# BASH FOR DATA SCIENTISTS

Oswald Campesato

# Bash for Data Scientists

# Bash for Data Scientists

**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
1-800-232-0223

O. Campesato. *Bash for Data Scientists*.
ISBN: 978-1-68392-973-4

Library of Congress Control Number: 2022948076

222324321 Printed on acid-free paper in the United States of America.

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

# *CONTENTS*

# *PREFACE*

## WHAT IS THE GOAL?

The goal of this book is to introduce readers to an assortment of powerful command line utilities that can be combined to create simple yet powerful shell scripts for processing datasets. The code samples and scripts use the bash shell, and typically involve small datasets so that you can focus on understanding the features of `grep,` `sed`, and `awk`. 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 datasets. 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 modest knowledge of shell programming.

You will acquire an understanding of how to use various bash commands, often as part of short shell scripts in later chapters. The chapters also contain simple use cases that illustrate how to perform various tasks involving datasets, such as switching the order of a two-column dataset (Chapter 1), removing control characters in a text file (Chapter 2), find specific lines and merge them (Chapter 3), reformatting a date field in a dataset (Chapter 5), and removing nested quotes (Chapter 6).

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 10.15.7 (macOS Catalina). Regarding their content: 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 size of this book.

## WHAT YOU NEED TO KNOW FOR THIS BOOK

You need some familiarity with working from the command line in a Unix-like environment. However, there are subjective prerequisites, such as a 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 book. 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 datasets before you process them in tools such as `Pandas`, after which you can do further reading to deepen your knowledge.

## HOW DO I SET UP A COMMAND SHELL?

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

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

A second method for Mac users is to open a new command shell on a MacBook from a command shell that is already visible simply by clicking `command+n` in that command shell, and your Mac will launch another command shell.

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

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

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

The answer to this question varies widely, mainly because the answer depends heavily on your objectives. The best answer is to try a new tool or technique from the book out on a problem or task you care about, professionally or personally. Precisely what that might be depends on who you are, as the needs of a data scientist, manager, student, or developer are all different. In addition, keep what you learned in mind as you tackle new data cleaning or manipulation challenges. Sometimes knowing a technique is possible will 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.

CHAPTER 1

# INTRODUCTION

This chapter introduces you to basic commands in Bash, such as navigating around the file system, listing files, and displaying the contents of files. This chapter is dense and contains a very eclectic mix of topics to quickly prepare you for later chapters. If you already have some knowledge of shell programming, you can probably skim quickly through this introductory chapter and proceed to Chapter 2.

The first part of this chapter starts with a brief introduction to some Unix shells, and then discusses 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 instructions for making them executable. As you will see, shell scripts contain Bash commands (and can optionally contain user-defined functions), so it's a good idea to learn about Bash commands before you can create shell scripts (which includes 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 5).

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 commands. Second, you can use the `crontab` utility to schedule the execution of your shell scripts. 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 and removing unwanted files. If you are completely new to Unix, 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 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, and today there are several variants available, such as HP/UX for HP machines and AIX for IBM machines. Linus 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` and `noclobber`. The various subcategories for Bourne Shell are listed as follows:

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

The different C-type shells are as follows:

- C shell (`csh`)
- TENEX/TOPS C shell (`tcsh`)

While the commands and the shell scripts in this book are based on the Bash shell, many of the commands also work in other shells (and if not, those other shells have a similar command to accomplish the same goal). Performing an Internet search for "how do I do <Bash command> in <shell name>" will often get you an answer. Sometimes the command is essentially the same, but with slightly different syntax, 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 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:

```
-rwxr-xr-x  1 root  wheel    31440 Sep 21  2020 sh
-rwxr-xr-x  1 root  wheel   110848 Sep 21  2020 dash
-r-xr-xr-x  1 root  wheel   623472 Sep 21  2020 bash
-rwxr-xr-x  1 root  wheel   529424 Sep 21  2020 tcsh
-r-xr-xr-x  1 root  wheel  1300256 Sep 21  2020 ksh
-rwxr-xr-x  1 root  wheel   529424 Sep 21  2020 csh
-rwxr-xr-x  1 root  wheel   637840 Sep 21  2020 zsh
```

A nice comparison matrix of the support for various features among the preceding shells is available online:

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

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 of this book, which 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, as shown here:

```
man cat
```

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 basic operations, such as displaying (or changing) the current directory, listing the contents of a directory, and displaying the contents of a file. The following set of commands shows you how to perform these operations, and you can execute a subset of these comments 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. If you are *currently* in the `/Users/jsmith` directory, 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 -`. The `cd` command on Windows merely displays the current directory and does not change the current directory (unlike the Unix `cd` command).

### *The history Command*

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

```
history
```

A short sample output of the preceding command (the maximum number of lines is 500) is here:

```
 1202  cat longfile.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 preceding snippet executes the same command that appears as line `1210` in the previous sample's output. However, `!cd` will search *backward* through the history of commands to find the *first* command that matches `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, especially something of the form* `!rm`*, which will remove one or more files. Hence, 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 *txt` command displays the following filenames (the actual display depends on the font size and the width of the command shell) on my Mac:

```
ReservedWords.txt   data2.txt   input-info.txt    longfile2.txt
abc.txt good-info.txt     longfile.txt names.txt
```

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

```
-rw-r--r--  1  oswaldcampesato   staff 2101  Jun   16    13:07
   input-info.txt
-rw-r--r--  1  oswaldcampesato   staff 29    Jun   16    13:07
   data2.txt
-rw-r--r--  1  oswaldcampesato   staff 407   Jun   16    13:07
   longfile2.txt
-rw-r--r--  1  oswaldcampesato   staff 408   Jun   16    13:07
   longfile.txt
-rw-r--r--  1  oswaldcampesato   staff 2267  Jun   16    13:07
   ReservedWords.txt
-rw-r--r--  1  oswaldcampesato   staff 1638  Jun   16    13:07
   good-info.txt
-rw-r--r--  1  oswaldcampesato   staff 45    Jun   16    13:07
   abc.txt
-rw-r--r--  1  oswaldcampesato   staff 35    Jun   16    13:07
names.txt
```

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

```
-rw-r--r--  1  oswaldcampesato   staff 35    Jun   16    13:07
   names.txt
-rw-r--r--  1  oswaldcampesato   staff 45    Jun   16    13:07
   abc.txt
-rw-r--r--  1  oswaldcampesato   staff 1638  Jun   16    13:07
   good-info.txt
-rw-r--r--  1  oswaldcampesato   staff 2267  Jun   16    13:07
   ReservedWords.txt
-rw-r--r--  1  oswaldcampesato   staff 408   Jun   16    13:07
   longfile.txt
```

```
-rw-r--r--  1  oswaldcampesato   staff 407    Jun   16    13:07
    longfile2.txt
-rw-r--r--  1  oswaldcampesato   staff 29     Jun   16    13:07
    data2.txt
-rw-r--r--  1  oswaldcampesato   staff 2101   Jun   16    13:07
input-info.txt
```

Here are the descriptions of all the listed columns in the preceding output:

- Column #1: represents the 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 the group of the owner
- Column #5: represents file size in bytes
- Column #6: the date and time when this file was created or last modified
- Column #7: represents the 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's listed. These (and other) initial values are described as follows:

- `-` 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)

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. Before doing so, let's invoke the `wc` (word count) command to display the number of lines, words, and characters in a text file, as shown here:

```
wc longfile.txt
37      80     408 longfile.txt
```

The preceding output shows that the file `longfile.txt` contains 37 lines, 80 words, and 408 characters, which means that the file size is actually quite small (despite its name).

### *The `cat` Command*

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

```
cat longfile.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 preceding command displays the entire contents of a file; however, there are several commands that display only a portion of a file, such as `less`, `more`, `page`, `head`, and `tail` (all of which are discussed later).

As another 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
```

Type the following command that contains the `?` meta character (discussed in detail later in this chapter):

```
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
```

### *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 longfile.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 longfile.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 longfile.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 longfile.txt` displays the last 4 lines of `longfile.txt`.

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

```
more longfile.txt
```

Press the `<spacebar>` to view the next screen full 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 ("`|`"), which enables you to "pipe" or redirect the output of one command to become the input of another command. The pipe command is useful 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 longfile.txt| head -2
```

A technical point: the preceding command creates two Bash processes (more about processes later) whereas the command `head -2 longfile.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 `longfile.txt`:

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

The preceding command displays the following text:

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

To display the line numbers for the preceding output, use the following:

```
cat -n longfile.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 longfile.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 5, you will learn how to display the lines in a text file that match a string or a pattern, as well as 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, and `x` = file 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.

Use the `chmod` command to set permissions for files. For example, if you need to set the permission `rwx rw- r--` for a file, use the following:

```
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.

To add additional permissions on the current file, use `+` to add permission to the user, group, or others and use `-` to remove the permissions. For example, given a file with the permissions `rwx rw- r--`, add the executable permission as follows:

```
chmod o+x filename
```

This command adds the `x` permission for `others`. Add the executable permission to all permission categories that is, for user, group, and others as follows:

```
chmod a+x filename
```

In the preceding command, the letter `a` means "all groups." Conversely, specify a `-` to remove a permission from all groups, as shown here:

```
chmod a-x filename
```

## HIDDEN FILES

An "invisible" file is one whose first character is the dot or period character (.). Bash programs (including the shell) use most of these files to store configuration information. Some common examples of hidden files include the following 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

To list invisible files, specify the `-a` option to `ls`:

```
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
```

Single dot .: This represents the current directory.
Double dot ..: This represents the parent directory.

## HANDLING PROBLEMATIC FILENAMES

Problematic filenames contain one or more whitespaces, hidden (non-printing) characters, or start 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 –abz` will not work correctly, because `-z` is interpreted as a switch for the `ls` command, and since there is no "z" option, you will see the following type of output:

```
ls: illegal option -- z
usage: ls [-@ABCFGHLOPRSTUWabcdefghiklmnopqrstuwx1%] [file ...]
```

In most cases, the best solution to this type of file is to rename the file. This can be done in your operating system if your client isn't a Unix 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 and then some of the more common variables.

### *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
```

Some interesting examples of setting an environment variable and also executing a command are described online:

*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 the 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 live in many directories, so operating systems provide a search path that lists the directories that the operating system searches for executable files. Add a directory to your path 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. These variables contain information available to the command shell and other programs.

The path variable is named `PATH` in `Bash` or `Path` in Windows (Bash is case-sensitive; Windows is not).

Setting the path in `Bash/Linux`:

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

To add the `Python` directory to the path for a particular session in `Bash`, use the following:

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

In Bourne shell or `ksh` shell, enter this command:

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

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

***Specifying Aliases and Environment Variables***

The following command defines an environment variable called `h1`:

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

Now if you enter the following command,

```
echo $h1
```

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 Jan 06 19:21 ssl-
    instructions.txt
-rw-r--r--  1 ocampesato  staff   12 Jan 06 19:21 output.txt
-rw-r--r--  1 ocampesato  staff   11 Jan 06 19:21 outfile.txt
```

```
-rwx------  1 ocampesato  staff   12 Jan 06 19:21 kyrgyzstan.txt
-rwx------  1 ocampesato  staff  478 Jan 06 19:21 iphonemeetup.txt
-rwx------  1 ocampesato  staff  146 Jan 06 19:21 checkin-
    commands.txt
-rwx------  1 ocampesato  staff   25 Jan 06 19:21 apple-care.txt
```

You can also define an alias that contains the Bash pipe (|) symbol:

```
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 via the commands which, whence, whereis, and whatis.

The preceding commands produce similar results to the which command. The which command gives the full path to whatever executable 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 on 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 version of the 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 6.

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
```

However, 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 with a specified delimiter (another word commonly used for `IFS`, especially when it's part of a command syntax, instead of being set as an outside variable) as well as a range of columns from an input stream. Some examples are here:

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

The following code snippet displays the contents of field 3 followed by field 1:

```
x="abc def ghi"
echo $x | cut -d" " -f3,2
```

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

```
def
```

Consider this code snippet:

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

The output of the preceding code snippet (`-c2-5` means "extract the characters in columns 2 through 5 from the variable") is here:

```
bc d
```

Listing 1.1 displays the content of `SplitName1.sh` that illustrates how to split a filename containing the "." character as a delimiter/`IFS`.

**Listing 1.1: 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 1.1 uses the `echo` command and the `cut` command 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, but in some cases, the results might be different than your expectations.

Listing 1.2 displays the content of `EchoCut.sh` that illustrates the differences that can occur when the `echo` command is used with the `cut` command.

**Listing 1.2: 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.

This seemingly minor detail is important when you write shell scripts that check the values contained in specific columns of text files, such as payroll files and other files with financial data. The solution involves the use of double quote marks (and sometimes the IFS variable that is discussed in Chapter 2) that you can see in the definition of x4.

## COMMAND SUBSTITUTION ("BACK TICK")

The *back tick* or command substitution feature of the Bourne shell 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 "`" (back tick) character. In Listing 1.3 below, the command between back ticks is 'ls *py'.

Listing 1.3 displays the content of CommandSubst.sh displays a subset of files in a directory.

**Listing 1.3: 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 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 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? You can redirect error messages to /dev/null, and you can also redirect error messages to a text file if you want to review them. Yet another option is to redirect stderr ("standard error") to stdout ("standard out"), which is beyond the scope of this chapter. For example, the following command shows you how to redirect errors from the ls command to /dev/null:

```
ls /tmp2 2>/dev/null
```

An intermediate error can cause an entire "pipeline" to fail. 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 the file myfile1.txt with this command:

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

However, each of the preceding commands inside the parentheses spawns a subshell (which is an extra process that consumes memory and increases CPU usage, both of which can sometimes be significant). 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.txt
```

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
```

Although cmd2 in the preceding code snippet does not recognize the value of name, 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) /tmp2 does not exist. However, 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 1.4 and Listing 1.5 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 has a list of files that are the destination for the `cp` command. Paste then merges the two files into output that could then be run to execute all the `cp` commands sequentially.

**Listing 1.4: list1**

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

**Listing 1.5: list2**

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

Listing 1.6 display the result of invoking the following command:

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

**Listing 1.6: list1.sh**

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

Listing 1.6 contains three `cp` commands that are the result of invoking the `paste` command. If you want to execute the commands in Listing 1.6, 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** *There is a blank line after the third line in the2 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 1.7 displays the content of `linepairs.csv`, which contains letter and number pairs, and Listing 1.8 contains `reversecolumns.sh`, which illustrates how to match the pairs even though the line breaks are in different places between numbers and letters.

**Listing 1.7: 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 1.8: 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 is joined (between "g" and "h" and "9 and 10"). We can make the necessary revision using the `sed` command (discussed in Chapter 4):

```
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 to reverse the order of two columns in a dataset. The purpose of the shell script in Listing 1.10 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 1.9 displays the content of `namepairs.csv` that contains the first name and last name of a set of people, and Listing 1.10 contains `reversecolumns.sh`, which illustrates how to reverse these two columns.

**Listing 1.9: namepairs.csv**

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

**Listing 1.10: reversecolumns.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 1.10 (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 1.10 is straightforward, there is a simpler way to execute this task: use the `cat` command and the `awk` command (discussed later). Specifically, compare the contents of `reversecolumns.sh` with the following single line of code that combines the `cat` command and the `awk` command to generate the same output:

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

The output from the preceding code snippet is here:

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

As you can see, there is a big difference in these two solutions. 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 data cleaning tasks. Another important point: document the commands as they become more complex, as they can be hard to interpret later by others (and even by yourself, if enough time has passed). A comment such as the following can be extremely helpful to interpreting code:

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

## WORKING WITH META CHARACTERS

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

The following meta characters are useful with regular expressions:

- The ? meta character refers to 0 or 1 occurrences of something.
- The + meta character refers to 1 or more occurrences of something.
- The * meta character refers to 0 or more occurrences of something.

**NOTE** *"Something" in the preceding descriptions can refer to a digit, letter, word, or more complex combinations.*

Some examples involving meta characters are shown here:

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 `a`, `a1`, `a2`, ..., `aa`, `ab`, `ac`, and so forth.

The pipe "|" meta character (which has a different context from the pipe symbol in 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 "`$`" meta character refers to the end of a line of text, and in regular expressions inside the `vi` editor, the "`$`" meta character refers to the last line in a file.

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

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

In the case of regular expressions, the "`^`" meta character can also mean "does not match." The next section contains some examples of the "`^`" meta character.

## 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 "^" meta character, and here are some examples of using "^" with character classes:

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

Based on what you have learned thus far, you can understand the purpose of the following regular expressions:

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

Chapter 4 contains a section that discusses regular expressions, which combine character classes and meta characters to create sophisticated expressions for matching complex string patterns (such as email addresses).

## WHAT ABOUT ZSH?

The latest version of OS X provides `zsh` 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
```

Bash and `zsh` have some features in common, as shown here:<UL>

- 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. However, 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 available online:

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

### *Switching between bash and zsh*

Type the following command to set 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** *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 set them, and also how to use aliases. You also learned about "sourcing" (also called "dotting") a shell script and how this changes variable behavior from calling a shell script in the normal fashion. In addition, you learned about meta characters that can be used in regular expressions.

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 `cut` command and `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 concepts from later chapters.

CHAPTER 2

# FILES AND DIRECTORIES

This chapter discusses files and directories and various useful Unix commands for managing them. You will learn how to use simple commands that can simplify your tasks.

The first part of this chapter shows you file permissions and how to set them according to your requirements. The second part of this chapter works with file-related commands, such as `touch`, `mv`, `cp`, and `rm`. The third part of this chapter contains shell commands for managing directories. The fourth 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

Unix supports file-related commands, such as `touch`, `cp`, and `rm` that enable you to create, copy, and remove files, respectively. The following subsections illustrate how to use these convenient commands.

### *Creating Files*

The `touch` command enables you to create an empty file. 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
```

The cp command provides several switches, including the following:

- the -a archive flag (for copying an entire directory tree)
- the -u update flag (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 subdirectories (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 content of CommandSubstCopy.sh that illustrates how to use a pair of back ticks (' ') 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 subdirectory.

The cp 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 subdirectory textfiles:

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

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

You need to use the following command to ensure that the subdirectory abc.txt is copied into textfiles:

```
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 directories (when it's invoked with the -r option). 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 the terms recursive, force, and interactive, respectively.

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

```
rm -rf *
```

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

The `-i` option is useful 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
```

Later, you will use command substitution, which redirects the output of one command as the input of another command. As a simple preview, the following command removes the files that are listed in the file `remove_list.txt`:

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

If there is a possibility that you might need some of the files that you intend to delete, you could create a directory and move those files into that directory, as shown here:

```
mkdir $HOME/backup-shell-scripts
mkdir *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 preexisting directory, it becomes a subdirectory of the destination directory.

## THE BASENAME, DIRNAME, AND FILE COMMANDS

The following code snippets show you how to extract the base portion of a filename, the directory portion, and the type of file, respectively.

```
$ x="/tmp/a.b.c"
$ basename $x
$ a.b.c

$ a="/tmp/a.b"
$ basename $a
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 *o*.txt` in a command shell, you will see information for all the text files in a directory that contain the letter `o`, similar to the following output:

```
-rw-r--r--  1 oswaldcampesato  staff    89 Jun 18 16:50 columns4.
    txt
-rw-r--r--  1 oswaldcampesato  staff    89 Jun 18 16:50 columns3.
    txt
-rw-r--r--  1 oswaldcampesato  staff    77 Jun 18 16:50 columns2.
    txt
-rw-r--r--  1 oswaldcampesato  staff    64 Jun 18 16:50 columns.
    txt
-rw-r--r--  1 oswaldcampesato  staff   407 Jun 18 16:44 longfile2.
    txt
-rw-r--r--  1 oswaldcampesato  staff   408 Jun 18 16:44 longfile.
    txt
-rw-r--r--  1 oswaldcampesato  staff  2101 Jun 18 16:44 input-
    info.txt
-rw-r--r--  1 oswaldcampesato  staff  1638 Jun 18 16:44 good-
    info.txt
-rw-r--r--  1 oswaldcampesato  staff  2267 Jun 18 16:44
ReservedWords.txt
```

If you run the command `wc long*`, you will see information about files that start with the string `long`, as shown here:

```
      37      80     408 longfile.txt
       3      80     407 longfile2.txt
      40     160     815 total
```

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

```
20       173    1420
```

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 some of these commands, along with some examples.

## THE more COMMAND AND THE less COMMAND

The `more` command enables you to view "pages" of the contents of a text 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 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 longfile.txt |wc
```

The preceding command displays the following output:

```
10      22     112
```

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

```
cat –n longfile.txt |head -4
```

The preceding command displays the following output that starts with 5 space characters, and the embedded `^I` characters are shown for emphasis:

```
     1^Ithe contents
     2^Iof this
     3^Ilong file
     4^Iare too long
```

## 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 *last* line of `test.txt` contains the string `aa`:

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

The following command displays three values:

```
tail longfile.txt |wc
10      15     84
```

The first number in both output listings is 10, which confirms that only the first 10 lines or the final 10 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, this command displays three lines:

```
$ head -3 longfile.txt
the contents
of this
long file
```

The next command displays the last three lines:

```
$ tail -3 longfile.txt
and Bourne shell
```

Note that the preceding pair of blank lines are the last two lines of `longfile.txt`.

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
```

## FILE COMPARISON COMMANDS

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 file-1 file-2` outputs three columns:

- column 1 = lines unique to `file-1`
- column 2 = lines unique to `file-2`
- column 3 = lines common to both

The following options allow you to suppress 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 file name, printing only the file name. 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 Unix 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, and they have values 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)`
- `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 +x filename
```

The following command sets the SUID bit on the "filename" permissions. (You probably won't need this option so it's not discussed, but you can find details online):

```
chmod u+s filename
# An ordinary user may execute "filename" with same privileges as
    the file's owner.
# (This does not apply to shell scripts.)
chmod 644 filename
```

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

```
chmod 444 filename
```

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

***The chown Command***

The `chown` command enables you to change permissions for files. 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***

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

```
chgrp internal *.txt
```

The `chown` and `chgrp` commands support recursion via the `-R` option.

***The umask and ulimit Commands***

Whenever you create a file in Unix, 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. You will either see a numeric value or the word `unlimited`.

## WORKING WITH DIRECTORIES

A *directory* is a file that acts like a container that stores file names and related information. All files, whether ordinary, special, or directory, are contained in directories.

UNIX uses 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 and the slash character (/), and all other directories are contained below it. The position of any file within the hierarchy is described by its *path name*. (which is sometimes written as path-name in addition to pathname).

Elements of a path name are separated by a /. A path name is *absolute* if it is described in relation to root, so absolute pathnames always begin with a /. These are some examples of absolute paths:

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

A path name can also be *relative* to your current working directory. Relative path names begin with ./ (such as `./myscript.sh`)

***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 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 and Relative Path Names***

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

A path name is *absolute* if it is described in relation to the root, so absolute path names always begin with a /. Here are some examples of absolute paths:

- `/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 partial display from the preceding command is here:

```
// files omitted for brevity
fg
fgrep
file
filebyproc.d
fileproviderctl
filtercalltree
find
find2perl
find2perl5.18
findrule
findrule5.18
findrule5.28
finger
fixproc
flex
flex++
fmt
fold
fontrestore
footprint
// files omitted for brevity
```

### *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 subdirectory `data` under the current directory:

```
mkdir -p test/data
```

The `-p` option forces the creation of missing intermediate directories (if any).

Sometimes you need to create a directory, but its parent directory or intermediate directories might not exist. In this case, `mkdir` issues an error message as follows:

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

You can use the `-p` option to create intermediate directories that are specified as the parent directories of the target directory:

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

The preceding command creates all the intermediate directories before creating the right-most directory:

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

***Removing Directories***

Use the `rm` command to remove non-empty directories and use the `rmdir` command to remove empty directories:

```
rmdir dirname1
```

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

```
rmdir dirname1 dirname2 dirname3
```

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

Since this directory is empty, you can remove it with this command:

```
rmdir test
```

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

```
rm -r test
```

or you can first remove the files and then remove the directory:

```
cd test
rm -rf *
cd ../
rmdir test
```

***Changing Directories***

You can 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
```

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

#### *Renaming Directories*

The `mv` (move) command can also be used to rename a directory. The syntax is as follows:

```
mv olddir newdir
```

## USING QUOTE CHARACTERS

There are three types of quotes characters: single quotes (‘), double quotes(“), and back quotes (`); all of these quotes occur in matching pairs (‘’, “”, or ``), and they are not mixed (e.g., `abc” causes an error).

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 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 shell or environment variables with the value of the referenced variable.

Text within a pair of *back quotes* 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.

Back quotes allow you 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 command works by first executing the command contained within back quotes:

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

This command displays the number of files in the user’s directory. Since the command is enclosed in back quotes, its output replaces the original back quotes text, as shown here:

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

When executed, this command prints the 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 when 1 and 2 are followed by the ">" symbol.

In this situation, 0 refers to standard input, 1 refers to standard output, and 2 refers to standard error. The patterns `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:

```
find . -print |grep "\.py$" 1>std1 2>std2
```

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

```
find . -print |grep "\.py$" 2>&1 1>both.txt
```

You can redirect the output of Unix commands in various ways, as shown in the following section.

## METACHARACTERS AND CHARACTER CLASSES

Unix 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.

Unix supports the following list of metacharacters:

- ? (matches 0 or 1): a? matches the string a (but not ab)
- `*` (matches 0 or more): a`*` matches the string aaa (but not baa)
- + (matches 1 or more): a+ matches aaa (but not baa)
- ^ (beginning 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 with Unix commands such as `ls` and `sed`, in shell scripts, and in editors such as `vi` (with subtle differences).

### *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 "!" means negation, as shown here:
[! abc ...] Matches any character other than those specified
[!a-z] Matches any character that is *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 matches 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 gave you an introduction to the Unix operating system, along with some Unix commands. You learned how to use the cp command for copying files, the rm command for removing files, and the mv command for moving files.

Next, you learned how to use the wc command for counting the number of lines in a text file, followed by an assortment of Unix commands such as cat, more, head, and tail. In addition, you saw how to create and rename a directory, along with how to work with relative paths.

CHAPTER 3

# USEFUL COMMANDS

This chapter discusses various Bash commands that you can use when working with datasets, such as splitting, sorting, and comparing datasets. You see examples of finding files in a directory and then searching for strings in those 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 datasets. This section also shows you how to sort files and find unique lines in files using the `sort` and `uniq` commands, respectively. The last portion 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 subdirectories; 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 to search files for a particular pattern.

Next you will see how to use the `tr` command, which is an executable that handles many commonly used text transformations, such as capitalization or removal of whitespace. In addition, you will see a use case for the `tr` command 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`, and `zmore`).

The fourth section introduces you to the `IFS` option, which is useful for extracting data from a range of columns in a dataset. You will also see how to use the `xargs` command 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 that 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 exactly two files, but 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 display 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 a 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 subfiles of a given file. By default, the subfiles 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 to divide the file `abc.txt` into files with 500 lines each:

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

For example, 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 the text file `test2.txt` contains the following lines:

```
aa
cc
bb
```

The following simple example sorts the lines in `test2.txt`:

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

The output of the preceding code snippet is here:

```
aa
bb
cc
```

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

```
Option  Description
-n      Numeric sort (ex: 10 sorts 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
```

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 increasing order based on their file size:

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

The output is here:

```
total 120
-rw-r--r--  1 oswaldcampesato  staff    12 Aug 15 13:55 test.txt
-rw-r--r--  1 oswaldcampesato  staff    29 Aug 15 13:55 data2.txt
-rw-r--r--  1 oswaldcampesato  staff    35 Aug 15 13:55 names.txt
-rwxr-xr-x  1 oswaldcampesato  staff    40 Aug 15 13:55
    CommandSubst.sh
-rw-r--r--  1 oswaldcampesato  staff    45 Aug 15 13:55 abc.txt
-rw-r--r--  1 oswaldcampesato  staff    64 Aug 15 13:55 columns.
    txt
-rw-r--r--  1 oswaldcampesato  staff    77 Aug 15 13:55 columns2.
    txt
-rw-r--r--  1 oswaldcampesato  staff    89 Aug 15 13:55 columns3.
    txt
-rw-r--r--  1 oswaldcampesato  staff    89 Aug 15 13:55 columns4.
    txt
-rw-r--r--  1 oswaldcampesato  staff   125 Aug 15 13:55 abc2.txt
-rw-r--r--  1 oswaldcampesato  staff   407 Aug 15 13:55 longfile2.
    txt
-rw-r--r--  1 oswaldcampesato  staff   408 Aug 15 13:55 longfile.
    txt
-rw-r--r--  1 oswaldcampesato  staff  1638 Aug 15 13:55 good-
    info.txt
-rw-r--r--  1 oswaldcampesato  staff  2101 Aug 15 13:55 input-
    info.txt
-rw-r--r--  1 oswaldcampesato  staff  2267 Aug 15 13:55
    ReservedWords.txt
```

Notice that the file listing is sorted based on the fifth column, which displays the file size of each file. 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 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 to remove duplicate lines.

## THE uniq COMMAND

The `uniq` command prints only the unique lines in a sorted text file and omits duplicates. As a simple example, suppose the file `test3.txt` contains the following lines:

```
abc
def
abc
abc
```

The following command 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 different.

## 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 (these `tab` characters are 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 many operations. For example, the `tr` command enables you to remove extraneous whitespaces in datasets, insert blank lines, print words on separate lines, and translate characters from one character set to another character set (i.e., from uppercase to lowercase, and vice versa). The following command capitalizes the letters in the variable `x`:

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

Another way to convert from lowercase to uppercase:

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

In addition to uppercase and lowercase, you can use the following `POSIX` characters classes (whose purpose is self-explanatory) in the `tr` command:

- `alnum`: alphanumeric characters
- `alpha`: alphabetic characters
- `cntrl`: control (non-printing) 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 commas "," 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 version of the `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 5), 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 'dot' is encountered, replace it with a 'dot' followed by a space, and do this for every such occurrence."

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 uppercase 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 way (one extra line of code) 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
```

However, both of the preceding code blocks are somewhat obscure (at least for novices), so it's probably better to use other tools, such as data frames in Pandas or RStudio.

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

## A SIMPLE USE CASE

The code sample in this section shows you how to use the `tr` command to replace the control character `^M` with a linefeed. Listing 3.4 displays the content of the dataset `controlm.csv`, which 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,BASOAB,,
   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 content 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 contains a variable for the input file and one for the output file, but you can simplify the `tr` command in Listing 3.5 by using hard-coded values for the filenames.

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

```
IDN,TEST,WEEK_MINUS1,WEEK0,WEEK1,WEEK2,WEEK3,WEEK4,WEEK10,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 current task is easily solved via the `tr` command. Note that additional data cleaning is required to handle the empty fields in the output.

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

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

The resulting output is shown here:

```
IDN|TEST|WEEK_MINUS1|WEEK0|WEEK1|WEEK2|WEEK3|WEEK4|WEEK10|
   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 dataset 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 5.

## 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 `AND` as well as `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 subdirectories)
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 subdirectories)
```

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, respectively.

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 `wget` command downloads any file from the Internet, an example of which is shown here:

```
wget https://raw.githubusercontent.com/uiuc-cse/data-fa14/gh-
    pages/data/iris.csv
```

Use the find command to find files and directories. Moreover, the –exec enables you to execute other Linux commands on the list of files or directories that are retrieved by the find command. Here is a simple example:

```
find . -name "*.txt" -type f
```

The next example shows you how to copy a set of files from the find command using the –exec option, followed by example that uses the xargs command:

```
mkdir /tmp/abc
find . -name "*.txt" -type f --exec cp /tmp/abc
find . -name "*.txt" -type f |xargs cp cp /tmp/abc
```

## 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. Some examples are here:

```
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 subdirectories) 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 subdirectories) 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 is 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, uncompress a tar file, and 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* (i.e., an archive file that is compressed via gzip technology) using the following methods:

*Method #1:*

```
tar -czvf archive.tar.gz [FILES]
```

*Method #2:*

```
tar -cavf archive.tar.gz [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. To compress with `bzip2`, use the following:

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

### *The zip Command*

The `zip` command is another utility for creating zip files. The first argument is the name of the zip file, followed by one or more filenames.

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

```
zip file.zip file1 file2 file3
```

This command creates a zip file that contains all the `txt` files in the current directory:

```
zip sample2.zip *.txt
```

Alternatively, you can use the Java `jar` command:

```
jar cvf sample3.zip *.txt
```

The `unzip` command will unzip the contents of a zip file, as shown here:

```
unzip sample1.zip
```

Alternatively, you can use the Java `jar` command to unzip a zip file:

```
jar xvf sample3.zip
```

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 of a file called test.gz that contains two lines of text:

```
zcat test.gz
The output from the preceding command is here:
```

A test file

# file test contains a line "A test file"

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). In the first case, we will use `IFS = "  "` and in the second we will use `IFS=","`. Suppose that the shell variable `data` is defined as follows:

```
data="name,sex,rollno,location"
```

To read each of the `data` elements into a variable, we can use `IFS`, as shown here:

```
oldIFS=$IFS
IFS=, now,
for item in $data;
do
  echo Item: $item
done
IFS=$oldIFS
```

The next section contains a code sample that relies on the value of `IFS` 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.

**Listing 3.6 datacolumns1.txt**

```
#23456789012345678901234567890
  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 loop that reads each line from the input file called `datacol-umns1.txt` and sets the `pound` variable equal to "" 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 to ensure that the `echo` command does not remove blank spaces. The output from Listing 3.7 is as follows:

```
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 content of the dataset `uneven.txt` that contains rows with a different number of columns. Listing 3.9 displays the content 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 input file so that the output appears with four columns in each row. 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
```

## THE alias COMMAND

The `alias` command enables you to define shortcut commands, as shown here:

```
alias lt='ls -lt'
alias ltr='ls -ltr'
alias ltr1='ls -ltr1'
alias ltm='ls -lt|more'
alias ltrm='ls -ltr|more'
alias latr='ls -latr'
alias ltr8='ls -ltr|tail -8'
alias ltd='ls -lt|grep "^d"'
alias ltdr='ls -ltr|grep "^d"'
alias mtd='ls -lt|grep "^d"'
alias ltdm='ls -lt|grep "^d"|more'
```

The preceding list of aliases is just a sample of the type of aliases that you can define in order to execute custom commands from the command line.

## 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 `xargs` command. You also learned about various ways to use the `tr` command, which is part of 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 about extracting column ranges of data, as well as the usefulness of the `IFS` option.

CHAPTER 4

# CONDITIONAL LOGIC AND LOOPS

This chapter introduces you to arrays, user input, relational and string operators, conditional logic, and various types of loops in Bash. In addition, various code samples illustrate the Bash features that are introduced in the chapter.

The first part of this chapter introduces arithmetic operations and operators, as well as working with arrays and with variables. You will also learn how to work with strings in Bash.

The second part of this chapter explains how to prompt users for input, along with operators to test for variables, files, and directories. This section introduces you to conditional logic with `if/elif` statements.

The third section discusses the `case/esac` statement, arithmetic operators and comparisons, and how to work with strings in shell scripts. The fourth section introduces `for` loops, `while` loops, and `until` loops. You will also learn how user-defined functions are created, and how to create menus of options.

## ARITHMETIC OPERATIONS AND OPERATORS

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

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

You can use command substitution to assign the result to a variable:

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

Another option is to use the `expr` command, as shown here:

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

The `expr` command supports arithmetic operations, as shown in the following example that adds two numbers:

```
#!/bin/sh
sum=`expr 2 + 2`
echo "The sum: $sum"
```

The preceding code block generates the following result:

The sum: 4

Spaces are required between operators and expressions (so 2+2 is incorrect), and expressions must be inside back tick characters (also called "inverted commas").

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

```
x=15
y=3

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

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

Other operators include modulus (%), equality (==), and inequality (!=). Here are some examples (assume that `x` and `y` have numeric values):

- `'expr $x % $y'` equals `0.`
- `[ $x == $y ]` returns `false.`
- `[ $x != $y ]` returns `true.`

Note the required spaces in the preceding expressions. All arithmetic calculations are done using long integers. One other point to keep in mind: in scripting, `$(())` avoids a fork/execute that occurs when you use the `expr` command.

## WORKING WITH ARRAYS

Arrays are available in many 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, virtually all online code samples use the term "array" in shell scripts. The syntax in Bash is different enough from other programming languages that it's worthwhile to use several examples to explore its behavior.

Listing 4.1 displays the content of `Array1.sh` that illustrates how to define an array and access elements in the array.

**Listing 4.1: 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 4.1 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 4.1 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 4.2 displays the content of the shell script `loadarray.sh` that initializes an array and then prints its values.

**Listing 4.2 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 4.2 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 back ticks.

Next, the two numeric variables `total1` and `total2` are initialized to 0, followed by a 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 4.2 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 4.3 displays the content of the shell script `update-array.sh` that shows you some operations you can perform on an initialized array.

**Listing 4.3 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 4.3 and you will see the following output:

```
I
I love deep dish pizza
0 1 2 3 4
I love deep pizza in Chicago
```

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
```

Listing 4.4 displays the contents of `fruits-array1.sh` that illustrates how to use an array and some operations that you can perform on arrays.

**Listing 4.4: 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 4.4 and you will see the following output:

```
first fruit: apple
first fruit: apple
last two: orange pear
substring: ban
length: 5
```

## ARRAYS AND TEXT FILES

Text files can contain different delimiters, such as CSV (comma-separated-values) files and TSV (tab-separated-values) files. Other delimiters include a colon ":", a pipe "|" symbol, and so forth. The delimiter is called the `IFS` (Internal Field Separator).

In Bash, you can populate an array with data from external files, such as text files and CSV files. Listing 4.5 displays the content of `names.txt` and Listing 4.6 displays the content 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 4.5: names.txt**

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

**Listing 4.6: array-from-file.sh**

```
#!/bin/bash

names="names.txt"
contents1=( `echo  "$names"` )

echo "First loop:"
for w in "${contents1[@]}"
do
  echo "$w"
done

IFS=""
names="names.txt"
contents1=( `echo  "$names"` )

echo "Second loop:"
for w in "${contents1[@]}"
do
  echo "$w"
done
```

Listing 4.6 initializes the array variable `contents1` by using command substitution with the `cat` command, followed by a loop that displays elements of the array `contents1`. The second loop is the same code as the first loop, but this time with the value of `IFS` equal to "", which has the effect of using the newline as a separator, one data element per row. Launch the code in Listing 4.6 and you will see the following output:

```
First loop:
Jane
```

```
Smith
John
Jones
Dave
Edwards
Second loop:
Jane Smith
John Jones
Dave Edwards
```

Listing 4.7 displays the content of `array-function.sh` that illustrates how to initialize an array and then display its contents in a user-defined function.

**Listing 4.7: 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 4.7 contains the `items()` function that displays the content of the `arr` array initialized prior to invoking this function. The output is shown here:

```
123
-abc
my data
```

Listing 4.8 displays the content of `array-loops1.sh` that illustrates how to determine the length of an initialized array and then display its contents via a loop.

**Listing 4.8: 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 4.8 contains straightforward code for initializing an array and displaying its values. Launch the code in Listing 4.8 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
```

## WORKING WITH VARIABLES

Earlier code samples contain examples of variables in Bash, and this section shows you how to assign values to variables. Always remember that Bash variables do *not* have any type-related information, which means that no distinction is made when assigning a number or a string to a variable (similar to 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
```

However, you *can* insert whitespace between text strings and variables in the `echo` command, as you saw in the previous code block.

Listing 4.10 displays the content of `variable-operations.sh` that illustrates how to update variables with different values.

**Listing 4.10: 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"

#error:
#c='expr $a + $b'
#error:
#c='echo "$a+$b"|bc'
```

Launch the code in Listing 4.10 and you will see the following output:

```
15
01112
123456789101112
6789101112
```

```
999456789101112
a: 5 b: 7 c: 12
```

## WORKING WITH OPERATORS FOR STRINGS AND NUMBERS

Bash provides a set of operators for making comparisons between string values and numeric values, some of which are discussed in this section using simple code snippets.

Use == in an expression for a string comparison and = to assign a value to a variable:

```
[ $a == $b ]
```

Use `-eq` for numeric comparisons:

```
[ $a -eq $b ]
```

Use `-gt` (greater than) to make a numeric comparison:

```
[ $a -gt 12 ]
```

Use `-lt` (less than) to make a numeric comparison:

```
[ $b -le 12 ]
```

Use == and a metacharacter to test if a string begins with `abc`:

```
[[ $string == abc* ]]
```

Use == and a metacharacter and quote marks to test if there is an exact match:

```
[[ $string == "abc*" ]]
```

Use the ^ and | symbols to find a list of usernames that start with `ab` or `xy`:

```
egrep "^ab|^xy" /etc/passwd|cut -d: -f1
```

The following code block adds two integers:

```
x1=1
x2=2
let x3=$x1+$x2
echo $x3
```

The output from the preceding block is here:

```
3
```

If you omit the word `let` in the preceding code block, the output is 1=2 instead of 3.

The following code snippet assigns a value to a variable and then references its value:

```
name=John && echo "My name is $name"
My name is John
```

Display a substring of a given string variable as follows:

```
myvar="Hello everyone I am machine learning engineer"
echo ${myvar:8:6}
eryone
```

Note that 8 specifies the starting position and 6 specifies the length of the substring.

Here is an interesting set of expressions:

```
myvar="code:555:777:/usr/bin":
echo "expression1:"
echo ${myvar#*:*:*:}
echo
echo "expression2:"
echo ${myvar#*:*:}
echo
echo "expression3:"
echo ${myvar%:*:*:*:}
```

The output from the preceding code block is here:

```
expression1:
/usr/bin:

expression2:
777:/usr/bin:

expression3:
code
```

## THE read COMMAND FOR USER INPUT

The following statement will read "n" characters from standard input into the variable `variable_name`:

```
read -n number_of_chars variable_name
```

The following code block reads a string of the specified number of characters from user input from the command line:

```
read -n 1 var
echo "result1: $var"

read -n 2 var
echo "result2: $var"

read -n 3 var
echo "result3: $var"
```

A sample output from the preceding code block is here:

```
aresult1: a
abresult2: ab
abcresult3: abc
```

## THE test COMMAND FOR VARIABLES, FILES, AND DIRECTORIES

The `test` command supports many options for testing files, strings, and numbers, as shown here:

- `-d file` The specified file exists and is a directory.
- `-e file` The specified file exists.
- `-r file` The specified file exists and is readable.
- `-s file` The specified file exists and has non-zero size.
- `-w file` The specified file exists and is writable.
- `-x file` The specified file exists and is executable.
- `-L file` The specified file exists and is a symbolic link.
- `f1 -nt f2` : File `f1` is newer than file `f2`.
- `f1 -ot f2` : File `f1` is older than file `f2`.
- `-n s1` : String `s1` has a nonzero length.
- `-z s1` : String `s1` has zero length.
- `s1 = s2` : String `s1` is the same as string `s2`.
- `s1 != s2` : String `s1` is not the same as string `s2`.
- `n1 -eq n2` : Integer `n1` is equal to integer `n2`.
- `n1 -ge n2` : Integer `n1` is greater than or equal to integer `n2`.
- `n1 -gt n2`: Integer `n1` is greater than integer `n2`.
- `n1 -le n2`: Integer `n1` is less than integer `n2`.
- `n1 -lt n2`: Integer `n1` is less than or equal to integer `n2`.
- `n1 -ne n2`: Integer `n1` is not equal to integer `n2`.
- `!` The not operator, which reverses the value of the following condition
- `-a` The `and` operator, which joins two conditions. Both conditions must be true for the overall result to be true.
- `-o` The `or` operator, which joins two conditions. If either condition is true, the overall result is true.
- `\( ... \)` You can group expressions within the test command by enclosing them within `\( and \)`.

Bash supports the following types of operators:

- arithmetic operators
- relational operators
- Boolean operators
- string operators
- file test operators

The following subsections contain examples that illustrate each of the preceding operator types.

### *Relational Operators*

Bash supports relational operators that are specific to numeric values: they will not work correctly for string values unless their value is numeric. For example, the following operators determine if a quantity is between 5 and 15 as well as in between "5" and "15" but not in between "five" and "fifteen." Suppose that the variable `a` equals 5 and variable `b` equals 15 in the following examples:

- `$a -eq $b` determines whether `$a` and `$b` are equal.
- `$a -ne $b` determines whether `$a` and `$b` are unequal.
- `$a -gt $b` determines whether `$a` is larger than `$b`.
- `$a -lt $b` determines whether `$a` is smaller than `$b`.
- `$a -ge $b` determines whether `$a` is larger or equal to `$b`.
- `$a -ge $b` determines whether `$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, such as `[ $a -eq $b ]`.

### *Boolean Operators*

Bash supports several Boolean operators. Suppose that variable `a` equals 5 and variable `b` equals 15 in the following examples.

```
[ ! false ] is true.
```

The `-o` flag is for a logical `OR` that is true if at least one operand is true:

```
[ $a -lt 20 -o $b -gt 100 ] is true if either operand is true.
```

The `-a` flag is a logical `AND` that is true only if *all* operands are true:

```
[ $a -lt 20 -a $b -gt 100 ] is true only if both operands are
    true.
```

### *String Operators*

The Bash shell supports various string-related operators. For example, suppose that the variable `a` equals "abc" and variable `b` equals "efg." Then,

- `[ $a = $b ]` is false.
- `[ $a != $b ]` is true.
- `[ -z $a ]` is false because `$a` has a non-zero length.
- `[ -n $a ]` is true because `$a` has a non-zero length.
- `[ $a ]` is false because `$a` is a non-empty string.

### *File Test Operators*

Bash supports operators to test various properties of files. For example, suppose that the variable `file` equals "test" that matches an existing file name whose size is 100 bytes and has read, write, and execute permissions:

- `if [ -b file ]` checks if a file is a block special file.
- `if [ -c file ]` checks if a file is a character special file.

- `if [ -d file ]` checks if a file is a directory.
- `if [ -f file ]` checks if a file is a file versus a directory or special file.
- `if [ -g file ]` checks if a file has its set group ID (SGID) bit set.
- `if [ -k file ]` checks if a file has its sticky bit set.
- `if [ -p file ]` checks if a file is a named `pipe`.
- `if [ -t file ]` checks if a file descriptor is open and associated with a terminal.
- `if [ -u file ]` checks if a file has its set user id (SUID) bit set.
- `if [ -r file ]` checks if a file is readable.
- `if [ -w file ]` checks if a file is writable.
- `if [ -x file ]` checks if a file is executable.
- `if [ -s file ]` checks if a file has a size greater than 0.
- `if [ -e file ]` checks if a file exists.

## CONDITIONAL LOGIC WITH if/else STATEMENTS

The following example shows you how to use `-lt` in an `if/else/elif` statement:

```
x=25
if [ $x -lt 30 ]
then
  echo "x is less than 30"
else
  echo "x is at least 30"
fi
```

The preceding code block illustrates how to use `if/else` conditional logic, and the output is exactly what you expected. Listing 4.11 displays the contents of the shell script `testvars.sh` that checks if the variable `x` is defined.

**Listing 4.11: testvars.sh**

```
x="abc"

if [ -n "$x" ]
then
  echo "x is defined: $x"
else
  echo "x is not defined"
fi
```

Listing 4.11 initializes `x` with the value `abc` and then performs if/else logic with the `-n` option to determine whether `x` contains a value. The output from Listing 4.11 is shown here:

```
x is defined: abc
```

Listing 4.12 displays the contents of the shell script `testvars2.sh` that checks if the variable `y` is undefined.

**Listing 4.12: testvars2.sh**

```
x="abc"

if [ -z "$y" ]
then
  y="def"
  echo "y is defined: $y"
else
  echo "y is defined: $y"
fi
```

Listing 4.12 initializes `x` with the value `abc` and then performs if/else logic with the `-z` option to determine whether `y` is uninitialized: if so, then `y` is initialized with the value `def` and its value is printed. Otherwise, the existing value of `y` is printed. The output from Listing 4.12 is shown here:

```
y is defined: def
```

## THE case/esac STATEMENT

The `case` 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 uppercase or lowercase "n" (for no) as well as "y" (for yes).

Listing 4.13 displays the content of `case1.sh` that checks various conditions in a `case/esac` statement.

**Listing 4.13: 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 4.13 initializes `x` with the value `abc` and then uses a `case` statement to compare the value of `x` with `a` and `c`, neither of which is true. Next, the case statement compares the value of `x` with a regular expression that starts with the letter `a`, and this *does* match the contents of the variable `x`. The output from Listing 4.12 is shown here:

```
x starts with a
```

Listing 4.14 shows you how to prompt users for an input string and then process that input via a `case/esac` statement.

**Listing 4.14: 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 4.14 prompts users to enter values for `fname`, `lname`, and `city`, initializes the variable `fullname` with the concatenation of `$fname` and `$lname`, and then prints the result.

The final block of code in Listing 4.14 checks if the value of `$city` starts with the string `San` or `Chicago`, and then prints an appropriate message.

Listing 4.15 displays the content of `StartChar.sh` that checks the type of the first character of a user-provided string.

**Listing 4.15: StartChar.sh**

```
while (true)
do
  echo -n "Enter a string (x to exit): "
  read var

  case ${var:0:1} in
       x) echo "Exiting program..."; break;;
  [0-9]*) echo "$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 4.15 contains a `while` loop that prompts users to enter a string and a case statement that compares the input string in the variable `var` with various regular expressions. If the first character is a digit, uppercase letter, or lowercase letter, then an appropriate message is printed. The default case is executed if the first three regular expressions do not match the first character of the variable `var`.

Listing 4.16 displays the content of `StartChar2.sh` that checks the type of the first pair of characters of a user-provided string.

**Listing 4.16: StartChar2.sh**

```
while (true)
do
  echo -n "Enter a string (x to exit): "
  read var

  case ${var:0:2} in
          x) echo "Exiting program..."; break;;
  [0-9][0-9]) echo "$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 4.16 extends the code in Listing 4.15 in the sense that the `case` statement in Listing 4.16 checks for two consecutive occurrences of the same data type: two digits, two uppercase letters, or two lowercase letters.

Listing 4.17 displays the content of `StartChar3.sh` that checks the type of the first character of a user-provided string.

**Listing 4.17: StartChar3.sh**

```
while (true)
do
  echo -n "Enter a string (x to exit): "
  read var

   case ${var:0:1} in
         x) echo "Exiting program..."; break;;
     [0-9]*) echo "$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 4.17 is essentially an alternate way to express the regular expressions in Listing 4.16, and the results will be the same.

## ARITHMETIC OPERATORS AND COMPARISONS

Conditions are usually enclosed in square brackets `[]`. There is a *space* between [or] and the operands: an error occurs if no space is provided. Hence, the following term is correct:

```
[ $var -eq 0 ]
```

The following term results in an error:

```
[$var -eq 0 ]
```

Performing arithmetic conditions over variables or values can be done as follows:

- `[ $var -eq 0 ]` # true if `$var` equals 0
- `[ $var -ne 0 ]` # true if `$var` is unequal to 0

Here are other important comparison operators that are useful for you to know:

- `-gt`: greater than
- `-lt`: less than
- `-ge`: greater than or equal to
- `-le`: less than or equal to

Multiple test conditions can be combined as follows:

```
[ $var1 -ne 0 -a $var2 -gt 2 ]  # using AND -a
[ $var -ne 0 -o var2 -gt 2 ] # OR -o
```

We can also test for various attributes of files and directories, as shown in the following examples.True if the variable is a regular file path or file name:

```
[ -f $var]
```

True if the variable is a file path or file name that is executable:

```
[ -x $var ]
```

True if the variable is a directory path or directory name:

```
[ -d $var ]
```

True if the variable is an existing file:

```
[ -e $var ]
```

True if the variable is a path of a character device file:

```
[ -c $var ]
```

True if the variable is a path of a block device file:

```
[ -b $var ]
```

True if the variable is a path of a file that is writable:

```
[ -w $var ]
```

True if the variable is a path of a file that is readable:

```
[ -r $var ]
```

True if the variable is a path of a symbolic link:

```
[ -L $var ]
```

An example of the `-e` switch is as follows:

```
fpath="/etc/passwd"
if [ -e $fpath ]; then
   echo File exists;
else
   echo Does not exist;
fi
```

For string comparison, use double square brackets since single brackets can result in errors.

Perform string comparisons as shown in the following examples. The presence of a space after and before = indicates a comparison, whereas the absence of a space indicates an assignment statement.

True when `str1` equals `str2`:

```
[[ $str1 = $str2 ]]
```

Another way to check for string equality:

```
[[ $str1 == $str2 ]]
```

True when `str1` and `str2` are different:

```
[[ $str1 != $str2 ]]
```

True when `str1` is alphabetically greater than `str2`:

```
[[ $str1 > $str2 ]]
```

True when `str1` is alphabetically lesser than `str2`:

```
[[ $str1 < $str2 ]]
```

True if `str1` is an empty string:

```
[[ -z $str1 ]]
```

True if `str1` is a non-empty string:

```
[[ -n $str1 ]]
```

Combine multiple conditions using the logical operators `&&` and `||` for `AND` and `OR`, respectively, using the following syntax:

```
if [[ -n $str1 ]] && [[ -z $str2 ]] ;
then
   commands;
fi

if [[ -n $str1 ]] || [[ -z $str2 ]] ;
then
   commands;
fi
```

An example of using the `&&` syntax is shown here:

```
str1="Not empty "
str2=""
if [[ -n $str1 ]] && [[ -z $str2 ]];
then
   echo str1 is non-empty and str2 is empty string.
fi
```

The output from launching the preceding block of code is as follows:

```
str1 is non-empty and str2 is empty string.
```

The `test` command can be used for performing condition checks. It helps to avoid usage of many braces. The same set of test conditions enclosed within `[]` can be used for the `test` command. For example:

```
if  [ $var -eq 0 ]; then echo "True"; fi
```

can be written as

```
if  test $var -eq 0 ; then echo "True"; fi
```

## WORKING WITH STRINGS IN SHELL SCRIPTS

Bash provides the `expr` command for finding the length of a string and finding substrings of a string using curly braces, both of which are discussed in this section.

The `expr` command enables you to find the length of a character string. Some examples are here:

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

### *Working with Strings*

Listing 4.18 displays the content of `substrings.sh` that illustrates how to use curly braces to find the substrings of a given string.

**Listing 4.18: 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}
```

Listing 4.18 initializes the variable x with the first ten letters of the alphabet and then uses the echo command to display various substrings of x. Launch the code in Listing 4.18 and you will see the following output:

```
abcdefghij
bcdefghij
fghij
hij
abcdefghij
ghij
ghij
```

The next portion of this chapter discusses additional types of loops that you can use in shell scripts.

## WORKING WITH LOOPS

Bash supports the `for` loop construct and the `while` loop construct. The following subsections illustrate how you can use each of these constructs.

### *Using a for loop*

The `for` loop syntax is slightly different from other languages (such as JavaScript and Java). The following code block shows you how to use a `for` loop to iterate through a set of files in the current directory.

```
for f in 'ls *txt'
do
  echo "file: $f"
done
```

Listing 4.19 displays the content of `renamefiles.sh2` that illustrates how to rename a set of files in a directory.

**Listing 4.19: 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 4.19 initializes the variable `suffix` with the value `txt`, followed by a `for` loop that processes all the files in the current directory that have the suffix `sh`. During each iteration, the variables' base and suffix are initialized with the initial portion and the suffix of the current file, after which the variable `newfile` is initialized with the same initial portion, but with a new suffix, `txt`. For example, the filename `abc.txt` is generated from an existing filename `abc.sh`.

Uncomment either the `mv` command or the `cp` command in Listing 4.17, launch the code again, and compare the results.

## WORKING WITH NESTED LOOPS

This section is mainly for fun: you will see how to use nested loops to displays a "triangular" output. Listing 4.9 displays the contents of `nestedloops.sh` that illustrates how to display an alternating set of symbols in a triangular fashion.

**Listing 4.9: 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 4.9 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 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 4.9, and you will see the following output:

```
@
# @
@ # @
# @ # @
@ # @ # @
# @ # @ # @
@ # @ # @ # @
# @ # @ # @ # @
@ # @ # @ # @ #
@ # @ # @ # @
@ # @ # @ #
@ # @ # @
@ # @ #
@ # @
@ #
@
```

## USING A while LOOP

Listing 4.20 displays the content of `while1.sh` that illustrates how to use a `while` loop to iterate through a set of numbers.

**Listing 4.20: while1.sh**

```
x=0
x=`expr $x + 1`
echo "new x: $x"

while (true)
do
  echo "x = $x"
  x=`expr $x + 1`
  if [ $x > 4 ]
  then
    break
  fi
done
```

Listing 4.20 initializes the variable x with the value 0, followed by a while loop that increments and prints the value of x until it is greater than 4.

Listing 4.21 displays the content of `wordfile.txt`. Listing 4.22 shows you how to use a `while` loop to iterate through a text file and convert the lines with an even number of words to uppercase and the lines with an odd number of words to lowercase.

**Listing 4.21: wordfile.txt**

```
abc def
def
abc ghi
abc
```

**Listing 4.22: 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 4.22 initializes the variables `infile` and `outfile` and then removes `outfile` unconditionally. Next, a `while` loop converts the contents of even-numbered rows in the file `wordfile.txt` to uppercase characters, and converts the contents of odd-numbered rows in the file `wordfile.txt` to lower case characters. In both cases, the converted text is redirected to `outfile`. Notice the use of  >>, which appends to file, whereas a single >

would truncate any existing text in the output file. Launch the code in Listing 4.22. The output creates the file `converted.txt`, whose contents are shown here:

```
ABC DEF
def
ABC GHI
abc
```

## THE while, case, AND if/elif/fi STATEMENTS

Listing 4.23 displays the content of `yesno.sh` that illustrates how to combine the `while`, `case`, and `if/elif/fi` statements in the same shell script.

**Listing 4.23: 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 4.23 contains a `while` loop that prompts users for an input string that is processed by a `case` statement. If the input string starts with a lowercase `n` or uppercase `N`, then the variable proceed is initialized with the value `false`. If the input string starts with a lowercase `y` or uppercase `Y`, then the variable proceed is initialized with the value `true`. In all other cases, the variable proceed is initialized with the value unknown.

```
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 4.24 illustrates how to use an `until` loop to iterate through a set of parameters.

**Listing 4.24 until1.sh**

```
x=5
until [ $x -gt 10 ];
do
  echo "x = $x"
  x=`expr $x + 1`
done
```

Listing 4.24 contains an `until` loop that increments the value of `x` (which is initially 5) until it exceeds the value 10. The value of `x` is incremented each time that its value is incremented inside the until loop.

## USER-DEFINED FUNCTIONS

Bash provides built-in functions and enables you to define your own functions. You can define functions to provide the required functionality. Here are simple rules to define a function in a shell script:

1. Function blocks begin with the keyword `function`, followed by the function name and parentheses ( ).
2. The code block within every function starts with a curly brace { and ends with another curly brace }.

A very simple custom function `Hello()` is defined here:

```
function Hello()
{
  echo "Hello World"
}

#invoke the preceding function:
Hello
```

The next example illustrates how to pass a parameter to a function:

```
Hello()
{
  echo "Hello $1"
}

#invoke the preceding function
Hello Bob
```

The next example uses conditional logic to check for the existence of a parameter and then prints the appropriate message:

```
function Hello()
{
  if [ "$1" != "" ]
  then
    echo "Hello $1"
  else
    echo "Please specify a name"
  fi
}

#invoke the preceding function
Hello $1
```

Place the preceding code block in the file `hello.sh`, make it executable (`chmod +x hello.sh`), and launch the code. You will see the following output:

```
Please specify a name
```

Launch the preceding code as follows:

```
./hello.sh pizza
```

The output from the preceding invocation is shown here:

```
Hello pizza
```

## CREATING A SIMPLE MENU FROM SHELL COMMANDS

Listing 4.25 displays the content of `AppendRow.sh` that illustrates how to update a set of users.

**Listing 4.25: AppendRow.sh**

```
dataFile="users.txt"

while (true)
do
  echo "List of Users"
  echo "============"
  cat users.txt 2>/dev/null

  echo "-----------------------------"
  echo "Enter 'x' to exit this menu"
  echo "Enter 'd' to delete all users"
  echo "-----------------------------"
  echo

  echo -n "First Name: "
  read fname

  if [ -n $fname -a $fname == "x" ]
  then
    break
  elif [ -n $fname -a $fname == "d" ]
  then
    rm $dataFile; touch $dataFile
    continue
  fi

  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
done
```

Listing 4.25 initializes the variable `dataFile` with the name of a text file. This is followed by a `while` loop that displays a menu of options and then prompts users for their input. The body of the loop contains several if/else conditional code blocks that check if the variables `fname` and `lname` have been initialized with user-specified values.

If `fname` is set equal to `x`, the program terminates, whereas if `fname` is set equal to `d`, the program deletes `$Datafile`. If `fname` and `lname` are set equal to other values, a new "record" containing their values is appended to `$Datafile`.

A sample invocation of Listing 4.25 is here, which illustrates the insertion of two users. This is followed by exiting the program.

```
List of Users
=============
------------------------------
Enter 'x' to exit this menu
Enter 'd' to delete all users
------------------------------

First Name: John
Last Name: Smith
List of Users
=============
John Smith
------------------------------
Enter 'x' to exit this menu
Enter 'd' to delete all users
------------------------------

First Name: Jane
Last Name: Edwards
List of Users
=============
John Smith
Jane Edwards
------------------------------
Enter 'x' to exit this menu
Enter 'd' to delete all users
------------------------------

First Name: x
```

## SUMMARY

This chapter started with an introduction to arrays as well as arithmetic operations and operators. Then you learned about working with strings in Bash. Next, you learned how to prompt users for input and then test variables for file types and directory types.

In addition, you learned conditional logic, starting with `if/elif` statements and the `case/esac` statement. You learned how to work with `for` loops, `while` loops, and `until` loops.

Finally, you saw how user-defined functions are created, as well as how to display a menu of options and then process the option that users have selected.

CHAPTER 5

# PROCESSING DATASETS WITH GREP AND SED

This chapter introduces you to the versatile `grep` command and `sed` command that can process an input text stream to generate a desired output text stream. Both of these commands work well with other Unix commands. 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 and the `sed` command with commands from previous chapters.

The first part of this chapter introduces the `grep` command used in isolation, combined with regular expression metacharacters (from Chapter 1) and 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 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. 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 fourth section contains a use case that illustrates how to use the `grep` command to find matching lines that are then merged to create a new dataset.

The fifth part of this chapter contains basic examples of the `sed` command, such as replacing and deleting strings, numbers, and letters. The sixth 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.

## 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  2022 abc.txt
-rw-r--r--  1 oswaldcampesato2  staff     6 Sep 21  2022
    control1.txt
-rw-r--r--  1 oswaldcampesato2  staff    27 Sep 28  2022 fiblist.
    txt
-rw-r--r--  1 oswaldcampesato2  staff    28 Sep 14  2022 dest
-rw-r--r--  1 oswaldcampesato2  staff    36 Sep 14  2022 source
-rw-r--r--  1 oswaldcampesato2  staff   195 Sep 28  2022
    Divisors.py
-rw-r--r--  1 oswaldcampesato2  staff   267 Sep 28  2022
    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 a special meaning may be quoted by preceding it with a backslash.

A regular expression may be followed by one of several repetition operators:

- "." 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 content of `lines.txt` that contains lines with words and 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. Note that 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 start with a period, followed by an asterisk (`*`), and then another period, all of which are "escaped" by a backslash that precedes each of these metacharacters (the right-most "$" is an end-of-line anchor):

```
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` and two `.txt`) 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 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: 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 performs 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 lists only the filenames that contain a successful match (note this matches contents of 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, and .csv:

```
grep -l abc *

abc1.txt
abc3.sh
abc.doc
```

The -l option lists only the filenames that contain a successful match:

```
grep –l abc *sh
```

Use the `-il` option to display the filenames that contain a specified string using a case insensitive match:

```
grep -il abc *doc
```

The preceding command is useful when you want to check for the occurrence of a string in Word documents.

The `-n` option specifies the 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 (it returns the 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 also specify a recursive search (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`, and `abcde`.

```
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 that 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 as follows.

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
```

Launch the following commands, where warnings and errors are redirected to `2>/dev/null`, so you will not see them:

```
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")? In this case, you would 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 a 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 they 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. Hence, add an extra command to a pipe if it adds value, otherwise start with a direct call to the files you want 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'` does not match `ba` or `ab` or `bb` (or anything else).

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 become 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 content 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 line 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 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) to distinguish otherwise similar things: for example, `John Smith` the *manager* must not be confused with `John Smith` the *programmer* in an employee dataset. 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 backslash (\) 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 word.
- `\>` = Match the empty string at the end of 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 subdirectories) 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 subdirectories) 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` to create very complex expressions.

Note that `grep -R hello .` also performs a search for the string `hello` in all files, including subdirectories, and follows the "one process" recommendation. However, the `find . -print` command searches for all files in all subdirectories, and you can pipe the output to `xargs grep hello` 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 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 subdirectories) to another directory with the `grep` command:

```
cp `grep -l abc *sh` /tmp
```

Yet another approach is to use back tick so that you can obtain additional information:

```
for file in 'find . -print'
do
   echo "Processing the file: $file"
   # now do something here
done
```

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 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 file names 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 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 invoke this command:

```
./findsvg.sh 1>11 2>22
```

Notice that the preceding command redirects error message (2>) to the file `22` and the results of the `jar/grep` command (1>) to the file `11` (and obviously you can specify different filenames).

## 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 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`, which 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 previous character), "?" (0 or 1 occurrence of previous character), and "|" (alternate matching). The egrep command is almost identical to the grep -E, along with some caveats that are described in the following online article:

*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 "|" to specify multiple words in a search pattern. A match is successful if *any* of the words in the search pattern appears. For example, 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 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, fgrep is available online:

*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`.

You can replace `echo $x` with a dataset 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 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.

## DELETE ROWS WITH MISSING VALUES

The code sample in this section shows you how to use the `awk` command to split the comma-separated fields in the rows of a dataset, where fields can contain nested quotes of arbitrary depth.

Listing 5.7 displays some of the rows in `titanic.csv` and Listing 5.8 displays the contents of the file `delete-empty-cols-grep.sh` that show you how to create a new dataset that contains only rows that are fully populated with data values.

**Listing 5.7: titanic.csv**

```
survived,pclass,sex,age,sibsp,parch,fare,embarked,class,who,adu
    lt_male,deck,embark_town,alive,alone
0,3,male,22.0,1,0,7.25,S,Third,man,True,,Southampton,no,False
1,1,female,38.0,1,0,71.2833,C,First,woman,False,C,Cherbourg,yes
    ,False
1,3,female,26.0,0,0,7.925,S,Third,woman,False,,Southampton,yes,
    True
1,1,female,35.0,1,0,53.1,S,First,woman,False,C,Southampton,yes,
    False
0,3,male,35.0,0,0,8.05,S,Third,man,True,,Southampton,no,True
0,3,male,,0,0,8.4583,Q,Third,man,True,,Queenstown,no,True
// rows omitted for brevity
0,3,male,25.0,0,0,7.05,S,Third,man,True,,Southampton,no,True
0,3,female,39.0,0,5,29.125,Q,Third,woman,False,,Queenstown,n
    o,False
0,2,male,27.0,0,0,13.0,S,Second,man,True,,Southampton,no,True
1,1,female,19.0,0,0,30.0,S,First,woman,False,B,Southampton,yes,
    True
0,3,female,,1,2,23.45,S,Third,woman,False,,Southampton,no,Fa
    lse
1,1,male,26.0,0,0,30.0,C,First,man,True,C,Cherbourg,yes,True
0,3,male,32.0,0,0,7.75,Q,Third,man,True,,Queenstown,no,True
```

**Listing 5.8: delete-empty-cols-grep.sh**

```
#field5,field4,field3,"field2,foo,bar",field1,field6,field7,"fieldZ"
input="titanic.csv"
output="titanic_clean.csv"

row_count1=`wc $input | awk  '{print $1}'`
echo "Number of input rows:  $row_count1"

# compare this code with the awk example in chapter 6:
cat $input |grep -v ",," > $output

row_count2='wc $output | awk '{print $1}''
echo "Number of output rows: $row_count2"

echo
echo "=> First five rows in $input:"
cat $input |head -6 |tail -5
echo "-------------------------"
echo

echo "=> First five rows in $output:"
cat $output |head -6 |tail -5
echo ""
```

Listing 5.8 starts by initializing the variables input and output with the values `titanic.csv` and `titanic_clean.csv`, respectively. Next, the variable `row_count1` is initialized with the number of rows from the input file, and then its value is printed.

The next code snippet uses a combination of the `cat` command and the `grep` command to find all rows that do not contain two consecutive commas (which represent missing values) and redirect that list of rows into the output file. In a similar fashion as the variable `row_count1`, the variable `row_count2` is initialized and its value is printed.

The next block of code in Listing 5.8 uses a combination of the `cat`, `head`, and `tail` commands to extract the first six rows of the input file and then the last five rows of the preceding output, which extracts rows 2 through 6 instead of rows 1 through 5.

Similarly, the final block of code in Listing 5.8 uses a combination of the `cat`, `head`, and `tail` commands to extract the first six rows of the output file and then the last five rows of the preceding output, which extracts rows 2 through 6 instead of rows 1 through 5. Launch the code in Listing 5.8 and you will see the following output:

```
Number of input rows:  892
Number of output rows: 183

=> First five rows in titanic.csv:
0,3,male,22.0,1,0,7.25,S,Third,man,True,,Southampton,no,False
```

```
1,1,female,38.0,1,0,71.2833,C,First,woman,False,C,Cherbourg,yes
    ,False
1,3,female,26.0,0,0,7.925,S,Third,woman,False,,Southampton,yes,
    True
1,1,female,35.0,1,0,53.1,S,First,woman,False,C,Southampton,yes,
    False
0,3,male,35.0,0,0,8.05,S,Third,man,True,,Southampton,no,True
-------------------------

=> First five rows in titanic_clean.csv:
1,1,female,38.0,1,0,71.2833,C,First,woman,False,C,Cherbourg,yes
    ,False
1,1,female,35.0,1,0,53.1,S,First,woman,False,C,Southampton,yes,
    False
0,1,male,54.0,0,0,51.8625,S,First,man,True,E,Southampton,no,
    True
1,3,female,4.0,1,1,16.7,S,Third,child,False,G,Southampton,yes,
    False
1,1,female,58.0,0,0,26.55,S,First,woman,False,C,Southampton,yes
    ,True
```

In Chapter 6, you will see how to perform the same task using the `awk` command, and you might be surprised to learn that the solution for this task is actually simpler using the `grep` command than the `awk` command.

## A SIMPLE USE CASE

The code sample in this section shows you how to use the `grep` command 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 5.9 displays the content of the file `test1.csv` that contains the initial dataset.

**Listing 5.9: 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 5.10 displays the content of the file `joinlines.sh` that illustrates how to merge the pairs of matching lines in `joinlines.csv`.

**Listing 5.10: 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 5.10 is here:

```
1,KLM,,1.4,,0.8,,1.2,,1.1,,,2.2,,,1.41,
    XYZ,,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,K
    LM,,0.03,,0.03,,0.04,0.04,,0.02,,,0.04,,5,
    XYZ,,4.73,,4.48,,4.49,4.40,,,4.59,,,4.63,
```

As you can see, the task in this section is easily solved via the `grep` command. Note that additional data cleaning is required to handle the empty fields in the output.

This concludes the portion of the chapter devoted to the `grep` command. The next portion discusses the `sed` command, along with various examples that illustrate some of its feature.

## WHAT IS THE sed COMMAND?

`sed` ("stream editor") is a utility that derives many of its commands from the `ed` line-editor (`ed` was the first Unix text editor). The `sed` command is a "non-interactive" stream-oriented editor that can be used to automate editing via shell scripts. This ability to modify an entire stream of data (which can be the contents of multiple files, in a manner similar to how `grep` behaves) as if you were inside an editor is not common in modern programming languages. This behavior allows some capabilities not easily duplicated elsewhere, while behaving exactly like any other command (such as `grep`, `cat`, `ls`, and `find`) in how it can accept data, output data, and pattern match with regular expressions.

Some of the more common uses for `sed` include printing matching lines, deleting matching lines, and finding/replacing 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:

- Reads an entire line from `stdin/file`.
- Removes any trailing newline.
- Places the line in its pattern buffer.
- Modifies the pattern buffer according to the supplied commands.
- Prints the pattern buffer to `stdout`.

## MATCHING STRING PATTERNS USING sed

The `sed` command requires you to specify a string 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 in the file using the `sed` command than to pipe it in with a different command. You can "feed" it data from another command if that other command adds value (such as adding line numbers, removing blank lines, or other similar helpful activities).

The `-n` option suppresses all output, and the `p` option prints the matching line. If you omit the `-n` option, then every line is printed, and the `p` option causes the matching line to be printed again. Hence, 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 prior command, you have instructed `sed` to substitute (`"s`) the first text pattern (/`abc`) with the second pattern (/`def`) and no further instructions(/`"`).

Deleting a text pattern is simply a matter of leaving the second pattern empty:

```
echo "abcdefabc" |sed "s/abc//"
```

The result is here:

```
defabc
```

As you see, this only removes the first occurrence of the pattern. You can remove all the occurrences of the pattern by adding the "global" terminal instruction (`/g"`):

```
echo "abcdefabc" |sed "s/abc//g"
```

The result of the preceding command is here:

```
def
```

Note that we are operating directly on the main stream with this command, as we are not using the `-n` tag. You can also suppress the main stream with `-n` and print the substitution, achieving the same output if you use the terminal `p` (print) instruction:

```
echo "abcdefabc" |sed -n "s/abc//gp"

def
```

For substitutions, either syntax will do, but that is not always true of other commands.

You can also remove digits instead of letters by using the numeric metacharacters as your regular expression match pattern (from Chapter 1):

```
ls svcc1234.txt |sed "s/[0-9]//g"
ls svcc1234.txt |sed -n "s/[0-9]//gp"
```

The result of either of the two preceding commands is here:

```
svcc.txt
```

Recall that the file `columns4.txt` contains the following text:

```
123 ONE TWO
456 three four
ONE TWO THREE FOUR
five 123 six
one two three
four five
```

The following `sed` command is instructed to identify the rows between 1 and 3, inclusive (`"1,3`), and delete (`d"`) them from the output:

```
cat columns4.txt  | sed "1,3d"
```

The output is here:

```
five 123 six
one two three
four five
```

The following `sed` command deletes a range of lines, starting from the line that matches `123` and continuing through the file until reaching the line that matches the string `five` (and also deleting as all the intermediate lines). The syntax should be familiar from the earlier matching example:

```
sed "/123/,/five/d" columns4.txt
```

The output is here:

```
one two three
four five
```

### *Replacing Vowels from a String or a File*

The following code snippet shows you how simple it is to replace multiple vowels from a string using the `sed` command:

```
echo "hello" | sed "s/[aeio]/u/g"
```

The output from the preceding code snippet is here:

```
Hullu
```

### *Deleting Multiple Digits and Letters from a String*

Suppose that we have a variable `x` that is defined as follows:

```
x="a123zAB 10x b 20 c 300 d 40w00"
```

Recall that an integer consists of one or more digits, so it matches the regular expression `[0-9]+`, which matches one or more digits. However, you need to specify the regular expression [0-9]* to remove every number from the variable `x`:

```
echo $x | sed "s/[0-9]//g"
```

The output of the preceding command is here:

```
azAB x b  c  d w
```

The following command removes all lowercase letters from the variable x:

```
echo $x | sed "s/[a-z]*//g"
```

The output of the preceding command is here:

```
123AB 10  20  300  4000
```

The following command removes all lowercase and uppercase letters from the variable x:

```
echo $x | sed "s/[a-z][A-Z]*//g"
```

The output of the preceding command is here:

```
123 10  20  300  4000
```

## SEARCH AND REPLACE WITH sed

The previous section showed you how to delete a range of rows of a text file, based on a start line and end line, using either a numeric range or a pair of strings. As deleting is just substituting an empty result for what you match, it should now be clear that a replace activity involves populating that part of the command with something that achieves your desired outcome. This section contains various examples that illustrate how to get the exact substitution you desire.

The following examples illustrate how to convert lowercase abc to uppercase ABC in sed:

```
echo "abc" |sed "s/abc/ABC/"
```

The output of the preceding command is here (which only works on one case of abc):

```
ABC
echo "abcdefabc" |sed "s/abc/ABC/g"
```

The output of the preceding command is here (/g" means it 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 replace 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 5.13 displays the contents of `update2.sh` that replace the occurrence of the string `hello` with the string `goodbye` in the files with the suffix `txt` in the current directory.

**Listing 5.13: update2.sh**

```
for f in `ls *txt`
do
  newfile="${f}_new"
  cat $f | sed -n "s/hello/goodbye/gp" > $newfile
  mv $newfile $f
done
```

Listing 5.13 contains a `for` loop that iterates over the list of text files with the `txt` suffix. For each such file, initialize the variable `newfile` that is created by appending the string `_new` to the first file (represented by the variable `f`). Next, replace the occurrences of `hello` with the string `goodbye` in each file `f`, and redirect the output to `$newfile`. Finally, rename `$newfile` to `$f` using the `mv` command.

If you want to perform the update in matching files in all subdirectories, replace the `for` statement with the following:

```
for f in `find . –print |grep "txt$"`
```

## DATASETS WITH MULTIPLE DELIMITERS

Listing 5.14 displays the content of the dataset `delimiter1.txt` that contains multiple delimiters: |, :, and ^. Listing 5.15 displays the content of `delimiter1.sh` that illustrates how to replace the various delimiters in `delimiter1.txt` with a single comma delimiter (,).

**Listing 5.14: delimiter1.txt**

```
1000|Jane:Edwards^Sales
2000|Tom:Smith^Development
3000|Dave:Del Ray^Marketing
```

**Listing 5.15: delimiter1.sh**

```
inputfile="delimiter1.txt"
cat $inputfile | sed -e 's/:/,/' -e 's/|/,/' -e 's/\^/,/'
```

As you can see, the second line in Listing 5.15 is simple yet powerful: you can extend the `sed` command with as many delimiters as you require to create a dataset with a single delimiter between values. The output from Listing 5.3 is shown here:

```
1000,Jane,Edwards,Sales
2000,Tom,Smith,Development
3000,Dave,Del Ray,Marketing
```

Do keep in mind that this kind of transformation can be a bit unsafe unless you have checked that your new delimiter is *not* already in use. For that a `grep` command is useful (you want 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. Specify `-n` when you want to suppress the printing of the basic stream output:

```
sed -n 's/foo/bar/'
```

Specify `-n` and end with `/p'` when you want to match the result only:

```
sed -n 's/foo/bar/p'
```

We briefly touched on using `-e` to do multiple substitutions, but it can also be used to combine other commands. This syntax lets us separate the commands in the last example:

```
sed -n -e 's/foo/bar/' -e 'p'
```

A more advanced example that hints at the flexibility of `sed` involves the insertion 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. It is possible to work through exactly what `sed` is doing by looking at each element of the command and comparing to the output, even if you don't know the syntax. (Sometimes you will encounter very complex instructions for `sed` without any documentation in the code.)

The output changes after every three characters and we know dot (`.`) matches any single character, so `.{3}` must be telling it to do that (with escape slashes \ because brackets are a special character for `sed`, and it won't interpret it properly if we just leave it as `.{3}`. The `n` is clear enough in the replacement column, so the `&\` must be somehow telling it to insert a character instead of replacing it. The terminal `g` command, of course, means to repeat. To clarify and confirm those guesses, take what you could infer and perform an Internet search.

## WORKING WITH DATASETS

The `sed` utility is very useful for manipulating the contents of text files. For example, you can print ranges of lines, subsets of lines that match a regular expression. You can also perform search-and-replace on the lines in a text file. This section contains examples that illustrate how to perform such functionality.

### *Printing Lines*

Listing 5.16 displays the content of `test4.txt` (doubled-spaced lines) that is used for several examples in this section.

**Listing 5.16: 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, where the basic output prints the whole file, which is why `def` does not get duplicated:

```
cat test4.txt  |sed -n "1,3p" |sed "s/abc/ABC/p"
ABC
ABC

def
```

***Character Classes and sed***

You can also use 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 lowercase letters:

```
cat columns4.txt | sed -n '/[0-9]/p'
```

The output from the preceding snippet is here:

```
one two three
one two
one two three four
one
one three
one four
```

The following code snippet illustrates how to match lines that contain lowercase letters:

```
cat columns4.txt | sed -n '/[a-z]/p'
```

The output from the preceding snippet is here:

```
123 ONE TWO
456 three four
five 123 six
```

The following code snippet illustrates how to match lines that contain the numbers 4, 5, or 6:

```
cat columns4.txt | sed -n '/[4-6]/p'
```

The output from the preceding snippet is here:

```
456 three four
```

The following code snippet illustrates how to match lines that start with any two characters followed by `EE`:

```
cat columns4.txt | sed -n '/^.\{2\}EE*/p'
```

The output from the preceding snippet is here:

```
ONE TWO THREE FOUR
```

***Removing Control Characters***

Listing 5.17 displays the contents of `controlchars.txt` that contains the `^M` control character. The code sample in this section shows you how to remove control characters via the `sed` just like any other character.

**Listing 5.17: controlchars.txt**

```
1 carriage return^M
2 carriage return^M
1 tab character^I
```

The following command removes the carriage return and the tab characters from the text file `ControlChars.txt`:

```
cat controlChars.txt | sed "s/^M//" |sed "s/   //"
```

You cannot see the tab character in the second `sed` command in the preceding code snippet; however, if you redirect the output to the file `nocontrol1.txt`, you can see that there are no embedded control characters in this new file by typing the following command:

```
cat –t nocontrol1.txt
```

## COUNTING WORDS IN A DATASET

Listing 5.18 displays the content of `WordCountInFile.sh` that illustrates how to combine various Bash commands to count the words (and their occurrences) in a file.

**Listing 5.18: wordcountinfile.sh**

```
# The file contents are converted to lowercase via the "tr"
    command
# sed removes commas and periods, then changes whitespace to
    newlines
# uniq needs each word on its own line to count the words
    properly
# Uniq filters unique words and the number of times they
    appeared
# The final sort orders the data by the wordcount.

cat "$1" | xargs -n1 | tr A-Z a-z | \
sed -e 's/\.//g' -e 's/\,//g' -e 's/ /\ /g' | \
sort | uniq -c | sort -nr
```

The previous multi-line command performs the following operations:

- Lists each word in each line of the file
- Shifts characters to lowercase
- Filters out periods and commas
- Changes space between words to linefeed,
- Removes duplicates and sorts numerically

## BACK REFERENCES IN sed

In the first part of the chapter describing `grep`, you learned about back references, and similar functionality is available with the `sed` command. The main difference is that the back references can also be used in the replacement section of the command.

The following `sed` command matches 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
```

## ONE-LINE sed COMMANDS

This section is intended to show more useful problems you can solve with a single line of `sed`, and to expose you to more switches and arguments that can be mixed and matched 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 addresses, 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 do I do <xxx> in sed" will likely either point you in the right direction or to an alternative Bash command that will be helpful.

Listing 5.19 displays the content 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 5.19: 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
```

Print the lines of `data4.txt` that do not 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 from the preceding code snippet 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 from the preceding code snippet is here:

```
line five
```

Print the contents of `data4.txt` and duplicate line five with this command:

```
sed '5p' < data4.txt
```

The output from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 to insert a `<TAB>` character.

Delete the tab characters in `data4.txt` with this command:

```
sed 's/  //g' <data4.txt
```

The output from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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 from the preceding code snippet 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
```

## POPULATE MISSING VALUES WITH THE sed COMMAND

The example in this section shows you how to update the values in datasets without using Python or Pandas. Although this approach is useful in some cases, Pandas does provide significant functionality that is often simpler than a Bash-based counterpart.

Listing 5.20 shows you the content of `missing-titanic-ages.sh` that shows you how to replace missing values in the `titanic.csv` dataset with the string `MISSING`, and then count the number of rows whose `age` value is missing.

**Listing 5.20: missing-titanic-ages.sh**

```
newfile="titanic-sed.csv"

# this command replaces missing values with MISSING:
cat titanic.csv |sed "s/,,/,MISSING,/g" > $newfile

cat $newfile | awk -F"," '
BEGIN { count = 0 }
{
   if ($4 ~ /MISSING/) { count += 1 }
}
END { print "number of missing age values:",count }
'
```

Listing 5.20 initializes the variable `newfile` with the name of CSV file whose missing values are replaced with the string `MISSING`, and the latter is performed by the subsequent code snippet that starts with the `cat` command.

The next portion of Listing 5.20 is an `awk` script that initializes the variable count with the value 0, and then increments this value whenever a row is encountered with a missing value for the `age` column. Launch the code in Listing 5.20 and you will see the following output:

```
number of missing age values: 177
```

## A DATASET WITH 1,000,000 ROWS

The code samples in this section shows you how to use `grep` to perform various comparisons on a dataset that contains 1,000,000 rows.

#### *Numeric Comparisons*

Listing 5.21 shows you how to check for a specific number (e.g., 58) and the occurrence of one, two, or three adjacent digits in the first field of each row.

**Listing 5.21: numeric_comparisons.sh**

```
filename="1000000_HRA_Records_short.csv"
echo "first loop:"
rownum=0
matches=0

while read line
do
field1=`echo $line | cut -d"," -f1`
  if [ rownum > 0 ]
  then
    if [ $field1 > 50 ]
    then
      matches='expr $matches + 1'
    fi
  fi
  rownum='expr $rownum + 1'
done < $filename
echo "matching records: $matches"

echo "second loop:"
rownum=0
matches=0
while read line
do
field1=`echo $line | cut -d"," -f1`
field4=`echo $line | cut -d"," -f4`
field5=`echo $line | cut -d"," -f5`

  if [ rownum > 0 ]
  then
```

```
    if [ $field1 > 50 -a "$field5" == "Support" ]
    then
      matches='expr $matches + 1'
    fi
  fi
  rownum='expr $rownum + 1'
done < $filename
echo "matching records: $matches"
```

Listing 5.21 initializes the variable `newfile` with the name of the CSV file and then initializes some scalar values. Next, a `while` loop processes each row (except for the first row) in the CSV file and initializes the value of the variable `field1` with the contents of the first field of the comma-delimited CSV file. If `field1` is greater than 50, then the variable `matches` is incremented.

The next portion of Listing 5.21 is another `while` loop that processes each row (except for the first row) in the CSV file and then extracts the first, fourth, and fifth fields with the following code block:

```
field1=`echo $line | cut -d"," -f1`
field4=`echo $line | cut -d"," -f4`
field5=`echo $line | cut -d"," -f5`
```

Next, a conditional statement checks whether the value of `field1` is greater than 40 and the value of `field5` equals the string Support, as shown here:

```
if [ $field1 > 40 -a "$field5" == "Support" ]
```

If the preceding statement is true, the variable `matches` is incremented. The final code snippet in the `while` loop updates the value of the variable `rownum` and then the next row in the CSV file is processed. Launch the code in Listing 5.21 and you will see the following output:

```
first loop:
matching records: 1001
second loop:
matching records: 153
```

### *Counting Adjacent Digits*

Listing 5.22 shows you how to find a specific number (e.g., 58) and the occurrence of one, two, or three adjacent digits in the first field of each row.

**Listing 5.22: adjacent_digits.sh**

```
filename="1000000_HRA_Records.csv"

echo "first:"
grep 58 $filename |wc
echo

echo "second:"
grep "[0-9]" $filename |wc
```

```
echo

echo "third:"
grep "[0-9][0-9]" $filename |wc
echo

echo "fourth:"
grep "[0-9][0-9][0-9]" $filename |wc
```

Listing 5.22 initializes the variable `filename` with the name of a CSV file, followed by four code blocks that contain a combination of the `grep` command and the `wc` command.

The first block determines the number of rows that contain the string 58, whereas the second code block determines the number of rows that contain a digit in the CSV file. The third block determines the number of rows that contain two consecutive digits, whereas the fourth code block determines the number of rows that contain three consecutive digits in the CSV file. Launch the code in Listing 5.22 and you will see the following output:

```
first:
  161740  453166 25044202

second:
 1000001 2799978 154857624

third:
 1000001 2799978 154857624

fourth:
 1000000 2799977 154857110
```

***Average Support Rate***

Listing 5.23 uses the Bash commands `echo`, `cut`, and `expr` to calculate the average rate for people who are over 50 and are in the Support department.

**Listing 5.23: average_rate.sh**

```
filename="1000000_HRA_Records.csv"

rownum=0
matches=0
total=0
num_records=0
min_rate=99999
max_rate=0

while read line
do
```

```
  if [ $rownum -gt 0 ]
  then
    field1=`echo $line | cut -d"," -f1`
   field4=`echo $line | cut -d"," -f4`
   field5=`echo $line | cut -d"," -f5`

    if [ $field1 > 40 -a "$field5" == "Support" ]
    then
      total=`expr $total + $field4`
      num_records='expr $num_records + 1'

      if [ $min_rate -gt $field4 ]
      then
        min_rate=$field4
      fi

      if [ $max_rate -lt $field4 ]
      then
        max_rate=$field4
      fi
    fi
  fi
  rownum=`expr $rownum + 1`
done < $filename

avg_rate='expr $total / $num_records'
echo "Number of Records:    $num_records"
echo "Minimum Rate:         $min_rate"
echo "Maximum Rate:         $max_rate"
echo "Sum of All Rates:     $total"
echo "Average Support Rate: $avg_rate"
```

Listing 5.23 initializes the variable `filename` as the name of a CSV file, followed by initializing a set of scalar variables. The main portion of Listing 5.23 consists of a `while` loop that processes each row after the first row in the CSV file, and then extracts the first, fourth, and fifth fields with the following code block:

```
field1=`echo $line | cut -d"," -f1`
field4=`echo $line | cut -d"," -f4`
field5=`echo $line | cut -d"," -f5`
```

Next, a conditional statement checks whether the value of `field1` is greater than 40 and the value of `field5` equals the string Support, as shown here:

```
if [ $field1 > 40 -a "$field5" == "Support" ]
```

If the preceding statement is true, the variables `total` and `num_records` are updated accordingly. Another pair of simple `if` statements determines whether the values of the variables `min_rate` and `max_rate` also need to be updated. The final code snippet in the `while` loop updates the value of the variable `rownum`, and then the next row in the CSV file is processed.

The final code block displays the values of the scalar variables that were initialized in the first section of this code sample:

```
avg_rate='expr $total / $num_records'
echo "Number of Records:    $num_records"
echo "Minimum Rate:        $min_rate"
echo "Maximum Rate:        $max_rate"
echo "Sum of All Rates:    $total"
echo "Average Support Rate: $avg_rate"
```

Launch the code in Listing 5.23 and you will see the following output:

```
Number of Records:    153
Minimum Rate:        107
Maximum Rate:        1489
Sum of All Rates:     118610
Average Support Rate: 775
```

## SUMMARY

This chapter showed you how to work with the `grep` utility, which is a powerful Unix 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 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 performed join-like operations using the `grep` command.

CHAPTER 6

# PROCESSING DATASETS 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 a programming language in a single command, which accepts standard input, gives standard output and uses regular expressions and metacharacters in the same way as other Unix commands. This functionality enables you to combine awk with other command line tools. Include commands in awk scripts because its versatility can make an awk script challenging to understand based on just a quick 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 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 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 to determine whether to print a line of text.

The fourth section illustrates how to "split" a text string that contains multiple "." characters as delimiters, 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, select a subset of columns from a dataset, and work with multi-line 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.

Please keep in mind the following points before you read this chapter. First, the datasets in this chapter are very short so that you can focus on learning the rich feature set of the `awk` command. After you have completed this chapter, you can use the `awk`, `grep`, and `sed` commands in any combination that you need to process your own datasets.

Datasets containing 25 terabytes of data have been successfully processed via `awk`. If you work with multi-terabyte datasets, most likely you will process them in a cloud-based environment. Moreover, the `awk` command is useful for data cleaning tasks involving datasets of almost any size, and some examples of such tasks are discussed in Chapter 7.

## THE awk COMMAND

The `awk` (Aho, Weinberger, and Kernighan) command has a C-like syntax and you can use this utility to perform very complex operations on numbers and text strings.

As a side comment, there is also the `gawk` command that is GNU `awk`, as well as the `nawk` command is "new" `awk` (neither command is discussed in this book). One advantage of `nawk` is that it allows you to set externally the value of an internal variable.

### *Built-in Variables that Control awk*

The `awk` command provides variables that you can change from their default values 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 default of one newline to two newlines, as shown here:

```
cat columns.txt | awk 'BEGIN { ORS ="\n\n" } ; { print $0 }'
```

Alternatively, you can include one or more print (or `printf`) statements in the body of an `awk` script, depending on the number of blank lines that you want to appear in the output after each input line. In fact, you can use conditional logic to print a different number of blank lines after a given input line.

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 (specified by you), then an associated action is performed (otherwise no action is performed). If the search pattern is not given, then `awk` performs the given actions for each record of the input. The default behavior if no action is given is to print all the records that match the given pattern. Finally, empty braces without any action results in nothing; i.e., the program will not perform the default printing operation. Note that each statement in the actions should be delimited by a semicolon.

To make the preceding paragraph more understandable, 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
```

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 "abcde" | awk 'BEGIN { FS = "" } ; { print NF }'
5
```

The following example illustrates several equivalent ways to specify `test.txt` as the input file for an `awk` command:

- `awk < test.txt '{ print $1 }'`
- `awk '{ print $1 }' < test.txt`
- `awk '{ print $1 }' test.txt`

Yet another way is shown here (but as we’ve discussed earlier, it can be inefficient, so only do it if the `cat` command adds value in some way):

```
cat test.txt | awk '{ print $1 }'
```

The preceding four ways to perform the same task illustrate why the inclusion of comments in `awk` scripts is generally a good idea. The next person to look at your code may not know/remember the syntax you are using (and that person might be you).

## 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 6.1 displays the content of `columns2.txt` and Listing 6.2 displays the content of the shell script `AlignColumns1.sh` that shows you how to align the columns in a text file.

**Listing 6.1: columns2.txt**

```
one two
three four
one two three four
five six
one two three
four five
```

**Listing 6.2: 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 6.2 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 6.2 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 supports 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 6.3 displays the content 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` (and most C-derived languages).

**Listing 6.3: Loop.sh**

```
awk '
BEGIN {}
{
  for(i=0; i<5; i++) {
    printf("%3d", i)
  }
}
END { print "\n" }
'
```

Listing 6.3 contains a `for` loop that prints numbers on the same line via the `printf()` statement. Notice that a new line is printed only in the `END` block of the code. The output from Listing 6.3 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 6.4 displays the content of `linepairs.csv` and Listing 6.5 displays the content of `deletelines.sh` that illustrates how to print alternating lines from the dataset `line-pairs.csv` that have exactly two columns.

**Listing 6.4: linepairs.csv**

```
a,b,c,d
e,f,g,h
1,2,3,4
5,6,7,8
```

**Listing 6.5: deletelines.sh**

```
inputfile="linepairs.csv"
outputfile="linepairsdeleted.csv"
awk ' NR%2 {printf "%s", $0; print ""; next}' < $inputfile >
    $outputfile
```

Listing 6.5 specifies `NR%2` to determine whether the current record number `NR` is divisible by 2, in which case, it prints the current line and then specifies `next` to skip 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 6.6 displays the content of `columns.txt`, and Listing 6.7 displays the content of `ColumnCount1.sh` that illustrates how to print the lines from the text file `columns.txt` that have exactly two columns.

**Listing 6.6: columns.txt**

```
one two three
one two
one two three four
one
one three
one four
```

**Listing 6.7: ColumnCount1.sh**

```
awk '
{
   if( NF == 2 ) { print $0 }
}
' columns.txt
```

Listing 6.7 is straightforward: if the current record contains an even number of fields, then the current line is printed (i.e., odd-numbered rows are skipped). The output from launching the code in Listing 6.7 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 content of `test4.txt` as a single line:

```
awk '{printf("%s", $0)}' test4.txt
```

The output of the preceding code snippet is here. See if you can tell what is happening before reading the explanation in the next paragraph:

```
Abcdefabcabc
```

Explanation: `%s` here is the record separator syntax for `printf();` with the end quote after it means the record separator is the empty field "". Our default record separator for `awk` is `/n` (newline): `printf` is stripping out all the newlines. The blank rows will vanish entirely, as all they have is the newline, so the result is that any actual text will be merged together with nothing between them. Had we added a space between the `%s` and the ending quote, there would be a space between each character block, plus an extra space for each newline.

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 6.8 displays the content of `digits.txt`, and Listing 6.9 displays the content of `digits.sh` that "joins" three consecutive lines of text in the file `digits.txt`.

**Listing 6.8: digits.txt**

```
1
2
3
4
5
6
7
8
9
```

**Listing 6.9: digits.sh**

```
awk -F" " '{
  printf("%d",$0)
  if(NR % 3 == 0) { printf("\n") }
}' digits.txt
```

Listing 6.9 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 6.10 displays the content of `columns2.txt`, and Listing 6.11 displays the content of `JoinLines.sh` that "joins" two consecutive lines of text in the file `columns2.txt`.

**Listing 6.10: columns2.txt**

```
one two
three four
one two three four
five six
one two three
four five
```

**Listing 6.11: JoinLines.sh**

```
awk '
{
   printf("%s",$0)
   if( $1 !~ /one/) { print " " }
}
' columns2.txt
```

The output from launching Listing 6.11 is here:

```
one two three four
one two three four five six
one two three four five
```

Notice that the code in Listing 6.11 depends on the presence of the string `one` as the first field in alternating lines of text. We are merging data based on matching a simple pattern, instead of tying it to record combinations.

To merge each pair of lines instead of merging based on matching a pattern, use the modified code in Listing 6.12.

**Listing 6.12: 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 6.13, 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 a puzzle if encountered in a program without a comment. It is doing exactly the same thing as Listing 6.12, but its purpose is less obvious because of the more compact syntax.

**Listing 6.13: 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, we can also match a regular expression, just as we did in `grep` and `sed`. Listing 6.14 displays the content of `Patterns1.sh` that uses metacharacters to match the beginning and the end of a line of text in the file `columns2.txt`.

**Listing 6.14: Patterns1.sh**

```
awk '
   /^f/    { print $1 }
   /two $/ { print $1 }
' columns2.txt
```

The output from launching Listing 6.14 is here:

```
one
five
four
```

Listing 6.15 displays the content of `RemoveColumns.txt` with lines that contain a different number of columns.

**Listing 6.15: columns3.txt**

```
123 one two
456 three four
one two three four
five 123 six
one two three
four five
```

Listing 6.16 displays the content of `MatchAlpha1.sh` that matches text lines starting with alphabetic characters as well as lines that contain numeric strings in the second column.

**Listing 6.16: MatchAlpha1.sh**

```
awk '
{
   if( $0 ~ /^[0-9]/) { print $0 }
   if( $0 ~ /^[a-z]+ [0-9]/) { print $0 }
}
' columns3.txt
```

The output from Listing 6.16 is here:

```
123 one two
456 three four
five 123 six
```

## PRINTING LINES USING CONDITIONAL LOGIC

Listing 6.17 displays the content of `products.txt` that contains three columns of information.

**Listing 6.17: 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 6.18 displays the content of `SplitFilename2.sh` that illustrates how to split a filename containing the "." character  to increment the numeric value of one of the components of the filename. Note that this code only works for a file name with exactly the expected syntax. It is possible to write more complex code to count the number of segments, or alternately, to say “change the field right before .zip,” which would only require the filename had a format matching the final two sections `(<anystructure>.number.zip)`.

**Listing 6.18: 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 6.18 is here:

```
05.20.144q.az.2.zip
```

## WORKING WITH POSTFIX ARITHMETIC OPERATORS

Listing 6.19 displays the content 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 6.19: mixednumbers.txt**

```
324.000-|10|983.000-
453.000-|30|298.000-
783.000-|20|347.000-
```

Listing 6.20 displays the content of `AddSubtract1.sh` that illustrates how to add the rows of numbers in Listing 6.19.

**Listing 6.20: 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 6.20 is here. 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 (the first position is a number, and the second position is either an empty value or a dash) and then captures the first position number into a variable. The `if` statement determines 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 up the values in the line.

## 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 the 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.

Often, you want random integers instead. Here is a user-defined function you can use to obtain a random nonnegative 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 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.

These code snippets do the following actions:

Prints each line preceded by the number of fields in each line:

```
awk '{print NF ":" $0}' short1.txt
```

Prints the right-most field in each line:

```
awk '{print $NF}' short1.txt
```

Prints the lines that contain more than 2 fields:

```
awk '{if(NF > 2) print }' short1.txt
```

Prints the value of the right-most field if the current line contains more than 2 fields:

```
awk '{if(NF > 2) print $NF }' short1.txt
```

Removes the leading and trailing whitespaces:

```
echo " a b c " | awk '{gsub(/^[ \t]+|[ \t]+$/,"");print}'
```

Prints 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
```

Prints 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 6.21 displays the content of the file `data.txt` that is used in various code samples in this section.

**Listing 6.21: 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 by the expand utility to change tabs into spaces, so the widths compared are actually the right-margin columns. This snippet prints every line that has at least one field:

```
awk 'NF > 0' data.txt
```

The output from the preceding code snippet shows you how to create a file whose contents do not include blank lines.

This snippet prints seven random numbers from 0 to 100, inclusive:

```
awk 'BEGIN { for (i = 1; i <= 7; i++)
print int(101 * rand()) }'
```

This snippet counts the lines in a file:

```
awk 'END { print NR }' < data.txt
```

This snippet prints 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.

This snippet inserts a duplicate of every line in a text file:

```
awk '{print $0, '\n', $0}' < data.txtThis snippet inserts a
    duplicate of every line in a text file and also remove blank
    lines:


awk '{print $0, "\n", $0}' < data.txt | awk 'NF > 0'
```

This snippet inserts 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 6.22 displays the content of `Fields2.sh` that illustrates how to print the words in a text string using the `awk` command.

**Listing 6.22: Fields2.sh**

```
echo "a b c d e"| awk '
{
  for(i=1; i<=NF; i++) {
     print "Field ",i,":",$i
  }
}
'
```

The output from Listing 6.22 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 6.23 and Listing 6.24 display the content `data1.csv` and `data2.csv`, respectively. Listing 6.25 displays the content 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 6.23: 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 6.24: 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 6.25: 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 6.25 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 6.25 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 `$3 == "past"` in `checkrows.sh` 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 6.26 displays the content of `FixedFieldCount1.sh` that illustrates how to print the words in a text string using the `awk` command.

**Listing 6.26: 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 6.26 is here:

```
aa bb cc
dd ee ff
gg hh
```

## PRINTING A DATASET IN A FIXED NUMBER OF COLUMNS

Listing 6.27 displays the content of `VariableColumns.txt` with lines of text that contain a different number of columns.

**Listing 6.27: VariableColumns.txt**

```
this is line one
this is line number one
this is the third and final line
```

Listing 6.28 displays the content of `Fields3.sh` that illustrates how to print the words in a text string using the `awk` command.

**Listing 6.28: 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 6.28 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 6.29 displays the content of `mixed-data.csv` with misaligned data values. In addition, the first line and final line in Listing 6.28 are empty lines, which will be removed by the shell script in this section.

**Listing 6.29: 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 6.30 displays the content of `mixed-data.sh` that illustrates how to realign the dataset in Listing 6.29.

**Listing 6.30: mixed-data.sh**

```
#--------------------------------------------------
# 1) remove blank lines
# 2) remove line feeds
# 3) print a line feed 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 6.30 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 6.30 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 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 places two records in each line of the new dataset.

Listing 6.31 displays the content of `mixed-data2.csv` with misaligned data values, followed by Listing 6.32, which displays the content of `aligned-data2.csv` with the correctly formatted dataset.

**Listing 6.31: 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 6.32: 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 6.33 displays the content of `mixed-data2.sh` that illustrates how to realign the dataset in Listing 6.31.

**Listing 6.33: 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 6.33 is similar to Listing 6.30. The main idea of the code 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 6.33.

You can generalize Listing 6.33 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.

## DISPLAYING A SUBSET OF COLUMNS IN A TEXT FILE

Listing 6.34 displays the content of `VariableColumns.txt` with lines of text that contain a different number of columns.

**Listing 6.34: products.txt**

```
MobilePhone 400  new
Tablet      300  new
Tablet      300  used
MobilePhone 200  used
MobilePhone 100  used
```

Listing 6.35 displays the content of `RemoveColumn.sh` that removes the first column from the text file.

**Listing 6.35: 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
```

Listing 6.36 displays the content of `RemoveColumns.sh` that contains an `awk` statement that prints all but the third column of a text file, followed by an `awk` statement that prints all but the third and seventh columns of a text file.

**Listing 6.36: RemoveColumns.sh**

```
filename="aligned-data2.csv"
echo "=> contents of $filename:"
cat $filename
echo ""

# print all but the third column:
echo "Skipping field 3:"
awk '
{
  for (i=1; i<=NF; i++) {
     if( i != 3 ) {
        printf("%s ", $i)
     }
  }
  print ""
}' $filename
echo ""

# print all but the third and seventh columns:
echo "Skipping field 3 and field 7:"
awk '
{
  for (i=1; i<=NF; i++) {
     if( (i != 3 ) && ( i != 7 ) ) {
        printf("%s ", $i)
     }
  }
  print ""
}' $filename
```

Listing 6.36 is an enhancement of the Listing 6.35, and it starts by initializing the variable `filename` with the name of a CSV file. Next, the loop in the first `awk` statement prints each column of each row in the CSV file unless it's the third column. The loop in the second `awk` statement prints each column of each row in the CSV file unless it's the third column or the seventh column. Launch the code in Listing 6.36 and you will see the following output:

```
=> contents of 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
```

```
Skipping field 3:
Sara, Jones, CA, Sally, Smith, 2000, IL
Dave, Jones, FL, John, Jones, 4000, CA
Dave, Jones, NY, Mike, Jones, 6000, NY
Tony, Jones, WA

Skipping field 3 and field 7:
Sara, Jones, CA, Sally, Smith, IL
Dave, Jones, FL, John, Jones, CA
Dave, Jones, NY, Mike, Jones, NY
Tony, Jones, WA
```

## SUBSETS OF COLUMN-ALIGNED ROWS IN DATASETS

The code sample in this section illustrates how to extract a subset of the existing columns and a subset of the rows. Listing 6.37 displays the content of `sub-rows-cols.txt` of the desired dataset that contains two columns from every even row of the file `aligned-data.txt`.

**Listing 6.37: sub-rows-cols.txt**

```
Sara, 1000
Dave, 3000
Dave, 5000
Tony, 7000
```

Listing 6.38 displays the content of `sub-rows-cols.sh` that illustrates how to generate the dataset in Listing 6.37. Most of the code is the same as Listing 6.33, with the new code shown in bold.

**Listing 6.38: 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.txt"

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 6.38 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 6.38. Notice that that the `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 to remove a trailing "," and a trailing space, as shown here:

```
cat temp-columns3 | sed 's/,$//' | sed 's/ $//' > $outputfile
```

There are other ways to perform the functionality in Listing 6.38, and the main purpose is to show you different techniques for combining various Bash commands.

## COUNTING WORD FREQUENCY IN DATASETS

Listing 6.39 displays the content of `WordCounts1.sh` that illustrates how to count the frequency of words in a file.

**Listing 6.39: 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 6.39 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 6.39 is here:

```
two      3
one      3
three    3
six      1
four     3
five     2
```

Listing 6.40 displays the content of `columns4.txt`, and Listing 6.41 displays the content of `WordCounts2.sh` that performs a case insensitive word count.

**Listing 6.40: columns4.txt**

```
123 ONE TWO
456 three four
ONE TWO THREE FOUR
five 123 six
one two three
four five
```

**Listing 6.41: 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 6.41 includes the following code snippet that converts the text in each input line to lowercase letters:

```
$0 = tolower($0)
```

The output from launching Listing 6.41 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 6.42 displays the content of `onlywords.sh` that contains three `awk` commands for displaying the words, integers, and alphanumeric strings, respectively, in a text string.

**Listing 6.42: 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 6.42 starts by initializing the variable `x` as a space separated set of tokens:

```
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 6.42 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 to retrieve only alphabetic strings from that dataset.

## DELETE ROWS WITH MISSING VALUES

The code sample in this section shows you how to use the `awk` command to split the comma-separated fields in the rows of a dataset, where fields can contain nested quotes of arbitrary depth.

Listing 6.43 displays some of the rows in `titanic.csv` and Listing 6.44 displays the content of the file `delete-empty-cols-awk.sh` that shows you how to create a new dataset whose rows are fully populated with data values.

**Listing 6.43: titanic.csv**

```
survived,pclass,sex,age,sibsp,parch,fare,embarked,class,who,
    adult_male,deck,embark_town,alive,alone
0,3,male,22.0,1,0,7.25,S,Third,man,True,,Southampton,no,False
1,1,female,38.0,1,0,71.2833,C,First,woman,False,C,Cherbourg,yes
    ,False
1,3,female,26.0,0,0,7.925,S,Third,woman,False,,Southampton,yes,
    True
1,1,female,35.0,1,0,53.1,S,First,woman,False,C,Southampton,yes,
    False
0,3,male,35.0,0,0,8.05,S,Third,man,True,,Southampton,no,True
0,3,male,,0,0,8.4583,Q,Third,man,True,,Queenstown,no,True
// rows omitted for brevity
0,3,male,25.0,0,0,7.05,S,Third,man,True,,Southampton,no,True
0,3,female,39.0,0,5,29.125,Q,Third,woman,False,,Queenstown,no,
    False
0,2,male,27.0,0,0,13.0,S,Second,man,True,,Southampton,no,True
1,1,female,19.0,0,0,30.0,S,First,woman,False,B,Southampton,yes,
    True
0,3,female,,1,2,23.45,S,Third,woman,False,,Southampton,no,
    False
1,1,male,26.0,0,0,30.0,C,First,man,True,C,Cherbourg,yes,True
0,3,male,32.0,0,0,7.75,Q,Third,man,True,,Queenstown,no,True
```

**Listing 6.44: delete-empty-cols-awk.sh**

```
input="titanic.csv"
output="titanic_clean.csv"

row_count1='wc $input | awk '{print $1}''
echo "Number of input rows:  $row_count1"

# compare the awk code with the grep example in chapter 5:
awk -F"," '
{
  if ($0 !~ /,,/) { print $0 }
}' < $input > $output

row_count2=`wc $output | awk  '{print $1}'`
echo "Number of output rows: $row_count2"

echo
echo "=> First five rows in $input:"
cat $input |head -6 |tail -5
echo "--------------------------"
echo

echo "=> First five rows in $output:"
cat $output |head -6 |tail -5
echo ""
```

Listing 6.44 starts by initializing the variables input and output with the values `titanic.csv` and `titanic_clean.csv`, respectively. The next code snippet is an `awk` command that extracts the rows from the CSV file that do not contain two consecutive commas (which indicate a missing field value) and redirects those rows to the output file.

The next code snippet initializes the variable `row_count2` with the rows 2 through 6 of the input file and displays their contents, followed by a code snippet that performs the same operation using the input file. Launch the code in Listing 6.44 and you will see the following output:

```
Number of input rows:  892
Number of output rows: 183

=> First five rows in titanic.csv:
0,3,male,22.0,1,0,7.25,S,Third,man,True,,Southampton,no,False
1,1,female,38.0,1,0,71.2833,C,First,woman,False,C,Cherbourg,yes
    ,False
1,3,female,26.0,0,0,7.925,S,Third,woman,False,,Southampton,yes,
    True
```

```
1,1,female,35.0,1,0,53.1,S,First,woman,False,C,Southampton,yes,
    False
0,3,male,35.0,0,0,8.05,S,Third,man,True,,Southampton,no,True
-------------------------

=> First five rows in titanic_clean.csv:
1,1,female,38.0,1,0,71.2833,C,First,woman,False,C,Cherbourg,yes
    ,False
1,1,female,35.0,1,0,53.1,S,First,woman,False,C,Southampton,yes,
    False
0,1,male,54.0,0,0,51.8625,S,First,man,True,E,Southampton,no,
    True
1,3,female,4.0,1,1,16.7,S,Third,child,False,G,Southampton,yes,
    False
1,1,female,58.0,0,0,26.55,S,First,woman,False,C,Southampton,yes
    ,True
```

As a quick refresher, the file `delete-empty-cols-grep.sh` contains the following code snippet that skips any records that contain two consecutive commas, which indicate a missing value for a feature:

```
cat $input |grep -v ",," > $output
```

Although the `awk` command is more powerful than `grep`, sometimes you can perform the same task with less (and more intuitive) code using the `grep` command. Hence, it's worthwhile for you to gain proficiency in `grep`, `sed`, and `awk` so that you can create shell scripts that are clear, concise, and can also be enhanced (or debugged) with less effort.

Moreover, you can take advantage of the power of `grep`, `sed`, and `awk` to perform pre-processing on datasets before you perform data cleaning tasks using utilities such as Pandas.

## WORKING WITH MULTI-LINE RECORDS IN AWK

Listing 6.45 displays the content of `employee.txt` and Listing 6.46 displays the content of `Employees.sh` that illustrates how to concatenate text lines in a file.

**Listing 6.45: 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 6.46: 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 6.46 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 various type 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 to split the comma-separated fields in the rows of a dataset, where fields can contain nested quotes of arbitrary depth.

Listing 6.47 displays the content of the file `quotes3.csv` that contains a "," delimiter and multiple quoted fields.

**Listing 6.47: quotes3.csv**

```
field5,field4,field3,"field2,foo,bar",field1,field6,field7,"fieldZ"
fname1,"fname2,other,stuff",fname3,"fname4,foo,bar",fname5
"lname1,a,b","lname2,c,d","lname3,e,f","lname4,foo,bar",lname5
```

Listing 6.48 displays the content of the file `delim1.sh` that illustrates how to replace the delimiters in `quotes3.csv` with a "," character.

**Listing 6.48: 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 6.48 is here:

```
LINE #1: field5,field4,field3,"field2,foo,bar",field1,field6,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,foo,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","lname4,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.

## ANOTHER USE CASE

The code sample in this section shows you how to use the `awk` command 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 6.49 displays the content of the file `dates2.csv` that contains a "," delimiter and three fields.

**Listing 6.49: dates2.csv**

```
Jane,Smith,20140805234658
Jack,Jones,20170805234652
Dave,Stone,20160805234655
John,Smith,20130805234646
Jean,Davis,20140805234649
Thad,Smith,20150805234637
Jack,Pruit,20160805234638
```

Listing 6.50 displays the content of `string2date2.sh` that converts the date field to a new format and shifts the new date to the first field.

**Listing 6.50: 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($0,9,2)":"substr($
    0,11,2)":"substr($0,13,2)}''

  # append newly formatted row to output file
  echo "${newdate},${fname},${lname}" >> $outputfile
done
```

Listing 6.50 initializes the `inputfile` and `outputfile` variables with the names of the two CSV files. The next code snippet removes those files, thereby ensuring that they are empty. The next portion of Listing 6.50 contains a `for` loop that iterates through the rows in the input file, and initializes the variables `fname`, `lname`, and `date1` with the contents of the first, second, and third fields, respectively, of each input line.

The next code block constructs the variable `newdate` via an `awk` statement that extracts the appropriate contents of each input row using the `substr()` function. The final portion of Listing 6.50 appends a new string to the `outputfile` that consists of the comma-separated values of `newdate`, `fname`, and `lname`. Launch Listing 6.50 and you will see the following output:

```
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
```

## A DATASET WITH 1,000,000 ROWS

The code samples in this section shows you how to use awk to perform various comparisons on a dataset that contains 1,000,000 rows.

### *Numeric Comparisons*

The code snippets in this section illustrate how to check for a specific number (e.g., 58) and the occurrence of one, two, or three adjacent digits in the first field of each row.

Listing 6.51 displays the content numeric_comparisons.sh that shows you how to work with a dataset that contains 1,000,000 rows.

**Listing 6.51: numeric_comparisons.sh**

```
filename="1000000_HRA_Records.csv"

echo "first:"
awk -F"," '{ if($1 > 50) { print $1 } }' < $filename | wc
echo

echo "second:"
awk -F"," '{ if($1 > 50 && $5 > 70) { print $4 } }' < $filename |
    wc
```

Listing 6.51 initializes the variable filename with a CSV filename, followed by an awk statement that counts the number of rows in which the first field is greater than 50.

The second and final awk statement counts the number of rows whose first field is greater than 50 and whose fifth field is greater than 70. Launch the code in Listing 6.48 and you will see the following output:

```
first:
  232161  232161  696484

second:
  232161  232161  696484

third:
  232161  232161 1011843
```

### *Counting Adjacent Digits*

The code snippet in this section illustrates how to check for a specific number (e.g., 58) and the occurrence of one, two, or three adjacent digits in the first field of each row.

**Listing 6.52: adjacent_digits.sh**

```
filename="1000000_HRA_Records.csv"

echo "first:"
awk -F"," '{ if($1 ~ /58/) { print $1} }'    < $filename  |wc
```

```
echo

echo "second:"
awk -F"," '{ if($1 ~ /[0-9]/) { print $1} }' <$filename  |wc
echo

echo "third:"
awk -F"," '{ if($1 ~ /[0-9][0-9]/) { print $1} }' < $filename |wc
echo

echo "fourth:"
awk -F"," '{ if($1 ~ /[0-9][0-9][0-9]/) { print $1} }' < $filename
    |wc
```

Listing 6.52 initializes the variable filename with a CSV filename, followed by three `awk` statements. The first `awk` statement counts the number of rows whose first field contains the string 58. The second `awk` statement counts the number of rows whose first field contains a digit, and the third `awk` statement counts the number of rows whose first field contains three consecutive digits. Launch the code in Listing 6.52 and you will see the following output:

```
first:
   23199       23199       69597

second:
 1000000       1000000     3000000

third:
 1000000       1000000     3000000

fourth:
       0       0           0
```

***Average Support Rate***

The code snippet in this section illustrates how to check for a specific number (e.g., 58) and the occurrence of one, two, or three adjacent digits in the first field of each row.

**Listing 6.53: average_rate.sh**

```
filename="1000000_HRA_Records.csv"

awk -F"," '
BEGIN { total = 0; num_records = 0 }
{
  if($1 > 40 && $5 == "Support") {
    total += $4
    num_records += 1
  }
}
END {
  avg_rate = total/num_records
  print "Number of Records:   ",num_records
  print "Average Support Rate:",avg_rate
}
' < $filename
```

Listing 6.53 initializes the variable `filename` with a CSV filename, followed by an `awk` statement that calculates the total value in the fourth field of the rows whose first field is greater than 40 and whose fifth field equals the string `Support`.

The final portion of Listing 6.53 calculates the `avg_rate`, which is the total value divided by the number of matching rows. Next, the `avg_rate` and `num_records` are displayed. Launch the code in Listing 6.53 and you will see the following output:

```
Number of Records:          77482
Minimum Rate:               100
Maximum Rate:               1500
Average Support Rate:       798.935
```

## SUMMARY

This chapter introduced the `awk` command, which is essentially an entire programming language packaged into a single Unix command.

We explored some of its built-in variables, as well as conditional logic, `while` loops, and `for` loops in `awk`, to manipulate the rows and columns in datasets. You then saw how to delete lines and merge lines in datasets, and 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, as well as some numeric functions.

In addition, you saw how to align columns in a dataset, delete columns, select a subset of columns from a dataset, and work with multi-line records in datasets. Finally, you saw some simple use cases involving nested quotes and date formats in a structured dataset.

At this point, you have all the tools necessary to do sophisticated data cleansing and processing. You should try to apply them to some task or problem of interest. The final step of the learning process is using the tools to solve real problems.

CHAPTER 7

# PROCESSING DATASETS (PANDAS)

This chapter shows you how to clean data, which includes missing data, incorrect data, and duplicate data.

The first part of this chapter contains several Pandas code samples that use Pandas to read CSV files and then calculate statistical values such as the mean, median, mode, and standard deviation.

The second part of this chapter uses Pandas to handle missing values in CSV files, starting with CSV files that contain a single column, followed by two-column CSV files. These code samples will prepare you to work with multi-column CSV files, such as the custom `bmi.csv` file and Titanic `titanic.csv` file.

After you have completed this chapter, you will be ready to learn how to "split" CSV files into subregions that are then processed via classification algorithms, such as kNN, decision trees, and random forests.

## PREREQUISITES FOR THIS CHAPTER

This chapter contains a mixture of Python-based code samples and an awk-based shell script. All the code samples are straightforward, and if you can follow the Pandas and awk-based code samples in Chapter 1, then you will most likely be able to understand the code samples in this chapter.

This chapter requires basic knowledge of Python and Pandas, such as creating Pandas data frames, as well as reading and writing CSV files. Knowledge of the awk programming language is required for three shell scripts that invoke the `awk` command if you decide to read those code samples.

In addition, the following list contains all the `import` statements that you will encounter in the Python code samples for this chapter:

- `from imblearn.over_sampling import SMOTE`
- `from scipy import stats`

- `from sklearn.covariance import EllipticEnvelope`
- `from sklearn.linear_model import LogisticRegression`
- `from sklearn.metrics import confusion_matrix`
- `from sklearn.metrics import classification_report`
- `from sklearn.model_selection import train_test_split`
- `import numpy as np`
- `import pandas as pd`
- `import random`

## ANALYZING MISSING DATA

This section contains subsections that describes types of missing data, common causes of missing data, and various ways to input values for missing data. Keep in mind that outlier detection, fraud detection, and anomaly detection pertain to *existing* data that is problematic, with varying degrees of severity.

By contrast, missing data presents a different type of issue that raises the following question: what can you do about the missing values? Is it better to discard data points (e.g., rows in a CSV file) with missing values or is it better to estimate reasonable values as a replacement for the missing values? Missing data can adversely affect a thorough analysis of a dataset, whereas erroneous data can increase bias and uncertainty.

At this point, you've undoubtedly realized that a single solution does not exist for every dataset: you need to perform an analysis on a case-by-case basis, after you have learned some of the techniques that might help you effectively address missing data values.

### *Causes of Missing Data*

There are various reasons for missing values in a dataset, some of which are listed here:

- values are unavailable
- values were improperly collected
- inaccurate data entry

Although you might be tempted to *always* replace a missing value with a concrete value, there are situations in which you cannot determine a value. As a simple example, a survey that contains questions for people under 30 will have a missing value for respondents who are over 30, and in this case, specifying a value for the missing value would be incorrect. With these details in mind, there are various ways to fill missing values, some of which are listed here:

- remove row with a high percentage of missing values (50% or larger)
- one-hot encoding for categorical data
- handling data inconsistency
- use the Imputer class from the scikit-learn library
- fill missing values with the value in an adjacent row
- replace missing data with the mean/median/mode value
- infer ("impute") the value for missing data
- delete rows with missing data

Once again, the technique that you select for filling missing values is influenced by various factors, such as

- how you want to process the data
- the type of data involved
- the cause of missing values (see above)

Although the most common technique involves the mean value for numeric features, someone needs to determine which technique is appropriate for a given feature.

However, if you are not confident that you can impute a reasonable value, consider deleting the row with a missing value, and then train a model with the imputed value and deleted row.

One problem that can arise after removing rows with missing values is that the resulting dataset is too small. In this case, consider using SMOTE (Synthetic Minority Oversampling Technique), which is discussed later in this chapter  to generate synthetic data.

## PANDAS, CSV FILES, AND MISSING DATA

This section contains several subsections with Python-based code samples that create Pandas data frames and then replace missing values in the data frames. First, we'll look at small CSV files with one column and then we'll look at small CSV files with two columns. Later, we'll look at skewed CSV files as well as multi-row CSV files.

### *Single Column CSV Files*

Listing 7.1 displays the content of the CSV file `one_char_column1.csv`, and Listing 7.2 displays the content of `one_char_column1.py` that fills in the missing values in the CSV file.

**Listing 7.1: one_char_column1.csv**

```
gender
Male
Male
NaN
Female
Male
```

**Listing 7.2: one_char_column1.py**

```
import pandas as pd

df1 = pd.read_csv('one_char_column1.csv')

print("=> initial dataframe contents:")
print(df1)
print()

df = df1.fillna("FEMALE")
```

```
print("dataframe after fillna():")
print(df)
print()
```

Listing 7.2 starts with two `import` statements and then initializes the Pandas data frame `df1` with the content of `one_char_column1.csv`, after which its contents are displayed. The next code block invokes the `fillna()` method to replace the missing values with the string `FEMALE`. Launch the code in Listing 7.2 and you will see the following output:

```
=> initial dataframe contents:
   gender
0    Male
1    Male
2     NaN
3  Female
4    Male

dataframe after fillna():
   gender
0    Male
1    Male
2  FEMALE
3  Female
4    Male
```

Listing 7.3 displays the content of the CSV file `one_char_column2.csv`, and Listing 7.4 displays the content of `one_char_column2.py` that fills in the missing values in the CSV file.

**Listing 7.3: one_char_column2.csv**

```
gender
Male
Male
Null
Female
Male
```

**Listing 7.4: one_char_column2.py**

```
import pandas as pd

df1 = pd.read_csv('one_char_column1.csv')

print("=> initial dataframe contents:")
print(df1)
print()
```

```
df = df1.fillna("FEMALE")
print("dataframe after fillna():")
print(df)
print()
```

Listing 7.4 starts with two `import` statements and then initializes the Pandas data frame `df1` with the content of `one_char_column1.csv`, after which its contents are displayed. The next code block invokes the `fillna()` method to replace the missing values with the string `FEMALE`. Launch the code in Listing 7.4 and you will see the following output:

```
=> initial dataframe contents:
   gender
0    Male
1    Male
2    Null
3  Female
4    Male

df after fillna():
   gender
0    Male
1    Male
2    Null
3  Female
4    Male

gender mode: Male

=> first mapped dataframe:
   gender
0    Male
1    Male
2  Female
3     NaN
4    Male

=> second mapped dataframe:
   gender
0    Male
1    Male
2  Female
3  Female
4    Male
```

Listing 7.5 displays the content of the CSV file `one_numeric_column.csv`, and Listing 7.6 displays the content of `one_numeric_column.py` that fills in the missing values in the CSV file.

**Listing 7.5: one_numeric_column.csv**

```
age
19
np.nan
16
NaN
17
```

**Listing 7.6: one_numeric_column.py**

```
import pandas as pd
import numpy  as np

df1 = pd.read_csv('one_numeric_column.csv')
df2 = df1.copy()
print("=> initial dataframe contents:")
print(df1)
print()

maxval = 12345
df1['age'] = df1['age'].map({'np.nan' : maxval})
print("=> dataframe after map():")
print(df1)
print()

# refresh contents of df1:
df1 = df2
df1['age'] = df1['age'].fillna(maxval)
print("=> refreshed dataframe after fillna():")
print(df1)
print()

df1 = df1.fillna(777)
print("dataframe after second fillna():")
print(df1)
print()

#print(df1.describe())
# error due to np.nan value:
#df1['age'].astype(int)
```

```
cols = df1.select_dtypes(np.number).columns
df1[cols] = df1[cols].fillna(9876)
print("df1 after third fillna():")
print(df1)
print()

# => this code block works:
#df1 = df1.replace('np.nan', 9876)
df1 = df1.replace({'np.nan': 9876})
print("df1 after replace():")
print(df1)
print()
```

Listing 7.6 starts with two `import` statements and then initializes the Pandas data frame `df1` with the content of `one_numeric_column.csv`, after which its contents are displayed. The next code block invokes the `fillna()` method to replace the missing values with the string `FEMALE`. Launch the code in Listing 7.6 and you will see the following output:

```
=> initial dataframe contents:
      age
0      19
1  np.nan
2      16
3     NaN
4      17

=> dataframe after map():
        age
0       NaN
1   12345.0
2       NaN
3       NaN
4       NaN

=> refreshed dataframe after fillna():
      age
0      19
1  np.nan
2      16
3   12345
4      17

dataframe after second fillna():
      age
```

```
0      19
1  np.nan
2      16
3   12345
4      17

df1 after third fillna():
      age
0      19
1  np.nan
2      16
3   12345
4      17

df1 after replace():
     age
0     19
1   9876
2     16
3  12345
4     17
```

***Two Column CSV Files***

Listing 7.7 displays the content of the CSV file `two_columns.csv`, and Listing 7.8 displays the content of `two_columns.py` that fills in the missing values in the CSV file.

**Listing 7.7: two_columns.csv**

```
gender,age
Male,19
Male,np.nan
NaN,16
Female,NaN
Male,17
```

**Listing 7.8: two_columns.py**

```
import pandas as pd

df1 = pd.read_csv('two_columns.csv')

print("=> initial dataframe contents:")
print(df1)
print()
```

```
df1 = df1.fillna("MISSING")
print("dataframe after fillna():")
print(df1)
print()

df1 = df1.replace({'np.nan': 99})
print("dataframe after replace():")
print(df1)
print()
```

Listing 7.8 starts with two `import` statements and then initializes the Pandas data frame `df1` with the content of `two_columns.csv`, after which its contents are displayed. The next code block invokes the `fillna()` method to replace the missing values with the string `FEMALE`. Launch the code in Listing 7.8 and you will see the following output:

```
=> initial dataframe contents:
    gender      age
0   Male         19
1   Male     np.nan
2   NaN          16
3   Female      NaN
4   Male         17

dataframe after fillna():
   gender      age
0     Male      19
1     Male  np.nan
2  MISSING      16
3   Female MISSING
4     Male      17

dataframe after replace():
   gender      age
0     Male      19
1     Male      99
2  MISSING      16
3   Female MISSING
4     Male      17
```

Listing 7.9 displays the content of the CSV file `two_columns2.csv`, and Listing 7.10 displays the content of `two_columns2.py` that fills in the missing values in the CSV file.

**Listing 7.9: two_columns2.csv**

```
gender,age
Male,19
```

```
Male,NaN
NaN,16
Female,18
Male,17
```

**Listing 7.10: two_columns2.py**

```
import pandas as pd

df1 = pd.read_csv('two_columns2.csv')
df2 = df1.copy()

print("=> initial dataframe contents:")
print(df1)
print()

# calculates the mean value on the
# 'age' column and skips NaN values:
full_avg = df1.mean()
print("full_avg:")
print(full_avg)
print()

avg = df1['age'].mean()
print("average age:",avg)
print()

#WRONG: *all* values are updated:
df1['age'] = df1['age'].mean()
print("updated all with same avg:")
print(df1)
print()

# refresh contents of df1:
df1 = df2

# fillna() replaces NaN with avg:
df1['age'] = df1['age'].fillna(avg)
print("updated age NaN with avg:")
print(df1)
print()

#this does not replace NaN with avg:
#df1 = df1.replace({'NaN': avg})
```

```
mode = df1['gender'].mode()[0]
print("mode:",mode)

df1['gender'] = df1['gender'].fillna(mode)
print("updated gender NaN with mode:")
print(df1)
print()
```

Listing 7.10 starts with two `import` statements and then initializes the Pandas data frame `df1` with the content of `two_columns2.csv`, after which its contents are displayed. The next code block invokes the `fillna()` method to replace the missing values with the string `FEMALE`. Launch the code in Listing 7.10 and you will see the following output:

```
=> initial dataframe contents:
   gender   age
0    Male  19.0
1    Male   NaN
2     NaN  16.0
3  Female  18.0
4    Male  17.0

full_avg:
age    17.5
dtype: float64

average age: 17.5

updated all with same avg:
   gender   age
0    Male  17.5
1    Male  17.5
2     NaN  17.5
3  Female  17.5
4    Male  17.5

updated age NaN with avg:
   gender   age
0    Male  19.0
1    Male  17.5
2     NaN  16.0
3  Female  18.0
4    Male  17.0
```

```
mode: Male
updated gender NaN with mode:
   gender   age
0    Male  19.0
1    Male  17.5
2    Male  16.0
3  Female  18.0
4    Male  17.0
```

## MISSING DATA AND IMPUTATION

In general, data cleaning involves one or more of the following tasks, which are specific to each dataset:

- Counting missing data values
- Removing/dropping redundant columns
- Assigning values to missing data
- Removing duplicate values
- Checking for incorrect values
- Ensuring uniformity of data
- Using the Imputer class (to fill with the mean, median, and `most_frequent`)

In addition, you can assign previous/next value to missing values using the following techniques:

- Random Value Imputation
- Multiple Imputation
- Matching and Hot Deck Imputation

Perform an online search for articles that contain details regarding any of the preceding techniques.

### *Counting Missing Data Values*

Listing 7.11 displays the content of `missing_values2.py` that illustrates how to count the missing data values in a Pandas data frame.

**Listing 7.11: missing_values2.py**

```
import pandas as pd
import numpy as np

"""
Count NaN values in one column:
df['column name'].isna().sum()

Count NaN values in an entire data frame:
df.isna().sum().sum()
```

```
Count NaN values in one row:
df.loc[[index value]].isna().sum().sum()
"""

data = {'column1': [100,250,300,450,500,np.nan,650,700,np.nan],
        'column2': ['X','Y',np.nan,np.nan,'Z','A','B',np.nan,np.
    nan],
        'column3':['XX',np.nan,'YY','ZZ',np.nan,np.nan,'AA',np.
    nan,np.nan]
       }

df = pd.DataFrame(data,columns=['column1','column2','column3'])
print("dataframe:")
print(df)

print("Missing values in 'column1':")
print(df['column1'].isna().sum())

print("Total number of missing values:")
print(df.isna().sum().sum())

print("Number of missing values for row index 7 (= row #8):")
print(df.loc[[7]].isna().sum().sum())
```

Listing 7.11 starts with two `import` statements and a comment block that explains the purpose of several Pandas methods pertaining to the sums of values and the `isna()` method for finding the `NaN` values in a dataset.

The next portion of Listing 7.11 initializes a dictionary with three arrays of values that are used to initialize the Pandas data frame `df`. Next, the missing values in column 1 are displayed, followed by the number of missing values in every column of `df`. The final code block displays the number of missing values for the row whose index is 7. Launch the code in Listing 7.11 and you will see the following output:

```
dataframe:
   column1 column2 column3
0    100.0       X      XX
1    250.0       Y     NaN
2    300.0     NaN      YY
3    450.0     NaN      ZZ
4    500.0       Z     NaN
5      NaN       A     NaN
6    650.0       B      AA
7    700.0     NaN     NaN
8      NaN     NaN     NaN
```

```
Missing values in 'column1':
2
Total number of missing values:
11
Number of missing values for row index 7 (= row #8):
2
```

The following site has additional Python code samples for data cleaning: *https://lvngd.com/blog/data-cleaning-with-python-pandas/*

***Drop Redundant Columns***

Listing 7.12 displays the content of `drop_columns.py` that illustrates how to remove redundant columns from a Pandas data frame.

**Listing 7.12: drop_columns.py**

```
import pandas as pd

# specify a valid CSV file here:
df1 = pd.read_csv("my_csv_file.csv") # <= specify your own CSV
    file

# remove redundant columns:
df2 = df1.drop(['url'],axis=1)

# remove columns with over 50% missing values
df3 = df2.dropna(thresh=half_count,axis=1)
```

Listing 7.12 initializes the Pandas data frame `df1` with the content of the CSV file `my_csv_file.csv`, and then initializes the Pandas data frame `df2` with the contents of `df1`, and then drops the column `url`, or some other column that exists in your CSV file. Finally, the Pandas data frame `df3` is initialized with the content of Pandas data frame `df2`, after which columns are dropped if they have more than 50% missing values.

***Remove Duplicate Rows***

*Data deduplication* refers to the task of removing row-level duplicate data values. Listing 7.13 displays the content of `duplicates.csv`, and Listing 7.14 displays the content of `duplicates.sh` that removes the duplicate rows and creates the CSV file `no_duplicates.csv` that contains unique rows.

**Listing 7.13: duplicates.csv**

```
Male,19,190,0
Male,19,190,0
Male,15,180,0
Male,15,180,0
Female,16,150,0
```

```
Female,16,150,0
Female,17,170,0
Female,17,170,0
Male,19,160,0
Male,19,160,0
```

**Listing 7.14: remove-duplicates.sh**

```
filename1="duplicates.csv"
filename2="no_duplicates.csv"

cat $filename1 | sort |uniq > $filename2
```

Listing 7.14 is straightforward: after initializing the variables `filename1` and `filename2` with the names of the input and output files, respectively, the only remaining code snippet contains a Unix pipe ("|") with a sequence of commands. The left-most command displays the contents of the input file, which is redirected to the `sort` command that sorts the input rows. The result of the `sort` command is redirected to the `uniq` command, which removes duplicates rows, and this result set is redirected to the file specified in the variable `filename2`.

The `sort` command must be followed by the `uniq` command: this is how the `uniq` command can remove adjacent duplicate rows. Launch the code in Listing 7.14 and you will see the output that is displayed in Listing 7.15.

**Listing 7.15: no_duplicates.csv**

```
Male,19,190,0
Female,16,150,0
Female,17,170,0
Male,15,180,0
Male,19,160,0
Male,19,190,0
Male,19,190,0
```

***Display Duplicate Rows***

The preceding example shows you how to find the unique rows, and the code sample in Listing 7.16 in this section shows you how to find the duplicate rows.

**Listing 7.16: find-duplicates.sh**

```
filename1="duplicates.csv"
sorted="sorted.csv"
unique="unique.csv"
multiple="multiple.csv"

# sorted rows:
cat $filename1 | sort > $sorted
```

```
# unique rows:
cat $sorted | uniq > $unique

# duplicates rows:
diff -u $sorted $unique |sed -e '1,3d' -e 's/^ //' -e 's/-//' >
    $multiple
```

Listing 7.16 starts by initializing several scalar variables as filenames that will contain CSV-based data. The remaining portion of Listing 7.16 consists of two statements with the `cat` command and another statement with the `diff` command, which populates the following three CSV files with data:

```
sorted.csv
unique.csv
multiple.csv
```

The first `cat` command pipes the content of `duplicates.csv` to the `sort` command, that in turn populates the CSV file `sorted.csv`, which contains the sorted set of rows from `duplicates.csv`. In addition, the duplicate rows (if any) in `sorted.csv` will appear as consecutive rows in `sorted.csv`.

The second `cat` command pipes the contents of `sorted.csv` to the `uniq` command that in turn populates `unique.csv` with the unique rows from `sorted.csv` so that the rows in the file `unique.csv` are unique.

Finally, the `diff` command highlights the differences in the content of `sorted.csv` and `unique.csv`. The output of the `diff` command is input for the `sed` command, which does three things:

- Remove the first three text lines from its input
- Remove an initial space character from its input
- Remove an initial hyphen (-) character from its input

After the `sed` command has completed, the output is redirected to the file `$multiple`, which contains duplicate rows. Launch the code in Listing 7.17 to see the content of `multiple.csv`:

```
Female,16,150,0
Female,16,150,0
Female,17,170,0
Female,17,170,0
Male,15,180,0
Male,15,180,0
Male,19,160,0
Male,19,160,0
Male,19,190,0
Male,19,190,0
```

***Uniformity of Data Values***

An example of the uniformity of data involves verifying that the data in a given feature contains the same units measure. For example, the following three items have numeric values with different units of measure, so they need to be updated in order to have the same unit of measure (either `kph` or `mph` or some other unit of measure that you specify):

```
50mph
50kph
100mph
20kph
```

Listing 7.18 displays the content of `same_units.sh` that illustrates how to ensure that the values in a set of strings have the same unit of measure.

**Listing 7.18: same_units.sh**

```
strings="120kph 100mph 50kph"
new_unit="fps"

for x in `echo $strings`
do
  number=`echo $x | tr -d [a-z][A-Z]`
  unit=`echo $x | tr -d [0-9][0-9]`
  echo "initial: $x"
  new_num="${number}${new_unit}"
  echo "new_num: $new_num"
  echo
done
```

Listing 7.18 starts by initializing the variables `strings` and `new_unit`, followed by a `for` loop that iterates through each string in the `strings` variable. During each iteration, the variables `number` and `unit` are initialized with the characters and digits, respectively, in the current string represented by the loop variable `x`.

Next, the variable `new_num` is initialized as the concatenation of the contents of `number` and `new_unit`. Launch the code in Listing 7.18 and you will see the following output:

```
initial: 120kph
new_num: 120fps

initial: 100mph
new_num: 100fps

initial: 50kph
new_num: 50fps
```

### *Too Many Missing Data Values*

Datasets with mostly `N/A` values, which is to say, 80% or more are `N/A` or `NaN` values, are always daunting, but not necessarily hopeless. As a simple first step, you can drop rows that contain `N/A` values, which might result in a loss of 99% of the data. A variation of the preceding all-or-nothing step for handling datasets with a majority of `N/A` values is as follows:

- Use a kNN imputer to fill in missing values in high value columns
- Drop low priority columns that have > 50% missing values
- Use a KNN imputer (again) to fill in the remaining missing values
- Try using 3 or 5 as the # of nearest neighbors

The preceding sequence attempts to prune insignificant data to concentrate on reconstructing the higher priority columns through data imputation. Of course, there is no guaranteed methodology for salvaging such a dataset, so you need some ingenuity as you experiment with datasets containing highly limited data values. If the dataset is highly imbalanced, consider *oversampling* before you drop columns and/or rows.

### *Categorical Data*

Categorical values are usually discrete and can easily be encoded by specifying a number for each category. If a category has `n` distinct values, then visualize the `nxn` identity matrix: each row represents one of the distinct values.

This technique is called *one-hot encoding*, and you can use the `OneHotEncoder` class in `scikit-learn` by specifying the dataset `X` and also the column index to perform one-hot encoding:

```
from scikit-learn.preprocessing import OneHotEncoder
ohc = OneHotEncoder(categorical_features = [0])
X = onehotencoder.fit_transform(X).toarray()
```

Since each one hot encoded row contains one 1 and (n-1) zero values, this technique is inefficient when `n` is large. Another technique involves the Pandas `map()` function that replaces string values with a single column that contains numeric values. For example, the following code block replaces `Male` and `Female` with 0 and 1, respectively:

```
values = {'Male' : 0, 'Female' : 1}
df['gender'] = df['gender'].map(values)
```

A variation of the preceding code is the following code block:

```
data['gender'].replace(0, 'Female',inplace=True)
data['gender'].replace(1, 'Male',inplace=True)
```

Another variation of the preceding code is this code block:

```
data['gender'].replace([0,1],['Male','Female'],inplace=True)
```

Keep in mind that the Pandas `map()` function converts invalid entries to `NaN`.

### *Data Inconsistency*

Data inconsistency occurs when distinct values are supposed to be the same value, such as "smith" and "SMITH" instead of "Smith." Another example would be "YES," "Yes," "YS," and "ys" instead of "yes." In all cases except for "ys," you can convert all the other strings to lowercase, which replaces all the strings with smith or yes, respectively.

Alternatively, you can use the Python-based Fuzzy Wuzzy module if there are too many distinct values to specify manually. This module identifies strings that are likely to be the same by comparing two strings and generating a numeric value, such that values closer 100 are more likely to represent the same string.

***Mean Value Imputation***

Listing 7.19 displays the content of `mean_imputation.py` that shows you how to replace missing values with the mean value of each feature.

**Listing 7.19: mean_imputation.py**

```
import numpy as np
import pandas as pd
import random

filename="titanic.csv"
df = pd.read_csv(filename)

# display null values:
print("=> Initial df.isnull().sum():")
print(df.isnull().sum())
print()

# replace missing ages with mean value:
df['age'] = df['age'].fillna(df['age'].mean())

"""
Or use median(), min(), or max():
df['age'] = df['age'].fillna(df['age'].median())
df['age'] = df['age'].fillna(df['age'].min())
df['age'] = df['age'].fillna(df['age'].max())
"""

# FILL MISSING DECK VALUES WITH THE mode():
mode = df['deck'].mode()[0]
#df['deck'] = df['deck'].fillna(mode)

print("=> new age and deck values:")
print([df[['deck','age']]])
```

Listing 7.19 starts with several `import` statements and then populates the Pandas data frame `df` with the contents of the CSV file `titanic.csv`. The next portion of Listing 7.19 displays the number of rows with null values, followed by setting the missing age values to the mean of the existing age values. The following comment block shows you how to initialize missing age values with the median, minimum, and maximum age values.

The next code snippet initializes the variable `mode` with the `mode` of the `deck` values, which are chapter strings (which means that we cannot use the mean of the `mode` values). Launch the code in Listing 7.19 and you will see the following output:

```
=> Initial df.isnull().sum():
survived        0
pclass          0
sex             0
age           177
sibsp           0
parch           0
fare            0
embarked        2
class           0
who             0
adult_male      0
deck          688
embark_town     2
alive           0
alone           0
dtype:      int64

=> new age and deck values:
[    deck        age
0      C  22.000000
1      C  38.000000
2      C  26.000000
3      C  35.000000
4      C  35.000000
..   ...        ...
886    C  27.000000
887    B  19.000000
888    C  29.699118
889    C  26.000000
890    C  32.000000

[891 rows x 2 columns]]
```

### *Random Value Imputation*

*Random value imputation* involves generating random values and using them to replace `null` values. Listing 7.20 displays the content of `random_imputation.py` that shows you how to replace missing values with random values that are selected from a given feature.

**Listing 7.20: random_imputation.py**

```
import numpy as np
import pandas as pd
import random

filename="titanic.csv"
df = pd.read_csv(filename)

# display null values:
print("=> Initial df.isnull().sum():")
print(df.isnull().sum())
print()

# replace missing ages with mean value:
df['age'] = df['age'].fillna(df['age'].mean())

#Randomise missing column data
def randomize_deck(df2):
  df = df2.copy()
  data = df["deck"]
  mask = data.isnull()
  samples = random.choices( data[~mask].values , k = mask.sum() )
  data[mask] = samples
  return df

# FILL MISSING DECK VALUES WITH RANDOM non-null values:
df = randomize_deck(df)

print("=> new age and deck values:")
print([df[['deck','age']]])
```

Listing 7.20 contains code that is similar to Listing 7.19, along with the user-defined Python function `randomize_deck()` that invokes the `choices()` method (that exists in the Python `random` class) to generate a set of random numbers and then replace `null` values in the `deck` feature with those random values.

The next code snippet invokes the `randomize_deck()` function to update the data frame `df` with non-null values, followed by a code snippet that displays the contents of the `deck` and `age` features in the data frame `df`. Launch the code in Listing 7.20 and you will see the following output:

```
=> Initial df.isnull().sum():
survived        0
pclass          0
```

```
sex                0
age              177
sibsp              0
parch              0
fare               0
embarked           2
class              0
who 0
adult_male         0
deck             688
embark_town        2
alive              0
alone              0
dtype:           int64

=> new age and deck values:
[    deck       age
0      D  22.000000
1      C  38.000000
2      C  26.000000
3      C  35.000000
4      B  35.000000
..   ...        ...
886    E  27.000000
887    B  19.000000
888    D  29.699118
889    C  26.000000
890    E  32.000000

[891 rows x 2 columns]]
=> see warning message when this code is launched
```

***Multiple Imputation***

Instead of using the same inputed value to replace every missing value in a dataset, another option involves replacing missing values with several imputed values.

As a simple example, multiple imputation can generate four imputed values for each missing value in a dataset. These values would typically be slightly different from each other in order to represent a certain degree of sampling variability.

Next, replace the missing values with a set of imputed values in order to generate a full dataset. The final step involves performing an analysis on each dataset, after which the inferences from each dataset can be combined. This technique is analogous to a random forest in which predictions are made by combining the results of multiple trees.

### *Matching and Hot Deck Imputation*

Hot deck imputation is performed by randomly selecting another row that has similar values on other variables and use its value in the row that contains missing values.

You can also impute values by using an existing value that appears in a similar data point, which is one technique that is used recommendation systems.

For example, suppose that data collected for a data set involves estimating a risk factor for each data point, where the risk is derived information from an associated document. However, the information might be insufficient to calculate an accurate risk value. One potential solution involves finding the nearest neighbors to the new data point and then calculating the average of the risk values in the nearest neighbors.

### *Is a Zero Value Valid or Invalid?*

In general, replacing a missing numeric value with zero is a risky choice: this value is obviously incorrect if the values of a feature are positive numbers between 1,000 and 5,000 (or some other range of positive numbers). For a feature that has numeric values, replacing a missing value with the mean of existing values can be better than the value zero (unless the average equals zero); also consider using the median value. For categorical data, consider using the mode to replace a missing value.

There are situations where you can use the mean of existing values to impute missing values but not the value zero, and vice versa. As a first example, suppose that an attribute contains the height in centimeters of a set of persons. In this case, the mean could be a reasonable imputation, whereas 0 suffers from

1. being an invalid value (nobody has height 0)
2. skewing statistical quantities, such as the mean and variance

You might be tempted to use the mean instead of 0 when the minimum allowable value is a positive number, but use caution when working with highly imbalanced datasets. As a second example, consider a small community of 50 residents where

1. 45 people have an average annual income of $50,000
2. 4 other residents have an annual income of $10,000,000
3. 1 resident has an unknown annual income

Although the preceding example might seem contrived, it's likely that the median income is preferable to the mean income, and certainly better than imputing a value of 0.

As a third example, suppose that a company generates weekly sales reports for multiple office branches, and a new office has been opened but has yet to make any sales. In this case, the use of the mean to impute missing values for this branch would produce fictitious results. Hence, it makes sense to use the value 0 for all sales-related quantities, which will accurately reflect the sales-related status of the new branch.

## SKEWED DATASETS

This section contains three files: a skewed CSV file, a shell script to generate a single output line, and another shell script that splits the preceding "one liner" into rows that contains four columns.

Listing 7.21 displays the content of the CSV file `skewed_four_columns.csv`, Listing 7.22 displays the content of `gen_one_line.sh`, and Listing 7.23 displays the content of `skewed_four_columns.sh` that generates output consisting of rows with an equal number of columns.

### Listing 7.21: skewed_four_columns.csv

```
survived,pclass,sex,age
0,3,male,22.0,
1,1,female,38.0,1,3,
female,26.0,1,1,female,35.0,
0,3,male,35.0,0,3,
male,23.0,0,1,male,54.0,0,3,male,2.0,1,3,female,27.0
1,2,
female,14.0,
1,3,female,4.0,
1,1,female,
58.0,
0,3,male,20.0,
0,3,
male,39.0,
0,3,female,14.0,
1,2,female,55.0,0,3,male,2.0,1,2,male,23.0,
0,3,female,31.0,
```

### Listing 7.22: gen_one_line.sh

```
filename="skewed_four_columns.csv"
cat $filename |sed "1d" | awk -F"," '{ printf("%s",$0) }'
```

Listing 7.22 uses the `sed` command to delete the first line of its input, which is the content of the CSV file. The output of the `sed` command is sent to the `awk` command, which prints each input line without a line feed, thereby generating a one-line string of the content of the CSV file. Launch the code in Listing 7.22 and you will see the following output:

```
0,3,male,22.0,1,1,female,38.0,1,3,female,26.0,1,1,female,35.0,0
    ,3,male,35.0,0,3,male,23.0,0,1,male,54.0,0,3,male,2.0,1,3,fe
    male,27.01,2,female,14.0,1,3,female,4.0,1,1,female,58.0,0,3,
    male,20.0,0,3,male,39.0,0,3,female,14.0,1,2,female,55.0,0,3,
    male,2.0,1,2,male,23.0,0,3,female,31.0,
```

Notice that the shell script `gen_one_line.sh` matches the initial portion of Listing 7.23 that is shown in bold, and that this shell script is not actually used in the solution for this task. The purpose of showing you the output from `gen_one_line.sh` is to ensure that you understand that its output is the input for the second `awk` command in Listing 7.22, which contains the code that splits the input text into lines of text that contain four fields.

**Listing 7.23: skewed_four_columns2.sh**

```
filename="skewed_four_columns.csv"
cat $filename |sed "1d" | awk -F"," '{ printf("%s",$0) }'
     | awk -F"," '
BEGIN { colCount = 4 }
{
  for(i=1; i<=NF; i++) {
     printf("%s,", $i)
     #if(i < colCount) { printf(",")}
     if(i % colCount == 0) { printf("\n") }
  }
}
' | sed -e 's/,$//' -e 's/,$//'
```

Listing 7.23 starts with the `cat` command that outputs the contents of the file specified in the variable `filename`, which becomes the input for the `sed` command that simply deletes the first line (i.e., the header line).

The output from the `sed` command is the input for the *first* `awk` command that generates a one-line string by invoking the `printf()` statement that suppresses a line feed after each input line. The output from the first `awk` command becomes the input for the *second* `awk` command, that starts by initializing the variable `colCount` to 4, which is the number of fields that we want to appear in each output row.

The body of the second `awk` command sequentially prints a field and a comma. When four fields have been printed, a line feed is also printed, which is how we generate four fields for every output row.

Finally, the output from the `awk` command generates a set of lines with a trailing comma in each line, along with two consecutive commas in the final output. These extra commas are removed with the following code snippet:

```
sed -e 's/,$//' -e 's/,$//'
```

Launch the code in Listing 7.23 and you will see the following output:

```
0,3,male,22.0
1,1,female,38.0
1,3,female,26.0
1,1,female,35.0
0,3,male,35.0
0,3,male,23.0
0,1,male,54.0
0,3,male,2.0
```

```
1,3,female,27.01
2,female,14.0,1
3,female,4.0,1
1,female,58.0,0
3,male,20.0,0
3,male,39.0,0
3,female,14.0,1
2,female,55.0,0
3,male,2.0,1
2,male,23.0,0
3,female,31.0
```

## CSV FILES WITH MULTI-ROW RECORDS

This section contains a CSV file with multi-row records such that each field is on a separate line (e.g., `survived:0`) instead of comma-separated field values for each record.

The solution is surprisingly simple when we use `awk`: set `RS` equal to the string pattern that separates records. In our case, we need to set `RS` equal to `\n\n`, after which `$0` will contain the contents of each multi-line record. In addition, specify `FS='\n'` so that get each line is treated as a field (i.e., `$1`, `$2`, and so forth).

Listing 7.24 displays the content of the CSV file `multi_line_rows.csv`, and Listing 7.25 displays the content of `multi_line_rows.sh`.

**Listing 7.24: multi_line_rows.csv**

```
survived:0
pclass:3
sex:male
age:22.0

survived:1
pclass:1
sex:female
age:38.0

survived:0
pclass:3
sex:male
age:35.0

survived:1
pclass:3
sex:female
age:27.0
```

**Listing 7.25: multi_line_rows.sh**

```
filename="multi_line_rows.csv"

cat $filename | awk '
BEGIN { RS="\n\n"; FS="\n" }
{
   # name/value pairs have this format:
   # survived:0 pclass:3 sex:male age:22.0
   split($1,arr,":"); printf("%s,",arr[2]);
   split($2,arr,":"); printf("%s,",arr[2]);
   split($3,arr,":"); printf("%s,",arr[2]);
   split($4,arr,":"); printf("%s\n",arr[2]);
}'
```

The main idea in Listing 7.25 is shown in bold, which specifies the value of `RS` (record separator) as two consecutive line feed characters and then specifies `FS` (field separator) as a line feed character. The main block of code splits the fields `$1`, `$2`, `$3`, and `$4` based on a colon (":") separator, and then prints the second field, which is the actual data value.

Note that `arr[1]` contains the *name* of the fields, such as `survived`, `pclass`, `sex`, or `age`, whereas `arr[2]` contains the *value* of the fields. Launch the code in Listing 7.25 and you will see the following output:

```
0,3,male,22.0
1,1,female,38.0
0,3,male,35.0
1,3,female,27.0
```

There is one more detail: Listing 7.25 does *not* display the header line with the names of the fields. Listing 7.26 shows you how to modify Listing 7.25 to generate the header line.

**Listing 7.26: multi_line_rows2.sh**

```
filename="multi_line_rows.csv"

cat $filename | awk '
BEGIN { RS="\n\n"; FS="\n"; count=0}
{
   if(count == 0) {
     count += 1
     split($1,arr,":"); header = arr[1]
     split($2,arr,":"); header = header "," arr[1]
     split($3,arr,":"); header = header "," arr[1]
     split($4,arr,":"); header = header "," arr[1]
     print header
   }
   # name/value pairs have this format:
   # survived:0 pclass:3 sex:male age:22.0
   split($1,arr,":"); printf("%s,",arr[2]);
   split($2,arr,":"); printf("%s,",arr[2]);
   split($3,arr,":"); printf("%s,",arr[2]);
   split($4,arr,":"); printf("%s\n",arr[2]);
}'
```

Listing 7.26 initializes the variable `count` with the value 0, followed by a conditional block of code that constructs the contents of the variable `header` (which will contain the names of the fields) by sequentially concatenating the field names. The contents of `header` are printed, and since the value of `count` has been incremented, this block of code is executed only once, which prevents the `header` line from being repeatedly displayed. Launch the code in Listing 7.26 and you will see the following output:

```
survived,pclass,sex,age
0,3,male,22.0
1,1,female,38.0
0,3,male,35.0
1,3,female,27.0
```

Other variations of the preceding code are also possible, such as changing the display order of the fields. Listing 7.27 displays the fields in reverse order: `age`, `sex`, `pclass`, and `survived`.

**Listing 7.27: multi_line_rows3.sh**

```
filename="multi_line_rows.csv"

cat $filename | awk '
BEGIN { RS="\n\n"; FS="\n"; count=0}
{
   # fields displayed in reverse order:
   if(count == 0) {
     count += 1
     split($4,arr,":"); header = arr[1]
     split($3,arr,":"); header = header "," arr[1]
     split($2,arr,":"); header = header "," arr[1]
     split($1,arr,":"); header = header "," arr[1]
     print header
   }

   # name/value pairs have this format:
   # survived:0 pclass:3 sex:male age:22.0
   split($1,arr,":"); survived = arr[2];
   split($2,arr,":"); pclass = arr[2];
   split($3,arr,":"); sex = arr[2];
   split($4,arr,":"); age = arr[2];

   # fields displayed in reverse order:
   printf("%s,%s,%s,%s\n",age, sex, pclass, survived)
}'
```

Listing 7.27 contains a conditional block of code that constructs the contents of the variable header by sequentially concatenating the field names *in reverse order*. The contents of header are printed, and since the value of `count` has been incremented, this block of code is executed only once, which prevents the header line from being repeatedly displayed.

The second block of code constructs an output string by initializing the variables `survived`, `pclass`, `sex`, and `age`, and then printing them *in reverse order*. Launch the code in Listing 7.27 and you will see the following output:

```
age,sex,pclass,survived
22.0,male,3,0
38.0,female,1,1
35.0,male,3,0
27.0,female,3,1
```

## COLUMN SUBSET AND ROW SUBRANGE OF THE TITANIC CSV FILE

At this point in the chapter, you have enough knowledge to create your own variations with respect to the order in which the fields are displayed. Note that the CSV file contains only a subset of the fields in the Titanic `CSV file.` As a final example for this section, Listing 7.28 displays a subset of the columns and a subrange of the rows in the Titanic dataset consisting of the passengers who survived.

**Listing 7.28: titanic-subrange.sh**

```
filename="titanic.csv"

cat $filename | awk -F"," '
BEGIN { start_row = 10; end_row=25; survived_count = 0
   print "=> The row range is from",start_row,"to",end_row,"\n"
}
{
   if(count == 0) {
     count += 1
     header = $3 "," $4 "," $7
     print header
   }
   if(count >= start_row && count <= end_row) {
      if($1 ~ /1/) {
         survived_count += 1
         print $3 "," $4 "," $7
      }
   }
   count += 1
}
END { print "\n=> Number of survivors:",survived_count}
'
```

Listing 7.28 starts by initializing some scalar variables that specify the start row number as 10 and end row number as 25 and want to print the rows between the 10th row and the 25th row in the `titanic.csv` dataset. The first portion of the main body of code performs a one-time display of the third, fourth, and seventh column headers.

The next portion of code is executed when `count` is between the start and end values that were previously initialized. Moreover, the code increments the variable `survived_count` if the first field equals one. Next, the variable count is incremented by 1 so that the initial code block is not executed again. Launch the code in Listing 7.28 and you will see the following output:

```
=> The row range is from 10 to 25

sex,age,fare
female,27.0,11.1333
female,14.0,30.0708
female,4.0,16.7
female,58.0,26.55
female,55.0,16.0
male,,13.0
female,,7.225
male,34.0,13.0
female,15.0,8.0292
male,28.0,35.5

=> Number of survivors: 10
```

Another way to solve this task involves a combination of the `head` and `tail` commands to extract the range of lines from row 10 through row 25, as shown here:

```
cat $filename | head -26 | tail -15
```

However, the preceding code snippet does not retain the header row, so you need to perform some further adjustments.

## DATA NORMALIZATION

Normalization is the process of scaling numeric columns in a dataset so that they have a common scale. In addition, the scaling is performed as follows:

1. scaling values to the range [0,1]
2. without losing information
3. without distorting any differences that exist in the ranges of values.

You can perform data normalization via the function `MinMaxScaler()` in the `Scikit-Learn` library.

### *Assigning Classes to Data*

Listing 7.29 displays the content of `product_prices.csv`, and Listing 7.30 displays the content of `assign_classes.py` that illustrates how to assign a class value to each row in a dataset.

**Listing 7.29: product_prices.csv**

```
item,price
product1,100
product2,200
product3,250
product4,300
product5,400
```

**Listing 7.30: assign_classes.py**

```
import pandas as pd

df = pd.read_csv("product_prices.csv")
print("contents of df:")
print(df)
print()

# define class ranges:
def class_value2(y):
  if y<=100:
    return '(1) 0 - 100'
  elif y<=200:
    return '(2) 100 - 200'
  elif y<=250:
    return '(3) 200 - 250'
  else:
    return '(4) 250+'

def class_value(y):
  if y<=100:
    return '1'
  elif y<=200:
    return '2'
  elif y<=250:
    return '3'
  else:
    return '4'

df['class1'] = df['price'].apply(class_value)
df['class2'] = df['price'].apply(class_value2)

print("contents of df:")
print(df)
```

Listing 7.30 initializes the Pandas data frame `df` with the contents of the CSV file `product_prices.csv` (shown in Listing 7.29) and displays its contents.

The next portion of Listing 7.30 is the Python function `class_value2` that returns a string whose contents are a range of values that are based on the parameter `y`. For example, if `y` is less than or equal to 100, the function returns the string `(1) 0 - 100`, and similar strings for larger values of `y`.

The next portion of Listing 7.30 is the Python function `class_value` that returns a string 1, 2, 3, or 4, depending on the parameter `y`. The last portion of Listing 7.30 initializes the column `class1` and `class2` in `df`, respectively, by invoking the `apply()` method with the Python functions `class_value` and `class_value2`. Launch the code in Listing 7.30 and you will see the following output:

```
contents of df:

       item  price

0  product1    100

1  product2    200

2  product3    250

3  product4    300

4  product5    400


contents of df:

       item  price class1        class2

0  product1    100      1    (1) 0 - 100

1  product2    200      2  (2) 100 - 200
```

```
2  product3    250      3  (3) 200 - 250
3  product4    300      4        (4) 250+
4  product5    400      4        (4) 250+
```

### *Other Data Cleaning Tasks*

As a quick review, here are additional tasks that belong to data cleaning that might be relevant to a given dataset:

- detect outliers/anomalies
- resolve missing data
- resolve incorrect data
- resolve duplicate data
- remove hidden control characters (ex: \t, ^L, ^M)
- remove HTML tags (such as <div> and <a>)
- handle diacritical marks
- check for gaps in sequences of data
- check for unusual distributions
- examine the actual data instead of relying on documentation

### *DeepChecks and Data Validation*

DeepChecks is a Python module that enables you to specify a set of rules to validate data in a dataset:

*https://deepchecks.com/*

A DeepChecks *suite* contains one or more *check*s, where a check displays pass/fail output, depending on the outcome of the check. Moreover, *conditions* can be added, modified, or removed from a check, and similarly for each check in a suite. If you have written Java unit tests, then DeepChecks might remind you of JUnit, which was written by Kent Beck, the creator of the "Extreme Programming" methodology.

## HANDLING CATEGORICAL DATA

A feature with categorical data can suffer from various issues, such as missing data, invalid data, or inconsistently formatted data. The following section discusses examples of inconsistent categorical data followed by a section that discusses how to map categorical data to numeric values.

### *Processing Inconsistent Categorical Data*

This section contains examples of processing inconsistent data values. For features that have very low cardinality, consider dropping those features, and similarly for numeric columns with zero or very low variance.

Next, check the contents of categorical columns for inconsistent spellings or errors. For example, suppose that a feature contains the values `M` and `F` (for male and female), along with a mixture of gender-related strings, some of which are in the following list:

```
male
Male
female
Female
m
f
M
F
```

The preceding categorical values for gender can be replaced with two categorical values (unless you have a valid reason to retain some of the other values). Moreover, if you are training a model whose analysis involves a single gender, then you need to determine which rows (if any) of a dataset must be excluded. Also check categorical data columns for redundant or missing whitespaces.

Check for data values that have multiple data types, such as a numerical column with numbers as numerals and some numbers as strings or objects. Ensure there are consistent data formats (such as integers or floating numbers) and that dates have the same format (for example, do not mix a `mm/dd/yyyy` date format with another date format, such as `dd/mm/yyyy`).

### *Mapping Categorical Data to Numeric Values*

Character data is often called *categorical data*, examples of which include people's names, home or work addresses, and email addresses. Many types of categorical data involve short lists of values. For example, the days of the week and the months in a year involve seven and twelve distinct values, respectively. Notice that the days of the week have a relationship: each day has a previous day and a next day, and this is similar for the months of a year.

The colors of an automobile are independent of each other: the color red is not "better" or "worse" than the color blue. However, cars of a certain color can have a statistically higher number of accidents, which is of interest to insurance companies, but we won't address this case here.

There are several well-known techniques for mapping categorical values to a set of numeric values. A simple example where you need to perform this conversion involves the gender feature in the Titanic dataset. This feature is one of the relevant features for training a machine learning model. The gender feature has `{M,F}` as its set of values. As you will see later in this chapter, Pandas makes it very easy to convert the pair of values `{M,F}` to the pair of values `{0,1}`.

Another mapping technique involves mapping a set of categorical values to a set of consecutive integer values. For example, the set `{Red, Green, Blue}` can be mapped to the set of integers `{0,1,2}`. The set `{Male, Female}` can be mapped to the set of integers `{0,1}`. The days of the week can be mapped to `{0,1,2,3,4,5,6}`. Note that the first day of the week depends on the country: in some cases, it's Sunday and in other cases, it's Monday.

Another technique is called *one-hot encoding*, which converts each value to a *vector* (check Wikipedia if you need a refresher regarding vectors). Thus, `{Male, Female}` can be represented by the vectors `[1,0]` and `[0,1]`, and the colors `{Red, Green, Blue}` can be represented by the vectors `[1,0,0], [0,1,0]`, and `[0,0,1]`.

If you vertically "line up" the two vectors for gender, they form a 2x2 identity matrix, and doing the same for the colors {R,G,B} will form a 3x3 identity matrix, as shown here:

```
[1,0,0]
[0,1,0]
[0,0,1]
```

If you are familiar with matrices, you probably noticed that the preceding set of vectors looks like the 3x3 identity matrix. In fact, this technique generalizes in a straightforward manner. Specifically, if you have n distinct categorical values, you can map each of those values to one of the vectors in an nxn identity matrix.

As another example, the set of titles {"Intern","Junior","Mid-Range", "Senior", "Project Leader","Dev Manager"} have a hierarchical relationship in terms of their salaries (which can also overlap, but we'll gloss over that detail for now).

Another set of categorical data involves the season of the year: {"Spring", "Summer", "Autumn", "Winter"} and while these values are generally independent of each other, there are cases in which the season is significant. For example, the values for the monthly rainfall, average temperature, crime rate, and foreclosure rate can depend on the season, month, week, or even the day of the year.

If a feature has a large number of categorical values, then a one-hot encoding will produce many additional columns for each datapoint. Since the majority of the values in the new columns equal 0, this can increase the sparsity of the dataset, which can result in more overfitting and adversely affect the accuracy of the machine learning algorithms you adopt during the training process.

One solution is to use a sequence-based solution in which N categories are mapped to the integers 1, 2, . . ., N. Another solution involves examining the row frequency of each categorical value. For example, suppose that N equals 20, and there are 3 categorical values for 95% of the values for a given feature. You can try the following:

1. Assign the values 1, 2, and 3 to those three categorical values.
2. Assign numeric values that reflect the relative frequency of those categorical values.
3. Assign the category "OTHER" to the remaining categorical values.
4. Delete the rows that whose categorical values belong to the 5%.

### *Mapping Categorical Data to One Hot Encoded Values*

Listing 7.31 displays the content of one_hot_encode.py that illustrates how to determine perform one-hot encoding on a CSV file.

**Listing 7.31: one_hot_encode.py**

```
import pandas as pd
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

```
filename="titanic.csv"
df = pd.read_csv(filename)

sns.countplot(x='class',data=df)
#plt.show()

print("=> class values:")
print(df['class'])
print()

# create ohe values:
ohe = pd.get_dummies(df['class'])
print("=> ohe:")
print(ohe)
```

Listing 7.31 starts with several `import` statements followed by a codes snippet that initializes the Pandas data frame `df` with the contents of the CSV file `titanic.csv`. The next snippet generates a chart that displays the relative frequency of each label (there are three such labels) in the `class` feature.

The next portion of Listing 7.31 displays the set of values in the `class` feature for the entire CSV file. The last portion of Listing 7.31 invokes the `get_dummies()` method that generates three new columns that represent a one-hot encoding for the categorical values `First`, `Second`, and `Third`, respectively. Launch the code in Listing 7.31 and you will see the following output:

```
=> class values:
0      Third
1      First
2      Third
3      First
4      Third
       ...
886   Second
887    First
888    Third
889    First
890    Third
Name: class, Length: 891, dtype: object

=> ohe:
    First  Second  Third
0       0       0      1
1       1       0      0
2       0       0      1
```

```
3        1       0      0
4        0       0      1
..     ...     ...    ...
886      0       1      0
887      1       0      0
888      0       0      1
889      1       0      0
890      0       0      1

[891 rows x 3 columns]
```

## WORKING WITH CURRENCY

The format for currency depends on the country, which includes different interpretations for a "," and "." in currency (and decimal values in general). For example, `1,124.78` equals "one thousand one hundred twenty-four point seven eight" in the US, whereas `1.124,78` has the same meaning in Europe (i.e., the "." symbol and the "," symbol are interchanged).

If you need to combine data from datasets that contain different currency formats, then you probably need to convert all the disparate currency formats to a single common currency format. There is another detail to consider: currency exchange rates can fluctuate on a daily basis, which can affect the calculation of taxes, late fees, and so forth. Although you might be fortunate enough where you won't have to deal with these issues, it's still worth being aware of them.

The next code sample shows you how to calculate the value between a pair of currencies, and also how to calculate the value of Bitcoin in USD. As an initial step, launch the following command from the command line:

```
pip3 install forex-python
```

Listing 7.32 displays the content of `convert_currency.py` that illustrates how to determine some currency values.

**Listing 7.32: convert_currency.py**

```
import pandas as pd

from forex_python.converter import CurrencyRates

curr = CurrencyRates()
usd = curr.get_rates('USD')
print("USD currency:",usd)
print()

#convert USD to EURO:
usd2eur = curr.get_rate('USD', 'EUR')
print("usd2eur:", usd2eur)
```

```
#rounded = print(round(usd2eur, 3))
#print("rounded:", rounded)

from forex_python.bitcoin import BtcConverter
# Bitcoin in USD:
btc = BtcConverter()
bcoin = btc.get_latest_price('USD')
print("bcoin:  ", bcoin)
```

Listing 7.32 starts with an `import` statement and then initializes the variables `curr` as an instance of the `CurrencyRates` class, and then initializes the variable `usd` with a set of currency USD currency rates.

The next code snippet initializes the currency exchange rate from `USD` to the `EUR` currency. The final code block initializes the variable `btc` as an instance of the `BtcConverter` class and then displays the latest Bitcoin price in `USD` currency. Launch the code in Listing 7.32 and you will see the following output:

```
USD currency: {'EUR': 0.8880994671403198, 'JPY':
    114.6714031971581, 'BGN': 1.7369449378330375, 'CZK':
    21.611900532859682, 'DKK': 6.608081705150977, 'GBP':
    0.7415452930728242, 'HUF': 316.50088809946715, 'PLN':
    4.067850799289521, 'RON': 4.392984014209592, 'SEK':
    9.275133214920071, 'CHF': 0.9213143872113678, 'ISK':
    127.708703374778, 'NOK': 8.84884547069272, 'HRK':
    6.685168738898757, 'RUB': 76.66412078152754, 'TRY':
    13.378685612788633, 'AUD': 1.4092362344582594, 'BRL':
    5.290586145648313, 'CAD': 1.2698934280639433, 'CNY':
    6.36101243339254, 'HKD': 7.795648312611013, 'IDR':
    14321.74955595027, 'INR': 74.77531083481351, 'KRW':
    1202.1403197158081, 'MXN': 20.541385435168742, 'MYR':
    4.185523978685613, 'NZD': 1.5126110124333927, 'PHP':
    51.079928952042636, 'SGD': 1.349822380106572, 'THB':
    33.195381882770874, 'ZAR': 15.242717584369451}

usd2eur: 0.8880994671403198
bcoin:   38515.2783
```

Listing 7.33 displays the content of `convert_currency2.py` that illustrates how to display side-by-side countries and their currency conversion rates.

**Listing 7.33: convert_currency2.py**

```
import pandas as pd

df = pd.DataFrame(data=['1000USD','2000EUR','3000EUR'],columns=
    ['price'])
print("Dataset:")
print(df)
print()

cc = pd.DataFrame({'from':['EUR','USD'],'to':['USD','EUR'],'rat
    e':[1.33,0.75]})
print("Conversion Table:")
print(cc)
print()
```

Listing 7.33 starts with an `import` statement to initialize the Pandas data frame `df` with three column titles, followed by the variable `cc` that contains currency conversion rates. The next code snippet displays the content of `cc` in a tabular format. Launch the code in Listing 7.33 and you will see the following output:

```
Dataset:
     price
0  1000USD
1  2000EUR
2  3000EUR

Conversion Table:
  from   to  rate
0  EUR  USD  1.33
1  USD  EUR  0.75
```

Listing 7.34 displays the content of `mixed_currency.csv`, and Listing 7.35 displays the content of `convert_currency3.sh` that illustrates how to replace a "," with a "." and vice versa when it's necessary so that strings have a valid `USD` currency format. The code does not check strings for invalid date formats.

**Listing 7.34: mixed_currency.csv**

```
product1|1129.95
product2|2110,99
product3|2.110,99
product4|2.110.678,99
product5|1,130.95
```

**Listing 7.35: convert_currency3.sh**

```
filename="mixed_currency.csv"

echo "=> INITIAL CONTENTS OF CSV FILE:"
cat $filename
echo

echo "=> UPDATED CONTENTS OF CSV FILE:"
awk -F"|" '
BEGIN { modified=0 }
{
   comma  = index($2,",")
   period = index($2,".")
   OLD2=$2

   if(comma > 0 && period == 0) {
     gsub(/,/,".",$2)
     modified += 1
     #print "comma(s) but no period:", $2
   }
   else if( comma > period ) {
     # replace right-most "," with "Z"
     gsub(/,/,"Z",$2)
     # replace "." with ","
     gsub(/\./,",",$2)
     # replace "Z" with "."
     gsub(/Z/,".",$2)
     modified += 1
     #print "comma(s) before period:", $2
   }
   NEW2=$2

   printf("OLD: %18s NEW: %15s\n",OLD2, NEW2)
}
END { print "=> Modified lines:",modified }
' < mixed_currency.csv
```

Listing 7.35 starts by initializing the variable `filename` as `mixed_currency.csv` and then displays its contents. The next portion of Listing 7.35 is an `awk` script that initializes the variables `comma` and `period` with the index of a comma (",") and period (".") for every input line from the file `mixed_currency.csv`. Unlike other programming languages, there is no explicit loop keyword in the code: instead, it's an implicit aspect of the `awk` programming language (which might take some time to become accustomed to this style).

The next block of conditional code checks for the presence of a comma and the absence of a period: if so, then the `gsub()` function replaces the comma (“,”) with a period (“.”) in the second field, which is the numeric portion of each line in Listing 7.35, and the variable modified is incremented. For example, the input line `product3|2110,99` is processed by the conditional block and replaces the contents of $2, which is the second field, with the value `2110.99`.

The next portion of code checks for the presence of a comma and a period where the location of the comma is on the *right side* of the period: if so, then three substitutions are performed. First, the right-most comma is replaced with the letter `Z`, after which the period is replaced with a comma, and then the `Z` is replaced with a period. Launch the code in Listing 7.35 with the following code snippet:

```
./convert-currency3.sh
```

You will see the following output:

```
=> INITIAL CONTENTS OF CSV FILE:
product1|1129.95
product2|2110,99
product3|2.110,99
product4|2.110.678,99
product5|1,130.95

=> UPDATED CONTENTS OF CSV FILE:
OLD:           1129.95     NEW:         1129.95
OLD:           2110,99     NEW:         2110.99
OLD:          2.110,99     NEW:        2,110.99
OLD:      2.110.678,99     NEW:    2,110,678.99
OLD:          1,130.95     NEW:        1,130.95
=> Modified lines: 3
```

## WORKING WITH DATES

The format for calendar dates varies among different countries, and this belongs to something called the *localization* of data (not to be confused with `i18n`, which is data internationalization). Some examples of date formats are shown here (and the first four are probably the most common):

```
MM/DD/YY
MM/DD/YYYY
DD/MM/YY
DD/MM/YYYY
YY/MM/DD
M/D/YY
D/M/YY
YY/M/D
MMDDYY
```

```
DDMMYY
YYMMDD
```

If you need to combine data from datasets that contain different date formats, then converting the disparate date formats to a single common date format will ensure consistency.

***Find Missing Dates***

Listing 7.36 displays the content of `pandas_missing_dates.py` that illustrates how to display missing dates from a range of dates.

**Listing 7.36: pandas_missing_dates.py**

```
import pandas as pd

# A dataframe from a dictionary of lists
data = {'Date': ['2022-01-18', '2022-01-20','2022-01-21','2022-
    01-24'],
       'Name': ['Joe', 'John', 'Jane', 'Jim']}
df = pd.DataFrame(data)

# Setting the Date values as index:
df = df.set_index('Date')

# to_datetime() converts string format to a DateTime object:
df.index = pd.to_datetime(df.index)

start_d="2022-01-18"
end_d="2022-01-25"

# display dates that are not in the sequence:
print("MISSING DATES BETWEEN",start_d,"and",end_d,":")
dates = pd.date_range(start=start_d, end=end_d).difference(df.
    index)

for date in dates:
  print("date:",date)
print()
```

Listing 7.36 starts with an `import` statement and then initializes the variable `data` as a dictionary of lists, which is used to initialize the Pandas data frame `df`. The next code snippet sets the field `Date` as the index, followed by converting the strings in `df` to `DateTime` objects, as shown here:

```
df.index = pd.to_datetime(df.index)
```

Then the variables `start_d` and `end_d` are initialized as string-based dates that represent a date range, after which the variable `dates` is initialized with a set of missing dates.

The final loop displays the missing dates. Launch the code in Listing 7.36 and you will see the following output:

```
MISSING DATES BETWEEN 2022-01-18 and 2022-01-25 :
date: 2022-01-19 00:00:00
date: 2022-01-22 00:00:00
date: 2022-01-23 00:00:00
date: 2022-01-25 00:00:00
```

### *Find Unique Dates*

Listing 7.37 displays the content of `multiple_dates.csv`, and Listing 7.38 displays the content of `pandas_misc1.py` that determines the unique years in Listing 7.37.

**Listing 7.37: multiple_dates.csv**

```
"dates","values"
2021-01-31,40
2021-02-28,45
2021-03-31,56
2021-04-30,NaN
2022-05-31,NaN
2022-06-30,140
2022-07-31,95
2022-08-31,40
2023-09-30,55
2023-10-31,NaN
2023-11-15,65
```

**Listing 7.38: pandas_misc1.py**

```
import pandas as pd

df = pd.read_csv('multiple_dates.csv', parse_dates=['dates'])
print("df:")
print(df)
print()

the_years = df['dates']
year_list = set(the_years)

arr1 = np.array([])
for long_year in year_list:
  year = str(long_year)
  short_year = year[0:4]
  arr1 = np.append(arr1,short_year)
```

```
unique_years = set(arr1)
print("unique_years:")
print(unique_years)
print()

unique_arr = np.array(pd.DataFrame.from_dict(unique_years))
print("unique_arr:")
print(unique_arr)
print()
```

Listing 7.38 starts with an `import` statement and then initializes the Pandas data frame `df` with the contents of the CSV file `multiple_dates.csv`. After initializing the variable `the_years` with the `dates` feature in `df`, and also initializing `year_list` with the distinct values in the variable `the_years`, a loop extracts the year value for each date and appends this value to the NumPy array `arr1`.

Next, `unique_years` is populated with the distinct values in the variable `arr1` and then the result is printed. The final code snippet initializes the variable `unique_arr` with the unique years from the dictionary `unique_years` and the result is displayed. Launch the code in Listing 7.38 and you will see the following output:

```
MISSING DATES BETWEEN 2022-01-18 and 2022-01-25
df:
       dates   values
0  2021-01-31    40.0
1  2021-02-28    45.0
2  2021-03-31    56.0
3  2021-04-30    NaN
4  2022-05-31    NaN
5  2022-06-30  140.0
6  2022-07-31   95.0
7  2022-08-31   40.0
8  2023-09-30   55.0
9  2023-10-31    NaN
10 2023-11-15   65.0

unique_years:
{'2022', '2023', '2021'}

unique_arr:
[['2022']
 ['2023']
 ['2021']]
```

### *Switch Date Formats*

Listing 7.39 displays the content of `standard_formats.csv`, and Listing 7.40 displays the content of `switching_date_formats.sh` that illustrates how to show missing dates from a range of dates.

**Listing 7.39: standard_dates.csv**

```
2021-01-31
2021-02-28
2022-04-30
2022-05-31
2023-10-31
2023-11-15
```

**Listing 7.40: switching_date_formats.sh**

```
file="standard_dates.csv"

echo "first output:"
cat $file | awk -F"-" ' { print $3 $2 $1 }'
echo

echo "second output:"
cat $file | awk -F"-" ' { print $3,$2,$1 }'
echo

echo "third output:"
cat $file | awk -F"-" ' { print $3 "-" $2 "-" $1 }'
```

Listing 7.40 starts by initializing the variable `file` with `standard_dates.csv`, whose contents are shown in Listing 7.39. The remaining portion of Listing 7.40 consists of three short code blocks, each of which contains an `awk` command that specifies a hyphen (-) as the delimiter for the dates. The values of `$1`, `$2`, and `$3` are the three hyphen-delimited values in each input line. For example, `$1`, `$2`, and `$3` equal 2021, 01, and 31, respectively, for the first input line `2021-01-31`.

The first `awk` command displays the date fields in the order `$3`, `$2`, and `$1`. The second `awk` command also displays the date fields in the order `$3`, `$2`, and `$1`, along with a comma delimiter. The third `awk` command also displays the date fields in the order `$3`, `$2`, and `$1`, along with a hyphen (-) delimiter. Note that if you want to display the date fields in the order month, day, and year, use the following statement: `print $2,$3,$1`. Launch the code in Listing 7.40 and you will see the following output:

```
first output:
31012021
28022021
30042022
31052022
```

```
31102023
15112023

second output:
31 01 2021
28 02 2021
30 04 2022
31 05 2022
31 10 2023
15 11 2023

third output:
31-01-2021
28-02-2021
30-04-2022
31-05-2022
31-10-2023
15-11-2023
```

## WORKING WITH IMBALANCED DATASETS

*Imbalanced datasets* contain at least once class that has significantly more values than another class in the dataset. For example, if class A has 99% of the data and class B has 1%, which classification algorithm would you use? Unfortunately, classification algorithms don't work as well with this type of imbalanced dataset.

However, there are various techniques that you can use to reduce the imbalance in a dataset. Regardless of the technique that you decide to use, keep in mind the following detail: *resampling techniques are only applied to the training data* (not the validation data or the test data).

In addition, if you perform k-fold cross validation on a training set, then oversampling is performed in each fold during the training step. To avoid data leakage, make sure that you do *not* perform oversampling prior to k-fold cross validation.

### *Data Sampling Techniques*

Data sampling techniques reduce the imbalance in imbalanced datasets, and some well-known techniques are as follows:

- random resampling: rebalances the class distribution
- random undersampling: deletes examples from the majority class
- random oversampling: duplicates data in the minority class
- SMOTE (Synthetic Minority Oversampling Technique)

*Random resampling* rebalances the class distribution by resampling the data space  to reduce the discrepancy between the number of rows in the majority class and the minority class. The *random undersampling* technique removes samples that belong to the majority class from the dataset, and involves the following:

- randomly remove samples from majority class
- can be performed with or without replacement
- alleviates imbalance in the dataset
- may increase the variance of the classifier
- may discard useful or important samples

However, random undersampling does not work well with extremely unbalanced datasets, such as a 99% and 1% split into two classes. Moreover, undersampling can result in losing information that is useful for a model.

*Random oversampling* generates new samples from a minority class: this technique duplicates examples from the minority class.

Another option to consider is the Python package `imbalanced-learn` in the `scikit-learn-contrib` project. This project provides various re-sampling techniques for datasets that exhibit class imbalance. More details are here:

*https://github.com/scikit-learn-contrib/imbalanced-learn*

Another well-known technique is called SMOTE, which involves data augmentation (i.e., synthesizing new data samples). SMOTE was initially developed by means of the kNN algorithm (other options are available), and it can be an effective technique for handling imbalanced classes. SMOTE is discussed in more detail in a later section.

### *Removing Noisy Data*

There are several techniques that attempt to remove so-called noisy data so that there is less ambiguity in the classification of the remaining data. Some of these techniques are listed here:

- Near Miss
- Condensed Nearest Neighbor (CNN)
- Tomek links
- ENN (Edited Nearest Neighbor)
- OSS (One-sided Selection)
- Neighborhood Cleaning Rule (NCR)

One potential drawback to CNN is due to its random choice of sample points. Tomek links is an undersampling technique that modified CNN in two ways. One improvement involves finding pairs of data points (x,y) that are cross-class nearest neighbors, which is to say, x and y belong to different classes and x and y also have the smallest Euclidean distance. After finding all such pairs, the values that belong to the majority class are removed. However, the efficacy of Tomek links does vary, and it's often used in conjunction with other undersampling techniques (including CNN).

ENN (Edited Nearest Neighbor) removes data points in the majority class that are misclassified as belonging to the minority class. ENN uses a "pairing" technique to find a matching nearest neighbor in the majority class that is paired with a "noisy" or ambiguous point that is located along the class boundary, and then removes the point in the majority.

The following code snippet for ENN links shows you how to import the appropriate class from `imblearn`:

```
from imblearn.under_sampling import EditedNearestNeighbours
// details omitted
undersampled = EditedNearestNeighbours(n_neighbors=5)
```

The default value for `n_neighbors` is 3, whereas the value 5 is specified in the preceding code snippet.

***Cost-sensitive Learning***

This section discusses the confusion matrix, which is not covered in this book. If you are unfamiliar with this matrix, perform an Internet search for articles that describe the details of the confusion matrix.

*Cost-sensitive learning* refers to assigning different costs for misclassification of data points, which is often relevant to imbalanced datasets. For example, the consequences of a type II error (false negative) are considerably worse than a type I error (false positive) for datasets pertaining to fraud detection, medical diagnosis (such as cancer), and so forth.

Specifically, the *cost* in cost-sensitive learning refers to the penalty that is assigned to an incorrect prediction. As such, the goal is to minimize the overall cost of a given model during the training step.

Another important concept is the *confusion matrix* for binary classification tasks, which has four possibilities:

- true positive
- false positive
- true negative
- false negative

An example of a confusion matrix is shown here, followed by the interpretation of the values in the confusion matrix:

```
[[60  4]
 [16 20]]
```

The four values in the preceding 2x2 matrix represent the following quantities:

```
True positive:  60
False positive:  4
True negative:  20
False negative: 16
```

Thus, the main diagonal consists of correct predictions, whereas the "off" diagonal consists of incorrect predictions. Cost-sensitive learning involves defining a cost matrix from the values in the confusion matrix. An example of a cost matrix is shown here:

```
[[0  5]
 [50 0]]
```

*The value 50 is much larger than the value 5 because 50 is the cost associated with a false negative, whereas 5 is associated with a false positive, and zero cost is associated with correct predictions.*

In this example, the associated cost function that we wish to minimize is shown here:

```
Cost = 50*FN + 5*FP
```

More information regarding the confusion matrix and associated metrics (accuracy, precision, recall, sensitivity, and F1 score) is accessible by performing an online search.

### *Detecting Imbalanced Data*

This step involves counting the number of rows that are associated with each class. First, read the dataset into a Pandas data frame (let's call it `df`) and then invoke the following code snippet:

```
df['your-target-column'].value_counts()
```

An example of the output from the preceding code snippet is here:

```
[OUT]
1    21000
0     6000
```

As you can see in the output, class 1 is more than 3 times larger than class 0, so this column in the dataset is imbalanced.

The Python-based open source library `Scikit-Learn` contains a vast set of algorithms for machine learning, some of which support a `class_weight` parameter, as listed here:

- LogisticRegression
- Perceptron
- RandomForest
- SVM

Set the value of the `class_weight` parameter equal to `balanced` before training the chosen model. Incidentally, it might also be worthwhile to train the chosen model with the default value for the `class_weight` parameter.

### *Rebalancing Datasets*

Let's return to the example of a dataset for which class `A` has 99% of the data and class `B` has 1%. Which classification algorithm would you use? The following list contains several well-known techniques for handling imbalanced datasets (not in any particular order):

- Random resampling (rebalances the class distribution)
- Random oversampling (duplicates data in the minority class)
- Random undersampling (deletes examples from the majority class)
- Algorithm Selection
- Cross-Validation for Imbalanced Data
- Generating Synthetic Data (ex: SMOTE)
- Performance Metric Selection

Several of the techniques in the preceding list are discussed in the following subsections.

***Specify stratify in Data Splits***

This step also is straightforward because `Scikit-Learn` supports a `stratify` parameter that ensures the data is split so that the train data and test data contain the same proportion of class values. For example, if a dataset contains 60% and 40%, respectively, of class `A` and class `B` in a column, then the train data and test data will contain the same proportions for class `A` and `B`.

The following code block demonstrates how to split a dataset (where `X` and y have already been initialized) so that the data is stratified:

```
from scikit-learn.model_selection import train_test_split

X_train,X_test,y_train,y_test = train_test_split(X,
                                                 y,
                                                 test_size=0.25,
                                                 random_state=42,
                                                 stratify=y)
```

## WHAT IS SMOTE?

SMOTE (Synthetic Minority Oversampling Technique) is a technique for synthesizing new samples for a dataset. This technique is based on linear interpolation:

Step 1: Select samples that are close in the feature space.
Step 2: Draw a line between the samples in the feature space.
Step 3: Draw a new sample at a point along that line.
A more detailed explanation of the SMOTE algorithm is here:

- Select a random sample "a" from the minority class.
- Find k nearest neighbors for that example.
- Select a random neighbor "b" from the nearest neighbors.
- Create a line L that connects "a" and "b."
- Randomly select one or more points "c" on line L.

If need be, you can repeat this process for the other (k-1) nearest neighbors to distribute the synthetic values more evenly among the nearest neighbors.

One disadvantage of SMOTE is that the creation of new data points does not take into account the majority class, which could result in some ambiguity if there is overlap between the minority and majority classes. However, variations SMOTE are more selective about generating synthetic samples, which is the topic of the next section.

## DATA WRANGLING

Data wrangling means different things to different people, which might cause some confusion unless people clarify what they mean when they talk about data wrangling. *Data wrangling* involves multiple steps that can include transforming one or more files. Here are some of the interpretations of data wrangling:

- It's part of a sequence of steps.
- Data wrangling transforms datasets.
- It's essentially the same as data cleaning.

This book adopts the approach of the first and second bullet items but not the third. Navigate to the following link that lists data wrangling as part of a six-step process:

*https://en.wikipedia.org/wiki/Data_wrangling*

In addition to the steps outlined in the preceding Web page, data wrangling can also involve the following tasks:

- transforming datasets from one format into another format (convert)
- creating new datasets from subsets of columns in existing datasets (extract)

As you can see, the preceding steps differentiate between converting data to a different format versus extracting data from multiple datasets to create new datasets. The conversion process can be considered a data cleaning task if only the first step is performed; i.e., there is no extraction step.

One additional comment: the interpretation of data wrangling in this chapter is convenient, but it's not a universally accepted standard definition. Hence, you are free to adopt your own interpretation of data wrangling (versus data cleaning) if you find one that better suits your needs. You may want to read more about data wrangling:

*https://pub.towardsai.net/tutorial-on-data-wrangling-college-towns-dataset-a0e8f8dfb6ae*

### *Data Transformation: What Does This Mean?*

In general, data cleaning involves a single data source (not necessarily in a CSV format), with some type of modification to the content of the data source (e.g., filling missing values, changing date formats, and so forth), without creating a second data source.

For example, suppose that the data source is a MySQL table called `employees` that contains employee-related information. After data cleaning tasks on the `employees` table are completed, the result will still be named the `employees` table. In database terminology, data cleaning is somewhat analogous to executing a SQL statement that involves a `SELECT` on a single table.

However, if two CSV files contain different date formats and you need to create a single CSV file that is based on the date columns, then there will be some type of conversion process that could be one of the following:

- Convert the first date format to the second date format.
- Convert the second date format to the first date format.
- Convert both date formats to a third date format.

In the case of financial data, you are likely to also encounter different currencies, which involves a conversion rate between a pair of currencies. Since currency conversion rates fluctuate, you need to decide the exchange rate to use for the data, which can be

- the exchange rate during the date that the CSV files were generated
- the current currency exchange rate
- some other mechanism

In addition, you might also need to convert the CSV files to XML documents, where the latter might be required to conform to an XML Schema, and perhaps also conform to XBRL, which is a requirement for business reporting purposes:

*https://en.wikipedia.org/wiki/XBRL*

As mentioned earlier, data transformation can involve two or more data sources to create yet another data source whose attributes are in the required format. Here are four scenarios of data transformation with just two data sources, A and B, where data from A and from B are combined to create data source C, where A, B, and C can have different file formats:

- all attributes in A and all attributes in B
- all attributes in A and some attributes in B
- a subset of the attributes in A and all attributes in B
- a subset of the attributes in A and some attributes in B

In database terminology, data transformation is somewhat analogous to executing a SQL statement that involves a `SELECT` statement on two or more database tables with a `JOIN` clause. Such SQL statements typically involve a subset of columns from each database table, which would correspond to selecting a subset of the features in the data sources.

There is also the scenario involving the *concatenation* of two or more data sources. If all data sources have the same attributes, then their concatenation is straightforward, but you might need to check for duplicate values. For example, if you want to load multiple CSV files into a database table that does not allow duplicates, then one solution involves concatenating the CSV files from the command line and then excluding the duplicate rows.

## A DATASET WITH 1,000,000 ROWS

The code sample in this section shows you how to use Pandas to perform various comparisons on a dataset that contains 1,000,000 rows. Listing 7.41 displays the content `numeric_comparisons.sh` that shows you how to work with a dataset that contains 1,000,000 rows.

### *Dataset Details*

Listing 7.41 displays the content of `read_1000000.py` that shows you how to display the details of a dataset that contains 1,000,000 rows.

**Listing 7.41: read_1000000.py**

```
import pandas as pd

filename="1000000_HRA_Records.csv"
df = pd.read_csv(filename)

print("df.describe():")
print(df.describe())
print()
print("df.info():")
print(df.info())
print()
```

Listing 7.41 reads the contents of a CSV file with one million records into the Pandas data frame `df`; this is followed by a code block that displays some statistics about the CSV file, followed by a second code block that describes the contents of the CSV file. Launch the code in Listing 7.41 and you will see the following output:

```
df.describe():
                Age  ...  YearsWithCurrManager
count  1000000.000000  ...      1000000.000000
mean        38.976191  ...            5.878886
std         12.403615  ...            6.016144
min         18.000000  ...            1.000000
25%         28.000000  ...            2.000000
50%         39.000000  ...            4.000000
75%         50.000000  ...            8.000000
max         60.000000  ...           40.000000

[8 rows x 26 columns]

df.info():
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 35 columns):
 #   Column                    Non-Null Count    Dtype
---  ------                    --------------    -----
 0   Age                      1000000 non-null  int64
 1   Attrition                 1000000 non-null  object
 2   BusinessTravel             1000000 non-null  object
 3   DailyRate                 1000000 non-null  int64
 4   Department                1000000 non-null  object
 5   DistanceFromHome           1000000 non-null  int64
 6   Education                 1000000 non-null  int64
 7   EducationField             1000000 non-null  object
 8   EmployeeCount              1000000 non-null  int64
 9   EmployeeNumber             1000000 non-null  int64
 10  EnvironmentSatisfaction    1000000 non-null  int64
 11  Gender                    1000000 non-null  object
 12  HourlyRate                 1000000 non-null  int64
 13  JobInvolvement             1000000 non-null  int64
 14  JobLevel                  1000000 non-null  int64
 15  JobRole                   1000000 non-null  object
 16  JobSatisfaction           1000000 non-null  int64
 17  MaritalStatus             1000000 non-null  object
 18  MonthlyIncome             1000000 non-null  int64
 19  MonthlyRate               1000000 non-null  int64
 20  NumCompaniesWorked        1000000 non-null  int64
 21  Over18                    1000000 non-null  object
 22  OverTime                  1000000 non-null  object
 23  PercentSalaryHike         1000000 non-null  int64
 24  PerformanceRating         1000000 non-null  int64
 25  RelationshipSatisfaction  1000000 non-null  int64
 26  StandardHours             1000000 non-null  int64
 27  StockOptionLevel          1000000 non-null  int64
 28  TotalWorkingYears         1000000 non-null  int64
 29  TrainingTimesLastYear     1000000 non-null  int64
 30  WorkLifeBalance           1000000 non-null  int64
 31  YearsAtCompany            1000000 non-null  int64
 32  YearsInCurrentRole        1000000 non-null  int64
 33  YearsSinceLastPromotion   1000000 non-null  int64
 34  YearsWithCurrManager      1000000 non-null  int64
dtypes: int64(26), object(9)
memory usage: 267.0+ MB
None
```

***Numeric Comparisons***

The code snippets in this section illustrate how to check for a specific number (e.g., 58) and the occurrence of one, two, or three adjacent digits in the first field of each row.

Listing 7.42 displays the content of `numeric_comparisons.sh` that shows you how to work with a dataset that contains 1,000,000 rows.

**Listing 7.42: numeric_comparisons.py**

```
import pandas as pd

filename="1000000_HRA_Records.csv"
filename="1000000_HRA_Records_short.csv"

df = pd.read_csv(filename)
age_filter = df['Age'] > 50
age = df[age_filter]
print("1) => Age Count:      ", age['Age'].sum())

Support = df['Department'] == "Support"
print("2) => Support count:  ", Support.sum())
```

```
AgeSupport = (df['Age'] > 58) & (df['Department'] == "Support")
print("3) => Age and Support:", AgeSupport.sum())
```

Listing 7.42 initializes the variable `filename` as the file that contains only 1,000 rows because the processing time for this file is much shorter than the processing time for the dataset that contains one million rows. However, feel free to use the entire CSV file for this code sample.

Next, the Pandas data frame `df` is initialized, followed by a code block that filters the rows in `df` based on the rows whose `Age` value is greater than 50. The number of such rows is counted, and that number is then displayed.

The next block of code initializes the variable `Support` with the set of rows from `df` whose `Department` value equals "`Support`," and then prints the number of rows in the variable `Support`. The final code block initializes the variable `AgeSupport` with the rows from `df` whose `Age` feature is greater than 58 and whose `Department` feature equals "`Support`," and then prints the number of rows in the variable `AgeSupport`. Launch the code in Listing 7.42 and you will see the following output:

```
1) => Age Count:      12534
2) => Support count:   153
3) => Age and Support: 7
```

### *Counting Adjacent Digits*

The code sample in Listing 7.43 shows you how to check for a specific number (e.g., 58) and the occurrence of one, two, or three adjacent digits in the first field of each row.

**Listing 7.43: adjacent_digits.py**

```
import pandas as pd

#filename="1000000_HRA_Records.csv"
filename="1000000_HRA_Records_short.csv"

df = pd.read_csv(filename)
age_filter = df['Age'] == 58

df_age = df[age_filter]
df_age_count = df_age['Age'].count()
print("1) => df_age_count = ", df_age_count)
print()

df['DailyRate'] = df['DailyRate'].astype(str)
three_digits = df['DailyRate'].str.extract(pat = '(\d{3})')
print("Three digits:",three_digits)
print()

df['DailyRate'] = df['DailyRate'].astype(str)
```

```
four_digits = df['DailyRate'].str.extract(pat = '(\d{4})')
print("Four digits: ",four_digits)
print()
```

Listing 7.43 starts by initializing the variable filename with a CSV file that is the "short" version of the CSV file containing 1,000,000 records that is located in the same directory as this Python file.

The next portion of Listing 7.43 initializes the variable `df` with the contents of the CSV file, and then initializes the variable `age_filter` to "filter" the records whose `Age` value equals 58, extracts that dataset from the data frame `df`, and then prints the result.

The next portion of Listing 7.43 converts the `DailyRate` feature to a string type, and then initializes the variable `three_digits` with the dataset whose records contain three consecutive digits in the `DailyRate` feature, after which the results are printed. The final portion of Listing 7.43 contains a block of code that searches for four consecutive digits in the `DailyRate` feature and then prints the results. Launch the code in Listing 7.43 and you will see the following output:

```
1) => df_age_count =  22

Three digits:
       0
0    200
1    720
2    140
3    131
4    711
..   ...
995  395
996  116
997  322
998  129
999  119

[1000 rows x 1 columns]

Four digits:
        0
0     NaN
1     NaN
2    1406
3    1316
4     NaN
..    ...
995   NaN
```

```
996  1162
997   NaN
998  1292
999  1190

[1000 rows x 1 columns]
```

Notice that the preceding output contains rows that do not match a sequence of four consecutive digits, and so the output is `NaN`.

**Listing 7.44: average_rate.py**

```
import pandas as pd

filename="1000000_HRA_Records.csv"
filename="1000000_HRA_Records_short.csv"

df = pd.read_csv(filename)

age_filter     = df['Age'] > 40
support_filter = df['Department'] == 'Support'

print("1) => Values for 'Support':")
df_support = df[support_filter]
total_rate = df_support['DailyRate'].sum()
row_count  = df_support['DailyRate'].count()
average_rate = total_rate/row_count
print("total_rate:", total_rate)
print("average:   ", average_rate)
print()

print("2) => Values for 'Age':")
df_age = df[age_filter]
total_rate = df_age['DailyRate'].sum()
row_count  = df_age['DailyRate'].count()
average_rate = total_rate/row_count
print("total_rate:", total_rate)
print("average:   ", average_rate)
print()

print("3) => Values for 'Age' and 'Support':")
df_age_support = df[age_filter & support_filter]
total_rate = df_age_support['DailyRate'].sum()
row_count  = df_age_support['DailyRate'].count()
```

```
average_rate = total_rate/row_count
print("total_rate:", total_rate)
print("average:   ", average_rate)
```

Listing 7.44 contains code with logic that is very similar to the code in Listing 7.43. In addition, Listing 7.44 contains a final code block that calculates the sum of all the `DailyRate` values, counts the number of such rows, and performs a division operation to calculate the average `DailyRate` value. Launch the code in Listing 7.44 and you will see the following output:

```
1) => Values for 'Support':
total_rate: 118610
average:    775.2287581699346

2) => Values for 'Age':
total_rate: 374756
average:    805.9268817204301

3) => Values for 'Age' and 'Support':
total_rate: 56175
average:    838.4328358208955
```

## SAVING CSV DATA TO XML, JSON, AND HTML FILES

Listing 7.45 shows you how to read the contents of a CSV file into a Pandas data frame and then save the data frame as a JSON, HTML, and XML file.

**Listing 7.45: adjacent_digits.py**

```
import pandas as pd

csvfile="short.csv"
xmlfile="short.xml"
jsonfile="short.json"
htmlfile="short.html"

df = pd.read_csv(csvfile)

df.to_html(htmlfile)
df.to_json(jsonfile)

#requires Pandas v1.3 (or higher):
df.to_xml(xmlfile)
```

```
# code samples for saving data frames to XML if you do not have
    v1.3:
# https://stackoverflow.com/questions/18574108/how-do-convert-a-
    pandas-dataframe-to-xml
```

Listing 7.45 initializes four variables with the names of a CSV file, XML file, JSON file, and an HTML file, respectively. Next, the variable `df` is initialized with the contents of the CSV file, after which its contents are saved as an HTML file as well as a JSON file.

The contents of the variable `df` are also saved as an XML file: note the comment indicating that Pandas v1.3 or higher is required to execute this code snippet. Launch the code in Listing 7.45 and you will see the following files in the same directory as Listing 7.45. The newly created files are shown in bold:

**`short.xml`**
**`short.json`**
**`short.html`**
`short.csv`

## SUMMARY

This chapter started with several Pandas code samples that use Pandas to read CSV files. We then analyzed missing data in CSV files that contain a single column, followed by two-column CSV files. In addition, you saw how to work with multi-column CSV files, such as the custom `bmi.csv` file and the Titanic `titanic.csv` CSV file.

Next, you learned about missing data and imputation, how to count missing data values, drop redundant columns, and remove duplicate rows via a Bash script. In addition, you learned about data uniformity, which involves verifying that the data values in a column have the same unit of measure (i.e., all values in meters, or all values in centimeters, and so forth).

Moreover, you learned about the data inconsistency that can arise when user-supplied data (such as inputs for text fields) allow for inconsistent values, such as `Y`, `ys`, and `Yes` as variations for the single value `YES`. You also saw that the value 0 can be a legitimate value, even though there are numerous situations where the value 0 is invalid.

Next, you saw a Bash script for processing datasets that contain a variable number of fields in each row and replacing them with datasets that have a fixed number of columns in each row. In addition, you saw how to process CSV files containing records that span multiple lines using a Bash script.

Furthermore, you learned about data normalization, which is the process of scaling numeric columns in a dataset so that they have a common scale. Then, you saw how to map categorical data to numeric values using one-hot encoding, and how to use the Pandas `map()` function. Next, you learned how to work with currency in datasets, as well as performing various tasks on date fields, such as finding missing dates, finding duplicate dates, and changing the format of dates.

You learned about imbalanced datasets and sampling techniques, and had a brief introduction to SMOTE for generating synthetic data. Finally, you learned how to use Pandas to work with a dataset that contains 1,000,000 rows.

CHAPTER 8

# NOSQL, SQLITE, AND PYTHON

This chapter introduces non-relational databases whose feature sets are well-suited to certain types of applications. Specifically, you will learn about NoSQL and MongoDB, which is a popular NoSQL database. Then you will see some of the features of SQLite and SQLAlchemy, as well as how to access both of them through Python scripts. This is followed by Python code that accesses MySQL.

The first section (which is roughly half of this chapter) introduces NoSQL, along with Python code samples to manage data in a MongoDB collection. To some extent, this section shows you how to perform operations in MongoDB that are counterparts to SQL commands. You will learn how to create a database in MongoDB, how to create a collection, and how to populate the collection with documents.

The second section shows you the NoSQL command for querying data from a NoSQL collection, as well as deleting document from a collection. You will also learn about Compass (a GUI tool for MongoDB) and PyMongo, which is a Python distribution for working with MongoDB.

The third section returns to MySQL, where you will see how to read MySQL data into a Pandas data frame and then save the data frame as an Excel spreadsheet. Although we won't discuss the details of Pandas and its rich functionality, the Pandas-related code is straightforward. If need be, you can also find many online tutorials that discuss various features of Pandas.

The fourth section provides a short description of SQLite, which is a database that is available on mobile devices, such as Android and iOS. As you can probably surmise from its name, SQLite supports a subset of SQL. You can invoke SQL commands in SQLite in various ways (such as SQLiteStudio) that are discussed in this section.

The final section provides an overview of SQLite, which is a command line tool for managing databases that is available on mobile devices. This section also introduces related tools, such as SQLiteStudio (an IDE for SQLite), DB Browser, and SQLiteDict.

## NON-RELATIONAL DATABASE SYSTEMS

There are several types of non-relational data stores, some of which are listed here:

- key-value store
- document store
- wide-column stores
- graph database

The following paragraphs contain a high-level description of the data stores in the preceding list of bullet items. Note that the details of NoSQL are deferred until later sections in this chapter.

A *key/value store* is analogous to a Python dictionary or a hash table (hash map) in Java. In abstract terms, a *key* can unlock a door to give you access to the contents on the other side of that door. In the case of key/value pairs of a key/value store, the *value* is the contents. The value can be anything, including a scalar, a data structure, or a concrete instance of a class. Although key/value stores provide limited functionality, they do provide high performance and are convenient as an in-memory cache.

A *document store* focuses on managing the storage of documents, which can involve XML, JSON, or binary formats. You can also view a document store as a generalization of a key/value store, where the values in these pairs are documents. In general, document stores also provide APIs to perform various operations, such as save, delete, update, and find documents.

- A *wide document store* provides column-based storage of name/value pairs, which includes documents. Keep in mind that a single column can consist of *multiple* columns, somewhat analogous to a table. Row keys provide access to individual columns, and columns with the same row key belong to the same row in the store. Examples of wide document stores include Bigtable (Google) and Cassandra (Facebook).

*Graph databases* are well-suited for more complex data models, such as those that exist in social networks. Each node in a graph database is a record and each edge between two nodes is a relationship between those two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships. Unsurprisingly, the complexity of their structure makes it correspondingly more difficult to access their contents in a straightforward manner.

### *Advantages of Non-relational Databases*

A NoSQL database is designed to provide fast access to data that may be stored in multiple locations (nodes). Important considerations include

- good scalability
- support for structured and non-structured data
- simpler updates to schemas
- shared nothing architecture

Due to the last point in the preceding bullet list, a DDB (distributed database) is a loosely coupled system in which each node operates on its own physical resources.

In some cases, a DDB will provide strong query abilities, whereas others focus on key-value data representation. A homogeneous DDB involves multiple databases with the same underlying DBMS, whereas a heterogeneous DDB involves multiple databases with *different* underlying DBMSs.

While distributed databases provide multiple advantages, they can also be more complex than a centralized DBMS.

## WHAT IS NOSQL?

Let's start with a clarification: in the early days, NoSQL usually meant "not SQL." More recently NoSQL has evolved to mean "not only SQL." Moreover, RDBMSs such as Oracle have adapted their databases to include support for non-structured data as well as semi-structured data. Nevertheless, RDBMSs are primarily about structured data, and NoSQL databases were designed for data types that are less structured (more about this later). In fact, some RDBMSs, such as Oracle, added support for NoSQL to the Oracle database.

NoSQL includes the data stores and graph databases that are discussed in the previous section. Recall that RDBMSs includes the normalization of database tables, whereas NoSQL data is denormalized, and `JOIN` operations are typically performed in the application code. NoSQL databases enable you to store and retrieve documents (often based on JSON) of variable length, and you can do so without defining a schema or even a table structure. In general, NoSQL databases do not support ACID (they lean toward eventual consistency), so they tend to have high speed transactions.

As you know, SQL stores data in tabular form with labelled rows and columns. By contrast, NoSQL databases have the notion of a "collection" that is analogous to an RDBMS table. A collection can contain multiple documents, where a document is analogous to a row in an RDBMS table.

Collections are not required to conform to a schema, which means that a collection can contain unrelated documents. Although you will probably populate each collection with documents that are logically related, you have a great deal of flexibility when making this decision.

Moreover, it's very easy to add new fields to one or more documents in a collection without updating a formal schema. Of course, if the documents in a collection have a highly similar (or identical) structure, then it's easier to insert, update, select, or delete such documents.

### *What is NewSQL?*

*NewSQL* refers to databases that provide the scalability of NoSQL databases and the transactional support of relational databases. Such databases can offer decentralized SQL support and often will provide support for dynamic JSON. Several examples of NewSQL databases include Snowflake, CockroachDB, and Spark SQL. More details regarding NewSQL and additional databases are available online:

*https://en.wikipedia.org/wiki/NewSQL*

## RDBMS VERSUS NOSQL: WHICH ONE TO USE?

Although RDBMSs and NoSQL databases can support the same types of data (and there are many types), they have different strengths: RDBMSs are suitable for structured data and NoSQL databases excel in their support for unstructured data. An important advantage of

NoSQL and key/value stores is their unlimited horizontal scalability, whereas RDBMSs have vertical expansion.

As you learned in previous chapters, RDBMSs are well-suited for data that can be stored in a tabular form. In addition, the structure of tables (i.e., their attributes along with their types) must be defined in advance. Furthermore, data is accessed through SQL queries, many of which are discussed in Chapter 2 and Chapter 4.

Some simple examples of "suitable data" include the details for customers and purchase orders (one-to-many relationship), along with purchase orders and lines items (also a one-to-many relationship).

Another example involves `students` and `classes` whose many-to-many relationship is replaced by a "join" table whose key is the union of 1) the attributes from the primary key for the `students` table and 2) the primary key for the `classes` table. As a result, both the `students` table and the `classes` table have a one-to-many relationship with the join table.

### *Good Data Types for NoSQL*

A NoSQL database is well-suited for documents, images, audio, and video, all of which have variable lengths (and different formats) and can be stored as single entities without adhering to the normalization process that you will see in Chapter 6. Although it's certainly possible for RDBMSs to manage these types of entities, write and read operations might require accessing different parts of an entity from multiple tables. Recall that normalization requires a `JOIN` keyword in SQL clauses that retrieve data from multiple tables, which in turn can adversely affect performance.

Moreover, the document model for NoSQL allows for fields to vary from document to document, all of which can belong to the same collection. In addition, more recent versions of MongoDB provide ACID compliance, and in conjunction with transaction support, this functionality can give MongoDB the look and feel of RDBMSs.

*The important point is to select the system (whenever possible) that best fits the requirements for your application.*

### *Some Guidelines for Selecting a Database*

MongoDB might be a better choice than an RDBMS under the following conditions:

- you need high data availability and fast, automatic, and instant data recovery
- you work with an unstable schema
- your services are mostly cloud-based, the native scale-out architecture that MongoDB comes with will be suitable for your business
- the architecture provides sharding, which aligns with horizontal scaling offered through cloud computing.

MySQL could be a better option under the following conditions:

- you are starting a business and the database is not going to scale
- the schema is fixed and its data structure will not change over time
- you want high-performance ability on a low budget
- you need a high transaction rate
- data security is the foremost priority

Of course, the preceding lists of bullet items only provide guidelines rather than a definitive list of criteria. Before you make a decision, make sure you perform a thorough evaluation of two types of databases based on a prioritized list of requirements.

### *NoSQL Databases*

The following list contains several NoSQL databases that are available for free:

- CockroachDB
- FaunaDB
- HarperDB
- RethinkDB

Before you decide to adopt one of the preceding databases, compare your list of requirements with each of these databases (and you might decide to adopt MySQL). If two of them are viable candidates, check for blog posts that contain a detailed comparison. If you decide to an application that uses each of those two databases, keep in mind that performance related issues generally arise when there is a high volume of data and/or many simultaneous transactions.

Now that you have an overview of some of the differences between RDBMSs and NoSQL databases, let's take a closer look at MongoDB, which is the topic of the next section.

## WHAT IS MONGODB?

MongoDB is a popular NoSQL database that supports NoSQL operations on data. An RDBMS allows you to create databases and tables and then insert data into those tables. By contrast, MongoDB supports the creation of databases and collections, after which you can insert documents into the collections (discussed in more detail shortly).

### *Features of MongoDB*

In addition to support for many standard query types, MongoDB offers the following features:

- sharding
- load balancing
- scalability
- schemas are optional
- support for indexes

### *Installing MongoDB*

There are two versions of MongoDB that you can install on your machine. The MongoDB community edition is available for download at the following website:

*https://docs.mongodb.com/manual/installation/#mongodb-community-edition-installation-tutorials*

Note that on MacOS you can use `brew` to install MongoDB. The MongoDB Enterprise edition is also downloadable:

*https://docs.mongodb.com/manual/administration/install-enterprise/*

In addition, you can use MongoDB with `Docker` (search online for tutorials and instructions).

***Launching MongoDB***

Use the command `mongo` without arguments, which then launches a command shell and connects to the URL `mongod://127.0.0.1:27017`.

The preceding URL is the default local server, and you're connected to the local host through port 27017. Type the following command to find the location of the `mongo` executable:

```
$ which mongo
/usr/local/bin/mongo
```

Now type `mongo` from a command shell to enter the `mongo` shell:

```
$ mongo
```

If everything has been set up correctly, you will see the following (or something similar):

```
MongoDB shell version v4.4.3
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&g
    ssapiServiceName=mongodb
Implicit session: session { "id" : UUID("2c254ca6-adf4-466f-
    8a87-a182d801ee0e") }
MongoDB server version: 4.4.3
// other details omitted for brevity
>
```

The `mongo` shell makes a connection to the `test` database, and you can verify the latter by typing the following in the MongoDB command shell:

```
> db
Test
```

Display the existing MongoDB databases with this command:

```
> show databases
admin    0.000GB
config   0.000GB
local    0.000GB
```

You can also replace `databases` with `dbs` in the preceding command. Note that `admin` and `local` are databases that are part of every MongoDB cluster.

## USEFUL MONGO APIS

With MongoDB, you can create one or more databases, where each database can contain one or more collections, and each collection can contain one or more JSON-based documents.

MongoDB supports CRUD operations (see below) that include `find()`, `insert()`, `update()`, and `delete()`, where these keywords can be suffixed with "One" or "Many" (e.g., `findOne()` and `findMany()`).

You can find data in a Mongo database with the following APIs:

- `db.collection.find()` lists all the documents in the collection.
- `db.collection.findOne()` lists only the first document in the collection.

Insert data with these MongoDB APIs:

- `db.collection.insert()` creates a new document in a collection.
- `db.collection.insertOne()` inserts a new document in a collection.
- `db.collection.insertMany()` inserts several new documents in a collection.

Update data with these MongoDB APIs:

- `db.collection.update()` modifies a document in a collection.
- `db.collection.updateOne()` modifies a single document in a collection.
- `db.collection.updateMany()` modifies multiple documents in a collection.
- `db.collection.replaceOne()` modifies a single document in a collection.

Delete data with these MongoDB APIs:

- `db.collection.remove()`: Delete a single document or all documents that match a specified filter.
- `db.collection.deleteOne()`: Delete, at most, a single document that matches a specified filter even though multiple documents may match the specified filter.
- `db.collection.deleteMany()`: Delete all documents that match a specified filter.

***Metacharacters in Mongo Queries***

MongoDB supports two metacharacters and a lowercase switch that you can use when you want to find substrings of a text string:

- `$` matches the end of a line
- `^` matches the beginning of a line
- `i` means "ignore case"

Consider the following list of names that we will use with the preceding bullet items:

```
Smith1
Smith2
Smith3
Smith4
Smith5
```

The following expression does not match anything in the list (it starts with a lowercase s instead of uppercase `S`):

```
/smith1/
```

The following expression matches `Smith1` in the list because of the i switch:

```
/smith1/i
```

The following expression matches `Smith1` through `Smith5` because `^S` matches any string that starts with a capital `S`:

```
/^S/
```

The following expression matches `Smith5` because `/^5/` matches any string that ends in the digit 5:

```
/5$/
```

## MONGODB COLLECTIONS AND DOCUMENTS

In simplified terms, think of a *collection* as a container-like entity that enables you to store documents. In addition, you can think of a *document* as a set of name/value pairs, where the values can be simple data types (e.g., numbers or strings) as well as arrays. Thus, MongoDB has a document-oriented data model instead of a table-oriented data model.

MongoDB's document-oriented model means that documents can be managed in their entirety instead of splitting them into components that are stored in different tables whose relationship must be defined, such as a one-to-many relationship that involves a foreign key.

Instead, you create a collection and simply insert documents in that collection. MongoDB provides APIs for managing the documents in a given collection. MongoDB performs lazy creation of databases and collections, which means that databases and collections are created after you insert the first document.

### *Document Format in MongoDB*

The documents in MongoDB are composed of field-and-value pairs and have the following structure:

```
{
   field1 → value1,
   field2 → value2,
   field3 → value3,
   ...
   fieldN → valueN
}
```

The value of a field can be any `JSON` data type, including other documents, arrays, and arrays of documents. In practice, you'll specify your documents using the `JSON` format.

## CREATE A MONGODB COLLECTION

Unlike an RDBMS, MongoDB does not have a `CREATE` command. Instead, you need to invoke the `use` command to create a database and then the `INSERT` command to insert a document, after which a database is created:

```
use temp
Insert a document in temp
```

Specifically, enter the following commands from the command line:

```
> use temp
switched to db temp
> db.temp.insertOne({"fname": "John", "lname": "Smith"})
```

```
{
   "acknowledged" : true,
   "insertedId" : ObjectId("603dad30876da25aabd36d5f")
}
```

You can also insert multiple documents, as shown here:

```
> doc1 = {"fname": "John", "lname": "Smith"}
{ "fname" : "John", "lname" : "Smith" }

> doc2 = {"fname": "Jane", "lname": "Jones"}
{ "fname" : "Jane", "lname" : "Jones" }

> doc3 = {"fname": "Dave", "lname": "Stone"}
{ "fname" : "Dave", "lname" : "Stone" }

> db.temp.insertMany([doc1,doc2,doc3])
{
    "acknowledged" : true,
    "insertedIds" : [
         ObjectId("603daee5876da25aabd36d60"),
         ObjectId("603daee5876da25aabd36d61"),
         ObjectId("603daee5876da25aabd36d62")
    ]
}
>
> db.temp.find()
{ "_id" : ObjectId("603dad30876da25aabd36d5f"), "fname" :
    "John", "lname" : "Smith" }
{ "_id" : ObjectId("603daee5876da25aabd36d60"), "fname" :
    "John", "lname" : "Smith" }
{ "_id" : ObjectId("603daee5876da25aabd36d61"), "fname" :
    "Jane", "lname" : "Jones" }
{ "_id" : ObjectId("603daee5876da25aabd36d62"), "fname" :
    "Dave", "lname" : "Stone" }
>
> db.temp.find({fname : "Jane"})
{ "_id" : ObjectId("603daee5876da25aabd36d61"), "fname" :
    "Jane", "lname" : "Jones" }
>
```

Now let's create a MongoDB *collection* called `cellphones` whose documents contain the attributes `year`, `os`, `model`, `color`, and `price`. Unlike an RDBMS that requires you to define the attributes of a database table, you can create a collection simply by inserting a document in that collection. Here is an example of inserting a row in the `cellphones` collection, which will create the `cellphones` collection if it does not already exist:

```
> use cellphones
switched to cellphones
> db
cellphones
> db.cellphones.insert({"year":"2017","os":"android","model":"pi
    xel2","color":"black","price":320})
```

The following document contains data for a specific cell phone in the `cellphones` collection:

```
{
 "_id" : ObjectId("600c626932e0e6419cee81a7"),
 "year" : "2017",
 "os"   : "android",
 "model" : "pixel2",
 "color" : "black",
 "price" : 320
}
```

Invoke the following command if you want to delete the cellphones collection:

```
> db.cellphones.drop()
```

## WORKING WITH MONGODB COLLECTIONS

This section contains a set of examples that illustrate how to use various MongoDB APIs for managing the data in the `cellphones` collection that was created in the previous section. The following sections contain code blocks that illustrate how to use the following APIs:

```
find()
insertOne()
insertMany()
aggregate()
```

### *Find All Android Phones*

NoSQL (and MongoDB) provide the `find()` function to query a collection for documents. The following query finds the `Android` cell phones in the `cellphones` collection:

```
> db.cellphones.find( {os: "android"} ).limit(1).pretty(){
 "_id" : ObjectId("600c63cf32e0e6419cee81ab"),
 "year" : "2017",
 "os"   : "android",
 "model" : "pixel2",
 "color" : "black",
 "price" : 28000
}
```

In addition to the `find()` function, the preceding query contains two additional functions. First, the `limit()` function is invoked to limit the number of rows that are returned in a query. Second, the `pretty()` function is invoked to display the output in a more aesthetically pleasing manner.

If you decide to omit the `pretty()` function in the preceding query, the output looks like this:

```
> db.cellphones.find( {os: "android"} ).limit(1){ "_id" : Obje
    ctId("600c63cf32e0e6419cee81ab"), "year" : "2017", "os" :
    "android", "model" : "pixel2", "color"  "black", "price" :
    320 }
```

### *Find All Android Phones in 2018*

NoSQL and MongoDB allow you to list multiple comma-separated conditions to specify Boolean `AND` logic on the specified conditions, an example of which is shown here:

```
> db.cellphones.find( {os: "android", year: "2018"} ).pretty(){
 "_id" : ObjectId("600c63cf32e0e6419cee81af"),
 "year" : "2018",
 "os" :   "android",
 "model" : "pixel3",
 "color" : "white",
 "price" : 700
}
```

### *Insert a New Item (Document)*

MongoDB provides the `insertOne()` function to insert a single document in a collection. An example of inserting (creating) a new document is as follows:

```
> db.cellphones.insertOne(
... {year: "2017", os: "bmw", color: "silver",
...  km: 28000, price: 39000}
... ){
 "acknowledged" : true,
 "insertedId" : ObjectId("600c6bc79445b834692e3b91")
}
```

### *Update an Existing Item (Document)*

MongoDB provides the `update()` function to update an existing document in a collection. For example, the following query specifies the condition that indicates the documents to be updated, and then passes the updated values as well as the `set` keyword:

```
> db.cellphones.update(
... { os: "bmw" },
... { $set: { os: "ios" }},
... { multi: true }
... )WriteResult({ "nMatched" : 5, "nUpserted" : 0, "nModified" :
    5 })
```

We need to use the `multi` parameter to update all the documents that meet the given condition. Otherwise, only one document will be updated.

### *Calculate the Average Price for Each Brand*

MongoDB provides the `aggregate()` function to group documents into similar groups. The subsequent query does the following:

1. groups the documents based on brands by selecting `$os` as `id`
2. specifies both the aggregation function which is `$avg`
3. specifies the field to be aggregated
4. inserts a single document in a collection

Here is the query that performs the preceding list of steps:

```
> db.cellphones.aggregate([
... { $group: { _id: "$make", avg_price: { $avg: "$price" }}}
... ]){ "_id" : "hyundai", "avg_price" : 36333.333333333336 }
{ "_id" : "ios", "avg_price" : 47400 }
{ "_id" : "android", "avg_price" : 35333.333333333336 }
```

If you are familiar with Pandas, the syntax is similar to the `groupby` function.

### *Calculate the Average Price for Each Brand in 2019*

This task is straightforward: start with the query in the preceding section and simply add another condition that specifies a value of 2019 for the `year`, as shown in bold in the following code:

```
> db.cellphones.aggregate([
... { $match: { year: "2019" }},
... { $group: { _id: "$os", avg_price: { $avg: "$price" }}}
... ]){ "_id" : "ios", "avg_price" : 53000 }
{ "_id" : "android", "avg_price" : 42000 }
{ "_id" : "ios",     "avg_price" : 41000 }
```

### *Import Data with mongoimport*

The `mongoimport` utility is a command line utility that enables you to import JSON, CSV, or TSV files into a MongoDB database. For example, you can import the CSV file `data.csv` into the `mytools` database with the following command:

```
mongoimport —db mytools —file /tmp/data.csv
```

## WHAT IS FUGUE?

`Fugue` a Python-based library that enables you to invoke SQL-like queries against Pandas data frames via `FugueSQL`. Install `Fugue` with the following command (specify a different version if you need to do so):

```
pip3.7 install fugue
```

Listing 8.1 displays the content of `fugue1.py` that illustrates how to populate a Pandas data frame and then invoke various SQL commands to retrieve a subset of the data from the Pandas data frame.

**Listing 8.1: fugue1.py**

```
import pandas as pd
from fugue_sql import fsql

df1 = pd.DataFrame({'fnames': ['john', 'dave', 'sara', 'eddy'],
                   'lnames': ['smith','stone','stein','bower'],
                   'ages':   [30,33,34,35],
                   'gender': ['m','m','f','m']})

print("=> data frame:")
print(df1)
print()

# Example #1: select users who are older than 33:
query_1 = """
SELECT fnames, lnames, ages, gender FROM df1
WHERE ages > 33
PRINT
"""

# display the extracted data:
fsql(query_1).run()
```

Listing 8.1 starts with `import` statements and then initializes the Pandas data frame `df1` with a set of data values. The next portion of Listing 8.1 constructs a query that retrieves the data values of all users who are older than 33. Launch the code in Listing 8.1 and you will see the following output:

```
collection:
=> data frame:
  fnames lnames  ages gender
0   john  smith    30      m
1   dave  stone    33      m
2   sara  stein    34      f
3   eddy  bower    35      m

ANTLR runtime and generated code versions disagree: 4.8!=4.9
ANTLR runtime and generated code versions disagree: 4.8!=4.9
ANTLR runtime and generated code versions disagree: 4.8!=4.9
ANTLR runtime and generated code versions disagree: 4.8!=4.9
PandasDataFrame
```

```
fnames:str|lnames:str|ages:long|gender:str
----------+----------+---------+----------
sara      |stein     |34       |f
eddy      |bower     |35       |m
Total count: 2
```

## WHAT IS COMPASS?

MongoDB Compass is a free GUI tool for MongoDB that enables you to manage data in a MongoDB database. Compass provides a GUI for accessing the contents of a MongoDB database. In addition, you can use this GUI to visually explore data and execute ad hoc queries. Compass is available for multiple platforms, such as Mac, Linux, and Window. Navigate to the following URL that contains instructions for downloading Compass:

*https://docs.mongodb.com/compass/master/install/*

After completing the installation, launch Compass and in the "Connect to Host" page, enter the following information:

```
Hostname: localhost
Port: 27107
Favorite Name: You Decide
```

Search online for more information if you are interested in using Compass with MongoDB.

## WHAT IS PYMONGO?

PyMongo is a Python distribution for working with MongoDB via Python. Install PyMongo on your machine with the following command:

```
pip3 install pymongo==3.11.2
```

The following website contains thorough documentation for learning how to use PyMongo:

*https://pymongo.readthedocs.io/en/stable/index.html*

Listing 8.2 displays the content of `pymongo1.py` that connects to the `mytools` MongoDB database.

**Listing 8.2: pymongo1.py**

```
import pymongo

# a client instance:
myclient = MongoClient("localhost",27017)

# connect to mytools:
db = myclient['mytools']

coll = db['weather']
print("collection:")
print(coll)
```

Listing 8.2 starts with an `import` statement and then initializes the variable `myclient` as an instance of the `MongoClient` class. Next, the variable `db` is initialized as database connection to the `mytools` database. The remaining code involves the variable `coll`, which is a reference to the `weather` collection, whose contents are then displayed. Launch the code in Listing 8.2 and you will see the following output:

```
collection:
Collection(Database(MongoClient(host=['localhost:27017'],
    document_class=dict, tz_aware=False, connect=True),
    'mytools'), 'weather')
```

In addition, `PyMongoArrow` enables you to load MongoDB result sets in several ways: as NumPy arrays, as Pandas data frames, or as Apache Arrow tables. Install `PyMongoArrow` with this command:

```
pip3 install pymongoarrow
```

This concludes the portion of the chapter regarding MongoDB. The next section returns to MySQL and discusses how to access a MySQL database via SQLAlchemy and Pandas.

## MYSQL, SQLALCHEMY, AND PANDAS

There are several ways to interact with a MySQL database, one of which is via SQLAlchemy. The Python code samples in subsequent sections rely on SQLAlchemy (which is briefly described in the next section) and Pandas.

### *What is SQLAlchemy?*

SQLAlchemy is an ORM (Object Relational Mapping), which serves as a "bridge" between Python code and a database. Install SQLAlchemy with this command:

```
pip3 install sqlalchemy
```

SQLAlchemy handles the task of converting Python function invocations into the appropriate SQL statements, as well as providing support for custom SQL statements. In addition, SQLAlchemy supports multiple databases, including MySQL, Oracle, PostgreSQL, and SQLite.

### *Read MySQL Data via SQLAlchemy*

The previous section showed you how to install SQLAlchemy. Install Pandas (if you haven't done so already) with this command:

```
pip3 install pandas
```

The Pandas functionality in the code samples involve the intuitively named `read_sql()` method and the related `read_sql_query()` method, both of which read the contents of a MySQL table.

Listing 8.3 displays the content of `read_sql_data.py` that reads the contents of the `people` table.

**Listing 8.3: read_sql_table.py**

```
from sqlalchemy import create_engine
import pymysql
import pandas as pd

engine = create_engine('mysql+pymysql://root:yourpassword@127.0.
    0.1',pool_recycle=3600)
dbConn = engine.connect()
frame  = pd.read_sql("select * from mytools.people", dbConn);

pd.set_option('display.expand_frame_repr', False)
print(frame)
dbConn.close()
```

Listing 8.3 starts with several `import` statements that are required to access a MySQL database. The next portion of code initializes the variable `engine` as a reference to MySQL, followed by `dbConn`, which is a database connection. Next, the variable `frame` is initialized with the rows in the `people` table. Launch the following command in a command shell:

```
python3 read_sql_table.py
```

You will see the following output:

```
  fname  lname age gender  country
0  john  smith  30      m      usa
1  jane  smith  31      f   france
2  jack  jones  32      m   france
3  dave  stone  33      m    italy
4  sara  stein  34      f  germany
5  eddy  bower  35      m    spain
```

Listing 8.4 displays the content of `sql_query.py` that reads the contents of the `people` table.

**Listing 8.4: sql_query.py**

```
from sqlalchemy import create_engine
import pymysql
import pandas as pd
engine = create_engine('mysql+pymysql://root:yourpassword@127.0.
    0.1',pool_recycle=3600)

query_1 = '''
select * from mytools.people
'''
```

```
print("create dataframe from table:")
df_2 = pd.read_sql_query(query_1, engine)

print("dataframe:")
print(df_2)
```

Listing 8.4 starts with several `import` statements followed by initializing the variable `engine` as a reference to a MySQL instance. Next, the variable `query_1` is defined as a string variable that specifies a SQL statement that selects all the rows of the `people` table, followed by the variable `df_2` (a data frame) that returns the result of executing the SQL statement specified in the variable `query_1`. The final code snippet displays the contents of the `people` table. Launch the following command in a command shell:

```
python3 sql_query.py
```

You will see the following output:

```
  fname  lname age gender  country
0  john  smith  30      m      usa
1  jane  smith  31      f   france
2  jack  jones  32      m   france
3  dave  stone  33      m    italy
4  sara  stein  34      f  germany
5  eddy  bower  35      m    spain
```

Launch the following Python script in a command shell:

```
python3 sql_query.py
```

You will see the following output:

```
  fname  lname age gender  country
0  john  smith  30      m      usa
1  jane  smith  31      f   france
2  jack  jones  32      m   france
3  dave  stone  33      m    italy
4  sara  stein  34      f  germany
5  eddy  bower  35      m    spain
```

## EXPORT SQL DATA FROM PANDAS TO EXCEL

Listing 8.5 displays the content of `sql_query_excel.py` that reads the contents of the `people` table into a Pandas data frame and then exports the latter to an Excel file.

**Listing 8.5: sql_query_excel.py**

```
from sqlalchemy import create_engine
import pymysql
import pandas as pd
```

```
engine = create_engine('mysql+pymysql://root:yourpassword@127.0.
    0.1',pool_recycle=3600)

query_1 = '''
select * from mytools.people
'''

print("create dataframe from table:")
df_2 = pd.read_sql_query(query_1, engine)

print("Contents of Pandas dataframe:")
print(df_2)

import openpyxl
print("saving dataframe to people.xlsx")
df_2.to_excel('people.xlsx', index=False)
```

Listing 8.5 contains several `import` statements followed by the variable `engine`, which is initialized to an "endpoint" from which a MySQL database can be accessed. The next code snippet initializes the variable `query_1` as a string that contains a simple SQL `SELECT` statement.

Next, the variable `df_2` is a Pandas data frame that was initialized as the result of invoking the SQL statement defined in the variable `query_1`. Next, the contents of `df_2` are displayed. The final portion of code in Listing 8.5 saves the contents of `df_2` to an Excel document called `people.xlsx`. Launch the following command in a command shell:

```
python3 sql_query_excel.py
```

The preceding command will generate the following output:

```
Creating dataframe from table people
Contents of Pandas dataframe:
   fname  lname age gender  country
0   john  smith  30      m      usa
1   jane  smith  31      f   france
2   jack  jones  32      m   france
3   dave  stone  33      m    italy
4   sara  stein  34      f  germany
..   ...    ...  ..    ...      ...
73  jane  smith  31      f   france
74  jack  jones  32      m   france
75  dave  stone  33      m    italy
76  sara  stein  34      f  germany
77  eddy  bower  35      m    spain

[78 rows x 5 columns]
saving dataframe to people.xlsx
```

**NOTE** *You might need to launch the earlier Python script using Python 3.7 instead of Python 3.8 or Python 3.9.*

The next section contains Pandas-related functionality that does not involve any database connectivity. Since the previous portion of this chapter contains Pandas-related functionality, it's a convenient location for this material. However, if you prefer, you can skip this section with no loss of continuity, and proceed to the next section that discusses SQLite.

## MYSQL AND CONNECTOR/PYTHON

MySQL provides a connector as another mechanism for connecting to a MySQL database. This section contains some simple Python code samples that rely on the connector to connect to a database and retrieve rows from a database table.

Before delving into the code samples, keep in mind that MySQL 8 uses `mysql_native_password` instead of `caching_sha2_password`. As a result, you need to specify a value for `auth_plugin` (which is not specified in various online code samples). Here is the error message:

```
mysql.connector.errors.NotSupportedError: Authentication
plugin 'caching_sha2_password' is not supported
```

The solution is highlighted in the Python code sample in the next section.

### *Establishing a Database Connection*

Listing 8.6 displays the content of `mysql_conn1.py` that illustrates how to establish a connector/Python database connection.

**Listing 8.6: mysql_conn1.py**

```
import mysql.connector

cnx = mysql.connector.connect(user='root',
                              password='yourpassword',
                              host='localhost',
                              database='employees',
                              auth_plugin='mysql_native_password')
cnx.close()
```

Listing 8.6 contains an `import` statement to set the appropriate path for Python 3.9. If the code executes correctly on your system without these two lines of code, then you can safely delete them.

The next code snippet is an `import` statement, followed by initializing the variable `cnx` as a database connection. Note the snippet shown in bold, which is required for MySQL 8 to connect to a MySQL database, as described in the introductory portion of this section. Launch the code in Listing 8.6 and if you don't see any error messages, then the code worked correctly.

### *Creating a Database Table*

Listing 8.7 displays the content of `create_fun_table.py` that illustrates how to establish a database connection and create a database table.

**Listing 8.7: create_fun_table.py**

```
my_table = (
    "CREATE TABLE `for_fun` ("
    "  `dept_no` char(4) NOT NULL,"
    "  `dept_name` varchar(40) NOT NULL,"
    "  PRIMARY KEY (`dept_no`), UNIQUE KEY `dept_name` (`dept_
    name`)"
    ") ENGINE=InnoDB")

DB_NAME = 'for_fun_db'

import mysql.connector
cnx = mysql.connector.connect(user='root',
                              password='yourpassword',
                              host='localhost',
                              database='mytools')
cursor = cnx.cursor()

try:
  print("Creating table {}: ".format(my_table), end='')
  cursor.execute(my_table)
except mysql.connector.Error as err:
  if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
    print("already exists.")
  else:
    print(err.msg)
else:
  print("Table created:",my_table)

cursor.close()

cnx.close()
```

Listing 8.7 starts by initializing the variable `my_table` as a string that contains a SQL statement for creating a MySQL table. The next portion of Listing 8.7 initializes the variable `cnx` as a connection to the `mytools` database, and then initializes the variable `cursor` as a database cursor.

The next portion of Listing 8.7 contains a `try/catch` block to create the table `for_fun` that is specified in the string variable `my_table`. The `except` block catches the connection-related error, and displays an appropriate message if the error occurred because the specified table already exists (or for some other reason).

Now launch the code in Listing 8.7, and if everything worked correctly, you will see the following output:

```
Creating table CREATE TABLE 'for_fun' (  'dept_no' char(4)
    NOT NULL,  'dept_name' varchar(40) NOT NULL,  PRIMARY
    KEY ('dept_no'), UNIQUE KEY 'dept_name' ('dept_name'))
    ENGINE=InnoDB: Table created: CREATE TABLE 'for_fun' (
    'dept_no' char(4) NOT NULL,  'dept_name' varchar(40) NOT
    NULL,  PRIMARY KEY ('dept_no'), UNIQUE KEY 'dept_name'
    ('dept_name')) ENGINE=InnoDB
```

Open a command shell, and from the MySQL prompt, enter the following command:

```
MySQL [mytools]> desc for_fun;
```

The output of the preceding statement is shown here:

```
+-----------+------------+------+-----+---------+-------+
| Field        | Type       | Null| Key | Default    | Extra |
+-----------+------------+------+-----+---------+-------+
| dept_no      | char(4)    | NO   | PRI | NULL       |       |
| dept_name    | varchar(40)|NO   | UNI | NULL       |       |
+-----------+------------+------+-----+---------+-------+
2 rows in set (0.060 sec)
```

***Reading Data from a Database Table***

Listing 8.8 displays the content of `mysql_pandas.py` that illustrates how to establish a database connection and retrieve the rows in a database table.

**Listing 8.8: mysql_pandas.py**

```
import mysql.connector

mydb = mysql.connector.connect(user='root',
                           password='yourpassword',
                           host='localhost',
                           database='employees',
                           auth_plugin='mysql_native_password')
mycursor = mydb.cursor()
# select all rows from the employees table:
mycursor.execute('SELECT * FROM employees')
```

```
import pandas as pd

# populate a Pandas data frame with the data:
table_rows = mycursor.fetchall()
df = pd.DataFrame(table_rows)

print("data frame:")
print(df)

mydb.close()
```

Listing 8.8 starts with the same `import` statement as Listing 8.7 and for the same purpose. The next code snippet is an `import` statement, followed by initializing the variable `cnx` as a database connection. Note the snippet shown in bold, which is required for MySQL 8 to connect to a MySQL database. Launch the code in Listing 8.8, and if everything worked correctly, you will see the following output:

```
=> Contents of data frame:
      0          1        2
0   1000       2000        Developer
1   2000       3000        Project Lead
2   3000       4000        Dev Manager
3   4000       4000        Senior Dev Manager
```

## WHAT IS SQLITE?

SQLite is a lightweight, portable, and open source RDBMS that is available on Windows, Linux, and MacOS, as well as Android and iOS. The following websites have more information:

*https://www.sqlite.org*

*https://www.sqlitetutorial.net/sqlite-commands/*

In addition, SQLite is ACID-compliant and implements most SQL standards. Let's look at some features of SQLite and the installation process, both of which are discussed in two subsections.

### *SQLite Features*

SQLite provides several useful features, some of which are listed here:

- doesn't require a separate server process or system to operate
- no system administration
- no external dependencies
- can operate in a serverless environment.
- available in multiple platforms (Unix, Linux, Mac, and Windows)
- ACID transactions
- full support for all features in SQL92

### SQLite Installation

Navigate to the following website and download the distribution for your operating system:
*https://www.sqlite.org/download.html*

The second step is to unzip the downloaded file in a convenient location, which we'll assume is the directory `$HOME/sqlite3_home`.

Note that if you have a MacBook, then the directory that contains the `sqlite3` executable is automatically in the `PATH` variable. Just to be sure, type the following command to see if the `sqlite3` is accessible:

```
which sqlite3
```

If the preceding command returns a blank line, then you need to include the path to the `bin` directory where `sqlite3` is located. For example, if the preceding directory is `$HOME/sqlite3_home/bin`, then update the `PATH` environment variable as follows:

```
export PATH=/$HOME/sqlite_home/bin:$PATH
```

The following sequence of commands show you how to launch `sqlite`, open a database, and display the contents of the `employees` table (which is created in the next section). Now type all the text that is displayed in bold:

```
sqlite3
sqlite> use sqlite3_mytools
sqlite> .open /Users/oswaldcampesato/sqlite3_mytools
sqlite> .tables
employees
sqlite> select * from employees;
1200|10000|BizDev
1100|10000|Sales
1000|10000|Developer
sqlite> .quit
```

The `.open` command opens existing databases and creates a new database, as shown above. The `employees` table was already created in an IDE, and you will see how to create that table (and any other table that you want) in the next section.

Although you can perform SQL operations from the command line, just like you can with MySQL, it's probably easier to work with SQLite in an IDE. In fact, a very robust IDE is SQLiteStudio, which is discussed in the next section.

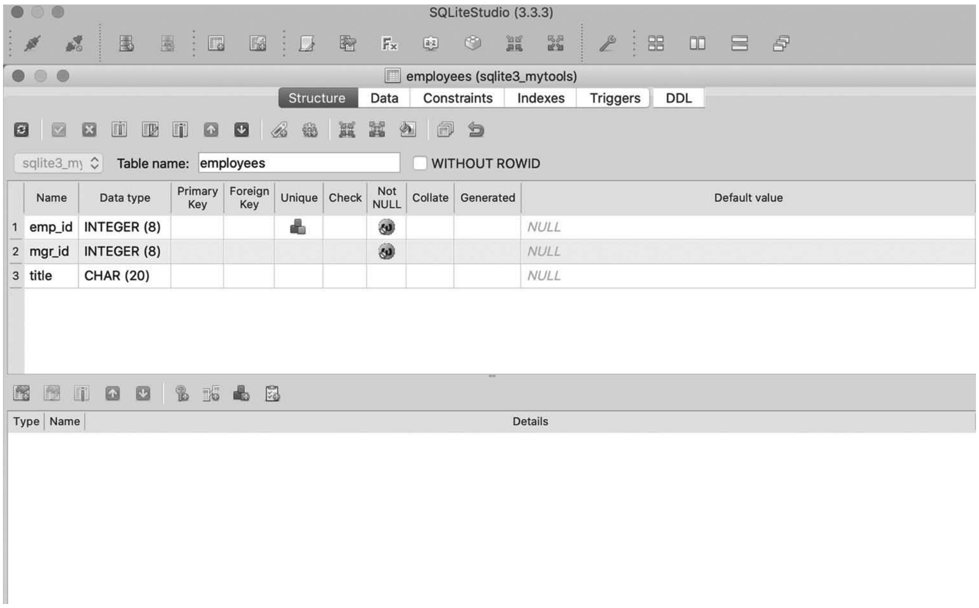
### SQLiteStudio Installation

SQLiteStudio is an open source IDE for SQLite that enables you to perform many database operations, such as creating, updating, and dropping tables and views. Navigate to the following websites, download the distribution for your operating system, and perform the specified installation steps:

*https://sqlitestudio.pl/*

*https://mac.softpedia.com/get/Developer-Tools/SQLiteStudio.shtml*

Figure 8.1 displays the structure of the `employees` table whose definition is the same as the `employees` table in the `mytools` database in MySQL.


SQLiteStudio (3.3.3)

employees (sqlite3_mytools)

Structure | Data | Constraints | Indexes | Triggers | DDL

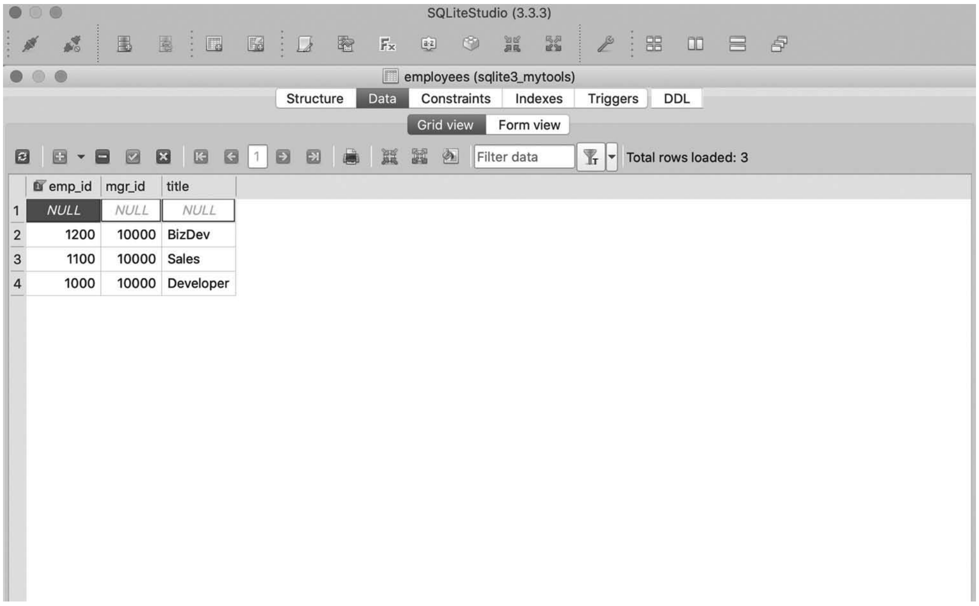
sqlite3_my Table name: employees WITHOUT ROWID

|  | Name | Data type | Primary Key | Foreign Key | Unique | Check | Not NULL | Collate | Generated | Default value |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | emp_id | INTEGER (8) |  |  | ● |  | ● |  |  | NULL |
| 2 | mgr_id | INTEGER (8) |  |  |  |  | ● |  |  | NULL |
| 3 | title | CHAR (20) |  |  |  |  |  |  |  | NULL |

| Type | Name | Details |
|---|---|---|

*Figure 8.1* The `employees` table.

Figure 8.2 displays a screenshot of three rows in the `employees` table, where you can insert a fourth row of data in the top row that is pre-populated with `NULL` values.


SQLiteStudio (3.3.3)

employees (sqlite3_mytools)

Structure | Data | Constraints | Indexes | Triggers | DDL

Grid view | Form view

Filter data Total rows loaded: 3

|  | emp_id | mgr_id | title |
|---|---|---|---|
| 1 | NULL | NULL | NULL |
| 2 | 1200 | 10000 | BizDev |
| 3 | 1100 | 10000 | Sales |
| 4 | 1000 | 10000 | Developer |

*Figure 8.2* Three rows in the `employees` table.

### *DB Browser for SQLite Installation*

DB Browser is an open source and visual-oriented tool for SQLite that enables you to perform various database-related operations, such as creating and updating files. Moreover, this tool enables you to manage data through an interface that resembles a spreadsheet.

Navigate to the following website, download the distribution for your operating system, and perform the specified installation steps:

*https://www.macupdate.com/app/mac/38584/db-browser-for-sqlite/download/secure*

The following website contains a multitude of URLs that provide details regarding the features of DB Browser:

*https://sqlitebrowser.org*

### *SQLiteDict (Optional)*

`SQLiteDict` is an open source tool that is a wrapper around `sqlite3`, and it's downloadable here:

*https://pypi.org/project/sqlitedict/*

SQLiteDict enables you to persist dictionaries to a file on the file system, as illustrated by the code in Listing 8.9.

**Listing 8.9: sqlitesavedict1.py**

```
# pip3 install sqlitedict

from sqlitedict import SqliteDict

mydict = SqliteDict('./my_db.sqlite', autocommit=True)
mydict['pasta'] = 'pasta'
mydict['pizza'] = 'pizza'

for key, value in mydict.iteritems():
  print("key:",key," value:",value)

# dictionary functions work:
print("length:",len(mydict))
mydict.close()

# a client instance:
myclient = MongoClient("localhost",27017)
```

Listing 8.9 contains an `import` statement followed by the variable `mydict`, which is initialized as a dictionary that includes the two strings `pasta` and `pizza`. The next code snippet displays the key/value pairs of `mydict`, followed by the length of the `mydict` dictionary. The next close snippet closes the dictionary and then launches a MongoDB client at the default port. Launch the code in Listing 8.9 and you will see the following output:

```
key: pasta  value: pasta
key: pizza  value: pizza
number of items: 2
```

Listing 8.9 shows you how to save key/value pairs, and Listing 8.10 illustrates how to read the content of the file that was saved in Listing 8.9.

**Listing 8.10: sqlitereaddict1.py**

```
# pip3 install sqlitedict

# read the contents of my_db.sqlite
# and note no autocommit=True
with SqliteDict('./my_db.sqlite') as mydict:
  print("old:", mydict['pasta'])
  mydict['pasta'] = u"more pasta"
  print("new:", mydict['pasta'])
  mydict['pizza'] = range(10)
  mydict.commit()
  # this is not persisted to disk:
  mydict['dish'] = u"deep dish"

# open the same file again:
with SqliteDict('./my_db.sqlite') as mydict:
  print("pasta:",mydict['pasta'])
  # this line will cause an error:
  #print("dish  value:",mydict['dish'])
```

Listing 8.10 contains a block of code that reads the existing value of `pasta` from `mydict`, updates its value, and then saves its new value. The final code block in Listing 8.10 reads the stored contents and displays the key/value pairs. Launch the code in Listing 8.10 and you will see the following output:

```
old: pasta
new: more pasta
pasta: more pasta
```

Check the online documentation for information regarding other functionality that is available through `sqlitedict`.

## WHAT IS TIMESCALEDB?

`Timescaledb` is a relational database that supports IoT datasets, and its home page is here:
*https://www.timescale.com*

There are two ways work with `timescaledb`: you can work with an online instance or you can install `timescaledb` on your local system, which is described in the next section.

*https://docs.timescale.com/timescaledb/latest/tutorials/nyc-taxi-cab/#introduction-to-iot-new-york-city-taxicabs*

### *Install Timescaledb (Macbook)*

Navigate to the following URL and download the appropriate distribution for your laptop:

*https://docs.timescale.com/install/latest/self-hosted/installation-macos/#install-self-hosted-timescaledb-on-macos-systems*

After you have downloaded and uncompressed the distribution for `timescaledb`, install it on a MacBook via the `homebrew` command line utility by performing the following steps (note the version number 2.7.0, shown in bold):

```
brew tap timescale/tap
brew install timescaledb

# for Catalina:
timescaledb-tune -conf-path /usr/local/var/postgres/postgresql.
    conf --yes
cd /usr/local/Cellar/timescaledb/2.7.0/bin
./timescaledb_move.sh
```

If need be, replace the hard-coded version number (shown in bold above) with the correct version number associated with your installation of `timescaledb`.

***Setting Up the TimescaleDB Extension***

On your laptop, connect to the `PostgreSQL` instance as the `postgres` superuser:

```
psql -U postgres -h localhost
```

If your connection is successful, you'll see a message like this, followed by the `psql` prompt:

```
psql (14.4)
```

Type "help" for help.

At the `psql` prompt, create an empty database named `tsdb`:

```
CREATE database tsdb;
```

Connect to the `tsdb` database that you created:

```
\c tsdb
```

Add the `TimescaleDB` extension:

```
CREATE EXTENSION IF NOT EXISTS timescaledb;
```

Now verify that the `TimescaleDB` extension is installed by using `\dx` command at the `psql` prompt, which will generate the following type of output:

```
sql
tsdb-# \dx
                                 List of installed extensions
 Name          | Version    | Schema     |Description
-------------+---------+-----------+------------------------
 plpgsql       | 1.0        | pg_catalog| PL/pgSQL procedural
                                          language
 timescaledb   | 2.7.0      | public     | Enables scalable
    inserts and complex queries for time-series data
(2 rows)
```

After creating the extension and the database, connect directly to your database using this command:

```
psql -U postgres -h localhost -d tsdb
```

### *The rides Table*

The first step involves downloading the CSV file `nyc_data_rides.csv` (a 400M dataset) by navigating to the following link and then clicking on the rounded button (shown in blue) that is displayed in the first section:

*https://docs.timescale.com/timescaledb/latest/tutorials/nyc-taxi-cab/*

The second step involves populating the table `rides` with data in the `CSV` file `nyc_data_rides.csv` by invoking the following command:

```
\COPY rides FROM nyc_data_rides.csv CSV;
```

Although you can upload small-to-medium sized `CSV` files quickly with the preceding command, it's also possible to upload `CSV` files in parallel, which is described in the next section.

### *The Parallel Copy Command*

A faster alternative for uploading `CSV` files to a database table is the `parallel-copy` command that creates multiple threads to import a `CSV` file in blocks of 5,000 rows, which reduces the execution time compared with a single thread.

Set `--workers <= CPUs` (or CPUs x 2) if they support hyperthreading. Replace the values for the connection string, database name, and file location that are relevant for your system in the following command:

```
timescaledb-parallel-copy --connection {CONNECTION STRING} --db-
    name {DATABASE NAME} --table rides --file {PATH TO nyc_data_
    rides.csv} --workers 4 --truncate --reporting-period 30s
```

Now that the `rides` table has been populated with data, execute the following `SQL` query that selects 5 rows from the `rides` table:

```
SELECT * FROM rides LIMIT 5;
```

### *Data Analysis*

The following `SQL` query returns the number of rides for the first 5 days:

```
SELECT date_trunc('day', pickup_datetime) as day, COUNT(*)
FROM rides
GROUP BY day
ORDER BY day;
```

The following code block displays the contents of `average_fare.sql`:

```
-- Calculate the daily average fare for rides
-- with only one passenger for first 7 days:
SELECT date_trunc('day', pickup_datetime)
AS day, avg(fare_amount)
FROM rides
WHERE passenger_count = 1
```

```
AND pickup_datetime <'2016-01-08'
GROUP BY day ORDER BY day;
```

## LARGE SCALE DATA IMPUTATION

`Scikit-learn` provides the `SimpleImputer` class that you can combine with `Pandas` to process datasets, an example of which is here:

```
from scikit-learn.impute import SimpleImputer
from scikit-learn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_
    size=.2, random_state=1066)

imp = SimpleImputer(missing_values=np.NaN, strategy='mean')

X_train = imp.fit_transform(X_train)
X_test = imp.transform(X_test)
```

Moreover, `SimpleImputer` can update missing values by setting the parameter `add_indicator` to `True`, as shown here:

```
imp = SimpleImputer(missing_values=np.NaN,strategy='mean',add_in-
dicator=True)
```

Despite the usefulness of the preceding code snippets, `Pandas` and `SimpleImputer` are not designed for large datasets (i.e., in the multi-terabyte range and beyond). Although there are `Python` packages (such as `Dask` and `Vaex`) that can handle such datasets, there can be significant transfer time for moving data from its source to its intended destination.

One cost-effective solution involves uploading the dataset to cloud-based storage, which manages terabytes of data and also supports a Linux-like execution environment.

## SUMMARY

This chapter introduced you to non-relational databases and some of their advantages. Then you learned about `NoSQL` and a `NoSQL` database called `MongoDB`. You saw how to create a database in `MongoDB` and a collection, as well as how to populate the collection with documents. You also saw how to query data from a `MongoDB` collection and how to delete a document from a collection.

Next, you learned about `Compass` (a GUI tool for `MongoDB`) and `PyMongo`, which is a `Python` distribution for working with `MongoDB`. You also learned about `DynamoDB`, which is a `NoSQL` database from Amazon. Then you saw how to read `MySQL` data into a `Pandas` data frame and then save the data frame as an `Excel` spreadsheet.

In addition, you learned about `SQLite`, which is a command line tool for managing databases that is available on mobile devices. Then you learned about related tools, such as `SQLiteStudio` (an IDE for `sqlite`), `DB Browser`, and `SQLiteDict`.

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 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