THIRD EDITION

"First Sobell taught people how to use Linux... now he teaches you the power of Linux. A must-have book for anyone who wants to take Linux to the next level."

—Jon "maddog" Hall, Executive Director, Linux International

A Practical Guide to Linux®

Commands, Editors, and Shell Programming

New Chapters on Python and MySQL—Covers Perl, too!

- Learn from hundreds of realistic, high-quality examples, and become a true Linux command-line guru!
- **NEW!** Covers busybox, Midnight Commander, screen, and sshfs/curlftpf
- Covers the Mac OS X command line and its unique tools
- 295-page reference covers 98 utilities, including Mac OS X commands!

For use with all popular versions of Linux, including Ubuntu™, Fedora™, openSUSE™, Red Hat®, Debian, Mageia, Mint, Arch, CentOS, and Mac OS X, too!

Mark G. Sobell

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for Previous Editions of *A Practical Guide to Linux® Commands, Editors, and Shell Programming*

"This book is a very useful tool for anyone who wants to 'look under the hood' so to speak, and really start putting the power of Linux to work. What I find particularly frustrating about man pages is that they never include examples. Sobell, on the other hand, outlines very clearly what the command does and then gives several common, easy-to-understand examples that make it a breeze to start shell programming on one's own. As with Sobell's other works, this is simple, straight-forward, and easy to read. It's a great book and will stay on the shelf at easy arm's reach for a long time."

—*Ray Bartlett*
Travel Writer

"Overall I found this book to be quite excellent, and it has earned a spot on the very front of my bookshelf. It covers the real 'guts' of Linux— the command line and its utilities—and does so very well. Its strongest points are the outstanding use of examples, and the Command Reference section. Highly recommended for Linux users of all skill levels. Well done to Mark Sobell and Prentice Hall for this outstanding book!"

—*Dan Clough*
Electronics Engineer and
Slackware Linux User

"Totally unlike most Linux books, this book avoids discussing everything via GUI and jumps right into making the power of the command line your friend."

—*Bjorn Tipling*
Software Engineer
ask.com

"This book is the best distro-agnostic, foundational Linux reference I've ever seen, out of dozens of Linux-related books I've read. Finding this book was a real stroke of luck. If you want to really understand how to get things done at the command line, where the power and flexibility of free UNIX-like OSes really live, this book is among the best tools you'll find toward that end."

—*Chad Perrin*
Writer, TechRepublic

"I moved to Linux from Windows XP a couple of years ago, and after some distro hopping settled on Linux Mint. At age 69 I thought I might be biting off more than I could chew, but thanks to much reading and the help of a local LUG I am now quite at home with Linux at the GUI level.

"Now I want to learn more about the CLI and a few months ago bought your book: *A Practical Guide to Linux® Commands, Editors, and Shell Programming, Second Edition*.

"For me, this book is proving to be the foundation upon which my understanding of the CLI is being built. As a comparative 'newbie' to the Linux world, I find your book a wonderful, easy-to-follow guide that I highly recommend to other Linux users."

—*John Nawell*
CQLUG (Central Queensland Linux User Group)

"I have the second edition of *A Practical Guide to Linux® Commands, Editors, and Shell Programming* and am a big fan. I used it while working as a Cisco support engineer. I plan to get the third edition as soon as it is released. We will be doing a ton of command-line work on literally 1000 boxes (IMS core nodes). I feel you have already given me a lot of tools with the second edition. I want to get your new book as soon as possible. The way you write works very well for my style of learning."

—*Robert Lingenfelter*
Support Engineer, VoIP/IMS

Praise for Other Books by Mark G. Sobell

"Since I'm in an educational environment, I found the content of Sobell's book to be right on target and very helpful for anyone managing Linux in the enterprise. His style of writing is very clear. He builds up to the chapter exercises, which I find to be relevant to real-world scenarios a user or admin would encounter. An IT/IS student would find this book a valuable complement to their education. The vast amount of information is extremely well balanced and Sobell manages to present the content without complicated asides and meandering prose. This is a 'must have' for anyone managing Linux systems in a networked environment or anyone running a Linux server. I would also highly recommend it to an experienced computer user who is moving to the Linux platform."

—*Mary Norbury*
IT Director
Barbara Davis Center
University of Colorado at Denver
from a review posted on slashdot.org

"I had the chance to use your UNIX books when I when was in college years ago at Cal Poly, San Luis Obispo, CA. I have to say that your books are among the best! They're quality books that teach the theoretical aspects and applications of the operating system."

—*Benton Chan*
IS Engineer

"The book has more than lived up to my expectations from the many reviews I read, even though it targets FC2. I have found something very rare with your book: It doesn't read like the standard technical text, it reads more like a story. It's a pleasure to read and hard to put down. Did I say that?! :-)"

—*David Hopkins*
Business Process Architect

"Thanks for your work and for the book you wrote. There are really few books that can help people to become more efficient administrators of different workstations. We hope (in Russia) that you will continue bringing us a new level of understanding of Linux/UNIX systems."

—*Anton Petukhov*

"Mark Sobell has written a book as approachable as it is authoritative."

—*Jeffrey Bianchine*
Advocate, Author, Journalist

"Excellent reference book, well suited for the sysadmin of a Linux cluster, or the owner of a PC contemplating installing a recent stable Linux. Don't be put off by the daunting heft of the book. Sobell has striven to be as inclusive as possible, in trying to anticipate your system administration needs."

—*Wes Boudville*
Inventor

"*A Practical Guide to Red Hat® Linux®* is a brilliant book. Thank you Mark Sobell."

—*C. Pozrikidis*
University of California at San Diego

"This book presents the best overview of the Linux operating system that I have found. . . . [It] should be very helpful and understandable no matter what the reader's background: traditional UNIX user, new Linux devotee, or even Windows user. Each topic is presented in a clear, complete fashion and very few assumptions are made about what the reader knows. . . . The book is extremely useful as a reference, as it contains a 70-page glossary of terms and is very well indexed. It is organized in such a way that the reader can focus on simple tasks without having to wade through more advanced topics until they are ready."

—*Cam Marshall*
Marshall Information Service LLC
Member of Front Range UNIX
Users Group [FRUUG]
Boulder, Colorado

"Conclusively, this is THE book to get if you are a new Linux user and you just got into RH/Fedora world. There's no other book that discusses so many different topics and in such depth."

—*Eugenia Loli-Queru*
Editor in Chief
OSNews.com

"I currently own one of your books, *A Practical Guide to Linux®*. I believe this book is one of the most comprehensive and, as the title says, practical guides to Linux I have ever read. I consider myself a novice and I come back to this book over and over again."

—*Albert J. Nguyen*

"Thank you for writing a book to help me get away from Windows XP and to never touch Windows Vista. The book is great; I am learning a lot of new concepts and commands. Linux is definitely getting easier to use."

—*James Moritz*

"I am so impressed by how Mark Sobell can approach a complex topic in such an understandable manner. His command examples are especially useful in providing a novice (or even an advanced) administrator with a cookbook on how to accomplish real-world tasks on Linux. He is truly an inspired technical writer!"

—*George Vish II*
Senior Education Consultant
Hewlett-Packard Company

"Overall, I think it's a great, comprehensive Ubuntu book that'll be a valuable resource for people of all technical levels."

—*John Dong*
Ubuntu Forum Council Member
Backports Team Leader

"The JumpStart sections really offer a quick way to get things up and running, allowing you to dig into the details of the book later."

—*Scott Mann*
Aztek Networks

"I would so love to be able to use this book to teach a class about not just Ubuntu or Linux but about computers in general. It is thorough and well written with good illustrations that explain important concepts for computer usage."

—*Nathan Eckenrode*
New York Local Community Team

"Ubuntu is gaining popularity at the rate alcohol did during Prohibition, and it's great to see a well-known author write a book on the latest and greatest version. Not only does it contain Ubuntu-specific information, but it also touches on general computer-related topics, which will help the average computer user to better understand what's going on in the background. Great work, Mark!"

—*Daniel R. Arfsten*
Pro/ENGINEER Drafter/Designer

"I read a lot of Linux technical information every day, but I'm rarely impressed by tech books. I usually prefer online information sources instead. Mark Sobell's books are a notable exception. They're clearly written, technically accurate, comprehensive, and actually enjoyable to read."

—*Matthew Miller*
Senior Systems Analyst/Administrator
BU Linux Project
Boston University Office
of Information Technology

"This is well-written, clear, comprehensive information for the Linux user of any type, whether trying Ubuntu on for the first time and wanting to know a little about it, or using the book as a very good reference when doing something more complicated like setting up a server. This book's value goes well beyond its purchase price and it'll make a great addition to the Linux section of your bookshelf."

—*Linc Fessenden*
Host of The LinuxLink TechShow
tllts.org

"The author has done a very good job at clarifying such a detail-oriented operating system. I have extensive Unix and Windows experience and this text does an excellent job at bridging the gaps between Linux, Windows, and Unix. I highly recommend this book to both 'newbs' and experienced users. Great job!"

—*Mark Polczynski*
Information Technology Consultant

"Your text, *A Practical Guide to Ubuntu Linux®, Third Edition,* is a well constructed, informative, superbly written text. You deserve an award for outstanding talent; unfortunately my name is not Pulitzer."

—*Harrison Donnelly*
Physician

"When I first started working with Linux just a short ten years or so ago, it was a little more difficult than now to get going. . . . Now, someone new to the community has a vast array of resources available on the web, or if they are inclined to begin with Ubuntu, they can literally find almost every single thing they will need in the single volume of Mark Sobell's *A Practical Guide to Ubuntu Linux®*.

"I'm sure this sounds a bit like hyperbole. Everything a person would need to know? Obviously not everything, but this book, weighing in at just under 1200 pages, covers so much so thoroughly that there won't be much left out. From install to admin, networking, security, shell scripting, package management, and a host of other topics, it is all there. GUI and command-line tools are covered. There is not really any wasted space or fluff, just a huge amount of information. There are screen shots when appropriate but they do not take up an inordinate amount of space. This book is information-dense."

—*JR Peck*
Editor
GeekBook.org

"I have been wanting to make the jump to Linux but did not have the guts to do so—until I saw your familiarly titled *A Practical Guide to Red Hat® Linux®* at the bookstore. I picked up a copy and am eagerly looking forward to regaining my freedom."

—*Carmine Stoffo*
Machine and Process Designer
to pharmaceutical industry

"I am currently reading *A Practical Guide to Red Hat® Linux®* and am finally understanding the true power of the command line. I am new to Linux and your book is a treasure."

—*Juan Gonzalez*

"Overall, *A Practical Guide to Ubuntu Linux®* by Mark G. Sobell provides all of the information a beginner to intermediate user of Linux would need to be productive. The inclusion of the Live DVD of the Gutsy Gibbon release of Ubuntu makes it easy for the user to test-drive Linux without affecting his installed OS. I have no doubts that you will consider this book money well spent."

—*Ray Lodato*
Slashdot contributor
www.slashdot.org

Blank

Excerpts of Chapters From

A Practical Guide to Linux® Commands, Editors, and Shell Programming

THIRD EDITION

Mark G. Sobell

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Blank

5

The Shell

EXCERPT

In This Chapter

The Command Line 126
Standard Input and Standard Output 133
Redirection 135
Pipelines 141
Running a Command in the Background 146
kill: Aborting a Background Job . . . 147
Filename Generation/Pathname Expansion 148
Builtins 153

Objectives

After reading this chapter you should be able to:

- Describe a simple command
- Understand command-line syntax and run commands that include options and arguments
- Explain how the shell interprets the command line
- Redirect output of a command to a file, overwriting the file or appending to it
- Redirect input for a command so it comes from a file
- Connect commands using a pipeline
- Run commands in the background
- Use special characters as wildcards to generate filenames
- Explain the difference between a stand-alone utility and a shell builtin

Blank

REDIRECTION

The term *redirection* encompasses the various ways you can cause the shell to alter where standard input of a command comes from and where standard output goes to. By default,

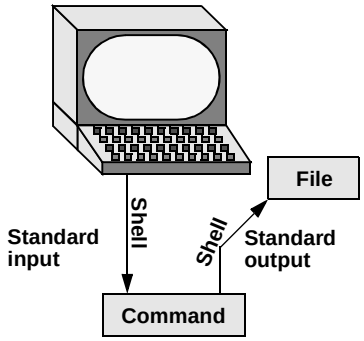

Figure 5-5 Redirecting standard output

the shell associates standard input and standard output of a command with the keyboard and the screen. You can cause the shell to redirect standard input or standard output of any command by associating the input or output with a command or file other than the device file representing the keyboard or the screen. This section demonstrates how to redirect input/output from/to text files and utilities.

REDIRECTING STANDARD OUTPUT

The *redirect output symbol* (>) instructs the shell to redirect the output of a command to the specified file instead of to the screen (Figure 5-5). The syntax of a command line that redirects output is

command [arguments] > *filename*

where *command* is any executable program (e.g., an application program or a utility), *arguments* are optional arguments, and *filename* is the name of the ordinary file the shell redirects the output to.

Figure 5-6 uses cat to demonstrate output redirection. This figure contrasts with Figure , where standard input *and* standard output are associated with the keyboard and screen. The input in Figure 5-6 comes from the keyboard. The redirect output symbol on the command line causes the shell to associate cat's standard output with the **sample.txt** file specified following this symbol.

Redirecting output can destroy a file I

caution Use caution when you redirect output to a file. If the file exists, the shell will overwrite it and destroy its contents. For more information see the tip "Redirecting output can destroy a file II" on page 139.

After giving the command and typing the text shown in Figure 5-6, the **sample.txt** file contains the text you entered. You can use cat with an argument of **sample.txt** to display this file. The next section shows another way to use cat to display the file.

```
$ cat > sample.txt
This text is being entered at the keyboard and
cat is copying it to a file.
Press CONTROL-D to indicate the
end of file.
CONTROL-D
$
```

Figure 5-6 cat with its output redirected

Figure 5-6 shows that redirecting standard output from cat is a handy way to create a file without using an editor. The drawback is that once you enter a line and press RETURN, you cannot edit the text until after you finish creating the file. While you are entering a line, the erase and kill keys work to delete text on that line. This procedure is useful for creating short, simple files.

Figure 5-7 shows how to run cat and use the redirect output symbol to *catenate* (join one after the other—the derivation of the name of the cat utility) several files into one larger file. The first three commands display the contents of three files: **stationery**, **tape**, and **pens**. The next command shows cat with three filenames as arguments. When you call it with more than one filename, cat copies the files, one at a time, to standard output. This command redirects standard output to the file **supply_orders**. The final cat command shows that **supply_orders** contains the contents of the three original files.

REDIRECTING STANDARD INPUT

Just as you can redirect standard output, so you can redirect standard input. The *redirect input symbol* (<) instructs the shell to redirect a command's input to come

```
$ cat stationery
2,000 sheets letterhead ordered:   October 7
$ cat tape
1 box masking tape ordered:        October 14
5 boxes filament tape ordered:     October 28
$ cat pens
12 doz. black pens ordered:        October 4

$ cat stationery tape pens > supply_orders

$ cat supply_orders
2,000 sheets letterhead ordered:   October 7
1 box masking tape ordered:        October 14
5 boxes filament tape ordered:     October 28
12 doz. black pens ordered:        October 4
```

Figure 5-7 Using cat to catenate files

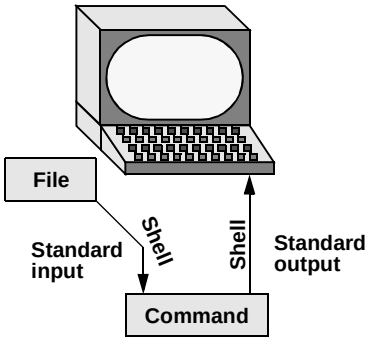

Figure 5-8 Redirecting standard input

from the specified file instead of from the keyboard (Figure 5-8). The syntax of a command line that redirects input is

command [arguments] < filename

where *command* is any executable program (such as an application program or a utility), *arguments* are optional arguments, and *filename* is the name of the ordinary file the shell redirects the input from.

Figure 5-9 shows cat with its input redirected from the **supply_orders** file created in Figure 5-7 and standard output going to the screen. This setup causes cat to display the **supply_orders** file on the screen. The system automatically supplies an EOF signal at the end of an ordinary file.

Utilities that take input from a file or standard input

Giving a cat command with input redirected from a file yields the same result as giving a cat command with the filename as an argument. The cat utility is a member of a class of utilities that function in this manner. Other members of this class of utilities include lpr, sort, grep, and Perl. These utilities first examine the command line that called them. If the command line includes a filename as an argument, the utility takes its input from the specified file. If no filename argument is present, the utility takes its input from standard input. It is the utility or program—not the shell or operating system—that functions in this manner.

```
$ cat < supply_orders
2,000 sheets letterhead ordered:   October 7
1 box masking tape ordered:        October 14
5 boxes filament tape ordered:     October 28
12 doz. black pens ordered:        October 4
```

Figure 5-9 cat with its input redirected

Redirecting output can destroy a file II

caution Depending on which shell you are using and how the environment is set up, a command such as the following can yield undesired results:

```
$ cat orange pear > orange
cat: orange: input file is output file
```

Although cat displays an error message, the shell destroys the contents of the existing **orange** file. The new **orange** file will have the same contents as **pear** because the first action the shell takes when it sees the redirection symbol (**>**) is to remove the contents of the original **orange** file. If you want to catenate two files into one, use cat to put the two files into a temporary file and then use mv to rename the temporary file:

```
$ cat orange pear > temp
$ mv temp orange
```

What happens in the next example can be even worse. The user giving the command wants to search through files **a**, **b**, and **c** for the word **apple** and redirect the output from grep (page 56) to the file **a.output**. Unfortunately the user enters the filename as **a output**, omitting the period and inserting a SPACE in its place:

```
$ grep apple a b c > a output
grep: output: No such file or directory
```

The shell obediently removes the contents of **a** and then calls grep. The error message could take a moment to appear, giving you a sense that the command is running correctly. Even after you see the error message, it might take a while to realize that you have destroyed the contents of **a**.

noclobber: PREVENTS OVERWRITING FILES

The shell provides the **noclobber** feature, which prevents you from overwriting a file using redirection. Enable this feature by setting **noclobber** using the command **set –o noclobber**. The same command with **+o** unsets **noclobber**. Under tcsh use **set noclobber** and **unset noclobber**. With **noclobber** set, if you redirect output to an existing file, the shell displays an error message and does not execute the command. Run under bash and tcsh, the following examples create a file using touch, set **noclobber**, attempt to redirect the output from echo to the newly created file, unset **noclobber**, and perform the same redirection:

bash

```
$ touch tmp
$ set -o noclobber
$ echo "hi there" > tmp
-bash: tmp: cannot overwrite existing file
$ set +o noclobber
$ echo "hi there" > tmp
```

tcsh

```
tcsh $ touch tmp
tcsh $ set noclobber
tcsh $ echo "hi there" > tmp
tmp: File exists.
tcsh $ unset noclobber
tcsh $ echo "hi there" > tmp
```

```
$ date > whoson
$ cat whoson
Wed Mar 27 14:31:18 PST 2013
$ who >> whoson
$ cat whoson
Wed Mar 27 14:31:18 PST 2013
sam        tty1         2013-03-27 05:00(:0)
max        pts/4        2013-03-27 12:23(:0.0)
max        pts/5        2013-03-27 12:33(:0.0)
zach       pts/7        2013-03-26 08:45 (172.16.192.1)
```

Figure 5-10 Redirecting and appending output

You can override **noclobber** by putting a pipe symbol (tcsh uses an exclamation point) after the redirect symbol (>|). In the following example, the user creates a file by redirecting the output of date. Next the user sets the **noclobber** variable and redirects output to the same file again. The shell displays an error message. Then the user places a pipe symbol after the redirect symbol, and the shell allows the user to overwrite the file.

```
$ date > tmp2
$ set -o noclobber
$ date > tmp2
-bash: tmp2: cannot overwrite existing file
$ date >| tmp2
```

For more information on using **noclobber** under tcsh, refer to page 397.

Do not trust noclobber

caution Appending output is simpler than the two-step procedure described in the preceding caution box but you must be careful to include both greater than signs. If you accidentally use only one greater than sign and the **noclobber** feature is not set, the shell will overwrite the **orange** file. Even if you have the **noclobber** feature turned on, it is a good idea to keep backup copies of the files you are manipulating in case you make a mistake.

Although it protects you from overwriting a file using redirection, **noclobber** does not stop you from overwriting a file using cp or mv. These utilities include the **–i** (interactive) option that helps protect you from this type of mistake by verifying your intentions when you try to overwrite a file. For more information see the tip "cp can destroy a file" on page 54.

APPENDING STANDARD OUTPUT TO A FILE

The *append output symbol* (>>) causes the shell to add new information to the end of a file, leaving existing information intact. This symbol provides a convenient way of catenating two files into one. The following commands demonstrate the action of the append output symbol. The second command accomplishes the catenation described in the preceding caution box:

```
$ cat orange
this is orange
$ cat pear >> orange
$ cat orange
this is orange
this is pear
```

The first command displays the contents of the **orange** file. The second command appends the contents of the **pear** file to the **orange** file. The final command displays the result.

Figure 5-10 shows how to create a file that contains the date and time (the output from date), followed by a list of who is logged in (the output from who). The first command in the example redirects the output from date to the file named **whoson**. Then cat displays the file. The next command appends the output from who to the **whoson** file. Finally cat displays the file containing the output of both utilities.

/dev/null: MAKING DATA DISAPPEAR

The **/dev/null** device is a *data sink*, commonly referred to as a *bit bucket*. You can redirect output you do not want to keep or see to **/dev/null**, and the output will disappear without a trace:

```
$ echo "hi there" > /dev/null
$
```

Reading from **/dev/null** yields a null string. The following command truncates the file named **messages** to zero length while preserving the ownership and permissions of the file:

```
$ ls -lh messages
-rw-rw-r--. 1 sam pubs 125K 03-16 14:30 messages
$ cat /dev/null > messages
$ ls -lh messages
-rw-rw-r--. 1 sam pubs 0 03-16 14:32 messages
```

Blank

6

THE vim EDITOR

IN THIS CHAPTER

Tutorial: Using vim to Create and Edit a File 161

Introduction to vim Features 168

Online Help 168

Command Mode: Moving the Cursor 174

Input Mode 178

Command Mode: Deleting and Changing Text 179

Searching and Substituting 183

Copying, Moving, and Deleting Text 190

The General-Purpose Buffer 191

Reading and Writing Files 193

The .vimrc Startup File 195

OBJECTIVES

After reading this chapter you should be able to:

- Use vim to create and edit a file
- View vim online help
- Explain the difference between Command and Input modes
- Explain the purpose of the Work buffer
- List the commands that open a line above the cursor, append text to the end of a line, move the cursor to the first line of the file, and move the cursor to the middle line of the screen
- Describe Last Line mode and list some commands that use this mode
- Describe how to set and move the cursor to a marker
- List the commands that move the cursor backward and forward by characters and words
- Describe how to read a file into the Work buffer
- Explain how to search backward and forward for text and how to repeat that search

EXCERPT

Blank

SEARCHING AND SUBSTITUTING

Searching for and replacing a character, a string of text, or a string that is matched by a regular expression is a key feature of any editor. The vim editor provides simple commands for searching for a character on the current line. It also provides more complex commands for searching for—and optionally substituting for—single and multiple occurrences of strings or regular expressions anywhere in the Work buffer.

SEARCHING FOR A CHARACTER

Find (**f**/**F**) You can search for and move the cursor to the next occurrence of a specified character on the current line using the **f** (Find) command. Refer to "Moving the Cursor to a Specific Character" on page 175.

Find (**t**/**T**) The next two commands are used in the same manner as the Find commands. The **t** command places the cursor on the character before the next occurrence of the specified character. The **T** command places the cursor on the character after the previous occurrence of the specified character.

A semicolon (**;**) repeats the last **f**, **F**, **t**, or **T** command.

You can combine these search commands with other commands. For example, the command **d2fq** deletes the text from the current character to the second occurrence of the letter **q** on the current line.

SEARCHING FOR A STRING

Search (**/**/**?**) The vim editor can search backward or forward through the Work buffer to find a string of text or a string that matches a regular expression (Appendix A). To find the next occurrence of a string (forward), press the forward slash (**/**) key, enter the text you want to find (called the *search string*), and press RETURN. When you press the slash key, vim displays a slash on the status line. As you enter the string of text, it is also displayed on the status line. When you press RETURN, vim searches for the string. If this search is successful, vim positions the cursor on the first character of the string. If you use a question mark (**?**) in place of the forward slash, vim searches for the previous occurrence of the string. If you need to include a forward slash in a forward search or a question mark in a backward search, you must quote it by preceding it with a backslash (\).

Two distinct ways of quoting characters

tip You use CONTROL-V to quote special characters in text that you are entering into a file (page 179). This section discusses the use of a backslash (****) to quote special characters in a search string. The two techniques of quoting characters are not interchangeable.

Next (**n**/**N**) The **N** and **n** keys repeat the last search but do not require you to reenter the search string. The **n** key repeats the original search exactly, and the **N** key repeats the search in the opposite direction of the original search.

If you are searching forward and vim does not find the search string before it gets to the end of the Work buffer, the editor typically *wraps around* and continues the search at the beginning of the Work buffer. During a backward search, vim wraps around from the beginning of the Work buffer to the end. Also, vim normally performs case-sensitive searches. Refer to "Wrap scan" (page 199) and "Ignore case in searches" (page 197) for information about how to change these search parameters.

NORMAL VERSUS INCREMENTAL SEARCHES

When vim performs a normal search (its default behavior), you enter a slash or question mark followed by the search string and press RETURN. The vim editor then moves the cursor to the next or previous occurrence of the string you are searching for.

When vim performs an incremental search, you enter a slash or question mark. As you enter each character of the search string, vim moves the highlight to the next or previous occurrence of the string you have entered so far. When the highlight is on the string you are searching for, you must press RETURN to move the cursor to the highlighted string. If the string you enter does not match any text, vim does not highlight anything.

The type of search that vim performs depends on the **incsearch** parameter (page 197). Give the command **:set incsearch** to turn on incremental searching; use **noincsearch** to turn it off. When you set the **compatible** parameter (page 168), vim turns off incremental searching.

SPECIAL CHARACTERS IN SEARCH STRINGS

Because the search string is a regular expression, some characters take on a special meaning within the search string. The following paragraphs list some of these characters. See also "Extended Regular Expressions" on page 1017.

The first two items in the following list (^ and **$**) always have their special meanings within a search string unless you quote them by preceding them with a backslash (\). You can turn off the special meanings within a search string for the rest of the items in the list by setting the **nomagic** parameter. For more information refer to "Allow special characters in searches" on page 196.

^ BEGINNING-OF-LINE INDICATOR

When the first character in a search string is a caret (also called a circumflex), it matches the beginning of a line. For example, the command **/^the** finds the next line that begins with the string **the**.

$ END-OF-LINE INDICATOR

A dollar sign matches the end of a line. For example, the command **/!$** finds the next line that ends with an exclamation point and **/ $** matches the next line that ends with a SPACE.

. ANY-CHARACTER INDICATOR

A period matches *any* character, anywhere in the search string. For example, the command **/l..e** finds **line**, **followed**, **like**, **included**, **all memory**, or any other word or character string that contains an **l** followed by any two characters and an **e**. To search for a period, use a backslash to quote the period (**\.**).

\> END-OF-WORD INDICATOR

This pair of characters matches the end of a word. For example, the command **/s\>** finds the next word that ends with an **s**. Whereas a backslash (\) is typically used to *turn off* the special meaning of a character, the character sequence \> has a special meaning, while > alone does not.

\< BEGINNING-OF-WORD INDICATOR

This pair of characters matches the beginning of a word. For example, the command **/\<The** finds the next word that begins with the string **The**. The beginning-of-word indicator uses the backslash in the same, atypical way as the end-of-word indicator.

* ZERO OR MORE OCCURRENCES

This character is a modifier that will match zero or more occurrences of the character immediately preceding it. For example, the command **/dis*m** will match the string **di** followed by zero or more **s** characters followed by an **m**. Examples of successful matches would include **dim**, or **dism**, and **dissm**.

[ ] CHARACTER-CLASS DEFINITION

Brackets surrounding two or more characters match any *single* character located between the brackets. For example, the command **/dis[ck]** finds the next occurrence of *either* **disk** or **disc**.

There are two special characters you can use within a character-class definition. A caret (^) as the first character following the left bracket defines the character class to be *any except the following characters.* A hyphen between two characters indicates a range of characters. Refer to the examples in Table 6-3.

Table 6-3 Search examples

Search string	What it finds
/and	Finds the next occurrence of the string **and** **Examples: sand and standard slander andiron**
/\<and\>	Finds the next occurrence of the word **and** **Example: and**
/^The	Finds the next line that starts with **The** **Examples:** **The . . .** **There . . .**
/^[0-9][0-9])	Finds the next line that starts with a two-digit number followed by a right parenthesis **Examples:** **77)…** **01)…** **15)…**
/\<[adr]	Finds the next word that starts with **a**, **d**, or **r** **Examples: apple drive road argument right**
/^[A-Za-z]	Finds the next line that starts with an uppercase or lowercase letter **Examples:** **will not find a line starting with the number 7 . . .** **Dear Mr. Jones . . .** **in the middle of a sentence like this . . .**

SUBSTITUTING ONE STRING FOR ANOTHER

A Substitute command combines the effects of a Search command and a Change command. That is, it searches for a string (regular expression) just as the **/** command

does, allowing the same special characters discussed in the previous section. When it finds the string or matches the regular expression, the Substitute command changes the string or regular expression it matches. The syntax of the Substitute command is

:[g][address]s/search-string/replacement-string[/option]

As with all commands that begin with a colon, vim executes a Substitute command from the status line.

THE SUBSTITUTE ADDRESS

If you do not specify an *address*, Substitute searches only the current line. If you use a single line number as the *address*, Substitute searches that line. If the *address* is two line numbers separated by a comma, Substitute searches those lines and the lines between them. Refer to "Line numbers" on page 197 if you want vim to display line numbers. Wherever a line number is allowed in the *address*, you might also use an *address* string enclosed between slashes. The vim editor operates on the next line that the *address* string matches. When you precede the first slash of the *address* string with the letter **g** (for global), vim operates on all lines in the file that the *address* string matches. (This **g** is not the same as the one that goes at the end of the Substitute command to cause multiple replacements on a single line; see "Searching for and Replacing Strings" on the next page).

Within the *address*, a period represents the current line, a dollar sign represents the last line in the Work buffer, and a percent sign represents the entire Work buffer. You can perform *address* arithmetic using plus and minus signs. Table 6-4 shows some examples of *address*es.

Table 6-4 Addresses

Address	Portion of Work buffer addressed
5	Line 5
77,100	Lines 77 through 100 inclusive
1,.	Beginning of Work buffer through current line
.,$	Current line through end of Work buffer
1,$	Entire Work buffer
%	Entire Work buffer
/pine/	The next line containing the word **pine**
g/pine/	All lines containing the word **pine**
.,.+10	Current line through tenth following line (11 lines in all)

SEARCHING FOR AND REPLACING STRINGS

An **s** comes after the *address* in the command syntax, indicating that this is a Substitute command. A delimiter follows the **s**, marking the beginning of the *search-string*. Although the examples in this book use a forward slash, you can use as a delimiter any character that is not a letter, number, blank, or backslash. You must use the same delimiter at the end of the *search-string*.

Next comes the *search-string*. It has the same format as the search string in the **/** command and can include the same special characters (page 185). (The *search-string* is a regular expression; refer to Appendix A for more information.) Another delimiter marks the end of the *search-string* and the beginning of the *replacement-string*.

The *replacement-string* replaces the text matched by the *search-string* and is typically followed by the delimiter character. You can omit the final delimiter when no option follows the *replacement-string;* a final delimiter is required if an option is present.

Several characters have special meanings in the *search-string*, and other characters have special meanings in the *replacement-string*. For example, an ampersand (**&**) in the *replacement-string* represents the text that was matched by the *search-string*. A backslash in the *replacement-string* quotes the character that follows it. Refer to Table 6-5 and Appendix A.

Table 6-5 Search and replace examples

Command	Result
:s/bigger/biggest/	Replaces the first occurrence of the string **bigger** on the current line with **biggest** **Example:** **bigger ⇨ biggest**
:1,.s/Ch 1/Ch 2/g	Replaces every occurrence of the string **Ch 1**, before or on the current line, with the string **Ch 2** **Examples:** **Ch 1 ⇨ Ch 2** **Ch 12 ⇨ Ch 22**
:1,$s/ten/10/g	Replaces every occurrence of the string **ten** with the string **10** **Examples:** **ten ⇨ 10** **often ⇨ of10** **tenant ⇨ 10ant**
:g/chapter/s/ten/10/	Replaces the first occurrence of the string **ten** with the string **10** on all lines containing the word **chapter** **Examples:** **chapter ten ⇨ chapter 10** **chapters will often ⇨ chapters will of10**

Table 6-5 Search and replace examples (continued)

Command	Result
:%s/\<ten\>/10/g	Replaces every occurrence of the word **ten** with the string **10** **Example:** **ten ⇨ 10**
:.,.+10s/every/each/g	Replaces every occurrence of the string **every** with the string **each** on the current line through the tenth following line **Examples:** **every ⇨ each** **everything ⇨ eachthing**
:s/\<short\>/"&"/	Replaces the word **short** on the current line with **"short"** (enclosed within quotation marks) **Example:** **the shortest of the short ⇨ the shortest of the "short"**

Normally, the Substitute command replaces only the first occurrence of any text that matches the *search-string* on a line. If you want a global substitution—that is, if you want to replace all matching occurrences of text on a line—append the **g** (global) option after the delimiter that ends the *replacement-string*. Another useful option, **c** (check), causes vim to ask whether you would like to make the change each time it finds text that matches the *search-string*. Pressing **y** replaces the *search-string*, **q** terminates the command, **l** (last) makes the replacement and quits, **a** (all) makes all remaining replacements, and **n** continues the search without making that replacement.

The *address* string need not be the same as the *search-string*. For example,

```
:/candle/s/wick/flame/
```

substitutes **flame** for the first occurrence of **wick** on the next line that contains the string **candle**. Similarly,

```
:g/candle/s/wick/flame/
```

performs the same substitution for the first occurrence of **wick** on each line of the file containing the string **candle** and

```
:g/candle/s/wick/flame/g
```

performs the same substitution for all occurrences of **wick** on each line that contains the string **candle**.

If the *search-string* is the same as the *address*, you can leave the *search-string* blank. For example, the command **:/candle/s//lamp/** is equivalent to the command **:/candle/s/candle/lamp/**.

MISCELLANEOUS COMMANDS

This section describes three commands that do not fit naturally into any other groups.

JOIN

Join (**J**)

The **J** (Join) command joins the line below the current line to the end of the current line, inserting a SPACE between what was previously two lines and leaving the cursor on this SPACE. If the current line ends with a period, vim inserts two SPACES.

You can always "unjoin" (break) a line into two lines by replacing the SPACE or SPACES where you want to break the line with a RETURN.

STATUS

Status (CONTROL-G)

The Status command, CONTROL-G, displays the name of the file you are editing, information about whether the file has been modified or is a readonly file, the number of the current line, the total number of lines in the Work buffer, and the percentage of the Work buffer preceding the current line. You can also use **:f** to display status information. Following is a sample status line:

```
"/usr/share/dict/words" [readonly] line 28501 of 98569 --28%-- col 1
```

. (PERIOD)

Repeat last command (**.**)

The **.** (period) command repeats the most recent command that made a change. If you had just given a **d2w** command (delete the next two words), for example, the **.** command would delete the next two words. If you had just inserted text, the **.** command would repeat the insertion of the same text. This command is useful if you want to change some occurrences of a word or phrase in the Work buffer. Search for the first occurrence of the word (use **/**) and then make the change you want (use **cw**). You can then use **n** to search for the next occurrence of the word and **.** to make the same change to it. If you do not want to make the change, give the **n** command again to find the next occurrence.

8

The Bourne Again Shell (bash)

In This Chapter

Startup Files 278
Redirecting Standard Error 282
Writing and Executing a Simple Shell Script 284
Job Control 294
Manipulating the Directory Stack 297
Parameters and Variables 300
Locale 316
Processes 323
History 326
Reexecuting and Editing Commands 328
Functions 346
Controlling bash: Features and Options 349
Processing the Command Line . . . 354

Objectives

After reading this chapter you should be able to:

- Describe the purpose and history of bash
- List the startup files bash runs
- Use three different methods to run a shell script
- Understand the purpose of the **PATH** variable
- Manage multiple processes using job control
- Redirect error messages to a file
- Use control operators to separate and group commands
- Create variables and display the values of variables and parameters
- List and describe common variables found on the system
- Reference, repeat, and modify previous commands using history
- Use control characters to edit the command line
- Create, display, and remove aliases and functions
- Customize the bash environment using the set and shopt builtins
- List the order of command-line expansion

EXCERPT

This chapter picks up where Chapter 5 left off by focusing on the Bourne Again Shell (bash). It notes where tcsh implementation of a feature differs from that of bash; if appropriate, you are directed to the page where the alternative implementation is discussed. Chapter 10 expands on this chapter, exploring control flow commands and more advanced aspects of programming the Bourne Again Shell. The bash home page is at www.gnu.org/software/bash. The bash info page is a complete Bourne Again Shell reference.

The Bourne Again Shell (bash) and the TC Shell (tcsh) are command interpreters and high-level programming languages. As command interpreters, they process commands you enter on the command line in response to a prompt. When you use the shell as a programming language, it processes commands stored in files called *shell scripts*. Like other languages, shells have variables and control flow commands (e.g., **for** loops and **if** statements).

When you use a shell as a command interpreter, you can customize the environment you work in. You can make the prompt display the name of the working directory, create a function or an alias for cp that keeps it from overwriting certain kinds of files, take advantage of keyword variables to change aspects of how the shell works, and so on. You can also write shell scripts that do your bidding—anything from a one-line script that stores a long, complex command to a longer script that runs a set of reports, prints them, and mails you a reminder when the job is done. More complex shell scripts are themselves programs; they do not just run other programs. Chapter 10 has some examples of these types of scripts.

Most system shell scripts are written to run under bash (or dash; next page). If you will ever work in single-user/recovery mode—when you boot the system or perform system maintenance, administration, or repair work, for example—it is a good idea to become familiar with this shell.

This chapter expands on the interactive features of the shell described in Chapter 5, explains how to create and run simple shell scripts, discusses job control, talks about locale, introduces the basic aspects of shell programming, talks about history and aliases, and describes command-line expansion. Chapter 9 covers interactive use of the TC Shell and TC Shell programming, and Chapter 10 presents some more challenging shell programming problems.

BACKGROUND

bash Shell

The Bourne Again Shell is based on the Bourne Shell (an early UNIX shell; this book refers to it as the *original Bourne Shell* to avoid confusion), which was written by Steve Bourne of AT&T's Bell Laboratories. Over the years the original Bourne Shell has been expanded, but it remains the basic shell provided with many commercial versions of UNIX.

Blank

USER-CREATED VARIABLES

The first line in the following example declares the variable named **person** and initializes it with the value **max**:

```
$ person=max
$ echo person
person
$ echo $person
max
```

Parameter substitution

Because the echo builtin copies its arguments to standard output, you can use it to display the values of variables. The second line of the preceding example shows that **person** does not represent **max**. Instead, the string **person** is echoed as **person**. The shell substitutes the value of a variable only when you precede the name of the variable with a dollar sign (**$**). Thus the command **echo $person** displays the value of the variable **person**; it does not display **$person** because the shell does not pass **$person** to echo as an argument. Because of the leading **$**, the shell recognizes that **$person** is the name of a variable, *substitutes* the value of the variable, and passes that value to echo. The echo builtin displays the value of the variable (not its name), never "knowing" you called it with the name of a variable.

Quoting the **$**

You can prevent the shell from substituting the value of a variable by quoting the leading **$**. Double quotation marks do not prevent the substitution; single quotation marks or a backslash (\) do.

```
$ echo $person
max
$ echo "$person"
max
$ echo '$person'
$person
$ echo \$person
$person
```

SPACES

Because they do not prevent variable substitution but do turn off the special meanings of most other characters, double quotation marks are useful when you assign values to variables and when you use those values. To assign a value that contains SPACES or

TABS to a variable, use double quotation marks around the value. Although double quotation marks are not required in all cases, using them is a good habit.

```
$ person="max and zach"
$ echo $person
max and zach
$ person=max and zach
bash: and: command not found
```

When you reference a variable whose value contains TABS or multiple adjacent SPACES, you must use quotation marks to preserve the spacing. If you do not quote the variable, the shell collapses each string of blank characters into a single SPACE before passing the variable to the utility:

```
$ person="max   and   zach"
$ echo $person
max and zach
$ echo "$person"
max   and   zach
```

Pathname expansion in assignments

When you execute a command with a variable as an argument, the shell replaces the name of the variable with the value of the variable and passes that value to the program being executed. If the value of the variable contains a special character, such as ***** or **?**, the shell *might* expand that variable.

The first line in the following sequence of commands assigns the string **max*** to the variable **memo**. All shells interpret special characters as special when you reference a variable that contains an unquoted special character. In the following example, the shell expands the value of the **memo** variable because it is not quoted:

```
$ memo=max*
$ ls
max.report
max.summary
$ echo $memo
max.report max.summary
```

Above, the shell expands the **$memo** variable to **max***, expands **max*** to **max.report** and **max.summary**, and passes these two values to echo. In the next example, the Bourne Again Shell *does not expand the string* because bash does not perform pathname expansion (page 148) when it assigns a value to a variable.

```
$ echo "$memo"
max*
```

All shells process a command line in a specific order. Within this order bash (but not tcsh) expands variables before it interprets commands. In the preceding echo command line, the double quotation marks quote the asterisk (*****) in the expanded value of **$memo** and prevent bash from performing pathname expansion on the expanded **memo** variable before passing its value to the echo command.

optional

Braces around variables

The *$VARIABLE* syntax is a special case of the more general syntax *${VARIABLE}*, in which the variable name is enclosed by **${}**. The braces insulate the variable name from adjacent characters. Braces are necessary when catenating a variable value with a string:

```
$ PREF=counter
$ WAY=$PREFclockwise
$ FAKE=$PREFfeit
$ echo $WAY $FAKE

$
```

The preceding example does not work as expected. Only a blank line is output because although **PREFclockwise** and **PREFfeit** are valid variable names, they are not initialized. By default the shell evaluates an unset variable as an empty (null) string and displays this value (bash) or generates an error message (tcsh). To achieve the intent of these statements, refer to the **PREF** variable using braces:

```
$ PREF=counter
$ WAY=${PREF}clockwise
$ FAKE=${PREF}feit
$ echo $WAY $FAKE
counterclockwise counterfeit
```

The Bourne Again Shell refers to command-line arguments using the positional parameters **$1**, **$2**, **$3**, and so forth up to **$9**. You must use braces to refer to arguments past the ninth argument: **${10}**. The name of the command is held in **$0** (page 458).

unset: REMOVES A VARIABLE

Unless you remove a variable, it exists as long as the shell in which it was created exists. To remove the *value* of a variable but not the variable itself, assign a null value to the variable. In the following example, set (page 460) displays a list of all variables and their values; grep extracts the line that shows the value of person.

```
$ echo $person
zach
$ person=
$ echo $person

$ set | grep person
person=
```

You can remove a variable using the unset builtin. The following command removes the variable **person**:

```
$ unset person
$ echo $person

$ set | grep person
$
```

Variable Attributes

This section discusses attributes and explains how to assign attributes to variables.

readonly: Makes the Value of a Variable Permanent

You can use the readonly builtin (not in tcsh) to ensure the value of a variable cannot be changed. The next example declares the variable **person** to be readonly. You must assign a value to a variable *before* you declare it to be readonly; you cannot change its value after the declaration. When you attempt to change the value of or unset a readonly variable, the shell displays an error message:

```
$ person=zach
$ echo $person
zach
$ readonly person
$ person=helen
bash: person: readonly variable
$ unset person
bash: unset: person: cannot unset: readonly variable
```

If you use the readonly builtin without an argument, it displays a list of all readonly shell variables. This list includes keyword variables that are automatically set as readonly as well as keyword or user-created variables that you have declared as readonly. See the next page for an example (**readonly** and **declare –r** produce the same output).

declare: Lists and Assigns Attributes to Variables

The declare builtin (not in tcsh) lists and sets attributes and values for shell variables. The typeset builtin (another name for declare) performs the same function but is deprecated. Table 8-3 lists five of these attributes.

Table 8-3 Variable attributes (declare)

Attribute	Meaning
–a	Declares a variable as an array (page 474)
–f	Declares a variable to be a function name (page 346)
–i	Declares a variable to be of type integer (page 306)
–r	Makes a variable readonly; also readonly (above)
–x	Makes a variable an environment variable; also export (page 468)

The following commands declare several variables and set some attributes. The first line declares **person1** and initializes it to **max**. This command has the same effect with or without the word **declare**.

```
$ declare person1=max
$ declare -r person2=zach
$ declare -rx person3=helen
$ declare -x person4
```

readonly and export

The readonly and export builtins are synonyms for the commands **declare –r** and **declare –x**, respectively. You can declare a variable without initializing it, as the preceding declaration of the variable **person4** illustrates. This declaration makes **person4** an environment variable so it is available to all subshells. Until **person4** is initialized, it has a null value.

You can list the options to declare separately in any order. The following is equivalent to the preceding declaration of **person3**:

```
$ declare -x -r person3=helen
```

Use the **+** character in place of **–** when you want to remove an attribute from a variable. You cannot remove the readonly attribute. After the following command is given, the variable **person3** is no longer exported, but it is still readonly:

```
$ declare +x person3
```

See page 468 for more information on exporting variables.

Listing variable attributes

Without any arguments or options, declare lists all shell variables. The same list is output when you run set (page 461) without any arguments.

If you call declare with options but no variable names, the command lists all shell variables that have the specified attributes set. For example, the command **declare –r** displays a list of all readonly variables. This list is the same as that produced by the **readonly** command without any arguments. After the declarations in the preceding example have been given, the results are as follows:

```
$ declare -r
declare -r BASHOPTS="checkwinsize:cmdhist:expand_aliases: ... "
declare -ir BASHPID
declare -ar BASH_VERSINFO='([0]="4" [1]="2" [2]="24" [3]="1" ... '
declare -ir EUID="500"
declare -ir PPID="1936"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand: ... "
declare -ir UID="500"
declare -r person2="zach"
declare -rx person3="helen"
```

The first seven entries are keyword variables that are automatically declared as readonly. Some of these variables are stored as integers (**–i**). The **–a** option indicates that **BASH_VERSINFO** is an array variable; the value of each element of the array is listed to the right of an equal sign.

Integer

By default, the values of variables are stored as strings. When you perform arithmetic on a string variable, the shell converts the variable into a number, manipulates it, and then converts it back to a string. A variable with the integer attribute is stored as an integer. Assign the integer attribute as follows:

```
$ declare -i COUNT
```

You can use declare to display integer variables:

```
$ declare -i
declare -ir BASHPID
declare -i COUNT
declare -ir EUID="1000"
declare -i HISTCMD
declare -i LINENO
declare -i MAILCHECK="60"
declare -i OPTIND="1"
...
```

Blank

12

The Python Programming Language

EXCERPT

In This Chapter

Invoking Python 564
Lists 569
Dictionaries 573
Control Structures 574
Reading from and Writing to Files 579
Pickle 582
Regular Expressions 583
Defining a Function 584
Using Libraries 585
Lambda Functions 589
List Comprehensions 590

Objectives

After reading this chapter you should be able to:

- Give commands using the Python interactive shell
- Write and run a Python program stored in a file
- Demonstrate how to instantiate a list and how to remove elements from and add elements to a list
- Describe a dictionary and give examples of how it can be used
- Describe three Python control structures
- Write a Python program that iterates through a list or dictionary
- Read from and write to a file
- Demonstrate exception processing
- Preserve an object using **pickle()**
- Write a Python program that uses regular expressions
- Define a function and use it in a program

INTRODUCTION

Python is a friendly and flexible programming language in widespread use everywhere from Fortune 500 companies to large-scale open-source projects. Python is an interpreted language: It translates code into *bytecode* (page 1059) at runtime and executes the bytecode within the Python virtual machine. Contrast Python with the C language, which is a compiled language. C differs from Python in that the C compiler *compiles* C source code into architecture-specific machine code. Python programs are not compiled; you run a Python program the same way you run a bash or Perl script. Because Python programs are not compiled, they are portable between operating systems and architectures. In other words, the same Python program will run on any system to which the Python virtual machine has been ported.

Object oriented

While not required to use the language, Python supports the object-oriented (OO) paradigm. It is possible to use Python with little or no understanding of object-oriented concepts and this chapter covers OO programming minimally while still explaining Python's important features.

Libraries

Python comes with hundreds of prewritten tools that are organized into logical libraries. These libraries are accessible to Python programs, but not loaded into memory at runtime because doing so would significantly increase startup times for Python programs. Entire libraries (or just individual modules) are instead loaded into memory when the program requests them.

Version

Python is available in two main development branches: Python 2.x and Python 3.x. This chapter focuses on Python 2.x because the bulk of Python written today uses 2.x. The following commands show that two versions of Python are installed and that the **python** command runs Python 2.7.3:

```
$ whereis python
python: /usr/bin/python /usr/bin/python2.7 /etc/python2.6 /etc/python ...
$ ls -l $(which python)
lrwxrwxrwx 1 root root 9 Apr 17 10:20 /usr/bin/python -> python2.7
$ python -V
Python 2.7.3
```

INVOKING PYTHON

This section discusses the methods you can use to run a Python program.

Interactive shell

Most of the examples in this chapter use the Python interactive shell because you can use it to debug and execute code one line at a time and see the results immediately. Although this shell is handy for testing, it is not a good choice for running longer, more complex programs. You start a Python interactive shell by calling the python utility (just as you would start a bash shell by calling bash). The primary Python

prompt is >>>. When Python requires more input to complete a command, it displays its secondary prompt (**...**).

```
$ python
Python 2.7.3 (default, Apr 20 2012, 22:39:59)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

While you are using the Python interactive shell, you can give Python a command by entering the command and pressing RETURN.

```
>>> print 'Good morning!'
Good morning!
```

Implied display

tip Within the Python interactive shell, Python displays output from any command line that does not have an action. The output is similar to what **print** would display, although it might not be exactly the same. The following examples show explicit **print** and implicit display actions:

```
>>> print 'Good morning!'
Good morning!
>>> 'Good morning!'
'Good morning!'

>>> print 2 + 2
4
>>> 2 + 2
4
```

Implied display allows you to display the value of a variable by typing its name:

```
>>> x = 'Hello'
>>> x
'Hello'
```

Implied display does not work unless you are running the Python interactive shell (i.e., Python does not invoke an implicit display action when it is run from a file).

Program file

Most of the time a Python program is stored in a text file. Although not required, the file typically has a filename extension of **.py**. Use chmod (page 741) to make the file executable. As explained on page 287, the **#!** at the start of the first line of the file instructs the shell to pass the rest of the file to **/usr/bin/python** for execution.

```
$ chmod 755 gm.py
$ cat gm.py
#!/usr/bin/python
print 'Good morning!'

$ ./gm.py
Good morning!
```

Blank

READING FROM AND WRITING TO FILES

Python allows you to work with files in many ways. This section explains how to read from and write to text files and how to preserve an object in a file using **pickle**.

FILE INPUT AND OUTPUT

open() The **open()** function opens a file and returns a file object called a *file handle;* it can open a file in one of several modes (Table 12-3). Opening a file in **w** (write) mode truncates the file; use **a** (append) mode if you want to add to a file. The following statement opens the file in Max's home directory named **test_file** in read mode; the file handle is named **f**.

```
f = open('/home/max/test_file', 'r')
```

Table 12-3 File modes

Mode	What it does
r	**Read only** Error if file does not exist.
w	**Write only** File is created if it does not exist. File is truncated if it exists.
r+	**Read and write** File is created if it does not exist.
a	**Append** File is created if it does not exist.
a+	**Append and read** File is created if it does not exist.
b	**Binary** Append to **r** or **w** to work with binary files.

Once the file is opened, you direct input and output using the file handle with one of the methods listed in Table 12-4. When you are finished working with a file, use **close()** to close it and free the resources the open file is using.

Table 12-4 File object methods

Method	Arguments	Returns or action
close()	None	Closes the file
isatty()	None	Returns *true* if the file is connected to a terminal; *false* otherwise
read()	Maximum number of bytes to read (optional)	Reads until EOF or specified maximum number of bytes; returns file as a string
readline()	Maximum number of bytes to read (optional)	Reads until NEWLINE or specified maximum number of bytes; returns line as a string
readlines()	Maximum number of bytes to read (optional)	Calls **readline()** repeatedly and returns a list of lines (iterable)
write(str)	String to be written	Writes to the file
writelines(strs)	List of strings	Calls **write()** repeatedly, once with each item in the list

The following example reads from **/home/max/test_file**, which holds three lines. It opens this file in read mode and assigns the file handle **f** to the open file. It uses the **readlines()** method, which reads the entire file into a list and returns that list. Because the list is iterable, Python passes to the **for** control structure one line from **test_file** each time through the loop. The **for** structure assigns the string value of this line to **ln**, which **print** then displays. The **strip()** method removes whitespace and/or a NEWLINE from the end of a line. Without **strip()**, **print** would output two NEWLINES: the one that terminates the line from the file and the one it automatically appends to each line it outputs. After reading and displaying all lines from the file, the example closes the file.

```
>>> f = open('/home/max/test_file', 'r')
>>> for ln in f.readlines():
...     print ln.strip()
...
This is the first line
and here is the second line
of this file.
>>> f.close()
```

The next example opens the same file in append mode and writes a line to it using **write()**. The **write()** method does not append a NEWLINE to the line it outputs, so you must terminate the string you write to the file with a **\n**.

```
>>> f = open('/home/max/test_file','a')
>>> f.write('Extra line!\n')
>>> f.close()
```

optional In the example that uses **for**, Python does not call the **readlines()** method each time through the **for** loop. Instead, it reads the file into a list the first time **readlines()** is called and then iterates over the list, setting **ln** to the value of the next line in the list each time it is called subsequently. It is the same as if you had written

```
>>> f = open('/home/max/test_file', 'r')
>>> lines = f.readlines()
>>> for ln in lines:
...     print ln.strip()
```

It is more efficient to iterate over the file handle directly because this technique does not store the file in memory.

```
>>> f = open('/home/max/test_file', 'r')
>>> for ln in f:
...     print ln.strip()
```

EXCEPTION HANDLING

An *exception* is an error condition that changes the normal flow of control in a program. Although you can try to account for every problem your code will need

optional

LAMBDA FUNCTIONS

Python supports *Lambda* functions—functions that might not be bound to a name. You might also see them referred to as *anonymous* functions. Lambda functions are more restrictive than other functions because they can hold only a single expression. In its most basic form, Lambda is another syntax for defining a function. In the following example, the object named **a** is a Lambda function and performs the same task as the function named **add_one**:

```
>>> def add_one(x):
...     return x + 1
...
>>> type (add_one)
<type 'function'>

>>> add_one(2)
3

>>> a = lambda x: x + 1
>>> type(a)
<type 'function'>

>>> a(2)
3
```

map() You can use the Lambda syntax to define a function inline as an argument to a function such as **map()** that expects another function as an argument. The syntax of the **map()** function is

map(func, seq1[, seq2, ...])

where *func* is a function that is applied to the sequence of arguments represented by *seq1* (and *seq2* ...). Typically the sequences that are arguments to **map()** and the

object returned by **map()** are lists. The next example first defines a function named **times_two()**:

```
>>> def times_two(x):
...     return x * 2
...
>>> times_two(8)
16
```

Next, the **map()** function applies **times_two()** to a list:

```
>>> map(times_two, [1, 2, 3, 4])
[2, 4, 6, 8]
```

You can define an inline Lambda function as an argument to **map()**. In this example the Lambda function is not bound to a name.

```
>>> map(lambda x: x * 2, [1, 2, 3, 4])
[2, 4, 6, 8]
```

LIST COMPREHENSIONS

List comprehensions apply functions to lists. For example, the following code, which does not use a list comprehension, uses **for** to iterate over items in a list:

```
>>> my_list = []
>>> for x in range(10):
...     my_list.append(x + 10)
...
>>> my_list
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

You can use a list comprehension to perform the same task neatly and efficiently. The syntax is similar, but a list comprehension is enclosed within square brackets and the operation (**x + 10**) precedes the iteration [**for x in range(10)**].

```
>>> my_list = [x + 10 for x in range(10)]
>>> my_list
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

The results when using a **for** structure and a list comprehension are the same. The next example uses a list comprehension to fill a list with powers of 2:

```
>>> potwo = [2**x for x in range(1, 13)]
>>> print potwo
[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
```

The next list comprehension fills a list with even numbers. The **if** clause returns values only if the remainder after dividing a number by 2 is 0 (**if x % 2 == 0**).

```
>>> [x for x in range(1,11) if x % 2 == 0]
[2, 4, 6, 8, 10]
```

PART VI

Command Reference

EXCERPT

Blank

sample OS X ← *OS X in an oval indicates this utility runs under Mac OS X only.*

Brief description of what the utility does.

*sample [**options**] **arguments***

Following the syntax line is a description of the utility. The syntax line shows how to run the utility from the command line. Options and arguments enclosed in brackets ([]) are not required. Enter words that appear in *this italic typeface* as is. Words that you must replace when you enter the command appear in ***this bold italic typeface***. Words listed as arguments to a command identify single arguments (for example, ***source-file***) or groups of similar arguments (for example, ***directory-list***). A note here indicates if the utility runs under Linux or OS X only. OS X

Arguments

This section describes the arguments you can use when you run the utility. The argument, as shown in the preceding syntax line, is printed in ***this bold italic typeface***.

Options

This section lists some of the options you can use with the command. Unless otherwise specified, you must precede options with one or two hyphens. Most commands accept a single hyphen before multiple options (page 128). Options in this section are ordered alphabetically by short (single-hyphen) options. If an option has only a long version (two hyphens), it is ordered by its long option. Following are some sample options:

––delimiter=*dchar*

–d *dchar*

This option includes an argument. The argument is set in a ***bold italic typeface*** in both the heading and the description. You substitute another word (filename, string of characters, or other value) for any arguments shown in ***this typeface***. Type characters that are in **bold type** (such as the **––delimiter** and **–d**) as is.

––make-dirs **–m** This option has a long and a short version. You can use either option; they are equivalent. This option description ends with **Linux** in a box, indicating it is available under Linux only. Options *not* followed by **Linux** or **OS X** are available under both operating systems. LINUX

–t (**table of contents**) This simple option is preceded by a single hyphen and not followed by arguments. It has no long version. The **table of contents** appearing in parentheses at the beginning of the description is a cue, suggestive of what the option letter stands for. This option description ends with **OS X** in a box, indicating it is available under OS X only. Options *not* followed by **Linux** or **OS X** are available under both operating systems. OS X

Discussion

This optional section describes how to use the utility and identifies any quirks it might have.

Notes

This section contains miscellaneous notes—some important and others merely interesting.

Examples

This section contains examples illustrating how to use the utility. This section is a tutorial, so it takes a more casual tone than the preceding sections of the description.

Blank

bzip2

Compresses or decompresses files

*bzip2 [**options**] [**file-list**]*
*bunzip2 [**options**] [**file-list**]*
*bzcat [**options**] [**file-list**]*
*bzip2recover [**file**]*

The bzip2 utility compresses files, bunzip2 restores files compressed using bzip2, and bzcat displays files compressed with bzip2.

Arguments

The ***file-list*** is a list of one or more ordinary files (no directories) that are to be compressed or decompressed. If ***file-list*** is empty or if the special option **–** is present, bzip2 reads from standard input. The **––stdout** option causes bzip2 to write to standard output.

Options

Under Linux, bzip2, bunzip2, and bzcat accept the common options described on page 718.

The Mac OS X version of bzip2 accepts long options

tip Options for bzip2 preceded by a double hyphen (**––**) work under Mac OS X as well as under Linux.

––stdout	**–c**	Writes the results of compression or decompression to standard output.
––decompress	**–d**	Decompresses a file that was compressed using bzip2. This option with bzip2 is equivalent to the bunzip2 command.
––fast *or* **––best**	***–n***	Sets the block size when compressing a file. The ***n*** is a digit from 1 to 9, where 1 (**––fast**) generates a block size of 100 kilobytes and 9 (**––best**) generates a block size of 900 kilobytes. The default level is 9. The **––fast** and **––best** options are provided for compatibility with gzip and do not necessarily yield the fastest or best compression.
––force	**–f**	Forces compression even if a file already exists, has multiple links, or comes directly from a terminal. The option has a similar effect with bunzip2.
––keep	**–k**	Does not delete input files while compressing or decompressing them.
––quiet	**–q**	Suppresses warning messages; does display critical messages.
––test	**–t**	Verifies the integrity of a compressed file. Displays nothing if the file is OK.
––verbose	**–v**	For each file being compressed, displays the name of the file, the compression ratio, the percentage of space saved, and the sizes of the decompressed and compressed files.

Discussion

The bzip2 and bunzip2 utilities work similarly to gzip and gunzip; see the discussion of gzip (page 839) for more information. Normally bzip2 does not overwrite a file; you must use **--force** to overwrite a file during compression or decompression.

Notes

The bzip2 home page is bzip.org.

The bzip2 utility does a better job of compressing files than gzip does.

Use the **--bzip2** modifier with tar (page 969) to compress archive files using bzip2.

See page 66 for additional information on and examples of using tar to create and unpack archives.

bzcat ***file-list*** Works like cat except it uses bunzip2 to decompress ***file-list*** as it copies files to standard output.

bzip2recover Attempts to recover a damaged file that was compressed using bzip2.

Examples

In the following example, bzip2 compresses a file and gives the resulting file the same name with a **.bz2** filename extension. The **-v** option displays statistics about the compression.

```
$ ls -l
-rw-r--r--  1 sam sam 737414 04-03 19:05 bigfile
$ bzip2 -v bigfile
  bigfile: 3.926:1, 2.037 bits/byte, 74.53% saved, 737414 in, 187806 out
$ ls -l
-rw-r--r--  1 sam sam 187806 04-03 19:05 bigfile.bz2
```

Next touch creates a file with the same name as the original file; bunzip2 refuses to overwrite the file in the process of decompressing **bigfile.bz2**. The **--force** option enables bunzip2 to overwrite the file.

```
$ touch bigfile
$ bunzip2 bigfile.bz2
bunzip2: Output file bigfile already exists.
$ bunzip2 --force bigfile.bz2
$ ls -l
-rw-r--r--  1 sam sam 737414 04-03 19:05 bigfile
```

cp

Copies files

*cp [**options**] **source-file destination-file***
*cp [**options**] **source-file-list destination-directory***

The cp utility copies one or more files. It can either make a copy of a single file (first syntax) or copy one or more files to a directory (second syntax). With the **–R** option, cp can copy directory hierarchies.

Arguments

The ***source-file*** is the pathname of the file that cp makes a copy of. The ***destination-file*** is the pathname cp assigns to the resulting copy of the file.

The ***source-file-list*** is a list of one or more pathnames of files that cp makes copies of. The ***destination-directory*** is the pathname of the directory in which cp places the copied files. With this syntax, cp gives each copied file the same simple filename as its ***source-file***.

The **–R** option enables cp to copy directory hierarchies recursively from the ***source-file-list*** into the ***destination-directory***.

Options

Under Linux, cp accepts the common options described on page 718. Options preceded by a double hyphen (**––**) work under Linux only. Except as noted, options named with a single letter and preceded by a single hyphen work under Linux and OS X.

––archive **–a** Attempts to preserve the owner, group, permissions, access date, and modification date of source file(s) while copying recursively without dereferencing symbolic links. Same as **–dpR**.

––backup **–b** If copying a file would remove or overwrite an existing file, this option makes a backup copy of the file that would be overwritten. The backup copy has the same name as the ***destination-file*** with a tilde (~) appended to it. When you use both **––backup** and **––force**, cp makes a backup copy when you try to copy a file over itself. For more backup options, search for **Backup options** in the **core utils** info page. LINUX

–d For each file that is a symbolic link, copies the symbolic link, not the file the link points to. Also preserves hard links in ***destination-files*** that exist between corresponding ***source-files***. This option is equivalent to **––no-dereference** and **––preserve=links**. See page 115 for information on dereferencing symbolic links. LINUX

––force **–f** When the ***destination-file*** exists but cannot be opened for writing, causes cp to try to remove ***destination-file*** before copying ***source-file***. This option is useful when the user copying a file does not have write permission to an

existing ***destination-file*** but does have write permission to the directory containing the ***destination-file***. Use this option with **–b** to back up a destination file before removing or overwriting it.

–H (**partial dereference**) For each file that is a symbolic link, copies the file the link points to, not the symbolic link itself. This option affects files specified on the command line; it does not affect files found while descending a directory hierarchy. This option treats files that are not symbolic links normally. Under OS X, works with **–R** only. See page 115 for information on dereferencing symbolic links.

––interactive **–i** Prompts you whenever cp would overwrite a file. If you respond with a string that starts with **y** or **Y**, cp copies the file. If you enter anything else, cp does not copy the file.

––dereference **–L** (**dereference**) For each file that is a symbolic link, copies the file the link points to, not the symbolic link itself. This option affects all files and treats files that are not symbolic links normally. Under OS X, works with **–R** only. See page 115 for information on dereferencing symbolic links.

––no-dereference **–P** (**no dereference**) For each file that is a symbolic link, copies the symbolic link, not the file the link points to. This option affects all files and treats files that are not symbolic links normally. Under OS X, works with **–R** only. See page 115 for information on dereferencing symbolic links.

––preserve[=*attr*] **–p** Creates a ***destination-file*** with the same owner, group, permissions, access date, and modification date as the ***source-file***. The **–p** option does not take an argument.

Without ***attr***, **––preserve** works as described above. The ***attr*** is a comma-separated list that can include **mode** (permissions and ACLs), **ownership** (owner and group), **timestamps** (access and modification dates), **links** (hard links), and **all** (all attributes).

––parents Copies a relative pathname to a directory, creating directories as needed. See the "Examples" section. LINUX

––recursive **–R** *or* **–r**
Recursively copies directory hierarchies including ordinary files. Under Linux, the **––no-dereference** (**–d**) option is implied: With the **–R**, **–r**, or **––recursive** option, cp copies the links (not the files the links point to). The **–r** and **––recursive** options are available under Linux only.

––update **–u** Copies only when the ***destination-file*** does not exist or when it is older than the ***source-file*** (i.e., this option will not overwrite a newer destination file). LINUX

––verbose **–v** Displays the name of each file as cp copies it.

–X Do not copy extended attributes (page 1044). OS X

Notes

Under Linux, cp dereferences symbolic links unless you use one or more of the **–R**, **–r**, **––recursive**, **–P**, **–d**, or **––no-dereference** options. As explained on the previous page, under Linux the **–H** option dereferences only symbolic links listed on the command line. Under Mac OS X, without the **–R** option, cp always dereferences symbolic links; with the **–R** option, cp does not dereference symbolic links (**–P** is the default) unless you specify **–H** or **–L**.

Many options are available for cp under Linux. See the **coreutils** info page for a complete list.

If the ***destination-file*** exists before you execute a cp command, cp overwrites the file, destroying its contents but leaving the access privileges, owner, and group associated with the file as they were.

If the ***destination-file*** does not exist, cp uses the access privileges of the ***source-file***. The user who copies the file becomes the owner of the ***destination-file*** and the user's login group becomes the group associated with the ***destination-file***.

Using the **–p** option (or **––preserve** without an argument) causes cp to attempt to set the owner, group, permissions, access date, and modification date to match those of the ***source-file***.

Unlike with the ln utility (page 856), the ***destination-file*** that cp creates is independent of its ***source-file***.

Under OS X version 10.4 and above, cp copies extended attributes (page 1044). The **–X** option causes cp not to copy extended attributes.

Examples

The first command makes a copy of the file **letter** in the working directory. The name of the copy is **letter.sav**.

```
$ cp letter letter.sav
```

The next command copies all files with a filename extension of **.c** to the **archives** directory, which is a subdirectory of the working directory. Each copied file retains its simple filename but has a new absolute pathname. The **–p** (**––preserve**) option causes the copied files in **archives** to have the same owner, group, permissions, access date, and modification date as the source files.

```
$ cp -p *.c archives
```

The next example copies **memo** from Sam's home directory to the working directory:

```
$ cp ~sam/memo .
```

The next example runs under Linux and uses the **––parents** option to copy the file **memo/thursday/max** to the **dir** directory as **dir/memo/thursday/max**. The find utility shows the newly created directory hierarchy.

```
$ cp --parents memo/thursday/max dir
$ find dir
dir
dir/memo
dir/memo/thursday
dir/memo/thursday/max
```

The following command copies the files named **memo** and **letter** into another directory. The copies have the same simple filenames as the source files (**memo** and **letter**) but have different absolute pathnames. The absolute pathnames of the copied files are **/home/sam/memo** and **/home/sam/letter**, respectively.

```
$ cp memo letter /home/sam
```

The final command demonstrates one use of the **–f** (**––force**) option. Max owns the working directory and tries unsuccessfully to copy **one** over another file (**me**) that he does not have write permission for. Because he has write permission to the directory that holds **me**, Max can remove the file but cannot write to it. The **–f** (**––force**) option unlinks, or removes, **me** and then copies **one** to the new file named **me**.

```
$ ls -ld
drwxrwxr-x    2 max max 4096 10-16 22:55 .
$ ls -l
-rw-r--r--    1 root root 3555 10-16 22:54 me
-rw-rw-r--    1 max max 1222 10-16 22:55 one
$ cp one me
cp: cannot create regular file 'me': Permission denied
$ cp -f one me
$ ls -l
-rw-r--r--    1 max max 1222 10-16 22:58 me
-rw-rw-r--    1 max max 1222 10-16 22:55 one
```

If Max had used the **–b** (**––backup**) option in addition to **–f** (**––force**), cp would have created a backup of **me** named **me~**. Refer to "Directory Access Permissions" on page 103 for more information.

cut

Selects characters or fields from input lines

*cut [**options**] [**file-list**]*

The cut utility selects characters or fields from lines of input and writes them to standard output. Character and field numbering start with 1.

Arguments

The ***file-list*** is a list of ordinary files. If you do not specify an argument or if you specify a hyphen (–) in place of a filename, cut reads from standard input.

Options

Under Linux, cut accepts the common options described on page 718. Options preceded by a double hyphen (––) work under Linux only. Options named with a single letter and preceded by a single hyphen work under Linux and OS X.

––characters=*clist*
–c *clist*
Selects the characters given by the column numbers in ***clist***. The value of ***clist*** is one or more comma-separated column numbers or column ranges. A range is specified by two column numbers separated by a hyphen. A range of ***–n*** means columns **1** through ***n; n–*** means columns ***n*** through the end of the line.

––delimiter=*dchar*
–d *dchar*
Specifies ***dchar*** as the input field delimiter. Also specifies ***dchar*** as the output field delimiter unless you use the **––output-delimiter** option. The default delimiter is a TAB character. Quote ***dchar*** as necessary to protect it from shell expansion.

––fields=*flist* **–f *flist***
Selects the fields specified in ***flist***. The value of ***flist*** is one or more comma-separated field numbers or field ranges. A range is specified by two field numbers separated by a hyphen. A range of ***–n*** means fields **1** through ***n; n–*** means fields ***n*** through the last field. The field delimiter is a TAB character unless you use the **–d** (**––delimiter**) option to change it.

––output-delimiter=*ochar*
Specifies ***ochar*** as the output field delimiter. The default delimiter is the TAB character. You can specify a different delimiter by using the **––delimiter** option. Quote ***ochar*** as necessary to protect it from shell expansion.

--only-delimited **–s** Copies only lines containing delimiters. Without this option, cut copies—but does not modify—lines that do not contain delimiters. Works only with the **–d** (**––delimiter**) option.

Notes

Although limited in functionality, cut is easy to learn and use and is a good choice when columns and fields can be selected without using pattern matching. Sometimes cut is used with paste (page 905).

Examples

For the next two examples, assume that an **ls –l** command produces the following output:

```
$ ls -l
total 2944
-rwxr-xr-x  1 zach pubs     259 02-01 00:12 countout
-rw-rw-r--  1 zach pubs    9453 02-04 23:17 headers
-rw-rw-r--  1 zach pubs 1474828 01-14 14:15 memo
-rw-rw-r--  1 zach pubs 1474828 01-14 14:33 memos_save
-rw-rw-r--  1 zach pubs    7134 02-04 23:18 tmp1
-rw-rw-r--  1 zach pubs    4770 02-04 23:26 tmp2
-rw-rw-r--  1 zach pubs   13580 11-07 08:01 typescript
```

The following command outputs the permissions of the files in the working directory. The cut utility with the **–c** option selects characters 2 through 10 from each input line. The characters in this range are written to standard output.

```
$ ls -l | cut -c2-10
otal 2944
rwxr-xr-x
rw-rw-r--
rw-rw-r--
rw-rw-r--
rw-rw-r--
rw-rw-r--
rw-rw-r--
```

The next command outputs the size and name of each file in the working directory. The **–f** option selects the fifth and ninth fields from the input lines. The **–d** option tells cut to use SPACEs, not TABs, as delimiters. The tr utility (page 987) with the **–s** option changes sequences of more than one SPACE character into a single SPACE; otherwise, cut counts the extra SPACE characters as separate fields.

```
$ ls -l | tr -s '  ' ' ' | cut -f5,9 -d' '

259 countout
9453 headers
1474828 memo
1474828 memos_save
7134 tmp1
4770 tmp2
13580 typescript
```

The last example displays a list of full names as stored in the fifth field of the **/etc/passwd** file. The **–d** option specifies that the colon character be used as the field delimiter. Although this example works under Mac OS X, **/etc/passwd** does not

contain information about most users; see "Open Directory" on page 1042 for more information.

```
$ cat /etc/passwd
root:x:0:0:Root:/:/bin/sh
sam:x:401:50:Sam the Great:/home/sam:/bin/zsh
max:x:402:50:Max Wild:/home/max:/bin/bash
zach:x:504:500:Zach Brill:/home/zach:/bin/tcsh
hls:x:505:500:Helen Simpson:/home/hls:/bin/bash

$ cut -d: -f5 /etc/passwd
Root
Sam the Great
Max Wild
Zach Brill
Helen Simpson
```

ditto OS X

Copies files and creates and unpacks archives

*ditto [**options**] **source-file destination-file***
*ditto [**options**] **source-file-list destination-directory***
*ditto **–c** [**options**] **source-directory destination-archive***
*ditto **–x** [**options**] **source-archive-list destination-directory***

The ditto utility copies files and their ownership, timestamps, and other attributes, including extended attributes (page 1044). It can copy to and from cpio and zip archive files, as well as copy ordinary files and directories. The ditto utility is available under OS X only. OS X

Arguments

The ***source-file*** is the pathname of the file that ditto is to make a copy of. The ***destination-file*** is the pathname that ditto assigns to the resulting copy of the file.

The ***source-file-list*** specifies one or more pathnames of files and directories that ditto makes copies of. The ***destination-directory*** is the pathname of the directory that ditto copies the files and directories into. When you specify a ***destination-directory***, ditto gives each of the copied files the same simple filename as its ***source-file***.

The ***source-directory*** is a single directory that ditto copies into the ***destination-archive***. The resulting archive holds copies of the contents of ***source-directory***, but not the directory itself.

The ***source-archive-list*** specifies one or more pathnames of archives that ditto extracts into ***destination-directory***.

Using a hyphen (**–**) in place of a filename or a directory name causes ditto to read from standard input or write to standard output instead of reading from or writing to that file or directory.

Options

You cannot use the **–c** and **–x** options together.

–c (**create archive**) Creates an archive file.

––help Displays a help message.

–k (**pkzip**) Uses the zip format, instead of the default cpio (page 758) format, to create or extract archives. For more information on zip, see the tip on page 65.

––norsrc (**no resource**) Ignores extended attributes. This option causes ditto to copy only data forks (the default behavior under Mac OS X 10.3 and earlier).

––rsrc (**resource**) Copies extended attributes, including resource forks (the default behavior under Mac OS X 10.4 and above). Also **–rsrc** and **–rsrcFork**.

–V (**very verbose**) Sends a line to standard error for each file, symbolic link, and device node copied by ditto.

–v (**verbose**) Sends a line to standard error for each directory copied by ditto.

–X (**exclude**) Prevents ditto from searching directories in filesystems other than the filesystems that hold the files it was explicitly told to copy.

–x (**extract archive**) Extracts files from an archive file.

–z (**compress**) Uses gzip (page 838) or gunzip to compress or decompress cpio archives.

Notes

The ditto utility does not copy the locked attribute flag (page 1046). The utility also does not copy ACLs.

By default ditto creates and reads *archives* (page 1055) in the cpio (page 758) format.

The ditto utility cannot list the contents of archive files; it can only create or extract files from archives. Use pax or cpio to list the contents of cpio archives, and use unzip with the **–l** option to list the contents of zip files.

Examples

The following examples show three ways to back up a user's home directory, including extended attributes (except as mentioned in "Notes"), while preserving timestamps and permissions. The first example copies Zach's home directory to the volume (filesystem) named **Backups**; the copy is a new directory named **zach.0228**:

```
$ ditto /Users/zach /Volumes/Backups/zach.0228
```

The next example copies Zach's home directory into a single cpio-format archive file on the volume named **Backups**:

```
$ ditto -c /Users/zach /Volumes/Backups/zach.0228.cpio
```

The next example copies Zach's home directory into a zip archive:

```
$ ditto -c -k /Users/zach /Volumes/Backups/zach.0228.zip
```

Each of the next three examples restores the corresponding backup archive into Zach's home directory, overwriting any files that are already there:

```
$ ditto /Volumes/Backups/zach.0228 /Users/zach
$ ditto -x /Volumes/Backups/zach.0228.cpio /Users/zach
$ ditto -x -k /Volumes/Backups/zach.0228.zip /Users/zach
```

The following example copies the **Scripts** directory to a directory named **ScriptsBackups** on the remote host **plum**. It uses an argument of a hyphen in place of ***source-directory*** locally to write to standard output and in place of ***destination-directory*** on the remote system to read from standard input:

```
$ ditto -c Scripts - | ssh plum ditto -x - ScriptsBackups
```

The final example copies the local startup disk (the root filesystem) to the volume named **Backups.root**. Because some of the files can be read only by **root**, the script must be run by a user with **root** privileges. The **–X** option keeps ditto from trying to copy other volumes (filesystems) that are mounted under **/**.

```
# ditto -X / /Volumes/Backups.root
```

tr

Replaces specified characters

*tr [**options**] **string1** [**string2**]*

The tr utility reads standard input and, for each input character, either maps it to an alternate character, deletes the character, or leaves the character as is. This utility reads from standard input and writes to standard output.

Arguments

The tr utility is typically used with two arguments, ***string1*** and ***string2***. The position of each character in the two strings is important: Each time tr finds a character from ***string1*** in its input, it replaces that character with the corresponding character from ***string2***.

With one argument, ***string1***, and the **–d** (**––delete**) option, tr deletes the characters specified in ***string1***. The option **–s** (**––squeeze-repeats**) replaces multiple sequential occurrences of characters in ***string1*** with single occurrences (for example, **abbc** becomes **abc**).

Ranges

A range of characters is similar in function to a character class within a regular expression (page 1013). GNU tr does not support ranges (character classes) enclosed within brackets. You can specify a range of characters by following the character that appears earlier in the collating sequence with a hyphen and the character that comes later in the collating sequence. For example, **1–6** expands to **123456**. Although the range A–Z expands as you would expect in ASCII, this approach does not work when you use the EBCDIC collating sequence, as these characters are not sequential in EBCDIC. See "Character Classes" for a solution to this issue.

Character Classes

A tr character class is not the same as the character class described elsewhere in this book. (GNU documentation uses the term *list operator* for what this book calls a *character class*.) You specify a character class as '*[:**class**:]*', where ***class*** is one of the character classes from Table VI-35. You must specify a character class in ***string1*** (and not ***string2***) unless you are performing case conversion (see the "Examples" section) or you use the **–d** and **–s** options together.

Table VI-35 Character classes

Class	Meaning
alnum	Letters and digits
alpha	Letters
blank	Whitespace

Table VI-35 Character classes (continued)

Class	Meaning
cntrl	CONTROL characters
digit	Digits
graph	Printable characters but not SPACES
lower	Lowercase letters
print	Printable characters including SPACES
punct	Punctuation characters
space	Horizontal or vertical whitespace
upper	Uppercase letters
xdigit	Hexadecimal digits

Options

Options preceded by a double hyphen (**––**) work under Linux only. Except as noted, options named with a single letter and preceded by a single hyphen work under Linux and OS X.

––complement **–c** Complements ***string1***, causing tr to match all characters *except* those in ***string1***.

––delete **–d** Deletes characters that match those specified in ***string1***. If you use this option with the **–s** (**––squeeze-repeats**) option, you must specify both ***string1*** and ***string2*** (see "Notes").

––help Summarizes how to use tr, including the special symbols you can use in ***string1*** and ***string2***. LINUX

––squeeze-repeats **–s** Replaces multiple sequential occurrences of a character in ***string1*** with a single occurrence of the character when you call tr with only one string argument. If you use both ***string1*** and ***string2***, the tr utility first translates the characters in ***string1*** to those in ***string2***; it then replaces multiple sequential occurrences of a character in ***string2*** with a single occurrence of the character.

––truncate-set1 **–t** Truncates ***string1*** so it is the same length as ***string2*** before processing input. LINUX

Notes

When ***string1*** is longer than ***string2***, the initial portion of ***string1*** (equal in length to ***string2***) is used in the translation. When ***string1*** is shorter than ***string2***, tr repeats the last character of ***string1*** to extend ***string1*** to the length of ***string2***. In this case tr departs from the POSIX standard, which does not define a result.

If you use the **–d** (**––delete**) and **–s** (**––squeeze-repeats**) options at the same time, tr first deletes the characters in ***string1*** and then replaces multiple sequential occurrences of a character in ***string2*** with a single occurrence of the character.

Examples

You can use a hyphen to represent a range of characters in ***string1*** or ***string2***. The two command lines in the following example produce the same result:

```
$ echo abcdef | tr  'abcdef' 'xyzabc'
xyzabc
$ echo abcdef | tr  'a-f' 'x-za-c'
xyzabc
```

The next example demonstrates a popular method for disguising text, often called ROT13 (rotate 13) because it replaces the first letter of the alphabet with the thirteenth, the second with the fourteenth, and so forth. The first line ends with a pipe symbol that implicitly continues the line (see the optional section on page 144) and causes bash to start the next line with a secondary prompt (page 311).

```
$ echo The punchline of the joke is ... |
> tr 'A-M N-Z a-m n-z' 'N-Z A-M n-z a-m'
Gur chapuyvar bs gur wbxr vf ...
```

To make the text intelligible again, reverse the order of the arguments to tr:

```
$ echo Gur chapuyvar bs gur wbxr vf ... |
> tr 'N-Z A-M n-z a-m' 'A-M N-Z a-m n-z'
The punchline of the joke is ...
```

The **--delete** option causes tr to delete selected characters:

```
$ echo If you can read this, you can spot the missing vowels! |
> tr --delete 'aeiou'
If y cn rd ths, y cn spt th mssng vwls!
```

In the following example, tr replaces characters and reduces pairs of identical characters to single characters:

```
$ echo tennessee | tr -s 'tnse' 'srne'
serene
```

The next example replaces each sequence of nonalphabetic characters (the complement of all the alphabetic characters as specified by the character class **alpha**) in the file **draft1** with a single NEWLINE character. The output is a list of words, one per line.

```
$ tr -c -s '[:alpha:]' '\n' < draft1
```

The next example uses character classes to upshift the string **hi there**:

```
$ echo hi there | tr '[:lower:]' '[:upper:]'
HI THERE
```