
}
END { printf ("sum = %s\n",sum) }
' diagonal.csv

echo "Off diagonal sum:"
awk -F"," '
BEGIN { sum = 0 }
{
  for (i=0; i<=NF; i++) { if(NR >= 1 && i+NR == NF+1)
{ sum += $i; } }
}
END { printf ("sum = %s\n",sum) }
' diagonal.csv
```

Listing 9.21 starts with an `awk` command that contains a loop that matches "diagonal" elements of the dataset, which is to say the first field of the first record, the second field of the second record, the third field of the third record, and so forth. This matching process is handled by the conditional logic inside the `for` loop.

The second part of Listing 9.21 contains an `awk` command that prints the off-diagonal elements of the dataset, using a very simple print statement.

The third part of Listing 9.21 contains an `awk` command that contains the same logic as the first `awk` command and then calculates the cumulative sum of the diagonal elements.

The fourth part of Listing 9.21 contains an `awk` command that contains logic that is similar to the first `awk` command, with the following variation:

```
if(NR >= 1 && i+NR == NF+1)
```

The preceding logic enables us to calculate the cumulative sum of the off-diagonal elements. The output from launching Listing 9.21 is here:

```
Main diagonal:
1
4
1
2
7
Off diagonal:
1
2
1
4
1
Main diagonal sum:
sum = 15
Off diagonal sum:
sum = 9
```

Listing 9.22, Listing 9.23, and Listing 9.24 display the contents of the `CSV` files `rain1.csv`, `rain2.csv`, and `rain3.csv.txt` that are used in several shell scripts in this section.

Listing 9.22: rain1.csv

```
1,0.10,53,15
2,0.12,54,16
3,0.19,65,10
4,0.25,86,23
5,0.18,57,17
6,0.23,79,34
7,0.34,66,21
```

Listing 9.23: rain2.csv

```
1,0.00,63,24
2,0.02,64,25
3,0.09,75,19
```

```
4,0.15,66,28
5,0.08,67,36
6,0.13,79,23
7,0.24,68,25
```

Listing 9.24: rain3.csv

```
1,1.00,83,34
2,0.02,84,35
3,1.09,75,19
4,0.15,86,38
5,1.08,87,36
6,0.13,79,33
7,0.24,88,45
```

ADDING VALUES FROM MULTIPLE DATASETS (1)

Listing 9.25 displays the contents of the shell script `rainfall1.sh` that adds the numbers in the corresponding fields of several CSV files and displays the results.

Listing 9.25: rainfall1.sh

```
# => Calculate COLUMN averages for multiple files

#columns in rain.csv:
#DOW,inches of rain, degrees F, humidity (%)

#files: rain1.csv, rain2.csv, rain3.csv
echo "FILENAMES:"
ls rain?.csv

awk -F',' '
{
  inches+=$2
  degrees+=$3
  humidity+=$4
}
END {
  printf("FILENAME: %s\n", FILENAME)
  printf("inches:   %.2f\n", inches/7)
  printf("degrees:  %.2f\n", degrees/7)
  printf("humidity: %.2f\n", humidity/7)
}
' rain?.csv
```

Listing 9.25 calculates the sum of the numbers in three columns (i.e., inches of rainfall, degrees Fahrenheit, and humidity as a percentage) in the datasets

specified by the expression `rain?.csv`, which in this particular example consists of the datasets `rain1.csv, rain2.csv,` and `rain3.csv`.

Thus, Listing 9.25 can handle multiple datasets (`rain1.csv` through `rain9.csv`). You can generalize this example to handle any dataset that starts with the string `rain` and ends with the suffix `csv` with the following expression:

```
rain*.csv
```

The output from launching Listing 9.25 is here:

```
FILENAMES:
rain1.csv     rain2.csv   rain3.csv
inches:   0.83
degrees:  217.71
humidity: 79.43
```

ADDING VALUES FROM MULTIPLE DATASETS (2)

Listing 9.26 displays the contents of the shell script `rainfall12.sh` that adds the numbers in the corresponding fields of several CSV files and displays the results.

Listing 9.26: rainfall2.sh

```
# => Calculate ROW averages for multiple files

#columns in rain.csv:
#DOW,inches of rain, degrees F, humidity (%)

#files: rain1.csv, rain2.csv, rain3.csv

awk -F',' '
{
  mon_rain[FNR]+=$2
  mon_degrees[FNR]+=$3
  mon_humidity[FNR]+=$4
  idx[FNR]++
}
END {
  printf("DAY INCHES DEGREES HUMIDITY\n")

    for(i=1; i<=FNR; i++){
    printf("%3d %-6.2f %-8.2f %-7.2f\n",
     i,mon_rain[i]/idx[i],mon_degrees[i]/idx[i],mon_
humidity[i]/idx[i])
  }
}
' rain?.csv
```

Listing 9.26 is similar to Listing 9.25, except that this code sample uses the value of FNR in order to calculate the average rainfall, degrees Fahrenheit, and percentage of humidity only for Monday. The output from launching Listing 9.26 is here:

```
DAY INCHES DEGREES HUMIDITY
  1 0.37   66.33    24.33
  2 0.05   67.33    25.33
  3 0.46   71.67    16.00
  4 0.18   79.33    29.67
  5 0.45   70.33    29.67
  6 0.16   79.00    30.00
  7 0.27   74.00    30.33
```

Listing 9.27, Listing 9.28, and Listing 9.29 display the contents of the dataset zain1.csv, zain2.csv, and rainz.csv.txt that are used in an upcoming shell script in this section.

Listing 9.27: zain1.csv

```
1,0.10,53,15
2,0.12,54,16
3,0.19,65,10
4,0.25,86,23
5,0.18,57,17
6,0.23,79,34
7,0.34,66,21
```

Listing 9.28: zain2.csv

```
1,0.00,63,24
2,0.02,64,25
3,0.09,75,19
4,0.15,66,28
5,0.08,67,36
6,0.13,79,23
7,0.24,68,25
```

Listing 9.29: zain3.csv

```
1,1.00,83,34
2,0.02,84,35
3,1.09,75,19
4,0.15,86,38
5,1.08,87,36
6,0.13,79,33
7,0.24,88,45
```

ADDING VALUES FROM MULTIPLE DATASETS (3)

Listing 9.30 displays the contents of the shell script `rainfall3.sh` that adds the numbers in the corresponding fields of several CSV files and displays the results.

Listing 9.30: rainfall3.sh

```
# => Calculate COLUMN averages for multiple files
(backtick)

#columns in rain.csv:
#DOW,inches of rain, degrees F, humidity (%)

# specify the list of CSV files (supports multiple
regexs)
files=`ls rain*csv zain*csv`

echo "FILES: `echo $files`"

awk -F',' '
{
  mon_rain[FNR]+=$2
  mon_degrees[FNR]+=$3
  mon_humidity[FNR]+=$4
  idx[FNR]++
}
END {
  printf("DAY INCHES DEGREES HUMIDITY\n")

  for(i=1; i<=FNR; i++){
    printf("%3d %-6.2f %-8.2f %-7.2f\n",
     i,mon_rain[i]/idx[i],mon_degrees[i]/idx[i],mon_
humidity[i]/idx[i])
  }
}
' `echo $files`
```

Listing 9.30 performs the same calculations as Listing 9.26, with the following variation: the datasets specified by the variable `files` that are defined by the regular expression `'ls rain*csv zain*csv'`. You can modify this regular expression to include any list of files that need to be processed. Notice that the final line of code in Listing 9.30 uses backtick substitution to expand the regular expression in the definition of the variable `files`:

```
' `echo $files`
```

As yet another variation, you can specify a file – let's call it filelist.txt - that contains a list of filenames that you want to process, and then replace the preceding line as follows:

```
' `cat filelist.txt`
```

The output from launching Listing 9.30 is here:

```
FILES: rain1.csv rain2.csv rain3.csv zain1.csv zain2.
csv zain3.csv
DAY INCHES DEGREES HUMIDITY
  1 0.37   66.33    24.33
  2 0.05   67.33    25.33
  3 0.46   71.67    16.00
  4 0.18   79.33    29.67
  5 0.45   70.33    29.67
  6 0.16   79.00    30.00
  7 0.27   74.00    30.33
```

CALCULATING COMBINATIONS OF FIELD VALUES

Listing 9.31 displays the contents of the shell script linear-combo.sh that computes various linear combinations of the columns in multiple datasets and displays one combined dataset as the output.

Listing 9.31: linear-combo.sh

```
# => combinations of columns
awk -F',' '
{
  $2 += $3 * 2 + $4 / 2
  $3 += $4 / 3 + $2 * $2 / 10
  $4 += $2 + $3
  $1 += $2 * 3 - $4 / 10
  printf("%d,%.2f,%.2f,%.2f\n",$1,$2,$3,$4)
}
' rain?.csv
```

Listing 9.31 processes the values of the datasets rain1.csv, rain2.csv, and rain3.csv whose contents are shown earlier in this section. The key observation to make is that the sequence of calculations in the calculations in the body of the awk statement involved inter-dependencies.

Specifically, the value of $2 is a linear combination of the values of $3 and $4. Next, the value of $3 is a linear combination of the value of $4 and $2, where the latter is *not* the original value from the datasets, but its calculated value. Third, the value of $4 is a linear combination of $2 and of $3, both of which are calculated values and not the values in the datasets. Finally, the value of $1 is a linear combination of the newly calculated values for $2 and $4.

As you can see, `awk` provides the flexibility to specify practically any combination of calculations (including non-linear combinations) in a very simple and sequential fashion. The output of Listing 9.31 is here:

```
194,113.60,1348.50,1477.10
196,116.12,1407.72,1539.84
204,135.19,1895.97,2041.16
187,183.75,3470.07,3676.82
202,122.68,1567.70,1707.38
194,175.23,3160.89,3370.12
207,142.84,2113.33,2277.17
201,138.00,1975.40,2137.40
202,140.52,2046.92,2212.44
201,159.59,2628.23,2806.82
203,146.15,2211.32,2385.47
203,152.08,2391.83,2579.91
199,169.63,2964.10,3156.73
206,148.74,2288.69,2462.43
183,184.00,3479.93,3697.93
182,185.52,3537.43,3757.95
200,160.59,2660.25,2839.84
179,191.15,3752.50,3981.65
178,193.08,3826.99,4056.07
195,174.63,3139.56,3347.19
173,198.74,4052.76,4296.50
```

SUMMARY

In this chapter, you have some examples of how to create some useful `bash` commands. First, you saw a `bash` script for handling text files containing multiline records. Next, you saw how to compute cumulative totals of numeric fields in records. Then you learned how to use the `split()` function inside a shell script with an `awk` command.

Then you saw how to use `awk` to process records that span multiple datasets in order to compute averages and total values. The multi-dataset tasks enabled you to learn more sophisticated shell scripts that operate on a set of "relational" tables, presented in the form of text files. Finally, you learned how to dynamically calculate various combinations of columns of numbers from multiple datasets.

CHAPTER 10

MISCELLANEOUS SHELL SCRIPTS

This chapter contains an eclectic assortment of `bash` scripts that illustrate how to perform various tasks involving multiple files and directories, working with compressed files, background processes, and printing simple reports.

The first part of this chapter shows you how to selectively copy and delete files from a directory tree. The second part of this chapter contains an assortment of shell scripts that create sub-directories in a directory, based on a set of strings. One of the scripts shows you how to check whether or not a given string is an existing sub-directory or a file.

The third part of this chapter discusses various commands for handling zip files, such as `gzip`, `gunzip`, and so forth. (See Chapter 4 for examples of the `tar` command for creating archive files.) The fourth part of this chapter shows you how to schedule tasks using the commands `at` and `crontab`. Next, you will learn about print-related commands and how to generate simple reports.

USING RM AND MV WITH DIRECTORIES

Before we look at the shell scripts, there are some simple tasks that you can perform with the `rm` command. For example, if you want to remove the directory `my_stuff` and all its contents, you can do so with this command:

```
rm -r my_stuff
```

Alternatively, you can use the `mv` command to move the directory `my_stuff` to the `/tmp` directory:

```
mv my_stuff /tmp
```

However, the preceding command works at most once; if the directory already exists in `/tmp`, then the preceding `mv` command will fail.

If you want to clone the entire contents of the directory `my_stuff` (including all the sub-directories) into the directory `my_stuff2`, use this command:

```
cp –r my_stuff my_stuff2
```

You can also use the `cp` or `mv` command in conjunction with back substitution to copy (or move) a subset of files in multiple sub-directories.

For example, suppose you have a directory with multiple sub-directories that contain Word documents and text files, as shown here:

```
myfiles/wordfiles/chapter1/chapter1.doc
myfiles/wordfiles/chapter2/chapter2.doc
myfiles/wordfiles/chapter3/chapter3.doc
myfiles/wordfiles/chapter4/chapter4.doc
myfiles/wordfiles/chapter1/data1.txt
myfiles/wordfiles/chapter2/data2.txt
myfiles/wordfiles/chapter3/data3.txt
myfiles/wordfiles/chapter4/data4.txt
```

Listing 10.1 displays the contents of the shell script `maketree.sh` that creates a set of sub-directories in the current directory.

Listing 10.1: maketree.sh

```
# remove myfiles
rm –r myfiles 2>/dev/null

# create subdirectories
mkdir -p myfiles/wordfiles/chapter1
mkdir -p myfiles/wordfiles/chapter2
mkdir -p myfiles/wordfiles/chapter3
mkdir -p myfiles/wordfiles/chapter4

# create empty files
touch myfiles/wordfiles/chapter1/chapter1.doc
touch myfiles/wordfiles/chapter2/chapter2.doc
touch myfiles/wordfiles/chapter3/chapter3.doc
touch myfiles/wordfiles/chapter4/chapter4.doc
touch myfiles/wordfiles/chapter1/data1.txt
touch myfiles/wordfiles/chapter2/data2.txt
touch myfiles/wordfiles/chapter3/data3.txt
touch myfiles/wordfiles/chapter4/data4.txt
```

Listing 10.1 removes the directory `myfiles`, and then uses the `mkdir -p` command to create a set of sub-directories, followed by the `touch` command to create empty Word documents and empty text files.

USING THE FIND COMMAND WITH DIRECTORIES

First, make sure that you launch the shell script `mktree.sh` in the preceding section in order to create the specified directory structure with the specified files.

The following command finds all the Word documents in the `myfiles` directory:

```
find myfiles -name "*.doc"
```

The following command copies all the Word documents from the `myfiles` directory to the /tmp directory:

```
cp `find myfiles -name "*.doc"` /tmp
```

You can perform similar operations with the text files by replacing occurrences of `*.doc` with `*.txt`.

Alternatively, the following command uses a combination of the `cp`, `find`, and `grep` command to copy all the Word documents from the `myfiles` directory to the `/tmp` directory:

```
cp `find myfiles |grep "\.doc$"` /tmp
```

Notice the pattern `"\.doc$"` in the `grep` command, which ensures that only files with the suffix `.doc` are found by the `grep` command. This pattern prevents the file (or directory) `mydoc_stuff` or `mydoc` from matching in the `grep` command.

However, there is one other point to keep in mind: it's possible to create the *directory* called `my.doc`, which will also match in the `grep` command. In this scenario, the `cp` command will display an error message that the directory `my.doc` was not copied.

CREATING A DIRECTORY OF DIRECTORIES

Listing 10.2 displays the contents of the shell script `makedirs.sh` that creates a set of sub-directories in the `names` directory.

Listing 10.2: makedirs.sh

```
#####################################################
# Define a variable with a list of directory names
# This method does not depend on an external file.
# Alternatively, place these names in a text file
# and then read the contents of that text file.
#####################################################

# change the value of mydir to whatever you need
mydir="names"

name_list="andrew-webber dave-jones jane-smith john-
smith keith-thompson"
```

```
if [ ! -d $mydir ]
then
  mkdir $mydir
  cd $mydir

  for name in `echo $name_list`
  do
    echo "creating directory $name in $mydir"
    mkdir $name
  done
else
  echo "Directory $mydir exists"
fi
```

Listing 10.2 initializes the variable `name_list` with a hard-coded list of names, each of which becomes a sub-directory of `$mydir`. The next portion of Listing 10.2 contains `if/else` logic to determine whether or not the `$mydir` directory exists. If it does not, then a loop iterates through the strings in `$name_list` and creates a corresponding sub-directory of `$mydir`.

Launch the code in Listing 10.2 and you will see the following output:

```
creating directory andrew-webber in names
creating directory dave-jones in names
creating directory jane-smith in names
creating directory john-smith in names
creating directory keith-thompson in names
```

However, if you invoke the code in Listing 10.2 a second time, you will see the following output:

```
Directory names exists
```

CLONING A SET OF SUB-DIRECTORIES

Listing 10.3 displays the contents of the directory `names` and Listing 10.4 displays the contents of the shell script `clonedirs1.sh` that populates the directory `names2` with the directories in the `names` directory.

Listing 10.3: names

```
john-smith
jane-smith
dave-jones
andrew-webber
keith-thompson
```

Listing 10.4: clonedirs1.sh

```
# make sure that you "cd" into the "names2" directory
for d in `ls ../names`
do
   echo "Creating directory $d"
   mkdir $d
done
```

Listing 10.4 contains a simple `for` loop that iterates through the contents of the (sibling) directory `names`, displays a message, and creates a corresponding directory in the `names2` directory.

Launch the code in Listing 10.4 and you will see the following output:

```
Creating directory andrew-webber
Creating directory dave-jones
Creating directory jane-smith
Creating directory john-smith
Creating directory keith-thompson
```

However, if you invoke the code in Listing 10.4 a second time, you will see multiple errors because the directories already exist, as shown here:

```
Creating directory andrew-webber
mkdir: andrew-webber: File exists
Creating directory dave-jones
mkdir: dave-jones: File exists
Creating directory jane-smith
mkdir: jane-smith: File exists
Creating directory john-smith
mkdir: john-smith: File exists
Creating directory keith-thompson
mkdir: keith-thompson: File exists
```

Listing 10.5 displays the contents of the shell script `clonedirs2.sh` that checks whether or not a directory already exists before populating the directory `names2` with the directories in the `names` directory.

Listing 10.5: clonedirs2.sh

```
# make sure that you "cd" into the "names2" directory
for d in `ls ../names`
do
  if [ -d $d ]
  then
    echo "Directory $d already exists"
  else
```

```
    echo "Creating directory $d"
    mkdir $d
  fi
done
```

Listing 10.5 contains a simple `for` loop that iterates through the contents of the (sibling) directory names. The `if/else` code block checks if the directory already exists; if so, a message is displayed; if not, the directory is created inside the `names2` directory.

Launch the code in Listing 10.5 and you will see the following output:

```
Directory andrew-webber already exists
Directory dave-jones already exists
Directory jane-smith already exists
Directory john-smith already exists
Directory keith-thompson already exists
```

Listing 10.6 displays the contents of the shell script `clonedirs3.sh` that first checks whether a file in `names2` is a file instead of a directory: if so, no directory is created. Next, if the file in `names2` is, in fact, a directory, the code checks whether or not a directory already exists before populating the directory `names2` with the directories in the `names` directory.

Listing 10.6: clonedirs3.sh

```
# make sure that you "cd" into the "names2" direc-
tory
for d in `ls ../names`
do
  if [ -f $d ]
  then
    echo "$d is a file (not a directory)"
  elif [ -d $d ]
  then
    echo "Directory $d already exists"
  else

    echo "Creating directory $d"
    mkdir $d
  fi
done
```

Listing 10.6 contains a simple `for` loop that iterates through the contents of the (sibling) directory names. The `if/else` code block checks if the directory already exists; if so, a message is displayed; if not, the directory is created in the `names2` directory.

Launch the code in Listing 10.6 and you will see the following output:

```
Directory andrew-webber already exists
Directory dave-jones already exists
Directory jane-smith already exists
Directory john-smith already exists
Directory keith-thompson already exists
```

Listing 10.7 displays the contents of the shell script `copy-file1.sh` that first checks whether a file in `names2` is a file instead of a directory: if so, no copy is created. Next, if the file in `names2` is, in fact, a directory, the code copies `file1` into this directory. If the file does not exist, the code creates a directory and copies `file1` into the directory.

Listing 10.7: copy-file1.sh

```
# make sure that you "cd" into the "names2" directory

echo "hello world" >/tmp/file1
file1="/tmp/file1"

for d in `ls`
do
  if [ -f $d ]
  then
    echo "Skipping copy command for $d"
  elif [ -d $d ]
  then
    echo "Copying $file1 into $d"
  else
    echo "Creating directory $d"
    mkdir $d
    echo "Copying $file1 into $d"
  fi
done
```

Listing 10.7 contains a simple `for` loop that iterates through the contents of the (sibling) directory names. The `if/else` code block checks if the directory already exists; if so, a message is displayed; if not, the directory is created in the `names2` directory.

Launch the code in Listing 10.7 and you will see the following output:

```
Copying /tmp/file1 into andrew-webber
Skipping copy command for clone-dirs1.sh
Skipping copy command for clone-dirs2.sh
Skipping copy command for clone-dirs3.sh
```

```
Skipping copy command for copy-file1.sh
Copying /tmp/file1 into dave-jones
Copying /tmp/file1 into jane-smith
Copying /tmp/file1 into john-smith
Copying /tmp/file1 into keith-thompson
```

EXECUTING FILES IN MULTIPLE DIRECTORIES

Suppose that you have executable shell scripts in multiple directories, and you want to execute all of them. One solution involves keeping track of the directories and the executable files in each of those directories, and then navigating to those directories to perform the necessary action.

Listing 10.8 displays the contents of the shell script `remove-files.sh` that removes different files based on the current directory.

Listing 10.8: remove-files.sh

```
# keep track of the top directory
topdir=`pwd`

for f in `ls`
do
  if [ -d $f ]
  then
    cd $topdir/$f

    if [ -f "remove.sh" ]
    then
      echo "executing remove.sh in $f"
      sh remove.sh
    else
      echo "cannot find remove.sh in $f"
    fi

    cd $topdir
  fi
done
```

Listing 10.8 assigns the current directory to the variable `topdir`, followed by a loop that processes the contents of the sub-directories of `topdir`. Notice the `if/else` logic to ensure that the current value of `$f` is actually a directory.

If `$f` is a directory, then perform a `cd` command into that directory. Note that the inner `if/else` logic checks that the shell script `remove.sh` exists before executing this shell script.

Another point to notice is the initial `sh` in the command `remove.sh` (shown in bold), which enables us to execute commands even if they do not have to execute permissions.

Now launch the code in Listing 10.8 and you will see the following output:

```
executing remove.sh in headers
removing text files
executing remove.sh in src
removing object files
```

THE CASE/ESAC COMMAND

As you already know, the `case/esac` command enables you to perform different actions that are based on the value of a user's input. Just to refresh your memory, here is a sample of the syntax of the `case` statement:

```
case $option in
  1) echo "Starting system backup...";;
  2) echo "Reading tape drive...";;
  x) echo "Are you really sure? (Y/n)"
esac
```

In general, you would display a menu of options and prompt users for their choice, and then execute the appropriate command.

Listing 10.9 displays the contents of the shell script `case-menu.sh` that displays a set of options and prompts users for their input.

Listing 10.9: case-menu.sh

```
menu()
{
  echo "1) Perform system backup"
  echo "2) Read tape drive"
  echo "x) Exit System"
  echo ""
  echo "Enter option:"
}

process_option()
{
  case $option in
    1) echo "Starting system backup...";;
    2) echo "Reading tape drive...";;
    x) echo "Are you really sure? (Y/n)"
       read val
       val=$val | tr '[:upper:]' '[:lower:]'
```

```
        if [ "$val" = "y" ]
        then
          echo "Exiting system ... goodbye"
          exit
        fi ;;
     *) echo "Exiting system ... goodbye"
        exit ;;
  esac
}

while(true)
do
  menu
  read option
  process_option
done
```

Listing 10.9 defines the `menu()` function that displays a set of options, followed by the `process_option()` function that processes the input that users have provided.

The response for the value 1 or the value 2 is straightforward, and you need to insert code that would actually do something.

The response for the value `x` is more complex, and it illustrates the flexibility of the `case/esac` statement. You can execute whatever you need to do in this option. In this example, the input value is converted to a lowercase value: if it equals lowercase `x`, then the program exits. Notice that the program also exists for all other input values.

The final portion of Listing 10.9 contains a loop that 1) displays the menu, 2) prompts users for an input value, and 3) processes the users' input value.

Launch the code in Listing 10.9 and select all the menu options, as shown here:

```
1) Perform system backup
2) Read tape drive
x) Exit System

Enter option:
1
Starting system backup...
1) Perform system backup
2) Read tape drive
x) Exit System

Enter option:
```

```
2
Reading tape drive...
1) Perform system backup
2) Read tape drive
x) Exit System

Enter option:
x
Are you really sure? (Y/n)

1) Perform system backup
2) Read tape drive
x) Exit System

Enter option:
4
Exiting system
```

COMPRESSING/UNCOMPRESSING FILES

This brief section contains a description of an assortment of commands for compressing and uncompressing files:

```
Compress:   Compress files
gunzip:     Uncompress gzipped files
gzip:       GNU alternative compression method
uncompress: Uncompress files
unzip:      List, test and extract compressed files in
a ZIP archive
zcat:       Cat a compressed file
zcmp:       Compare compressed files
zdiff:      Compare compressed files
zmore:      File perusal filter for crt viewing of
compressed text
```

The `uuencode` utility encodes binary files (images, sound files, compressed files, etc.) into ASCII characters, making them suitable for transmission as an attachment or even in the body of an email message. This is especially useful where `MIME` (multimedia) encoding is not available. The `uudecode` decodes the files created via the `uuencode` command.

THE DD COMMAND

The `man` page for the `dd` command describes `dd` as a utility that "copies standard input to standard output." Despite this innocuous description, the `dd` command is actually quite powerful, and can even convert ASCII characters

into EBCDIC, which is a data format for mainframes, and vice versa. The syntax for the `dd` command is as follows:

```
dd if=SOURCE of=TARGET bs=BLOCK_SIZE count=COUNT
```

The `if` and `of` options in the preceding snippet are for the input file and the output file, respectively. The `bs` option is for the block size (usually a multiple of 512), and `count` is an integer that specifies the number of blocks to be copied. Note that `bs` and `count` are both optional. If `count` is omitted, then `dd` copies data from the input file until it reaches the end of file (EOF) marker.

In order to copy a partition into a file, use this command:

```
dd if=/dev/sda1 of=sda1_partition.img
```

Note that `/dev/sda1` in the preceding snippet is the device path for the partition.

Restore the partition using the backup with this command:

```
dd if=sda1_partition.img of=/dev/sda1
```

Generate the file `data.file` of 100kb as follows:

```
dd if=/dev/zero bs=100k count=1 of=data.file
```

The preceding command creates a file `data.file` that is filled with zeros with a size of 100kb.

The next command creates the file `junk.data` that is exactly 1MB in size:

```
dd if=/dev/zero of=junk.data bs=1M count=1
1+0 records in
1+0 records out
1048576 bytes (1.0 MB) copied, 0.00767266 s, 137 MB/s
```

THE `crontab` COMMAND

The `crontab` command allows you to schedule the execution of tasks on a regular basis. For instance, you can schedule the execution of shell scripts on a flexible schedule instead of manually invoking those shell scripts.

Keep in mind that `crontab` tasks do *not* inherit the environment of a specific user, which means that you must ensure that all required environment variables are set properly (such as invoking a script that contains those variables).

You can schedule a task to run based on the following:

```
an hourly, daily, or weekly basis
a specific day of the month
a specific month or year
```

The following command displays the currently scheduled jobs for your machine:

```
crontab
```

You need to be either the root user or use `sudo` in order to modify `crontab` (in the `/usr/bin` directory) with the following command:

```
crontab -e
```

The following command replaces the current set of jobs with the jobs in the file `crontab.new`:

```
crontab –r < crontab.new
```

UNCOMPRESSING FILES AS A CRON JOB

Listing 10.10 displays the contents of `uncompress.sh` that illustrates how to uncompress a set of "zip" files in separate directories.

Listing 10.10 uncompress.sh

```
#!/bin/sh

zipfiles="`ls *zip`"

if [ "$zipfiles" != "" ]
then
  for f in `echo $zipfiles`
  do
    echo "Processing file: $f"
    f1=`echo $f | cut -d"." -f1`
    f2=`echo $f | cut -d"." -f2`
    newdir="${f1}-new"

    echo "Creating directory: $newdir"
    mkdir -p $newdir
    cp $f $newdir
    cd $newdir
    echo "Uncompressing file: $f"
    jar xvf $f
    cd ../
  done
else
  echo "No zip files found"
fi
```

Listing 10.10 initializes the variable `zipfiles` with the list of zip files in the current directory, followed by an `if/else` block that checks whether or not any zip files exist by checking the value of the variable `zipfiles`.

If `zipfiles` is non-empty, a `for` loop iterates through each file in the `zipfiles` variable, and then constructs a string called `newdir` by appending the string `–new` to the prefix `f1` of the current filename.

The next section in Listing 10.10 invokes the `mkdir -p` command in order to create a new directory named `newdir`. After invoking the cd command to enter this new directory, the current zip file (which is the value of the loop variable f) is uncompressed into this directory. Notice the invocation of `cd ..` (which is the last statement in the loop) in order to return to the parent directory.

SCHEDULED COMMANDS AND BACKGROUND PROCESSES

The section contains an assortment of `bash` commands that can be useful for various tasks and shell scripts. You can schedule a command to run at a specific time via the `at` command or via `cron`. You can also schedule a command to run in the background via the ampersand ("&") symbol.

How to Schedule Tasks

The `at` job control command executes a given set of commands at a specified time. Superficially, it resembles `cron`; however, the `at` command is chiefly useful for a one-time execution of a command set. For example, the following snippet prompts for a set of commands to execute at that time:

```
at 2pm January 15
```

These commands should be shell-script compatible because users are typing in an executable shell script one line at a time. Input terminates with a `ctrl-d`.

Use either the `-f` option or input redirection (<) when you need the `at` command to read a command list from a file.

The `batch` job control command is similar to the `at` command; however, it runs a command list when the system load is less than .8. The `batch` command can also read commands from a file with the `-f` option.

The `nohup` *Command*

The `nohup` ("no hangup") command enables you to execute another command and continue to execute that command even after you have logged out. Here is a simple example:

```
nohup myscript.sh
```

By default, both standard output and standard error (if any) are redirected to the text file `nohup.out` in the current directory.

Executing Commands Remotely

`Bash` supports remote execution of various commands. You can remotely log into another machine using the `rsh` command. If you want to use a secure connection, use the `ssh` instead of `rsh`.

After logging into a remote system, you can execute various commands between the two systems. For example, `rcp` is the counterpart to the `cp` command. Other commands include `rsync`, `ftp`, `ping`, and `telnet`. Perform an online search to learn the details of these commands.

How to Schedule Tasks in the Background

The & switch runs a process in the background. Note that & does *not* lower the priority of a currently executing process. This option is convenient for invoking two commands in the same shell:

```
find . -name "*txt" -print > text_files &
tail -f text_files
```

The preceding find command runs in the background as it searches for all files with the txt suffix and redirects the output to text_files. The tail command displays the contents of text_files whenever this file is updated.

You can bring the previous background process to the foreground with this command:

```
fg %1
```

HOW TO TERMINATE PROCESSES

The kill <number> PID command enables you to terminate tasks or processes by specifying a signal value for <number> and a process id for PID. The value of <number> can be 1, 2, 3, 6, 9, 14, or 15.

For example, both of the following commands will terminate a process with a "hangup" signal:

```
kill -1 2345
kill -s HUP 2345
```

You can terminate a process whose PID is 2345 with kill -3 2345 and kill -9 2345, which are signals QUIT and KILL, respectively. However, the signal QUIT can be "caught" via the "trap" command, whereas the signal KILL forces a process to terminate immediately (i.e., as soon as the operating system can execute this command). Moreover, the signal KILL cannot be caught via the "trap" command and it cannot be ignored.

NOTE *Only superuser can send a signal to another user's processes.*

Perform an online search for more details regarding the kill command, or check the man page with this command:

```
man kill
```

Terminating Multiple Processes

The kill <number> PID command enables you to terminate multiple processes by specifying a list of processes, an example of which is shown here:

```
kill -1 5000 5010 5135 68238
```

Another scenario involves multiple child processes that have the same parent process. For example, when you launch multiple browser sessions in Chrome or Firefox, each browser session will have an associated PID. There are two ways to terminate all the child processes and the parent process. One

technique is to terminate the parent process that appears in the `monitor` utility. However, if you need to terminate these processes from the command line, you can invoke the following command:

```
ps -ef |grep -i firefox | awk '{print $2}' | xargs kill -9
```

PROCESS-RELATED COMMANDS

The `ps` command displays information about processes on your machine. For example, if you type the following command at the command line:

```
ps –ef | more | head -7
```

you will see output that is similar to the following:

```
UID   PID  PPID   C STIME   TTY           TIME CMD
  0     1     0   0 Thu05AM ??        34:15.35 /sbin/
launchd
  0    12     1   0 Thu05AM ??         0:10.58 /usr/
libexec/kextd
  0    14     1   0 Thu05AM ??         1:44.60 /usr/
sbin/notifyd
  0    15     1   0 Thu05AM ??         2:18.59 /usr/
sbin/securityd -i
  0    17     1   0 Thu05AM ??        46:41.08 /usr/
libexec/configd
  0    18     1   0 Thu05AM ??         1:26.37 /usr/
sbin/syslogd
```

The `kill` command enables you to terminate processes. For example, the following command terminates the Chrome browser that is running on your machine:

```
kill -9 Chrome
```

If you execute commands in a shell script, you can find the process id of the currently executing process from `$$`, whereas `$!` contains the process id of the process that recently switched to the background.

The `uptime` command displays how long the system has been running, and sample output is here:

```
23:21  up 1 day, 12:59, 9 users, load averages: 2.60
3.26 3.42
```

How to Monitor Processes

The `top` command displays information about currently running tasks, such as the "top" users of the CPU, the amount of memory that they consume, priorities of processes, and so forth. A sample output from the `top` command is here:

```
Processes: 322 total, 2 running, 320 sleeping, 1382
threads            21:35:32
Load Avg: 2.08, 1.69, 1.44  CPU usage: 3.85% user,
2.65% sys, 93.49% idle
SharedLibs: 142M resident, 42M data, 22M linkedit.
MemRegions: 85784 total, 2587M resident, 57M private,
577M shared.
PhysMem: 8135M used (1543M wired), 55M unused.
VM: 1331G vsize, 634M framework vsize, 48085(0)
swapins, 136883(0) swapouts.
Networks: packets: 3488298/4488M in, 1873020/198M out.
Disks: 2042619/22G read, 346447/14G written.
PID   COMMAND      %CPU TIME     #TH   #WQ  #PORT MEM
PURG   CMPRS  PGRP PPID
5207  top          3.9  00:00.67 1/1   0    20    2892K+
0B     0B     5207 491
5092  cupsd        0.0  00:00.06 3     1    43    2420K
0B     0B     5092 1
5089- Office365Ser 0.0  00:00.21 8     2    154   4676K
0B     0B     5089 1
5084  quicklookd   0.0  00:00.16 4     1    86    4344K
32K    0B     5084 1
5043- mdworker32   0.0  00:00.46 3     1    51    5772K
0B     0B     5043 1
4989  mdworker     0.0  00:00.57 4     1    48    21M
0B     0B     4989 1
4943  netbiosd     0.0  00:00.06 2     2    30    500K
0B     1884K  4943 1
```

Perform an online search for more details about the options that are available for the `top` command.

CHECKING EXECUTION RESULTS

Whenever you need to determine the status of the most recently executed command, check the value of `$?`. This term is assigned the value that is specified by the `exit` statement of the most recently executed command. By convention, an exit value of 0 indicates successful execution, and non-zero values indicate a non-successful result.

Listing 10.11 displays the contents of `parent.sh` that illustrates how to check the result of executing a shell script from inside another shell script.

Listing 10.11 parent.sh

```
#check result of shell script

echo "first time:"
./cmdargs.sh pasta
echo "result = $?"

echo "second time:"
./cmdargs.sh
echo "result = $?"

echo "third time:"
cmdargs.sh
echo "result = $?"
```

Listing 10.11 invokes the shell script `cmdargs.sh` three times: the first time with the parameter `pasta`, and the subsequent invocations with no parameters. Each time that `cmdargs.sh` is launched, the value of `$?` is displayed in the current shell script, which is the exit status of `cmdargs.sh`.

Listing 10.12 displays the contents of `cmdargs.sh` that illustrates how to check the result of executing a shell script from inside another shell script.

Listing 10.12 cmdargs.sh

```
if [ $# -eq 0 ]
then
  echo "Usage: $0 <filename>"
  exit 1
fi

exit 0
```

Listing 10.12 checks if any command-line arguments are specified when `cmdargs.sh` is launched from the command line. If not, a message is displayed and the shell script terminates with an exit value of 1. Otherwise, the shell script terminates with an exit value of 0.

Launch the preceding shell script as follows:

```
./cmdargs.sh
The output is shown here:
first time:
result = 0
second time:
Usage: ./cmdargs.sh <filename>
result = 1
third time:
Usage: ./cmdargs.sh <filename>
result = 1
```

SYSTEM MESSAGES AND LOG FILES

Bash provides the dmesg command to display system-related messages. A sample output from the dmesg command is here:

```
hibernate image path: /var/vm/sleepimage
AirPort_Brcm43xx::powerChange: System Sleep
hibernate_alloc_pages act 865005, inact 768286, anon
589882, throt 0, spec 30749, wire 407619, wireinit
241382
hibernate_setup(0) took 0 ms
sizeof(IOHibernateImageHeader) == 512
kern_open_file_for_direct_io(0) took 249 ms
Opened file /var/vm/sleepimage, size 8589934592,
partition base 0x0, maxio 400000 ssd 0
hibernate image major 1, minor 0, blocksize 512,
pollers 5
en1: BSSID changed to 68:7f:74:cb:05:f8
wlEvent: en1 en1 Link DOWN virtIf = 0
AirPort: Link Down on en1. Reason 8 (Disassociated
because station leaving).
en1::IO80211Interface::postMessage bssid changed
LE is supported - Disable LE meta event
        0 [Time 1380032186] [Message hibernate_page_
list_setall(preflight 0) start 0xffffff80fb3d5000,
0xffffff80fb433000
```

The /var directory on your machine contains various files and directories. A sample of its contents is here:

```
en1::IO80211Interface::postMessage bssid changed
total 10872
-rw-r--r--@  1 root            wheel       12 Sep 12
2012 CDIS.custom
drwxrwx---  33 root            admin     1122 Sep 24
02:11 DiagnosticMessages
```

As you can see, the root user is the owner of many of these directories, so you need to use sudo in order to view the contents of those directories.

A portion of the aptly named file /var/log/zzz.log is here:

```
Wed Sep 12 14:22:43 201 [SleepServicesD] /SourceCache/
SleepServicesD_executables/SleepServicesD-1.43/
SleepServicesD/ModeConfig.m:41 Waiting for
IOPlatformPluginFamily to load ...
Wed Sep 12 14:22:48 201 [SleepServicesD] /SourceCache/
SleepServicesD_executables/SleepServicesD-1.43/
```

```
SleepServicesD/ModeConfig.m:41 Waiting for
IOPlatformPluginFamily to load ...
Wed Sep 12 14:22:49 201 [SleepServicesD] /SourceCache/
SleepServicesD_executables/SleepServicesD-1.43/
SleepServicesD/ModeConfig.m:41 Waiting for
IOPlatformPluginFamily to load ...
```

Disk Usage Commands

The `du` command displays "disk usage" information and the following invocation displays the number of kilobytes for the files in the current directory:

```
du –s –k .
```

Replace "-k" with "-m" to see the number of megabytes.

The `df` command shows the "disk free" space available, and a sample invocation is here:

```
Filesystem 1024-blocks      Used Available Capacity
iused   ifree %iused  Mounted on
/dev/disk1   487184384 462012772  24915612    95%
115567191 6228903   95%   /
```

The `df` command with the –h option produces this type of output:

```
Filesystem      Size   Used  Avail Capacity  iused
ifree %iused  Mounted on
/dev/disk1     893Gi  800Gi   92Gi    90% 17489436
4277477843    0%   /
devfs          191Ki  191Ki    0Bi   100%      661
0  100%   /dev
map -hosts       0Bi    0Bi    0Bi   100%        0
0  100%   /net
map auto_home    0Bi    0Bi    0Bi   100%        0
0  100%   /home
```

TRAPPING AND IGNORING SIGNALS

Listing 10.13 displays the contents of `trap1.sh` that illustrates how to trap a signal in `bash`. In this example, a message is displayed when users resize their command shell in less than 10 seconds.

Listing 10.13 Trap1.sh

```
#!/bin/sh

maxCount="10"

screenchange()
{
   echo "Caught a signal from the trap statement"
}
```

```
echo "Resize the current command shell."
trap screenchange SIGWINCH

COUNT=0
while [ $COUNT -lt $maxCount ] ; do
  COUNT=$(($COUNT + 1))
  sleep 1
done
```

Listing 10.13 initializes the variable `maxCount` with the value 10, followed by the definition of the `screenchange` function, which merely displays a message via the echo statement.

Next, the `echo` statement prompts users to resize their command shell, followed by the `trap` statement that invokes the `screenchange` function if the command shell is resized in less than 10 seconds.

The final portion of Listing 10.13 is a `for` loop that iterates 10 times (i.e., the value assigned to the variable `maxCount`), and sleeps for one second during each iteration.

Listing 10.14 displays the contents of `trap2.sh` that illustrates how to ignore a signal in a shell script.

Listing 10.14 trap2.sh

```
#!/bin/sh
trap "" SIGINT
 echo "This program will sleep for 10 seconds and
cannot be killed with"
echo "control-c."
sleep 10
```

Listing 10.14 starts with the `trap` statement that "traps" the `SIGINT` signal, which occurs when you press `ctrl-c` at the command line. The next portion of Listing 10.14 displays a message and then sleeps for 10 seconds. Launch the code in Listing 10.14 and try to terminate the program by pressing `ctrl-c` at the command line.

ARITHMETIC WITH THE BC AND DC COMMANDS

The `bash` shell does not differentiate between strings and numbers (integers or real numbers). However, the `bc` command enables you to work with numbers, examples of which are shown here:

```
bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
```

```
4+3
7
4.5 + 3.7
8.2
```

Alternatively, you can enter the following commands from the command line:

```
echo "2+3" |bc
5
echo "4.5+3.7" |bc
8.2
```

You can assign values to variables and perform arithmetic operations with the `bc` command, as shown here:

```
x = 4
y = 7
echo "$x * $y" |bc
28
```

WORKING WITH THE DATE COMMAND

The `date` command provides information about the current date in various formats by specifying different command-line options. The `date` command can be useful for generating a quasi-random filename. Alternatively, you can also use the current process id that is stored in `$$`, as well as the `tempfile` command to generate filenames.

```
#!/bin/bash
# Exercising the 'date' command
echo "The number of days since the year's beginning is
`date +%j`."
# Needs a leading '+' to invoke formatting.
# %j gives day of year.
echo "The number of seconds elapsed since 01/01/1970
is `date +%s`."
#  %s yields number of seconds since "BASH epoch"
began,
#+ but how is this useful?

prefix=temp
suffix=$(date +%s)  # The "+%s" option to 'date' is
GNU-specific.
filename=$prefix.$suffix
echo "Temporary filename = $filename"
```

```
#  It's great for creating "unique and random" temp
filenames,
#+ even better than using $$.
$ date
Thu May 20 23:09:04 IST 2010
```

The epoch time can be printed as follows:

```
$ date +%s

1290047248
```

You can convert a date string into epoch as follows:

```
$ date --date "Thu Nov 18 08:07:21 IST 2010" +%s

1290047841
```

The `--date` option is used to provide a date string as input. However, we can use any date formatting options to print output. Feeding input date from a string can be used to find out the weekday, given the date. Here is a simple example:

```
$ date --date "Jan 20 2001" +%A

Saturday
```

Use a combination of format strings prefixed with + as an argument for the date command to print the date in the format of your choice. For example:

```
$ date "+%d %B %Y"

20 May 2010
```

We can set the date and time as follows:

```
date -s "Formatted date string"
```

An example of how to use the preceding syntax is here:

```
date -s "21 June 2009 11:01:22"
```

Listing 10.15 displays the contents of `DateInfo1.sh` that performs date-related manipulation in conjunction with a `case/esac` statement.

Listing 10.15 DateInfo1.sh

```
# a script for the following:
# display the day of the week
# display the current month
# execute some command

TOP=`pwd`
today=`date`
dayOfWeek=`echo $today |cut -d" " -f1`
theMonth=`echo $today  |cut -d" " -f2`
```

```
dateDir="${dayOfWeek}-${theMonth}"
newDir="$TOP/$dateDir"

echo "Today: $today"
echo "Day of Week: $dayOfWeek"
echo "The Month:   $theMonth"
echo "Directory:   $newDir"

if [ ! -d $newDir ]; then
  mkdir -p $newDir
fi

case $dayOfWeek in
  Mon) echo "Monday"
       #execute command1
       ;;
  Tue) echo "Tuesday"
       #execute command2
       ;;
  Wed) echo "Wednesday"
       #execute command3
       touch "$newDir/monday-news"
       ;;
  Thu) echo "Thursday"
       #execute command4
       ;;
  Fri) echo "Friday"
       #execute command5
       ;;
  Sat) echo "Saturday"
       #execute command6
       ;;
  Sun) echo "Sunday"
       #execute command7
       ;;
esac
```

Listing 10.15 starts by initializing the variable `TOP` with the current directory, followed by initializing the variable `today` with the current date. The next section in Listing 10.15 initializes some date-related variables, as shown here:

```
dayOfWeek=`echo $today |cut -d" " -f1`
theMonth=`echo $today  |cut -d" " -f2`
dateDir="${dayOfWeek}-${theMonth}"
newDir="$TOP/$dateDir"
```

Next, several `echo` statements display the values of the variables that are initialized in the preceding code block.

The final portion of Listing 10.15 is a `case/esac` code block that compares the value of the variable `dayOfWeek` with each day of the week. Notice that each possibility has a "commented out" statement, which you can substitute with a command that you want to execute on each day of the week.

PRINT-RELATED COMMANDS

The print-related commands enable you to perform various tasks, such as printing a file using the `lp` command, the `pr` command, or the `lpr` command. You can use the `lpstat` command to check the status of a printer and scheduled print jobs. The `lpq` command displays the queue status of the named printer.

Keep in mind that each waiting or active file has an assigned print job number, which you can reference when performing operations on those files, such as deleting a file (discussed later). If your system includes a printer, here is an example of using the `lpr` command to print a file called `myscript.sh`:

```
lpr myscript.sh
```

Lengthy files take longer to print, and the good news is that you can send additional files to the printer while a file is being printed. The `lpq` command displays the files that are in the "print queue:"

```
lpq
lp is ready and printing
Rank   Owner      Job  Files
Total Size
active root       155  /etc/passwd
1030 bytes
```

Use the `lprm` to cancel printing of a file by specifying the print job number of that file. For example:

```
lprm 155
```

The preceding command cancels printing of job number 155. However, only the user who requested that a file be printed (or the root user) can cancel printing of the file.

Creating a Report with the `printf()` Command

Listing 10.16 displays the contents of `SimpleReport.sh` that illustrates how to update a set of users.

Listing 10.16 SimpleReport.sh

```
DataFile="users.txt"
NAME="John Doe"
ADDRESS="1234 Appian Way, SF"
```

```
PHONE="(555) 555-5555"
GPA="3.885"

# display the aligned output:
printf "%20s | %30s | %14s | %5s\n" "Name" "Address"
"Phone Number" "GPA"
printf "%20s | %30s | %14s | %5.2f\n" "$NAME"
"$ADDRESS" "$PHONE" "$GPA"
```

Listing 10.16 starts by initializing several variables with simulated data pertaining to a hypothetical user. The second portion of Listing 10.16 invokes the `printf` statement twice in order to display the user-related information in column-aligned format.

The output from Listing 10.16 is here:

```
 Name                |      Address |    Phone Number |     GPA
 John Doe            |      1234 Main Street, SF, CA | (555)
555-5555 |  3.88
```

CHECKING UPDATES IN A LOGFILE

Listing 10.17 displays the contents of `CheckLogUpdates.sh` that illustrates how to periodically check the last line in a log file to determine the status of a system. This shell script simulates the status of a system by appending a new row that is based on the current timestamp.

The shell script sleeps for a specified number of seconds, and on the third iteration the script appends a row with an error status in order to simulate an error. In the case of a shell script that is monitoring a live system, the error code is obviously generated outside the shell script.

Listing 10.17 CheckLogUpdates.sh

```
DataFile="mylogfile.txt"
OK="okay"
ERROR="error"
sleeptime="2"
loopcount=0

rm -f $DataFile 2>/dev/null; touch $DataFile
newline="`date` SYSTEM IS OKAY"
echo $newline >> $DataFile

while (true)
do
  loopcount=`expr $loopcount + 1`

  echo "sleeping $sleeptime seconds..."
  sleep $sleeptime
```

```
  echo "awake again..."

  lastline=`tail -1 $DataFile`

  if [ "$lastline" == "" ]

  then
    continue
  fi

  okstatus=`echo $lastline |grep -i $OK`
  badstatus=`echo $lastline |grep -i $ERROR`

  if [ "$okstatus" != "" ]
  then
    echo "system is normal"
    if [ $loopcount == "3" ]
    then
       newline="`date` SYSTEM IS OKAY"
    else
       newline="`date` SYSTEM ERROR"
    fi
    echo $newline >> $DataFile
  elif [ "$badstatus" != "" ]
  then
    echo "Error in logfile: $lastline"
    break
  fi
done
```

Listing 10.17 starts by initializing some variables with hard-coded values that pertain to a log file that is specified via the variable `DataFile`, as shown here:

```
DataFile="mylogfile.txt"
OK="okay"
ERROR="error"
sleeptime="2"
loopcount=0
```

The next section of code unconditionally deletes the file `DataFile` in order to ensure that it's initially empty whenever this shell script is invoked.

The main portion of Listing 10.17 is a `while` loop that contains several subsections. The first subsection sleeps for `$sleeptime` seconds and then checks the last line in the text file defined by the variable `DataFile`. If the last line is empty, then the continue statement returns execution to the top of the `while` loop.

Otherwise, the next subsection initializes the variables `okstatus` and `badstatus` by invoking the `grep` command to find occurrences of the variables `OK` and `ERROR` that are initialized with the values okay and error, respectively. If the variable `okstatus` is non-empty, then a message is appended to the log file, depending on the value of the variable `loopcount`. Notice that two different messages can be appended to the log file: if an error message is appended, then this message will be detected during a subsequent iteration of the while loop.

On the other hand, if `okstatus` is empty and the variable `badstatus` is non-empty, then an error message is appended to the log file and the `break` statement exits the `while` loop. A sample invocation of Listing 10.17 is here:

```
sleeping 2 seconds...
awake again...
system is normal
sleeping 2 seconds...
awake again...
Error in logfile: Fri Sep 27 21:28:53 PDT 2013 SYSTEM
ERROR
```

This code sample is admittedly contrived, but you can use the same (or similar) logic in case you need to parse the contents of a log file to check for error messages, after which you can provide some type of status update.

LISTING ACTIVE USERS ON A MACHINE

There are various commands for finding out information about users on a Unix system, such as `finger`, `who`, `uname`, and `whoami`. The `who` command lists information about various users, as shown here:

```
ocampesato console  Aug 17 08:15
ocampesato ttys000  Aug 17 08:16
ocampesato ttys001  Aug 17 08:16
. . .
ocampesato ttys011  Aug 17 08:16
ocampesato ttys012  Aug 17 08:16
```

The `w` command provides additional information about users, as shown here:

```
22:30  up 28 days, 14:18, 14 users, load averages:
1.14 1.33 1.90
USER     TTY      FROM              LOGIN@  IDLE WHAT
ocampesato console  -      Thu05AM 28days -
ocampesato s011     -      Thu05AM    10 vi shell-
                           programming-outline.txt
ocampesato s012     -      Thu05AM 2days -bash
ocampesato s000     -      Thu05AM 27:01 -bash
```

```
ocampesato s002      -      Thu05AM  3:31 vi demo-list.
                            txt
ocampesato s001      -      Thu05AM 2days -bash
ocampesato s004      -      Thu05AM 33:52 vi todo-
                            events.txt
ocampesato s007      -      Thu05AM 2days -bash
```

You can also view the users who are logged into a Unix machine with the `users` command. If you have a Macbook, you will probably be the only person logged into your machine.

MISCELLANEOUS COMMANDS

The `who` command displays a list of logged in users. If you are the only user on your Macbook, you will see something like this:

The `df` command displays the amount of free disk space, and when you invoke this command with the `-k` option, the output looks something like this:

```
Filesystem    1024-blocks      Used Available Capacity
iused      ifree %iused  Mounted on
/dev/disk1      936490368 839326208  96908160    90%
17483073 4277484206    0%   /
devfs                189       189         0   100%
655          0  100%   /dev
map -hosts             0         0         0   100%
0          0  100%   /net
map auto_home          0         0         0   100%
0          0  100%   /home
```

The `man` and `info` commands provide a summary for `bash` commands. An

example of the initial portion of the output from `man ls` is here:

```
LS(1)                    BSD General Commands Manual
LS(1)

NAME
     ls -- list directory contents

SYNOPSIS
     ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1] [file
...]

DESCRIPTION
     For each operand that names a file of a type other
than directory,
```

```
     ls displays its name as well as any requested,
associated informa-
     tion.  For each operand that names a file of type
directory, ls
     displays the names of files contained within that
directory, as
     well as any requested, associated information.
```

The `curl` command is a utility that reads the contents of a URL from the command line. For example, you can read the contents of the Google homepage and redirect its contents to a file with the following command:

```
curl https://www.google.com >y1
```

The `sleep` command is the shell equivalent of a wait loop. It pauses for a specified number of seconds, doing nothing. It can be useful for timing or in processes running in the background, checking for a specific event every so often (polling).

```
sleep 4     # Pauses 4 seconds.
```

Note: the sleep command defaults to seconds, but minute, hours, or days may also be specified.

```
sleep 4 h   # Pauses 4 hours
```

The `su` command enables you to switch to the "superuser," whereas the `sudo` command enables you to execute a specific command as superuser, but without switching to the superuser.

The `make` utility is primarily for compiling C or C++ programs, but you can use this command for other purposes as well. By default, the `make` utility searches for the file `Makefile` ("big make"), followed by the file `makefile` ("small make"). You can specify a different file via the "-f" switch, as shown here:

```
make –f myfile
```

The `tee` command enables you to redirect a copy of the output of a command to a file, as shown here:

```
ls –l | tee /tmp/files
```

You can append to an existing file with the "-a" option, as shown here:

```
ls –l | tee –a /tmp/files
```

The `nice` command changes the scheduling priority of a command, which means that you can increase or decrease its priority.

The `sync` command synchronizes the contents of your hard disk with the contents of in-memory buffers.

The `finger` command provides user-related information for a particular user.

The `cal` command displays calendar-related information. Execute the `cal` command with no arguments if you want information about the current month and year, and execute `cal 2020` if you want information about all the months in the year 2020 (or some other year). For example, if the current month and year are February 2020, the cal command displays the following output:

```
   February 2020
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

The `mktemp` enables you to create temporary file names, which are useful for storing intermediate results during the execution of long-running shell scripts.

SUMMARY

In this chapter, you learned how to selectively copy and delete files from a directory tree. Then you saw how to create sub-directories in a directory, based on a set of strings. One of those shell scripts shows you how to check whether or not a given string is an existing sub-directory or a file.

In addition, you learned how to work with compression-related files such as gzip, gunzip, and so forth. Then you saw how to schedule tasks with the `at` and `crontab` commands, and also how to terminate processes with the `kill` command. Finally, you learned an assortment of miscellaneous commands, such as displaying logged-in users, displaying disk-related information, and displaying calendar-related information.

At this point, there is one more thing to say: Congratulations! You have completed a fast-paced yet dense book, and if you are a `bash` neophyte, the material will probably keep you busy for many hours. The examples in the chapters provide a solid foundation, and the appendix contains additional examples and use cases to further illustrate how the `bash` commands work together. 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!

INDEX