



Quick answers to common problems

Linux Shell Scripting Cookbook

Second Edition

Over 110 practical recipes to solve real-world shell problems, guaranteed to make you wonder how you ever lived without them

Shantanu Tushar
Sarath Lakshman

[PACKT] open source*
PUBLISHING community experience distilled

Linux Shell Scripting Cookbook

Second Edition

Over 110 practical recipes to solve real-world shell problems, guaranteed to make you wonder how you ever lived without them

Shantanu Tushar

Sarath Lakshman

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Linux Shell Scripting Cookbook

Second Edition

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2011

Second edition: May 2013

Production Reference: 1140513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-274-2

www.packtpub.com

Cover Image by Parag Kadam (paragvkadam@gmail.com)

Credits

Authors

Shantanu Tushar
Sarath Lakshman

Reviewers

Rajeshwari K.
John C. Kennedy
Anil Kumar
Sudhendu Kumar
Aravind SV

Acquisition Editor

Kartikey Pandey

Lead Technical Editor

Ankita Shashi

Technical Editors

Jalasha D'costa
Amit Ramadas
Lubna Shaikh

Project Coordinator

Shiksha Chaturvedi

Proofreader

Linda Morris

Indexer

Hemangini Bari

Production Coordinator

Shantanu Zagade

Cover Work

Shantanu Zagade

About the Authors

Shantanu Tushar is an advanced GNU/Linux user since his college days. He works as an application developer and contributes to the software in the KDE projects.

Shantanu has been fascinated by computers since he was a child, and spent most of his high school time writing C code to perform daily activities. Since he started using GNU/Linux, he has been using shell scripts to make the computer do all the hard work for him. He also takes time to visit students at various colleges to introduce them to the power of Free Software, including its various tools. Shantanu is a well-known contributor in the KDE community and works on Calligra, Gluon and the Plasma subprojects. He looks after maintaining Calligra Active – KDE's office document viewer for tablets, Plasma Media Center, and the Gluon Player. One day, he believes, programming will be so easy that everybody will love to write programs for their computers.

Shantanu can be reached by e-mail on shantanu@kde.org, shantanutushar on [identi.ca](https://www.instagram.com/shantanutushar)/twitter, or his website <http://www.shantanutushar.com>.

I would like to thank my friends and family for the support and encouragement they've given me, especially to my sweet sister for her patience when I couldn't get time to talk to her. I am particularly thankful to Sinny Kumari for patiently testing the scripts to make sure they function properly and Sudhendu Kumar for helping me with the recipe on GNU Screen.

I must also thank Krishna, Madhusudan, and Santosh who introduced me to the wonderful world of GNU/Linux and Free Software. Also, a big thanks to all the reviewers of the book for taking the time to painfully go through every minute detail in the book and help me in improving it. I am also thankful to the whole team at Packt Publishing, without whose efforts and experience, this second edition wouldn't have happened.

Sarath Lakshman is a 23 year old who was bitten by the Linux bug during his teenage years. He is a software engineer working in ZCloud engineering group at Zynga, India. He is a life hacker who loves to explore innovations. He is a GNU/Linux enthusiast and hactivist of free and open source software. He spends most of his time hacking with computers and having fun with his great friends. Sarath is well known as the developer of SLYINUX (2005)—a user friendly GNU/Linux distribution for Linux newbies. The free and open source software projects he has contributed to are PiTiVi Video editor, SLYINUX GNU/Linux distro, Swathantra Malayalam Computing, School-Admin, Istanbul, and the Pardus Project. He has authored many articles for the *Linux For You* magazine on various domains of FOSS technologies. He had made a contribution to several different open source projects during his multiple Google Summer of Code projects. Currently, he is exploring his passion about scalable distributed systems in his spare time. Sarath can be reached via his website <http://www.sarathlakshman.com>.

About the Reviewers

Rajeshwari K. received her B.E degree (Information Science and Engineering) from VTU in 2004 and M. Tech degree (Computer Science and Engineering) from VTU in 2009. From 2004 to 2007 she handled a set of real-time projects and did some freelancing. Since 2010 she has been working as Assistant Professor at BMS College of Engineering in the department of Information Science and Engineering. She has a total of five years' experience in teaching in Computer Science subjects.

BMS College of Engineering, Bangalore is one of the autonomous colleges running under VTU with high acclamation nationwide.

Her research interests include operating systems and system-side programming.

John C. Kennedy has been administering Unix and Linux servers and workstations since 1997. He has experience with Red Hat, SUSE, Ubuntu, Debian, Solaris, and HP-UX. John is also experienced in Bash shell scripting and is currently teaching himself Python and Ruby. John has also been a Technical Editor for various publishers for over 10 years specializing in books related to open source technologies.

When John is not geeking out in front of either a home or work computer, he helps out with a German Shepherd Rescue in Virginia by fostering some great dogs or helping with their IT needs.

I would like to thank my family (my wonderful wife, Michele, my intelligent and caring daughter Denise, and my terrific and smart son, Kieran) for supporting the (sometimes) silly things and not so silly things I do. I'd also like to thank my current foster dogs for their occasional need to keep their legs crossed a little longer while I test things out from the book and forget they are there.

Anil Kumar is a software developer. He received his Computer Science undergraduate degree from BITS Pilani. He has work experience of more than two years in the field of Web Development and Systems. Besides working as a software developer, Anil is an open source evangelist and a blogger. He currently resides in Bangalore. He can be contacted at anil.18june@gmail.com.

Sudhendu Kumar has been a GNU/Linux user for more than five years. Presently being a software developer for a networking giant, in free time, he also contributes to KDE.

I would like to thank the publishers for giving me this opportunity to review the book. I hope readers find the book useful and they enjoy reading it.

Aravind SV has worked with various Unix-like systems and shells over many years. You can contact him at aravind.sv+shellbook@gmail.com.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

*Dedicated to my parents who taught me how to think and reason, and to be
optimistic in every situation in life*

—Shantanu Tushar

Table of Contents

Preface	1
Chapter 1: Shell Something Out	7
Introduction	8
Printing in the terminal	10
Playing with variables and environment variables	13
Function to prepend to environment variables	17
Math with the shell	19
Playing with file descriptors and redirection	21
Arrays and associative arrays	27
Visiting aliases	29
Grabbing information about the terminal	31
Getting and setting dates and delays	32
Debugging the script	36
Functions and arguments	37
Reading the output of a sequence of commands	40
Reading n characters without pressing the return key	43
Running a command until it succeeds	44
Field separators and iterators	45
Comparisons and tests	48
Chapter 2: Have a Good Command	53
Introduction	53
Concatenating with cat	54
Recording and playing back of terminal sessions	57
Finding files and file listing	58
Playing with xargs	68
Translating with tr	73
Checksum and verification	77
Cryptographic tools and hashes	80

Sorting unique and duplicates	83
Temporary file naming and random numbers	89
Splitting files and data	90
Slicing filenames based on extension	92
Renaming and moving files in bulk	95
Spell checking and dictionary manipulation	97
Automating interactive input	99
Making commands quicker by running parallel processes	102
Chapter 3: File In, File Out	105
Introduction	106
Generating files of any size	106
The intersection and set difference (A-B) on text files	107
Finding and deleting duplicate files	110
Working with file permissions, ownership, and the sticky bit	113
Making files immutable	118
Generating blank files in bulk	119
Finding symbolic links and their targets	120
Enumerating file type statistics	121
Using loopback files	124
Creating ISO files and hybrid ISO	127
Finding the difference between files, patching	130
Using head and tail for printing the last or first 10 lines	132
Listing only directories – alternative methods	135
Fast command-line navigation using pushd and popd	136
Counting the number of lines, words, and characters in a file	138
Printing the directory tree	139
Chapter 4: Texting and Driving	143
Introduction	143
Using regular expressions	144
Searching and mining a text inside a file with grep	147
Cutting a file column-wise with cut	154
Using sed to perform text replacement	158
Using awk for advanced text processing	162
Finding the frequency of words used in a given file	168
Compressing or decompressing JavaScript	170
Merging multiple files as columns	173
Printing the nth word or column in a file or line	174
Printing text between line numbers or patterns	175
Printing lines in the reverse order	176
Parsing e-mail addresses and URLs from a text	177

Removing a sentence in a file containing a word	178
Replacing a pattern with text in all the files in a directory	180
Text slicing and parameter operations	181
Chapter 5: Tangled Web? Not At All!	183
Introduction	184
Downloading from a web page	184
Downloading a web page as plain text	187
A primer on cURL	188
Accessing Gmail e-mails from the command line	192
Parsing data from a website	194
Image crawler and downloader	195
Web photo album generator	198
Twitter command-line client	201
Creating a "define" utility by using the Web backend	206
Finding broken links in a website	209
Tracking changes to a website	211
Posting to a web page and reading the response	214
Chapter 6: The Backup Plan	217
Introduction	217
Archiving with tar	218
Archiving with cpio	224
Compressing data with gzip	226
Archiving and compressing with zip	230
Faster archiving with pbzip2	231
Creating filesystems with compression	232
Backup snapshots with rsync	234
Version control based backup with Git	237
Creating entire disk images with fsarchiver	240
Chapter 7: The Old-boy Network	243
Introduction	243
Setting up the network	244
Let us ping!	250
Listing all the machines alive on a network	254
Running commands on a remote host with SSH	257
Transferring files through the network	261
Connecting to a wireless network	265
Password-less auto-login with SSH	267
Port forwarding using SSH	269
Mounting a remote drive at a local mount point	270
Network traffic and port analysis	271

Creating arbitrary sockets	274
Sharing an Internet connection	275
Basic firewall using iptables	276
Chapter 8: Put on the Monitor's Cap	279
Introduction	279
Monitoring disk usage	280
Calculating the execution time for a command	285
Collecting information about logged in users, boot logs, and boot failures	288
Listing the top 10 CPU consuming processes in an hour	291
Monitoring command outputs with watch	293
Logging access to files and directories	294
Logfile management with logrotate	296
Logging with syslog	297
Monitoring user logins to find intruders	299
Remote disk usage health monitor	303
Finding out active user hours on a system	305
Measuring and optimizing power usage	308
Monitoring disk activity	309
Checking disks and filesystems for errors	310
Chapter 9: Administration Calls	313
Introduction	313
Gathering information about processes	314
Killing processes and send or respond to signals	324
Sending messages to user terminals	327
Gathering system information	329
Using /proc for gathering information	330
Scheduling with cron	331
Writing and reading the MySQL database from Bash	335
User administration script	340
Bulk image resizing and format conversion	344
Taking screenshots from the terminal	347
Managing multiple terminals from one	348
Index	351

Preface

GNU/Linux is one of the most powerful and flexible operating systems in the world. In modern computing, there is absolutely no space where it is not used—from servers, portable computers, mobile phones, tablets to supercomputers, everything runs Linux. While there are beautiful and modern graphical interfaces available for it, the shell still remains the most flexible way of interacting with the system.

In addition to executing individual commands, a shell can follow commands from a script, which makes it very easy to automate tasks. Examples of such tasks are preparing reports, sending e-mails, performing maintenance, and so on. This book is a collection of chapters which contain recipes to demonstrate real-life usages of commands and shell scripts. You can use these as a reference, or an inspiration for writing your own scripts. The tasks will range from text manipulation to performing network operations to administrative tasks.

As with everything, the shell is only as awesome as you make it. When you become an expert at shell scripting, you can use the shell to the fullest and harness its true power. *Linux Shell Scripting Cookbook* shows you how to do exactly that!

What this book covers

Chapter 1, Shell Something Out, is an introductory chapter for understanding the basic concepts and features in Bash. We discuss printing text in the terminal, doing mathematical calculations, and other simple functionalities provided by Bash.

Chapter 2, Have a Good Command, shows commonly used commands that are available with GNU/Linux. This chapter travels through different practical usage examples that users may come across and that they could make use of. In addition to essential commands, this second edition talks about cryptographic hashing commands and a recipe to run commands in parallel, wherever possible.

Chapter 3, File In, File Out, contains a collection of recipes related to files and filesystems. This chapter explains how to generate large-size files, installing a filesystem on files, mounting files, and creating ISO images. We also deal with operations such as finding and removing duplicate files, counting lines in a file collecting details about files, and so on.

Chapter 4, Texting and Driving, has a collection of recipes that explains most of the command-line text processing tools well under GNU/Linux with a number of task examples. It also has supplementary recipes for giving a detailed overview of regular expressions and commands such as `sed` and `awk`. This chapter goes through solutions to most of the frequently used text processing tasks in a variety of recipes. It is an essential read for any serious task.

Chapter 5, Tangled Web? Not At All!, has a collection of shell-scripting recipes that talk to services on the Internet. This chapter is intended to help readers understand how to interact with the Web using shell scripts to automate tasks such as collecting and parsing data from web pages. This is discussed using POST and GET to web pages, writing clients to web services. The second edition uses new authorization mechanisms such as OAuth for services such as Twitter.

Chapter 6, The Backup Plan, shows several commands used for performing data back up, archiving, compression, and so on. In addition to faster compression techniques, this second edition also talks about creating entire disk images.

Chapter 7, The Old-boy Network, has a collection of recipes that talks about networking on Linux and several commands useful for writing network-based scripts. The chapter starts with an introductory basic networking primer and goes on to cover usages of `ssh` – one of the most powerful commands on any modern GNU/Linux system. We discuss advanced port forwarding, setting up raw communication channels, configuring the firewall, and much more.

Chapter 8, Put on the Monitor's Cap, walks through several recipes related to monitoring activities on the Linux system and tasks used for logging and reporting. The chapter explains tasks such as calculating disk usage, monitoring user access, and CPU usage. In this second edition, we also learn how to optimize power consumption, monitor disks, and check their filesystems for errors.

Chapter 9, Administration Calls, has a collection of recipes for system administration. This chapter explains different commands to collect details about the system and user management using scripting. We also discuss bulk image resizing and accessing MySQL databases from the shell. New in this edition is that we learn how to use the GNU Screen to manage multiple terminals without needing a window manager.

What you need for this book

Basic user experience with any GNU/Linux platform will help you easily follow the book. We have tried to keep all the recipes in the book precise and as simple to follow as possible. Your curiosity for learning with the Linux platform is the only prerequisite for the book. Step-by-step explanations are provided for solving the scripting problems explained in the book. In order to run and test the examples in the book, a Ubuntu/Debian Linux installation is recommended, however, any other Linux distribution is enough for most of the tasks. You will find the book to be a straightforward reference to essential shell-scripting tasks, as well as a learning aid to code real-world efficient scripts.

Who this book is for

If you are a beginner, or an intermediate user, who wants to master the skill of quickly writing scripts to perform various tasks without reading entire man pages, this book is for you. You can start writing scripts and one-liners by simply looking at a similar recipe and its descriptions without any working knowledge of shell scripting or Linux. Intermediate or advanced users, as well as system administrators or developers and programmers, can use this book as a reference when they face problems while coding.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We create a function called `repeat` that has an infinite `while` loop, which attempts to run the command passed as a parameter (accessed by `$@`) to the function."

A block of code is set as follows:

```
if [ $var -eq 0 ]; then echo "True"; fi
can be written as
if test $var -eq 0 ; then echo "True"; fi
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
while read line;
do something
done < filename
```

Any command-line input or output is written as follows:

```
# mkdir /mnt/loopback
# mount -o loop loopbackfile.img /mnt/loopback
```

New terms and **important words** are shown in bold.



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Shell Something Out

In this chapter, we will cover:

- ▶ Printing in the terminal
- ▶ Playing with variables and environment variables
- ▶ Function to prepend to environment variables
- ▶ Math with the shell
- ▶ Playing with file descriptors and redirection
- ▶ Arrays and associative array
- ▶ Visiting aliases
- ▶ Grabbing information about the terminal
- ▶ Getting and setting dates and delays
- ▶ Debugging the script
- ▶ Functions and arguments
- ▶ Reading output of a sequence of commands in a variable
- ▶ Reading n characters without pressing the return key
- ▶ Running a command until it succeeds
- ▶ Field separators and iterators
- ▶ Comparisons and tests

Introduction

Unix-like systems are amazing operating system designs. Even after many decades, Unix-style architecture for operating systems serves as one of the best designs. One of the important features of this architecture is the command-line interface, or the shell. The shell environment helps users to interact with and access core functions of the operating system. The term **scripting** is more relevant in this context. Scripting is usually supported by interpreter-based programming languages. Shell scripts are files in which we write a sequence of commands that we need to perform and are executed using the shell utility.

In this book we are dealing with **Bash (Bourne Again Shell)**, which is the default shell environment for most GNU/Linux systems. Since GNU/Linux is the most prominent operating system on Unix-style architecture, most of the examples and discussions are written by keeping Linux systems in mind.

The primary purpose of this chapter is to give readers an insight into the shell environment and become familiar with the basic features that the shell offers. Commands are typed and executed in a shell terminal. When a terminal is opened, a prompt is available which usually has the following format:

```
username@hostname$
```

Or:

```
root@hostname #
```

or simply as \$ or #.

\$ represents regular users and # represents the administrative user root. Root is the most privileged user in a Linux system.



It is usually a bad idea to directly use the shell as the root user (administrator) to perform tasks. This is because typing errors in your commands have the potential to do more damage when your shell has more privileges. So, it is recommended to log in as a regular user (your shell will denote that as \$ in the prompt, and # when running as root), and then use tools such as `sudo` to run privileged commands. Running a command such as `sudo <command> <arguments>` will run it as root.

A shell script is a text file that typically begins with a shebang, as follows:

```
#!/bin/bash
```

Shebang is a line on which `#!` is prefixed to the interpreter path. `/bin/bash` is the interpreter command path for Bash.

Execution of a script can be done in two ways. Either we can run the script as a command-line argument to `bash` or we can grant execution permission to the script so it becomes executable.

The script can be run with the filename as a command-line argument as follows (the text that starts with `#` is a comment, you don't have to type it out):

```
$ bash script.sh # Assuming script is in the current directory.
```

Or:

```
$ bash /home/path/script.sh # Using full path of script.sh.
```

If a script is run as a command-line argument for `bash`, the shebang in the script is not required.

If required, we can utilize the shebang to facilitate running the script on its own. For this, we have to set executable permissions for the script and it will run using the interpreter path that is appended to `#!` to the shebang. This can be set as follows:

```
$ chmod a+x script.sh
```

This command gives the `script.sh` file the executable permission for all users. The script can be executed as:

```
$ ./script.sh #./ represents the current directory
```

Or:

```
$ /home/path/script.sh # Full path of the script is used
```

The kernel will read the first line and see that the shebang is `#!/bin/bash`. It will identify `/bin/bash` and execute the script internally as:

```
$ /bin/bash script.sh
```

When a shell is started, it initially executes a set of commands to define various settings such as prompt text, colors, and much more. This set of commands are read from a shell script at `~/.bashrc` (or `~/.bash_profile` for login shells) located in the home directory of the user. The Bash shell also maintains a history of commands run by the user. It is available in the `~/.bash_history` file.



~ denotes your home directory, which is usually `/home/user` where `user` is your username or `/root` for the root user.

A login shell is the shell which you get just after logging in to a machine. However, if you open up a shell while logged in to a graphical environment (such as GNOME, KDE, and so on), then it is not a login shell.

In Bash, each command or command sequence is delimited by using a semicolon or a new line. For example:

```
$ cmd1 ; cmd2
```

This is equivalent to:

```
$ cmd1
```

```
$ cmd2
```

Finally, the # character is used to denote the beginning of unprocessed comments. A comment section starts with # and proceeds up to the end of that line. The comment lines are most often used to provide comments about the code in the file or to stop a line of code from being executed.

Now let us move on to the basic recipes in this chapter.

Printing in the terminal

The **terminal** is an interactive utility by which a user interacts with the shell environment. Printing text in the terminal is a basic task that most shell scripts and utilities need to perform regularly. As we will see in this recipe, this can be performed via various methods and in different formats.

How to do it...

`echo` is the basic command for printing in the terminal.

`echo` puts a newline at the end of every `echo` invocation by default:

```
$ echo "Welcome to Bash"
```

```
Welcome to Bash
```

Simply, using double-quoted text with the `echo` command prints the text in the terminal. Similarly, text without double quotes also gives the same output:

```
$ echo Welcome to Bash
```

```
Welcome to Bash
```

Another way to do the same task is by using single quotes:

```
$ echo 'text in quotes'
```

These methods may look similar, but some of them have a specific purpose and side effects too. Consider the following command:

```
$ echo "cannot include exclamation - ! within double quotes"
```

This will return the following output:

```
bash: !: event not found error
```

Hence, if you want to print special characters such as `!`, either do not use them within double quotes or escape them with a special escape character (`\`) prefixed with it, like so:

```
$ echo Hello world !
```

Or:

```
$ echo 'Hello world !'
```

Or:

```
$ echo "Hello world \!" #Escape character \ prefixed.
```

The side effects of each of the methods are as follows:

- ▶ When using `echo` without quotes, we cannot use a semicolon, as it acts as a delimiter between commands in the Bash shell
- ▶ `echo hello; hello` takes `echo hello` as one command and the second `hello` as the second command
- ▶ Variable substitution, which is discussed in the next recipe, will not work within single quotes

Another command for printing in the terminal is `printf`. It uses the same arguments as the `printf` command in the C programming language. For example:

```
$ printf "Hello world"
```

`printf` takes quoted text or arguments delimited by spaces. We can use formatted strings with `printf`. We can specify string width, left or right alignment, and so on. By default, `printf` does not have newline as in the `echo` command. We have to specify a newline when required, as shown in the following script:

```
#!/bin/bash
#Filename: printf.sh

printf "%-5s %-10s %-4s\n" No Name Mark
printf "%-5s %-10s %-4.2f\n" 1 Sarath 80.3456
printf "%-5s %-10s %-4.2f\n" 2 James 90.9989
printf "%-5s %-10s %-4.2f\n" 3 Jeff 77.564
```

We will receive the formatted output:

No	Name	Mark
1	Sarath	80.35
2	James	91.00
3	Jeff	77.56

How it works...

`%s`, `%c`, `%d`, and `%f` are format substitution characters for which an argument can be placed after the quoted format string.

`%-5s` can be described as a string substitution with left alignment (`-` represents left alignment) with width equal to 5. If `-` was not specified, the string would have been aligned to the right. The width specifies the number of characters reserved for that variable. For `Name`, the width reserved is 10. Hence, any name will reside within the 10-character width reserved for it and the rest of the characters will be filled with space up to 10 characters in total.

For floating point numbers, we can pass additional parameters to round off the decimal places.

For marks, we have formatted the string as `%-4.2f`, where `.2` specifies rounding off to two decimal places. Note that for every line of the format string a newline (`\n`) is issued.

There's more...

While using flags for `echo` and `printf`, always make sure that the flags appear before any strings in the command, otherwise Bash will consider the flags as another string.

Escaping newline in echo

By default, `echo` has a newline appended at the end of its output text. This can be avoided by using the `-n` flag. `echo` can also accept escape sequences in double-quoted strings as an argument. When using escape sequences, use `echo` as `echo -e "string containing escape sequences"`. For example:

```
echo -e "1\t2\t3"
```

```
1 2 3
```

Printing a colored output

Producing a colored output on the terminal is very interesting and is achieved by using escape sequences.

Colors are represented by color codes, some examples being, reset = 0, black = 30, red = 31, green = 32, yellow = 33, blue = 34, magenta = 35, cyan = 36, and white = 37.

To print a colored text, enter the following command:

```
echo -e "\e[1;31m This is red text \e[0m"
```

Here, `\e[1;31m` is the escape string that sets the color to red and `\e[0m` resets the color back. Replace 31 with the required color code.

For a colored background, reset = 0, black = 40, red = 41, green = 42, yellow = 43, blue = 44, magenta = 45, cyan = 46, and white=47, are the color codes that are commonly used.

To print a colored background, enter the following command:

```
echo -e "\e[1;42m Green Background \e[0m"
```

Playing with variables and environment variables

Variables are essential components of every programming language and are used to hold varying data. Scripting languages usually do not require variable type declaration before its use as they can be assigned directly. In Bash, the value for every variable is string, regardless of whether we assign variables with quotes or without quotes. Furthermore, there are variables used by the shell environment and the operating environment to store special values, which are called **environment variables**. Let us look at how to play with some of these variables in this recipe.

Getting ready

Variables are named with the usual naming constructs. When an application is executing, it will be passed a set of variables called environment variables. To view all the environment variables related to a terminal, issue the `env` command. For every process, environment variables in its runtime can be viewed by:

```
cat /proc/$PID/environ
```

Set `PID` with a process ID of the process (`PID` always takes an integer value).

For example, assume that an application called **gedit** is running. We can obtain the process ID of `gedit` with the `pgrep` command as follows:

```
$ pgrep gedit
12501
```

You can obtain the environment variables associated with the process by executing the following command:

```
$ cat /proc/12501/environ
GDM_KEYBOARD_LAYOUT=usGNOME_KEYRING_PID=1560USER=slynuxHOME=/home/slynux
```



Note that many environment variables are stripped off for convenience.
The actual output may contain numerous variables.

The aforementioned command returns a list of environment variables and their values. Each variable is represented as a name=value pair and are separated by a null character (`\0`). If you can substitute the `\0` character with `\n`, you can reformat the output to show each variable=value pair in each line. Substitution can be made using the `tr` command as follows:

```
$ cat /proc/12501/environ | tr '\0' '\n'
```

Now, let us see how to assign and manipulate variables and environment variables.

How to do it...

A variable can be assigned as follows:

```
var=value
```

`var` is the name of a variable and `value` is the value to be assigned. If `value` does not contain any space character (such as space), it need not be enclosed in quotes, Otherwise it is to be enclosed in single or double quotes.

Note that `var = value` and `var=value` are different. It is a usual mistake to write `var =value` instead of `var=value`. The later one is the assignment operation, whereas the earlier one is an equality operation.

Printing contents of a variable is done using by prefixing `$` with the variable name as follows:

```
var="value" #Assignment of value to variable var.
```

```
echo $var
```

Or:

```
echo ${var}
```

We will receive an output as follows:

```
value
```

We can use variable values inside `printf` or `echo` in double quotes:

```
#!/bin/bash
#Filename :variables.sh
fruit=apple
count=5
echo "We have $count ${fruit}(s)"
```

The output will be as follows:

```
We have 5 apple(s)
```

Environment variables are variables that are not defined in the current process, but are received from the parent processes. For example, `HTTP_PROXY` is an environment variable. This variable defines which proxy server should be used for an Internet connection.

Usually, it is set as:

```
HTTP_PROXY=192.168.1.23:3128
export HTTP_PROXY
```

The `export` command is used to set the `env` variable. Now any application, executed from the current shell script, will receive this variable. We can export custom variables for our own purposes in an application or shell script that is executed. There are many standard environment variables that are available for the shell by default.

For example, `PATH`. A typical `PATH` variable will contain:

```
$ echo $PATH
/home/slynux/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games
```

When given a command for execution, the shell automatically searches for the executable in the list of directories in the `PATH` environment variable (directory paths are delimited by the `:"` character). Usually, `$PATH` is defined in `/etc/environment` or `/etc/profile` or `~/.bashrc`. When we need to add a new path to the `PATH` environment, we use:

```
export PATH="$PATH:/home/user/bin"
```

Or, alternately, we can use:

```
$ PATH="$PATH:/home/user/bin"
$ export PATH
```

```
$ echo $PATH
/home/slynux/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
sbin:/bin:/usr/games:/home/user/bin
```


Here we have added `/home/user/bin` to `PATH`.

Some of the well-known environment variables are `HOME`, `PWD`, `USER`, `UID`, `SHELL`, and so on.



When using single quotes, variables will not be expanded and will be displayed as is. This means:

`$ echo '$var' will print $var`

Whereas, `$ echo "$var"` will print the value of the `$var` variable if defined or nothing at all if it is not defined.

There's more...

Let us see more tips associated with standard and environment variables.

Finding the length of a string

Get the length of a variable value using the following command:

```
length=${#var}
```

For example:

```
$ var=12345678901234567890$  
echo ${#var}  
20
```

The `length` parameter will bear the number of characters in the string.

Identifying the current shell

To identify the shell which is currently being used, we can use the `SHELL` variable, like so:

```
echo $SHELL
```

Or:

```
echo $0
```

For example:

```
$ echo $SHELL  
/bin/bash
```

```
$ echo $0  
/bin/bash
```

Checking for super user

UID is an important environment variable that can be used to check whether the current script has been run as a root user or regular user. For example:

```
If [ $UID -ne 0 ]; then
    echo Non root user. Please run as root.
else
    echo Root user
fi
```

The UID value for the root user is 0.

Modifying the Bash prompt string (username@hostname:~\$)

When we open a terminal or run a shell, we see a prompt string such as `user@hostname: /home/$`. Different GNU/Linux distributions have slightly different prompts and different colors. We can customize the prompt text using the `PS1` environment variable. The default prompt text for the shell is set using a line in the `~/ .bashrc` file.

- ▶ We can list the line used to set the `PS1` variable as follows:


```
$ cat ~/.bashrc | grep PS1
PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w\$ '
```
- ▶ To set a custom prompt string, enter the following command:


```
slynux@localhost: ~$ PS1="PROMPT>"
PROMPT> Type commands here # Prompt string changed.
```
- ▶ We can use colored text using the special escape sequences such as `\e[1;31` (refer to the *Printing in the terminal* recipe of this chapter).

There are also certain special characters that expand to system parameters. For example, `\u` expands to username, `\h` expands to hostname, and `\w` expands to the current working directory.

Function to prepend to environment variables

Environment variables are often used to store a list of paths of where to search for executables, libraries, and so on. Examples are `$PATH`, `$LD_LIBRARY_PATH`, which will typically look like this:

```
PATH=/usr/bin:/bin
LD_LIBRARY_PATH=/usr/lib:/lib
```

This essentially means that whenever the shell has to execute binaries, it will first look into `/usr/bin` followed by `/bin`.

A very common task that one has to do when building a program from source and installing to a custom path is to add its `bin` directory to the `PATH` environment variable. Let's say in this case we install `myapp` to `/opt/myapp`, which has binaries in a directory called `bin` and libraries in `lib`.

How to do it...

A way to do this is to say it as follows:

```
export PATH=/opt/myapp/bin:$PATH
export LD_LIBRARY_PATH=/opt/myapp/lib:$LD_LIBRARY_PATH
```

`PATH` and `LD_LIBRARY_PATH` should now look something like this:

```
PATH=/opt/myapp/bin:/usr/bin:/bin
LD_LIBRARY_PATH=/opt/myapp/lib:/usr/lib:/lib
```

However, we can make this easier by adding this function in `.bashrc`:

```
prepend() { [ -d "$2" ] && eval $1="\$2':'\$${1}" && export $1; }
```

This can be used in the following way:

```
prepend PATH /opt/myapp/bin
prepend LD_LIBRARY_PATH /opt/myapp/lib
```

How it works...

We define a function called `prepend()`, which first checks if the directory specified by the second parameter to the function exists. If it does, the `eval` expression sets the variable with the name in the first parameter equal to the second parameter string followed by `:` (the path separator) and then the original value for the variable.

However, there is one caveat, if the variable is empty when we try to prepend, there will be a trailing `:` at the end. To fix this, we can modify the function to look like this:

```
prepend() { [ -d "$2" ] && eval $1="\$2\${$1:+':'\$${1}}\" && export $1 ;
}
```



In this form of the function, we introduce a shell parameter expansion of the form:

`${parameter:+expression}`

This expands to `expression` if parameter is set and is not null.

With this change, we take care to try to append `:` and the old value if, and only if, the old value existed when trying to prepend.

Math with the shell

Arithmetic operations are an essential requirement for every programming language. In this recipe, we will explore various methods for performing arithmetic operations in shell.

Getting ready

The Bash shell environment can perform basic arithmetic operations using the commands `let`, `(())`, and `[]`. The two utilities `expr` and `bc` are also very helpful in performing advanced operations.

How to do it...

1. A numeric value can be assigned as a regular variable assignment, which is stored as a string. However, we use methods to manipulate as numbers:

```
#!/bin/bash
```

```
no1=4;
```

```
no2=5;
```

2. The `let` command can be used to perform basic operations directly. While using `let`, we use variable names without the `$` prefix, for example:

```
let result=no1+no2
```

```
echo $result
```

- ❑ Increment operation:

```
$ let no1++
```

- ❑ Decrement operation:

```
$ let no1--
```

- ❑ Shorthands:

```
let no+=6
```

```
let no-=6
```

These are equal to `let no=no+6` and `let no=no-6` respectively.

- Alternate methods:

The `[]` operator can be used in the same way as the `let` command as follows:

```
result=$(( no1 + no2 )
```

Using the `$` prefix inside `[]` operators are legal, for example:

```
result=$(( $no1 + 5 )
```

`(())` can also be used. `$` prefixed with a variable name is used when `(())` operator is used, as follows:

```
result=$(( no1 + 50 ))
```

`expr` can also be used for basic operations:

```
result=`expr 3 + 4`
```

```
result=$((expr $no1 + 5)
```

All of the preceding methods do not support floating point numbers, and operate on integers only.

3. `bc`, the precision calculator is an advanced utility for mathematical operations. It has a wide range of options. We can perform floating point operations and use advanced functions as follows:

```
echo "4 * 0.56" | bc
2.24
```

```
no=54;
result=`echo "$no * 1.5" | bc`
echo $result
81.0
```

Additional parameters can be passed to `bc` with prefixes to the operation with semicolon as delimiters through `stdin`.

- **Decimal places scale with `bc`:** In the following example the `scale=2` parameter sets the number of decimal places to 2. Hence, the output of `bc` will contain a number with two decimal places:

```
echo "scale=2;3/8" | bc
0.37
```

- ❑ **Base conversion with bc:** We can convert from one base number system to another one. Let us convert from decimal to binary, and binary to octal:

```
#!/bin/bash
```

```
Desc: Number conversion
```

```
no=100
```

```
echo "obase=2;$no" | bc
```

```
1100100
```

```
no=1100100
```

```
echo "obase=10;ibase=2;$no" | bc
```

```
100
```

- ❑ Calculating squares and square roots can be done as follows:

```
echo "sqrt(100)" | bc #Square root
```

```
echo "10^10" | bc #Square
```

Playing with file descriptors and redirection

File descriptors are integers that are associated with file input and output. They keep track of opened files. The best-known file descriptors are `stdin`, `stdout`, and `stderr`. We even can redirect the contents of one file descriptor to another. This recipe shows examples on how to manipulate and redirect with file descriptors.

Getting ready

While writing scripts we use standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`) frequently. Redirection of an output to a file by filtering the contents is one of the essential things we need to perform. While a command outputs some text, it can be either an error or an output (nonerror) message. We cannot distinguish whether it is output text or an error text by just looking at it. However, we can handle them with file descriptors. We can extract text that is attached to a specific descriptor.

File descriptors are integers associated with an opened file or data stream. File descriptors 0, 1, and 2 are reserved as follows:

- ▶ 0: `stdin` (standard input)
- ▶ 1: `stdout` (standard output)
- ▶ 2: `stderr` (standard error)

How to do it...

1. Redirecting or saving output text to a file can be done as follows:

```
$ echo "This is a sample text 1" > temp.txt
```

This would store the echoed text in `temp.txt` by truncating the file, the contents will be emptied before writing.

2. To append text to a file, consider the following example:

```
$ echo "This is sample text 2" >> temp.txt
```

3. You can view the contents of the file as follows:

```
$ cat temp.txt
```

```
This is sample text 1
```

```
This is sample text 2
```

4. Let us see what a standard error is and how you can redirect it. `stderr` messages are printed when commands output an error message. Consider the following example:

```
$ ls +
```

```
ls: cannot access +: No such file or directory
```

Here `+` is an invalid argument and hence an error is returned.



Successful and unsuccessful commands

When a command returns after an error, it returns a nonzero exit status. The command returns zero when it terminates after successful completion. The return status can be read from special variable `?` (run `echo $?` immediately after the command execution statement to print the exit status).

The following command prints the `stderr` text to the screen rather than to a file (and because there is no `stdout` output, `out.txt` will be empty):

```
$ ls + > out.txt
```

```
ls: cannot access +: No such file or directory
```

In the following command, we redirect `stderr` to `out.txt`:

```
$ ls + 2> out.txt # works
```

You can redirect `stderr` exclusively to a file and `stdout` to another file as follows:

```
$ cmd 2>stderr.txt 1>stdout.txt
```

It is also possible to redirect `stderr` and `stdout` to a single file by converting `stderr` to `stdout` using this preferred method:

```
$ cmd 2>&1 output.txt
```

Or the alternate approach:

```
$ cmd &> output.txt
```

5. Sometimes, the output may contain unnecessary information (such as debug messages). If you don't want the output terminal burdened with the `stderr` details then you should redirect the `stderr` output to `/dev/null`, which removes it completely. For example, consider that we have three files `a1`, `a2`, and `a3`. However, `a1` does not have the read-write-execute permission for the user. When you need to print the contents of files starting with `a`, we use the `cat` command. Set up the test files as follows:

```
$ echo a1 > a1
$ cp a1 a2 ; cp a2 a3;
$ chmod 000 a1 #Deny all permissions
```

While displaying contents of the files using wildcards (`a*`), it will show an error message for file `a1` as it does not have the proper read permission:

```
$ cat a*
cat: a1: Permission denied
a1
a1
```

Here, `cat: a1: Permission denied` belongs to the `stderr` data. We can redirect the `stderr` data into a file, whereas `stdout` remains printed in the terminal. Consider the following code:

```
$ cat a* 2> err.txt #stderr is redirected to err.txt
a1
a1
```

```
$ cat err.txt
cat: a1: Permission denied
```

Take a look at the following code:

```
$ cmd 2>/dev/null
```

When redirection is performed for `stderr` or `stdout`, the redirected text flows into a file. As the text has already been redirected and has gone into the file, no text remains to flow to the next command through pipe (`|`), and it appears to the next set of command sequences through `stdin`.

6. However, there is a way to redirect data to a file, as well as provide a copy of redirected data as `stdin` for the next set of commands. This can be done using the `tee` command. For example, to print `stdout` in the terminal as well as redirect `stdout` into a file, the syntax for `tee` is as follows:

```
command | tee FILE1 FILE2
```

In the following code, the `stdin` data is received by the `tee` command. It writes a copy of `stdout` to the `out.txt` file and sends another copy as `stdin` for the next command. The `cat -n` command puts a line number for each line received from `stdin` and writes it into `stdout`:

```
$ cat a* | tee out.txt | cat -n
cat: a1: Permission denied
    1a1
    2a1
```

Examine the contents of `out.txt` as follows:

```
$ cat out.txt
a1
a1
```

Note that `cat: a1: Permission denied` does not appear because it belongs to `stderr`. The `tee` command can read from `stdin` only.

By default, the `tee` command overwrites the file, but it can be used with appended options by providing the `-a` option, for example, `$ cat a* | tee -a out.txt | cat -n`.

Commands appear with arguments in the format: `command FILE1 FILE2 ...` or simply `command FILE`.

7. We can use `stdin` as a command argument. It can be done by using `-` as the filename argument for the command as follows:

```
$ cmd1 | cmd2 | cmd -
```

For example:

```
$ echo who is this | tee -
who is this
who is this
```

Alternately, we can use `/dev/stdin` as the output filename to use `stdin`.

Similarly, use `/dev/stderr` for standard error and `/dev/stdout` for standard output. These are special device files that correspond to `stdin`, `stderr`, and `stdout`.

How it works...

For output redirection, `>` and `>>` operators are different. Both of them redirect text to a file, but the first one empties the file and then writes to it, whereas the later one adds the output to the end of the existing file.

When we use a redirection operator, the output won't print in the terminal but it is directed to a file. When redirection operators are used, by default, they operate on standard output. To explicitly take a specific file descriptor, you must prefix the descriptor number to the operator.

`>` is equivalent to `1>` and similarly it applies for `>>` (equivalent to `1>>`).

When working with errors, the `stderr` output is dumped to the `/dev/null` file. `/dev/null` is a special device file where any data received by the file is discarded. The null device is often known as a **black hole** as all the data that goes into it is lost forever.

There's more...

A command that reads `stdin` for input can receive data in multiple ways. Also, it is possible to specify file descriptors of our own using `cat` and pipes, for example:

```
$ cat file | cmd
$ cmd1 | cmd
```

Redirection from a file to a command

By using redirection, we can read data from a file as `stdin` as follows:

```
$ cmd < file
```

Redirecting from a text block enclosed within a script

Sometimes we need to redirect a block of text (multiple lines of text) as standard input. Consider a particular case where the source text is placed within the shell script.

A practical usage example is writing a logfile header data. It can be performed as follows:

```
#!/bin/bash
cat<<EOF>log.txt
LOG FILE HEADER
This is a test log file
Function: System statistics
EOF
```

The lines that appear between `cat <<EOF >log.txt` and the next `EOF` line will appear as the `stdin` data. Print the contents of `log.txt` as follows:

```
$ cat log.txt
LOG FILE HEADER
This is a test log file
Function: System statistics
```

Custom file descriptors

A file descriptor is an abstract indicator for accessing a file. Each file access is associated with a special number called a file descriptor. 0, 1, and 2 are reserved descriptor numbers for `stdin`, `stdout`, and `stderr`.

We can create our own custom file descriptors using the `exec` command. If you are already familiar with file programming with any other programming language, you might have noticed modes for opening files. Usually, the following three modes are used:

- ▶ Read mode
- ▶ Write with truncate mode
- ▶ Write with append mode

`<` is an operator used to read from the file to `stdin`. `>` is the operator used to write to a file with truncation (data is written to the target file after truncating the contents). `>>` is an operator used to write to a file by appending (data is appended to the existing file contents and the contents of the target file will not be lost). File descriptors can be created with one of the three modes.

Create a file descriptor for reading a file, as follows:

```
$ exec 3<input.txt # open for reading with descriptor number 3
```

We could use it in the following way:

```
$ echo this is a test line > input.txt
$ exec 3<input.txt
```

Now you can use file descriptor 3 with commands. For example, we will use `cat <&3` as follows:

```
$ cat<&3
this is a test line
```

If a second read is required, we cannot re-use the file descriptor 3. It is required that we reassign the file descriptor 3 for read using `exec` for making a second read.

Create a file descriptor for writing (truncate mode) as follows:

```
$ exec 4>output.txt # open for writing
```

For example:

```
$ exec 4>output.txt
$ echo newline >&4
$ cat output.txt
newline
```

Create a file descriptor for writing (append mode) as follows:

```
$ exec 5>>input.txt
```

For example:

```
$ exec 5>>input.txt
$ echo appended line >&5
$ cat input.txt
newline
appended line
```

Arrays and associative arrays

Arrays are a very important component for storing a collection of data as separate entities using indexes. Regular arrays can use only integers as their array index. On the other hand, Bash also supports associative arrays that can take a string as their array index. Associative arrays are very useful in many types of manipulations where having a string index makes more sense. In this recipe, we will see how to use both of these.

Getting ready

To use associate arrays, you must have Bash Version 4 or higher.

How to do it...

1. An array can be defined in many ways. Define an array using a list of values in a line as follows:

```
array_var=(1 2 3 4 5 6)

#Values will be stored in consecutive locations starting from
index 0.
```

Alternately, define an array as a set of index-value pairs as follows:

```
array_var[0]="test1"
array_var[1]="test2"
array_var[2]="test3"
array_var[3]="test4"
array_var[4]="test5"
array_var[5]="test6"
```

2. Print the contents of an array at a given index using the following commands:

```
echo ${array_var[0]}
test1
index=5
echo ${array_var[$index]}
test6
```

3. Print all of the values in an array as a list using the following commands:

```
$ echo ${array_var[*]}
test1 test2 test3 test4 test5 test6
```

Alternately, you could use:

```
$ echo ${array_var[@]}
test1 test2 test3 test4 test5 test6
```

4. Print the length of an array (the number of elements in an array) as follows:

```
$ echo ${#array_var[*]}
6
```

There's more...

Associative arrays have been introduced to Bash from Version 4.0 and they are useful entities to solve many problems using the hashing technique. Let us go into more detail.

Defining associative arrays

In an associative array, we can use any text data as an array index. Initially, a declaration statement is required to declare a variable name as an associative array. This can be done as follows:

```
$ declare -A ass_array
```

After the declaration, elements can be added to the associative array using two methods as follows:

- ▶ By using inline index-value list method, we can provide a list of index-value pairs:


```
$ ass_array=( [index1]=val1 [index2]=val2 )
```
- ▶ Alternately, you could use separate index-value assignments:


```
$ ass_array[index1]=val1
$ ass_array[index2]=val2
```

For example, consider the assignment of price for fruits using an associative array:

```
$ declare -A fruits_value
$ fruits_value=( [apple]='100dollars' [orange]='150 dollars' )
```

Display the content of an array as follows:

```
$ echo "Apple costs ${fruits_value[apple]}"
Apple costs 100 dollars
```

Listing of array indexes

Arrays have indexes for indexing each of the elements. Ordinary and associative arrays differ in terms of index type. We can obtain the list of indexes in an array as follows:

```
$ echo ${!array_var[*]}
```

Or, we can also use:

```
$ echo ${!array_var[@]}
```

In the previous `fruits_value` array example, consider the following command:

```
$ echo ${!fruits_value[*]}
orange apple
```

This will work for ordinary arrays too.

Visiting aliases

An **alias** is basically a shortcut that takes the place of typing a long-command sequence. In this recipe, we will see how to create aliases using the `alias` command.

How to do it...

There are various operations you can perform on aliases, these are as follows:

1. An alias can be created as follows:

```
$ alias new_command='command sequence'
```

Giving a shortcut to the install command, `apt-get install`, can be done as follows:

```
$ alias install='sudo apt-get install'
```

Therefore, we can use `install pidgin` instead of `sudo apt-get install pidgin`.

2. The `alias` command is temporary; aliasing exists until we close the current terminal only. To keep these shortcuts permanent, add this statement to the `~/.bashrc` file. Commands in `~/.bashrc` are always executed when a new shell process is spawned:

```
$ echo 'alias cmd="command seq"' >> ~/.bashrc
```

3. To remove an alias, remove its entry from `~/.bashrc` (if any) or use the `unalias` command. Alternatively, `alias example=` should unset the alias named `example`.
4. As an example, we can create an alias for `rm` so that it will delete the original and keep a copy in a backup directory:

```
alias rm='cp $@ ~/backup && rm $@'
```



When you create an alias, if the item being aliased already exists, it will be replaced by this newly aliased command for that user.

There's more...

There are situations when aliasing can also be a security breach. See how to identify them.

Escaping aliases

The `alias` command can be used to alias any important command, and you may not always want to run the command using the alias. We can ignore any aliases currently defined by escaping the command we want to run. For example:

```
$ \command
```

The `\` character escapes the command, running it without any aliased changes. While running privileged commands on an untrusted environment, it is always good security practice to ignore aliases by prefixing the command with `\`. The attacker might have aliased the privileged command with his/her own custom command to steal the critical information that is provided by the user to the command.

Grabbing information about the terminal

While writing command-line shell scripts, we will often need to heavily manipulate information about the current terminal, such as the number of columns, rows, cursor positions, masked password fields, and so on. This recipe helps in collecting and manipulating terminal settings.

Getting ready

`tput` and `stty` are utilities that can be used for terminal manipulations. Let us see how to use them to perform different tasks.

How to do it...

There are specific information you can gather about the terminal as shown in the following list:

- ▶ Get the number of columns and rows in a terminal by using the following commands:
`tput cols`
`tput lines`
- ▶ To print the current terminal name, use the following command:
`tput longname`
- ▶ To move the cursor to a 100,100 position, you can enter:
`tput cup 100 100`
- ▶ Set the background color for the terminal using the following command:
`tputsetb n`
`n` can be a value in the range of 0 to 7.
- ▶ Set the foreground color for text by using the following command:
`tputsetf n`
`n` can be a value in the range of 0 to 7.
- ▶ To make text bold use this:
`tput bold`

- ▶ To start and end underlining use this:
`tput smul`
`tput rmul`
- ▶ To delete from the cursor to the end of the line use the following command:
`tput ed`
- ▶ While typing a password, we should not display the characters typed. In the following example, we will see how to do it using `stty`:

```
#!/bin/sh
#Filename: password.sh
echo -e "Enter password: "
stty -echo
read password
stty echo
echo
echo Password read.
```



The `-echo` option in the preceding command disables the output to the terminal, whereas `echo` enables output.

Getting and setting dates and delays

Many applications require printing dates in different formats, setting date and time, and performing manipulations based on date and time. Delays are commonly used to provide a wait time (such as 1 second) during the program execution. Scripting contexts, such as monitoring a task every 5 seconds, demands the understanding of writing delays in a program. This recipe will show you how to work with dates and time delays.

Getting ready

Dates can be printed in variety of formats. We can also set dates from the command line. In Unix-like systems, dates are stored as an integer, which denotes the number of seconds since 1970-01-01 00:00:00 UTC. This is called **epoch** or **Unix time**. Let us see how to read dates and set them.

How to do it...

It is possible to read the dates in different formats and also to set the date. This can be accomplished with these steps:

1. You can read the date as follows:

```
$ date
Thu May 20 23:09:04 IST 2010
```

2. The epoch time can be printed as follows:

```
$ date +%s
1290047248
```

We can find out epoch from a given formatted date string. You can use dates in multiple date formats as input. Usually, you don't need to bother about the date string format that you use if you are collecting the date from a system log or any standard application generated output. Convert the date string into epoch as follows:

```
$ date --date "Thu Nov 18 08:07:21 IST 2010" +%s
1290047841
```

The `--date` option is used to provide a date string as input. However, we can use any date formatting options to print the output. Feeding the input date from a string can be used to find out the weekday, given the date.

For example:

```
$ date --date "Jan 20 2001" +%A
Saturday
```

The date format strings are listed in the table mentioned in the *How it works...* section:

3. Use a combination of format strings prefixed with `+` as an argument for the `date` command to print the date in the format of your choice. For example:

```
$ date "+%d %B %Y"
20 May 2010
```

4. We can set the date and time as follows:

```
# date -s "Formatted date string"
```


For example:

```
# date -s "21 June 2009 11:01:22"
```

- Sometimes we need to check the time taken by a set of commands. We can display it using the following code:

```
#!/bin/bash
#Filename: time_take.sh
start=$(date +%s)
commands;
statements;

end=$(date +%s)
difference=$(( end - start))
echo Time taken to execute commands is $difference seconds.
```

 An alternate method would be to use `time <scriptpath>` to get the time that it took to execute the script.

How it works...

While considering dates and time, epoch is defined as the number of seconds that have elapsed since midnight proleptic **Coordinated Universal Time (UTC)** of January 1, 1970, not counting leap seconds. Epoch time is very useful when you need to calculate the difference between two dates or time. You may find out the epoch times for two given timestamps and take the difference between the epoch values. Therefore, you can find out the total number of seconds between two dates.

To write a date format to get the output as required, use the following table:

Date component	Format
Weekday	%a (for example, Sat) %A (for example, Saturday)
Month	%b (for example, Nov) %B (for example, November)
Day	%d (for example, 31)
Date in format (mm/dd/yy)	%D (for example, 10/18/10)
Year	%Y (for example, 10) %Y (for example, 2010)
Hour	%I or %H (For example, 08)
Minute	%M (for example, 33)
Second	%S (for example, 10)

Date component	Format
Nano second	%N (for example, 695208515)
Epoch Unix time in seconds	%s (for example, 1290049486)

There's more...

Producing time intervals is very essential when writing monitoring scripts that execute in a loop. Let us see how to generate time delays.

Producing delays in a script

To delay execution in a script for a particular period of time, use `sleep:$ sleepno_of_seconds`. For example, the following script counts from 0 to 40 by using `tput` and `sleep`:

```
#!/bin/bash
#Filename: sleep.sh
echo -n Count:
tput sc

count=0;
while true;
do
    if [ $count -lt 40 ];
    then
        let count++;
        sleep 1;
        tput rc
        tput ed
        echo -n $count;
    else exit 0;
    fi
done
```

In the preceding example, a variable `count` is initialized to 0 and is incremented on every loop execution. The `echo` statement prints the text. We use `tput sc` to store the cursor position. On every loop execution we write the new count in the terminal by restoring the cursor position for the number. The cursor position is restored using `tput rc`. This clears text from the current cursor position to the end of the line, so that the older number can be cleared and the count can be written. A delay of 1 second is provided in the loop by using the `sleep` command.

Debugging the script

Debugging is one of the critical features that every programming language should implement to produce race-back information when something unexpected happens. Debugging information can be used to read and understand what caused the program to crash or to act in an unexpected fashion. Bash provides certain debugging options that every sysadmin should know. This recipe shows how to use these.

How to do it...

We can either use Bash's inbuilt debugging tools or write our scripts in such a manner that they become easy to debug, here's how:

1. Add the `-x` option to enable debug tracing of a shell script as follows:

```
$ bash -x script.sh
```

Running the script with the `-x` flag will print each source line with the current status. Note that you can also use `sh -x script`.

2. Debug only portions of the script using `set -x` and `set +x`. For example:

```
#!/bin/bash
#Filename: debug.sh
for i in {1..6};
do
    set -x
    echo $i
    set +x
done
echo "Script executed"
```

In the preceding script, the debug information for `echo $i` will only be printed, as debugging is restricted to that section using `-x` and `+x`.

3. The aforementioned debugging methods are provided by Bash built-ins. But they always produce debugging information in a fixed format. In many cases, we need debugging information in our own format. We can set up such a debugging style by passing the `_DEBUG` environment variable.

Look at the following example code:

```
#!/bin/bash
function DEBUG()
{
    [ "$_DEBUG" == "on" ] && $@ || :
}
```

```
for i in {1..10}
do
    DEBUG echo $i
done
```

We can run the above script with debugging set to "on" as follows:

```
$ _DEBUG=on ./script.sh
```

We prefix `DEBUG` before every statement where debug information is to be printed. If `_DEBUG=on` is not passed to the script, debug information will not be printed. In Bash, the command `:` tells the shell to do nothing.

How it works...

The `-x` flag outputs every line of script as it is executed to `stdout`. However, we may require only some portions of the source lines to be observed such that commands and arguments are to be printed at certain portions. In such conditions we can use `set builtin` to enable and disable debug printing within the script.

- ▶ `set -x`: This displays arguments and commands upon their execution
- ▶ `set +x`: This disables debugging
- ▶ `set -v`: This displays input when they are read
- ▶ `set +v`: This disables printing input

There's more...

We can also use other convenient ways to debug scripts. We can make use of shebang in a trickier way to debug scripts.

Shebang hack

The shebang can be changed from `#!/bin/bash` to `#!/bin/bash -xv` to enable debugging without any additional flags (`-xv` flags themselves).

Functions and arguments

Like any other scripting languages, Bash also supports functions. Let us see how to define and use functions.

How to do it...

We can create functions to perform tasks and we can also create functions that take parameters (also called arguments) as you can see in the following steps:

1. A function can be defined as follows:

```
function fname()
{
    statements;
}
Or alternately,
fname()
{
    statements;
}
```

2. A function can be invoked just by using its name:

```
$ fname ; # executes function
```

3. Arguments can be passed to functions and can be accessed by our script:

```
fname arg1 arg2 ; # passing args
```

Following is the definition of the function `fname`. In the `fname` function, we have included various ways of accessing the function arguments.

```
fname()
{
    echo $1, $2; #Accessing arg1 and arg2
    echo "$@"; # Printing all arguments as list at once
    echo "$*"; # Similar to $@, but arguments taken as single entity
    return 0; # Return value
}
```

Similarly, arguments can be passed to scripts and can be accessed by `script:$0` (the name of the script):

- ❑ `$1` is the first argument
- ❑ `$2` is the second argument
- ❑ `$n` is the *n*th argument
- ❑ `"$@"` expands as `"$1" "$2" "$3"` and so on
- ❑ `"$*"` expands as `"$1c$2c$3"`, where *c* is the first character of IFS
- ❑ `"$@"` is used more often than `"$*"` since the former provides all arguments as a single string

There's more...

Let us explore through more tips on Bash functions.

The recursive function

Functions in Bash also support recursion (the function that can call itself). For example, `F()`
`{ echo $1; F hello; sleep 1; }.`



Fork bomb

We can write a recursive function, which is basically a function that calls itself:

```
:() { :|:& };;:
```

It infinitely spawns processes and ends up in a denial-of-service attack. `&` is postfixed with the function call to bring the subprocess into the background. This is a dangerous code as it forks processes and, therefore, it is called a fork bomb.

You may find it difficult to interpret the preceding code. See the Wikipedia page http://en.wikipedia.org/wiki/Fork_bomb for more details and interpretation of the fork bomb.

It can be prevented by restricting the maximum number of processes that can be spawned from the config file at `/etc/security/limits.conf`.

Exporting functions

A function can be exported—like environment variables—using `export`, such that the scope of the function can be extended to subprocesses, as follows:

```
export -f fname
```

Reading the return value (status) of a command

We can get the return value of a command or function in the following way:

```
cmd;
```

```
echo $?;
```

`$?` will give the return value of the command `cmd`.

The return value is called **exit status**. It can be used to analyze whether a command completed its execution successfully or unsuccessfully. If the command exits successfully, the exit status will be zero, otherwise it will be a nonzero value.

We can check whether a command terminated successfully or not by using the following script:

```
#!/bin/bash
#Filename: success_test.sh
CMD="command" #Substitute with command for which you need to test the
exit status
$CMD
if [ $? -eq 0 ];
then
    echo "$CMD executed successfully"
else
    echo "$CMD terminated unsuccessfully"
fi
```

Passing arguments to commands

Arguments to commands can be passed in different formats. Suppose `-p` and `-v` are the options available and `-k N` is another option that takes a number. Also, the command takes a filename as argument. It can be executed in multiple ways as shown:

- ▶ `$ command -p -v -k 1 file`
- ▶ `$ command -pv -k 1 file`
- ▶ `$ command -vpk 1 file`
- ▶ `$ command file -pvk 1`

Reading the output of a sequence of commands in a variable

One of the best-designed features of shell scripting is the ease of combining many commands or utilities to produce output. The output of one command can appear as the input of another, which passes its output to another command, and so on. The output of this combination can be read in a variable. This recipe illustrates how to combine multiple commands and how its output can be read.

Getting ready

Input is usually fed into a command through `stdin` or arguments. Output appears as `stderr` or `stdout`. While we combine multiple commands, we usually use `stdin` to give input and `stdout` to provide an output.

In this context, the commands are called **filters**. We connect each filter using pipes, the piping operator being `|`. An example is as follows:

```
$ cmd1 | cmd2 | cmd3
```

Here we combine three commands. The output of `cmd1` goes to `cmd2` and output of `cmd2` goes to `cmd3` and the final output (which comes out of `cmd3`) will be printed, or it can be directed to a file.

How to do it...

We typically use pipes and use them with the subshell method for combining outputs of multiple files. Here's how:

1. Let us start with combining two commands:

```
$ ls | cat -n > out.txt
```

Here the output of `ls` (the listing of the current directory) is passed to `cat -n`, which in turn puts line numbers to the input received through `stdin`. Therefore, its output is redirected to the `out.txt` file.

2. We can read the output of a sequence of commands combined by pipes as follows:

```
cmd_output=$(COMMANDS)
```

This is called **subshell method**. For example:

```
cmd_output=$(ls | cat -n)
echo $cmd_output
```

Another method, called **back quotes** (some people also refer to it as **back tick**) can also be used to store the command output as follows:

```
cmd_output=`COMMANDS`
```

For example:

```
cmd_output=`ls | cat -n`
echo $cmd_output
```

Back quote is different from the single-quote character. It is the character on the `~` button in the keyboard.

There's more...

There are multiple ways of grouping commands. Let us go through a few of them.

Spawning a separate process with subshell

Subshells are separate processes. A subshell can be defined using the `()` operators as follows:

```
pwd;
(cd /bin; ls);
pwd;
```

When some commands are executed in a subshell, none of the changes occur in the current shell; changes are restricted to the subshell. For example, when the current directory in a subshell is changed using the `cd` command, the directory change is not reflected in the main shell environment.

The `pwd` command prints the path of the working directory.

The `cd` command changes the current directory to the given directory path.

Subshell quoting to preserve spacing and the newline character

Suppose we are reading the output of a command to a variable using a subshell or the back quotes method. We always quote them in double quotes to preserve the spacing and newline character (`\n`). For example:

```
$ cat text.txt
1
2
3

$ out=$(cat text.txt)
$ echo $out
1 2 3 # Lost \n spacing in 1,2,3

$ out="$(cat tex.txt)"
$ echo$out
1
2
3
```

Reading *n* characters without pressing the return key

`read` is an important Bash command to read text from the keyboard or standard input. We can use `read` to interactively read an input from the user, but `read` is capable of much more. Most of the input libraries in any programming language read the input from the keyboard; but string input termination is done when *return* is pressed. There are certain critical situations when *return* cannot be pressed, but the termination is done based on a number of characters or a single character. For example, in a game, a ball is moved upward when `+` is pressed. Pressing `+` and then pressing *return* every time to acknowledge the `+` press is not efficient. In this recipe we will use the `read` command that provides a way to accomplish this task without having to press *return*.

How to do it...

You can use various options of the `read` command to obtain different results as shown in the following steps:

1. The following statement will read *n* characters from input into the `variable_name` variable:

```
read -n number_of_chars variable_name
```

For example:

```
$ read -n 2 var
$ echo $var
```

2. Read a password in the nonechoed mode as follows:

```
read -s var
```

3. Display a message with `read` using:

```
read -p "Enter input:" var
```

4. Read the input after a timeout as follows:

```
read -t timeout var
```

For example:

```
$ read -t 2 var
#Read the string that is typed within 2 seconds into variable var.
```

5. Use a delimiter character to end the input line as follows:

```
read -d delim_char var
```

For example:

```
$ read -d ":" var
hello:#var is set to hello
```

Running a command until it succeeds

When using your shell for everyday tasks, there will be cases where a command might succeed only after some conditions are met, or the operation depends on an external event (such as a file being available to download). In such cases, one might want to run a command repeatedly until it succeeds.

How to do it...

Define a function in the following way:

```
repeat()
{
    while true
    do
        $@ && return
    done
}
```

Or, add this to your shell's `rc` file for ease of use:

```
repeat() { while true; do $@ && return; done }
```

How it works...

We create a function called `repeat` that has an infinite `while` loop, which attempts to run the command passed as a parameter (accessed by `$@`) to the function. It then returns if the command was successful, thereby exiting the loop.

There's more...

We saw a basic way to run commands until they succeed. Let us see what we can do to make things more efficient.

A faster approach

On most modern systems, `true` is implemented as a binary in `/bin`. This means that each time the aforementioned `while` loop runs, the shell has to spawn a process. To avoid this, we can use the `:` shell built-in, which always returns an exit code 0:

```
repeat() { while :; do $@ && return; done }
```

Though not as readable, this is certainly faster than the first approach.

Adding a delay

Let's say you are using `repeat()` to download a file from the Internet which is not available right now, but will be after some time. An example would be:

```
repeat wget -c http://www.example.com/software-0.1.tar.gz
```

In the current form, we will be sending too much traffic to the web server at `www.example.com`, which causes problems to the server (and maybe even to you, if say the server blacklists your IP for spam). To solve this, we can modify the function and add a small delay as follows:

```
repeat() { while :; do $@ && return; sleep 30; done }
```

This will cause the command to run every 30 seconds.

Field separators and iterators

The **internal field separator (IFS)** is an important concept in shell scripting. It is very useful while manipulating text data. We will now discuss delimiters that separate different data elements from single data stream. An internal field separator is a delimiter for a special purpose. An internal field separator is an environment variable that stores delimiting characters. It is the default delimiter string used by a running shell environment.

Consider the case where we need to iterate through words in a string or **comma separated values (CSV)**. In the first case we will use `IFS=" "` and in the second, `IFS=","`. Let us see how to do it.

Getting ready

Consider the case of CSV data:

```
data="name,sex,rollno,location"
To read each of the item in a variable, we can use IFS.
oldIFS=$IFS
IFS=, now,
for item in $data;
do
```

```
        echo Item: $item
    done
```

```
IFS=$oldIFS
```

The output is as follows:

```
Item: name
Item: sex
Item: rollno
Item: location
```

The default value of IFS is a space component (newline, tab, or a space character).

When IFS is set as `,` the shell interprets the comma as a delimiter character, therefore, the `$item` variable takes substrings separated by a comma as its value during the iteration.

If IFS is not set as `,` then it would print the entire data as a single string.

How to do it...

Let us go through another example usage of IFS by taking the `/etc/passwd` file into consideration. In the `/etc/passwd` file, every line contains items delimited by `:`. Each line in the file corresponds to an attribute related to a user.

Consider the input: `root:x:0:0:root:/root:/bin/bash`. The last entry on each line specifies the default shell for the user. To print users and their default shells, we can use the IFS hack as follows:

```
#!/bin/bash
#Desc: Illustration of IFS
line="root:x:0:0:root:/root:/bin/bash"
oldIFS=$IFS;
IFS=":"
count=0
for item in $line;
do

    [ $count -eq 0 ]  && user=$item;
    [ $count -eq 6 ]  && shell=$item;
    let count++
done;
IFS=$oldIFS
echo $user's shell is $shell;
```

The output will be:

```
root's shell is /bin/bash
```

Loops are very useful in iterating through a sequence of values. Bash provides many types of loops. Let us see how to use them:

► Using a `for` loop:

```
for var in list;
do
    commands; # use $var
done
list can be a string, or a sequence.
```

We can generate different sequences easily.

`echo {1..50}` can generate a list of numbers from 1 to 50. `echo {a..z}` or `{A..Z}` or `{a..h}` can generate lists of alphabets. Also, by combining these we can concatenate data.

In the following code, in each iteration, the variable `i` will hold a character in the range `a` to `z`:

```
for i in {a..z}; do actions; done;
```

The `for` loop can also take the format of the `for` loop in C. For example:

```
for((i=0;i<10;i++))
{
    commands; # Use $i
}
```

► Using a `while` loop:

```
while condition
do
    commands;
done
```

For an infinite loop, use `true` as the condition.

► Using a `until` loop:

A special loop called `until` is available with Bash. This executes the loop until the given condition becomes true. For example:

```
x=0;
until [ $x -eq 9 ]; # [ $x -eq 9 ] is the condition
do
    let x++; echo $x;
done
```


Comparisons and tests

Flow control in a program is handled by comparison and test statements. Bash also comes with several options to perform tests that are compatible with the Unix system-level features. We can use `if`, `if else`, and logical operators to perform tests and certain comparison operators to compare data items. There is also a command called `test` available to perform tests. Let us see how to use these.

How to do it...

We will have a look at all the different methods used for comparisons and performing tests:

- ▶ Using an `if` condition:

```
if condition;  
then  
    commands;  
fi
```
- ▶ Using `else if` and `else`:

```
if condition;  
then  
    commands;  
else if condition; then  
    commands;  
else  
    commands;  
fi
```

Nesting is also possible with `if` and `else`. The `if` conditions can be lengthy, to make them shorter we can use logical operators as follows:



- ▶ `[condition] && action; # action executes if the condition is true`
- ▶ `[condition] || action; # action executes if the condition is false`

`&&` is the logical AND operation and `||` is the logical OR operation. This is a very helpful trick while writing Bash scripts.

- ▶ Performing mathematical comparisons: Usually conditions are enclosed in square brackets `[]`. Note that there is a space between `[` or `]` and operands. It will show an error if no space is provided. An example is as follows:

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

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

```
[ $var -eq 0 ] # It returns true when $var equal to 0.
[ $var -ne 0 ] # It returns true when $var is not equal to 0
```

Other important operators are as follows:

- `-gt`: Greater than
- `-lt`: Less than
- `-ge`: Greater than or equal to
- `-le`: Less than or equal to

Multiple test conditions can be combined as follows:

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

- ▶ Filesystem related tests: We can test different filesystem-related attributes using different condition flags as follows:
 - `[-f $file_var]`: This returns true if the given variable holds a regular file path or filename
 - `[-x $var]`: This returns true if the given variable holds a file path or filename that is executable
 - `[-d $var]`: This returns true if the given variable holds a directory path or directory name
 - `[-e $var]`: This returns true if the given variable holds an existing file
 - `[-c $var]`: This returns true if the given variable holds the path of a character device file
 - `[-b $var]`: This returns true if the given variable holds the path of a block device file
 - `[-w $var]`: This returns true if the given variable holds the path of a file that is writable
 - `[-r $var]`: This returns true if the given variable holds the path of a file that is readable
 - `[-L $var]`: This returns true if the given variable holds the path of a symlink

An example of the usage is as follows:

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

- String comparisons: While using string comparison, it is best to use double square brackets, since the use of single brackets can sometimes lead to errors.

Two strings can be compared to check whether they are the same in the following manner:


- ❑ `[[$str1 = $str2]]`: This returns true when `str1` equals `str2`, that is, the text contents of `str1` and `str2` are the same
- ❑ `[[$str1 == $str2]]`: It is an alternative method for string equality check

We can check whether two strings are not the same as follows:

- ❑ `[[$str1 != $str2]]`: This returns true when `str1` and `str2` mismatch

We can find out the alphabetically smaller or larger string as follows:

- ❑ `[[$str1 > $str2]]`: This returns true when `str1` is alphabetically greater than `str2`
- ❑ `[[$str1 < $str2]]`: This returns true when `str1` is alphabetically lesser than `str2`

 Note that a space is provided after and before `=`, if it is not provided, it is not a comparison, but it becomes an assignment statement.

- ❑ `[[-z $str1]]`: This returns true if `str1` holds an empty string
- ❑ `[[-n $str1]]`: This returns true if `str1` holds a nonempty string

It is easier to combine multiple conditions using logical operators such as `&&` and `||` in the following code:

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

For example:

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

Output:

```
str1 is nonempty and str2 is empty string.
```

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

For example:

```
if [ $var -eq 0 ]; then echo "True"; fi  
can be written as  
if test $var -eq 0 ; then echo "True"; fi
```


2

Have a Good Command

In this chapter, we will cover:

- ▶ Concatenating with `cat`
- ▶ Recording and playingback of terminal sessions
- ▶ Finding files and file listing
- ▶ Playing with `xargs`
- ▶ Translating with `tr`
- ▶ Checksum and verification
- ▶ Cryptographic tools and hashes
- ▶ Sorting unique and duplicates
- ▶ Temporary file naming and random numbers
- ▶ Splitting files and data
- ▶ Slicing filenames based on extension
- ▶ Renaming and moving files in bulk
- ▶ Spell checking and dictionary manipulation
- ▶ Automating interactive input
- ▶ Making commands quicker by running parallel processes

Introduction

Unix-like systems have the privilege of having the best command-line tools. They help us achieve many tasks making our work easier. While each command has a specific focus, with practice you'll be able to solve complex problems by combining two or more commands. Some frequently used commands are `grep`, `awk`, `sed`, and `find`.

Mastering the Unix/Linux command line is an art; you will get better at using it as you practice and gain experience. This chapter will introduce you to some of the most interesting and useful commands.

Concatenating with `cat`

`cat` is one of the first commands that a command-line warrior must learn. It is usually used to read, display, or concatenate the contents of a file, but `cat` is capable of more than just that. We even scratch our heads when we need to combine standard input data, as well as data from a file using a single-line command. The regular way of combining the `stdin` data, as well as file data, is to redirect `stdin` to a file and then append two files. But we can use the `cat` command to do it easily in a single invocation. In this recipe we will see basic and advanced usages of `cat`.

How to do it...

The `cat` command is a very simple and frequently used command and it stands for concatenate.

The general syntax of `cat` for reading contents is:

```
$ cat file1 file2 file3 ...
```

This command concatenates data from the files specified as command-line arguments.

- ▶ To print contents of a single file:

```
$ cat file.txt  
This is a line inside file.txt  
This is the second line inside file.txt
```
- ▶ To print contents of more than one file:

```
$ cat one.txt two.txt  
This is line from one.txt  
This is line from two.txt
```

How it works...

`cat` can be used in a variety of ways, let's walk through some of these now.

The `cat` command can not only read from files and concatenate the data, but can also read the input from the standard input.

To read from the standard input, use a pipe operator as follows:

```
OUTPUT_FROM_SOME_COMMANDS | cat
```

Similarly, we can concatenate content from input files along with standard input using `cat`. Combine `stdin` and data from another file, as follows:

```
$ echo 'Text through stdin' | cat - file.txt
```

In this example, `-` acts as the filename for the `stdin` text.

There's more...

The `cat` command has a few other options for viewing files. Let's go through them.

Getting rid of extra blank lines

Sometimes text files may contain two or more blank lines together. If you need to remove the extra blank lines, use the following syntax:

```
$ cat -s file
```

For example:

```
$ cat multi_blanks.txt
```

```
line 1
```

```
line2
```

```
line3
```

```
line4
```

```
$ cat -s multi_blanks.txt # Squeeze adjacent blank lines
```

```
line 1
```

```
line2
```

```
line3
```

```
line4
```


Alternately, we can remove all blank lines using `tr`, as discussed in the *Translating with tr* recipe in this chapter.

Displaying tabs as ^I

It is hard to distinguish tabs and repeated space characters. While writing programs in languages such as Python, tabs and spaces have different meanings for indentation purposes. Therefore, the use of tab instead of spaces causes problems in indentation. It may become difficult to track where the misplacement of the tab or space occurred by looking through a text editor. `cat` has a feature that can highlight tabs. This is very helpful in debugging indentation errors. Use the `-T` option with `cat` to highlight tab characters as `^I`. An example is as follows:

```
$ cat file.py
def function():
    var = 5
        next = 6
    third = 7
```

```
$ cat -T file.py
def function():
^Ivar = 5
        next = 6
^Ithird = 7^I
```

Line numbers

By using the `-n` flag for the `cat` command will output each line with a line number prefixed. It is to be noted that the `cat` command never changes a file; instead it produces an output on `stdout` with modifications to input according to the options provided. For example:

```
$ cat lines.txt
line
line
line

$ cat -n lines.txt
 1 line
 2 line
 3 line
```



-n will make the `cat` command output line numbers even for blank lines. If you want to skip numbering blank lines, use the `-b` option.

Recording and playing back of terminal sessions

When you need to show somebody how to do something in the terminal, or you need to prepare a tutorial on how to do something through the command line, you would normally type the commands manually and show them. Or, you could record a screencast and playback the video to them. There are other options for doing this. Using the commands `script` and `scriptreplay`, we can record the order and timing of the commands and save the data to text files. Using these files, others can replay and see the output of the commands on the terminal until the playback is complete.

Getting ready

The `script` and `scriptreplay` commands are available in most of the GNU/Linux distributions. Recording the terminal sessions to a file will be interesting. You can create tutorials of command-line hacks and tricks to achieve a task by recording the terminal sessions. You can also share the recorded files for others to playback and see how to perform a particular task using the command line.

How to do it...

We can start recording the terminal session using the following commands:

```
$ script -t 2> timing.log -a output.session
type commands;
...
..
exit
```



Note that this recipe will not work with shells that do not support redirecting only `stderr` to a file, such as the `csh` shell.

Two configuration files are passed to the `script` command as arguments. One file is for storing timing information (`timing.log`) at which each of the commands is run, whereas the other file (`output.session`) is used for storing the command output. The `-t` flag is used to dump timing data to `stderr`. Here, you will see, `2>` is used to redirect `stderr` to `timing.log`.

By using the two files, `timing.log` (stores timing information) and `output.session` (stores command output information), we can replay the sequence of command execution as follows:

```
$ scriptreplay timing.log output.session
# Plays the sequence of commands and output
```

How it works...

Usually, we record desktop videos to prepare tutorials. However, videos require considerable amount of storage. On the other hand, a terminal script file is just a text file, usually only in the order of kilobytes.

You can share the `timing.log` and `output.session` files to anyone who wants to replay a terminal session in their terminal.

Finding files and file listing

`find` is one of the great utilities in the Unix/Linux command-line toolbox. It is a very useful command for shell scripts; however, many people do not use it to its fullest effectiveness. This recipe deals with most of the common ways to utilize `find` to locate files.

Getting ready

The `find` command uses the following strategy: `find` descends through a hierarchy of files, matches the files that meet specified criteria, and performs some actions. Let's go through different use cases of `find` and its basic usages.

How to do it...

To list all the files and folders from the current directory to the descending child directories, use the following syntax:

```
$ find base_path
```

`base_path` can be any location from which `find` should start descending (for example, `/home/slynux/`).

An example of this command is as follows:

```
$ find . -print
# Print lists of files and folders
```

. specifies current directory and .. specifies the parent directory. This convention is followed throughout the Unix filesystem.

The `-print` argument specifies to print the names (path) of the matching files. When `-print` is used, '\n' will be the delimiting character for separating each file. Also, note that even if you omit `-print`, the `find` command will print the filenames by default.

The `-print0` argument specifies each matching filename printed with the delimiting character '\0'. This is useful when a filename contains a space character.

There's more...

In this recipe we have learned the usage of the most commonly-used `find` command with an example. The `find` command is a powerful command-line tool and it is armed with a variety of interesting options. Let us take a look at them.

Search based on filename or regular expression match

The `-name` argument specifies a matching string for the filename. We can pass wildcards as its argument text. The `*.txt` command matches all the filenames ending with `.txt` and prints them. The `-print` option prints the filenames or file paths in the terminal that matches the conditions (for example, `-name`) given as options to the `find` command.

```
$ find /home/slynux -name "*.txt" -print
```

The `find` command has an option `-iname` (ignore case), which is similar to `-name` but it matches filenames while ignoring the case.

For example:

```
$ ls
example.txt  EXAMPLE.txt  file.txt
$ find . -iname "example*" -print
./example.txt
./EXAMPLE.txt
```

If we want to match either of the multiple criteria, we can use OR conditions as shown in the following:

```
$ ls
new.txt  some.jpg  text.pdf
$ find . \( -name "*.txt" -o -name "*.pdf" \) -print
./text.pdf
./new.txt
```

The previous command will print all of the .txt and .pdf files, since the `find` command matches both .txt and .pdf files. `\(` and `\)` are used to treat `-name "*.txt" -o -name "*.pdf"` as a single unit.

The `-path` argument can be used to match the file path for files that match the wildcards. `-name` always matches using the given filename. However, `-path` matches the file path as a whole. For example:

```
$ find /home/users -path "*/slynux/*" -print
This will match files as following paths.
/home/users/list/slynux.txt
/home/users/slynux/eg.css
```



The `-regex` argument is similar to `-path`, but `-regex` matches the file paths based on regular expressions.

Regular expressions are an advanced form of wildcard matching, which enables us to specify text with patterns. By using patterns, we can make matches to the text and print them. A typical example of text matching using regular expressions is: parsing all e-mail addresses from a given pool of text. An e-mail address takes the form `name@host.root`. So, it can be generalized as `[a-z0-9]+@[a-z0-9]+\.[a-z0-9]+`. The `+` sign signifies that the previous class of characters can occur one or more times, repeatedly, in the characters that follow.

The following command matches the .py or .sh files:

```
$ ls
new.PY  next.jpg  test.py
$ find . -regex ".*\(\.py\|\.sh\) $"
./test.py
```

Similarly, using `-iregex` ignores the case for the regular expressions that are available. For example:

```
$ find . -iregex ".*\\(\\.py\\|\\.sh\\)$"
./test.py
./new.PY
```



We will learn more about regular expressions in *Chapter 4, Texting and Driving*.

Negating arguments

`find` can also exclude things that match a pattern using `!`:

```
$ find . ! -name "*.txt" -print
```

This will match all the files whose names do not end in `.txt`. The following example shows the result of the command:

```
$ ls
list.txt  new.PY  new.txt  next.jpg  test.py
```

```
$ find . ! -name "*.txt" -print
.
./next.jpg
./test.py
./new.PY
```

Search based on the directory depth

When the `find` command is used, it recursively walks through all the subdirectories as much as possible, until it reaches the leaf of the subdirectory tree. We can restrict the depth to which the `find` command traverses using some depth parameters given to `find`. `-maxdepth` and `-mindepth` are the parameters.

In most of the cases, we need to search only in the current directory. It should not further descend into the subdirectories from the current directory. In such cases, we can restrict the depth to which the `find` command should descend using depth parameters. To restrict `find` from descending into the subdirectories from the current directory, the depth can be set as 1. When we need to descend to two levels, the depth is set as 2, and so on for the rest of the levels.

For specifying the maximum depth we use the `-maxdepth` level parameter. Similarly, we can also specify the minimum level at which the descending should start. If we want to start searching from the second level onwards, we can set the minimum depth using the `-mindepth` level parameter. To restrict the `find` command to descend to a maximum depth of 1, use the following command:

```
$ find . -maxdepth 1 -name "f*" -print
```

This command lists all the files whose names begin with "f", but only from the current directory. If there are subdirectories they are not printed or traversed. Similarly, `-maxdepth 2` traverses up to at most two descending levels of subdirectories.

`-mindepth` is similar to `-maxdepth`, but it sets the least depth level for the `find` traversal. It can be used to find and print the files that are located with a minimum level of depth from the base path. For example, to print all the files whose names begin with "f", and are at least two subdirectories distant from the current directory, use the following command:

```
$ find . -mindepth 2 -name "f*" -print
./dir1/dir2/file1
./dir3/dir4/f2
```

Even if there are files in the current directory or `dir1` and `dir3`, it will not be printed.



`-maxdepth` and `-mindepth` should be specified as the third argument to the `find` command. If they are specified as the fourth or further arguments, it may affect the efficiency of `find` as it has to do unnecessary checks (for example, if `-maxdepth` is specified as the fourth argument and `-type` as the third argument, the `find` command first finds out all the files having the specified `-type` and then finds all of the matched files having the specified depth. However, if the depth were specified as the third argument and `-type` as the fourth, `find` could collect all the files having at most the specified depth and then check for the file type, which is the most efficient way to search.

Search based on file type

Unix-like operating systems treat every object as a file. There are different kinds of files, such as regular file, directory, character devices, block devices, symlinks, hardlinks, sockets, FIFO, and so on.

The file search can be filtered out using the `-type` option. By using `-type`, we can specify to the `find` command that it should only match files having a specified type.

List only directories including descendants as follows:

```
$ find . -type d -print
```

It is hard to list directories and files separately. But `find` helps to do it. List only regular files as follows:

```
$ find . -type f -print
```

List only symbolic links as follows:

```
$ find . -type l -print
```

You can use the `type` arguments from the following table to properly match the required file type:

File type	Type argument
Regular file	<code>f</code>
Symbolic link	<code>l</code>
Directory	<code>d</code>
Character special device	<code>c</code>
Block device	<code>b</code>
Socket	<code>s</code>
FIFO	<code>p</code>

Search on file times

Unix/Linux filesystems have three types of timestamps on each file. They are as follows:

- ▶ **Access time** (`-atime`): It is the last timestamp of when the file was accessed by a user
- ▶ **Modification time** (`-mtime`): It is the last timestamp of when the file content was modified
- ▶ **Change time** (`-ctime`): It is the last timestamp of when the metadata for a file (such as permissions or ownership) was modified



There is no such thing as creation time in Unix.

`-atime`, `-mtime`, and `-ctime` are the time parameter options available with `find`. They can be specified with integer values in "number of days". These integer values are often attached with `-` or `+` signs. The `-` sign implies less than, whereas the `+` sign implies greater than. For example:

- ▶ Print all the files that were accessed within the last seven days as follows:

```
$ find . -type f -atime -7 -print
```
- ▶ Print all the files that are having access time exactly seven-days old as follows:

```
$ find . -type f -atime 7 -print
```
- ▶ Print all the files that have an access time older than seven days as follows:

```
$ find . -type f -atime +7 -print
```

Similarly, we can use the `-mtime` parameter for search files based on the modification time and `-ctime` for search based on the change time.

`-atime`, `-mtime`, and `-ctime` are time-based parameters that use the time metric in days. There are some other time-based parameters that use the time metric in minutes. These are as follows:

- ▶ `-amin` (access time)
- ▶ `-mmin` (modification time)
- ▶ `-cmin` (change time)

For example:

To print all the files that have an access time older than seven minutes, use the following command:

```
$ find . -type f -amin +7 -print
```

Another good feature available with `find` is the `-newer` parameter. By using `-newer`, we can specify a reference file to compare with the timestamp. We can find all the files that are newer (older modification time) than the specified file with the `-newer` parameter.

For example, find all the files that have a modification time greater than that of the modification time of a given `file.txt` file as follows:

```
$ find . -type f -newer file.txt -print
```

Timestamp manipulation flags for the `find` command are very useful for writing the system backup and maintenance scripts.

Search based on file size

Based on the file sizes of the files, a search can be performed as follows:

```
$ find . -type f -size +2k
# Files having size greater than 2 kilobytes
```

```
$ find . -type f -size -2k
# Files having size less than 2 kilobytes
```

```
$ find . -type f -size 2k
# Files having size 2 kilobytes
```

Instead of k we can use different size units such as the following:

- ▶ b: 512 byte blocks
- ▶ c: Bytes
- ▶ w: Two-byte words
- ▶ k: Kilobyte (1024 bytes)
- ▶ M: Megabyte (1024 kilobytes)
- ▶ G: Gigabyte (1024 megabytes)

Deleting based on the file matches

The `-delete` flag can be used to remove files that are matched by `find`.

Remove all the `.swp` files from the current directory as follows:

```
$ find . -type f -name "*.swp" -delete
```

Match based on the file permissions and ownership

It is possible to match files based on the file permissions. We can list out the files having specified file permissions as follows:

```
$ find . -type f -perm 644 -print
# Print files having permission 644
```

`-perm` specifies that `find` should only match files with their permission set to a particular value. Permissions are explained in more detail in the *File permissions, ownership, and the sticky bit* in Chapter 3, *File In, File Out*.

As an example usage case, we can consider the case of the Apache web server. The PHP files in the web server require proper permissions to execute. We can find out the PHP files that don't have proper execute permissions as follows:

```
$ find . -type f -name "*.php" ! -perm 644 -print
```

We can also search files based on ownership of the files. The files owned by a specific user can be found out using the `-user USER` option.

The `USER` argument can be a username or UID.

For example, to print the list of all files owned by the user `slynux`, you can use the following command:

```
$ find . -type f -user slynux -print
```

Executing commands or actions with find

The `find` command can be coupled with many of the other commands using the `-exec` option. It is one of the most powerful features that comes with `find`.

Consider the example in the previous section. We used `-perm` to find out the files that do not have proper permissions. Similarly, in the case where we need to change the ownership of all files owned by a certain user (for example, `root`) to another user (for example, `www-data`, the default Apache user in the web server), we can find all the files owned by `root` by using the `-user` option and using `-exec` to perform the ownership change operation.



You must run the `find` command as root if you want to change ownership of files or directories.

Let's have a look at the following example:

```
# find . -type f -user root -exec chown slynux {} \;
```

In this command, `{ }` is a special string used with the `-exec` option. For each file match, `{ }` will be replaced with the filename for `-exec`. For example, if the `find` command finds two files `test1.txt` and `test2.txt` with owner `slynux`, the `find` command will perform:

```
chown slynux { }
```

This gets resolved to `chown slynux test1.txt` and `chown slynux test2.txt`.



Sometimes we don't want to run the command for each file. Instead, we might want to run it a fewer times with a list of files as parameters. For this, we use `+` instead of `;` in the `exec` syntax.

Another usage example is to concatenate all the C program files in a given directory and write it to a single file, say, `all_c_files.txt`. We can use `find` to match all the C files recursively and use the `cat` command with the `-exec` flag as follows:

```
$ find . -type f -name "*.c" -exec cat {} \;>all_c_files.txt
```

`-exec` is followed by any command. `{}` is a match. For every matched filename, `{}` is replaced with the filename.


To redirect the data from `find` to the `all_c_files.txt` file, we have used the `>` operator instead of `>>` (append) because the entire output from the `find` command is a single data stream (`stdin`). `>>` is necessary only when multiple data streams are to be appended to a single file.

For example, to copy all the `.txt` files that are older than 10 days to a directory `OLD`, use the following command:

```
$ find . -type f -mtime +10 -name "*.txt" -exec cp {} OLD \;
```

Similarly, the `find` command can be coupled with many other commands.

[



-exec with multiple commands

We cannot use multiple commands along with the `-exec` parameter. It accepts only a single command, but we can use a trick. Write multiple commands in a shell script (for example, `commands.sh`) and use it with `-exec` as follows:

```
-exec ./commands.sh {} \;
```

]

`-exec` can be coupled with `printf` to produce a very useful output. For example:

```
$ find . -type f -name "*.txt" -exec printf "Text file: %s\n" {} \;
```

Skipping specified directories when using the find command

Skipping certain subdirectories for performance improvement is sometimes required while doing a directory search and performing an action. For example, when programmers look for particular files on a development source tree, which is under the version control system such as Git, the source hierarchy will always contain the `.git` directory in each of the subdirectories (`.git` stores version-control-related information for every directory). Since version-control-related directories do not produce useful output, they should be excluded from the search. The technique of excluding files and directories from the search is known as **pruning**. It can be performed as follows:

```
$ find devel/source_path \( -name ".git" -prune \) -o \( -type f -print \)
```

Instead of `\(-type -print \)`, use required filter.

The preceding command prints the name (path) of all the files that are not from the `.git` directories.

Here, `\(-name ".git" -prune \)` is the exclude portion, which specifies that the `.git` directory should be excluded and `\(-type f -print \)` specifies the action to be performed. The actions to be performed are placed in the second block `-type f -print` (the action specified here is to print the names and path of all the files).

Playing with xargs

We use pipes to redirect `stdout` (standard output) of a command to `stdin` (standard input) of another command. For example:

```
cat foo.txt | grep "test"
```

Some of the commands accept data as command-line arguments rather than a data stream through `stdin` (standard input). In that case, we cannot use pipes to supply data through command-line arguments.

We should try alternate methods. `xargs` is a command that is very helpful in handling standard input data to the command-line argument conversions. It can manipulate `stdin` and convert to command-line arguments for the specified command. Also, `xargs` can convert any one-line or multiple-line text inputs into other formats, such as multiple lines (specified number of columns) or a single line and vice versa.

All Bash users love one-liner commands, which are command sequences that are joined by using the pipe operator, but do not use the semicolon terminator (`;`) between the commands used. Crafting one-line commands makes tasks more efficient and simpler to solve. It requires proper understanding and practice to formulate one-liners for solving text processing problems. `xargs` is one of the important components for building one-liner commands.

Getting ready

When using the pipe operator, the `xargs` command should always be the first thing to appear after the operator. `xargs` uses standard input as the primary data stream source. It uses `stdin` and executes another command by providing command-line arguments for that executing command using the `stdin` data source. For example:

```
command | xargs
```

How to do it...

The `xargs` command can supply arguments to a command by reformatting the data received through `stdin`.

`xargs` can act as a substitute that can perform similar actions as the `-exec` argument in the case of the `find` command. Let's see a variety of hacks that can be performed using the `xargs` command.

- **Converting multiple lines of input to a single-line output:** Multiple-line input can be converted simply by removing the newline character and replacing with the " " (space) character. '\n' is interpreted as a newline character, which is the delimiter for the lines. By using `xargs`, we can ignore all the newlines with space so that multiple lines can be converted into a single-line text as follows:

```
$ cat example.txt # Example file
1 2 3 4 5 6
7 8 9 10
11 12
```

```
$ cat example.txt | xargs
1 2 3 4 5 6 7 8 9 10 11 12
```

- **Converting single-line into multiple-line output:** Given a maximum number of arguments in a line = `n`, we can split any `stdin` (standard input) text into lines of `n` arguments each. An argument is a piece of a string delimited by " " (space). Space is the default delimiter. A single line can be split into multiple lines as follows:

```
$ cat example.txt | xargs -n 3
1 2 3
4 5 6
7 8 9
10 11 12
```

How it works...

The `xargs` command is appropriate to be applied to many problem scenarios with its many options. Let's see how these options can be used wisely to solve problems.

We can also use our own delimiter towards separating arguments. To specify a custom delimiter for input, use the `-d` option as follows:

```
$ echo "splitXsplitXsplitXsplit" | xargs -d X
split split split split
```

In the preceding code, `stdin` contains a string consisting of multiple `x` characters. We can use `x` as the input delimiter by using it with `-d`. Here, we have explicitly specified `x` as the input delimiter, whereas in the default case `xargs` takes the **internal field separator** (space) as the input delimiter.

By using `-n` along with the previous command, we can split the input into multiple lines having two words each as follows:

```
$ echo "splitXsplitXsplitXsplit" | xargs -d X -n 2
split split
split split
```

There's more...

We have learned how to format `stdin` to different output as arguments from the previous examples. Now, let's learn how to supply this formatted output as arguments to commands.

Passing formatted arguments to a command by reading `stdin`

Write a small custom `echo` script for better understanding of example usages with `xargs` to provide command arguments:

```
#!/bin/bash
#Filename: cecho.sh

echo $*'#'
```

When arguments are passed to the `cecho.sh` shell, it will print the arguments terminated by the `#` character. For example:

```
$ ./cecho.sh arg1 arg2
arg1 arg2 #
```

Let's have a look at a problem:

- I have a list of arguments in a file (one argument in each line) to be provided to a command (say, `cecho.sh`). I need to provide arguments in two methods. In the first method, I need to provide one argument each for the command as follows:

```
./cecho.sh arg1
./cecho.sh arg2
./cecho.sh arg3
```

Or, alternately, I need to provide two or three arguments each for each execution of the command. For two arguments each, it would be similar to the following:

```
./cecho.sh arg1 arg2
./cecho.sh arg3
```

- In the second method, I need to provide all arguments at once to the command as follows:

```
./cecho.sh arg1 arg2 arg3
```

Run the preceding commands and note the output before going through the following section.

These problems can be solved using `xargs`. We have the list of arguments in a file called `args.txt`. The contents are as follows:

```
$ cat args.txt
arg1
arg2
arg3
```

For the first problem, we can execute the command multiple times with one argument per execution, therefore, use:

```
$ cat args.txt | xargs -n 1 ./cecho.sh
arg1 #
arg2 #
arg3 #
```

For executing a command with `x` arguments per each execution, use:

```
INPUT | xargs -n X
```

For example:

```
$ cat args.txt | xargs -n 2 ./cecho.sh
arg1 arg2 #
arg3 #
```

For the second problem, in order to execute the command at once with all the arguments, use:

```
$ cat args.txt | xargs ./ccat.sh
arg1 arg2 arg3 #
```

In the preceding examples, we have supplied command-line arguments directly to a specific command (for example, `cecho.sh`). We could only supply the arguments from the `args.txt` file. However, in real time, we may also need to add a constant parameter with the command (for example, `cecho.sh`), along with the arguments taken from `args.txt`. Consider the following example with the format:

```
./cecho.sh -p arg1 -l
```

In the preceding command execution `arg1` is the only variable text. All others should remain constant. We should read arguments from a file (`args.txt`) and supply it as:

```
./cecho.sh -p arg1 -l
./cecho.sh -p arg2 -l
./cecho.sh -p arg3 -l
```


To provide a command execution sequence as shown, `xargs` has an option `-I`. By using `-I`, we can specify a replacement string that will be replaced while `xargs` expands. When `-I` is used with `xargs`, it will execute as one command execution per argument.

Let's do it as follows:

```
$ cat args.txt | xargs -I {} ./cecho.sh -p {} -l
-p arg1 -l #
-p arg2 -l #
-p arg3 -l #
```

`-I {}` specifies the replacement string. For each of the arguments supplied for the command, the `{}` string will be replaced with arguments read through `stdin`.



When used with `-I`, the command is executed in a loop. When there are three arguments the command is executed three times along with the command `{}`. Each time `{}` is replaced with arguments one by one.

Using `xargs` with `find`

`xargs` and `find` are best friends. They can be combined to perform tasks easily. Usually, people combine them in the wrong way. For example:

```
$ find . -type f -name "*.txt" -print | xargs rm -f
```

This is dangerous. It may sometimes cause removal of unnecessary files. Here, we cannot predict the delimiting character (whether it is `\n` or `' '`) for the output of the `find` command. Many of the filenames may contain a space character (`' '`) and hence, `xargs` may misinterpret it as a delimiter (for example, `"hell text.txt"` is misinterpreted by `xargs` as `"hell"` and `"text.txt"`).

Hence, we must use `-print0` along with `find` to produce an output with a delimited character null (`\0`) whenever we use the `find` output as the `xargs` input.

Let's use `find` to match and list of all the `.txt` files and remove them using `xargs`:

```
$ find . -type f -name "*.txt" -print0 | xargs -0 rm -f
```

This removes all `.txt` files. `xargs -0` interprets that the delimiting character is `\0`.

Counting the number of lines of C code in a source code directory

This is a task most programmers do, that is, counting all C program files for **Lines of Code (LOC)**. The code for this task is as follows:

```
$ find source_code_dir_path -type f -name "*.c" -print0 | xargs -0 wc -l
```



If you want more statistics about your source code, there is a utility called **SLOccount**, which is very useful. Modern GNU/Linux distributions usually have packages or you can get it from <http://www.dwheeler.com/sloccount/>.

While and subshell trick with stdin

`xargs` is restricted to providing arguments in limited ways to supply arguments. Also, `xargs` cannot supply arguments to multiple sets of commands. For executing commands with collected arguments from the standard input, we have a very flexible method. A subshell with a `while` loop can be used to read arguments and execute commands in a trickier way as follows:

```
$ cat files.txt | ( while read arg; do cat $arg; done )
# Equivalent to cat files.txt | xargs -I {} cat {}
```

Here, by replacing `cat $arg` with any number of commands using a `while` loop, we can perform many command actions with the same arguments. We can also pass the output to other commands without using pipes. Subshell `()` tricks can be used in a variety of problematic environments. When enclosed within subshell operators, it acts as a single unit with multiple commands inside, like so:

```
$ cmd0 | ( cmd1;cmd2;cmd3 ) | cmd4
```

If `cmd1` is `cd /`, within the subshell, the path of the working directory changes. However, this change resides inside the subshell only. `cmd4` will not see the directory change.

Translating with tr

`tr` is a small and beautiful command in the Unix command-warrior toolkit. It is one of the important commands frequently used to craft beautiful one-liner commands. It can be used to perform substitution of characters, deletion of the characters, and squeezing of repeated characters from the standard input. It is often called **translate**, since it can translate a set of characters to another set. In this recipe we will see how to use `tr` to perform basic translation between sets.

Getting ready

`tr` accepts input only through `stdin` (standard input) and cannot accept input through command-line arguments. It has the following invocation format:

```
tr [options] set1 set2
```

Input characters from `stdin` are mapped from `set1` to `set2` and the output is written to `stdout` (standard output). `set1` and `set2` are character classes or a set of characters. If the length of sets is unequal, `set2` is extended to the length of `set1` by repeating the last character, or else, if the length of `set2` is greater than that of `set1`, all the characters exceeding the length of `set1` are ignored from `set2`.

How to do it...

To perform translation of characters in the input from uppercase to lowercase, use the following command:

```
$ echo "HELLO WHO IS THIS" | tr 'A-Z' 'a-z'
```

'A-Z' and 'a-z' are the sets. We can specify custom sets as needed by appending characters or character classes.

'ABD-}', 'aA.', 'a-ce-x', 'a-c0-9', and so on are valid sets. We can define sets easily. Instead of writing continuous character sequences, we can use the 'startchar-endchar' format. It can also be combined with any other characters or character classes. If startchar-endchar is not a valid continuous character sequence, they are then taken as a set of three characters (for example, startchar, -, and endchar). You can also use special characters such as '\t', '\n', or any ASCII characters.

How it works...

By using `tr` with the concept of sets, we can map characters from one set to another set easily. Let's go through an example on how to use `tr` for encrypting and decrypting numeric characters:

```
$ echo 12345 | tr '0-9' '9876543210'
87654 #Encrypted
```

```
$ echo 87654 | tr '9876543210' '0-9'
12345 #Decrypted
```

Let's look at another interesting example.

ROT13 is a well-known encryption algorithm. In the ROT13 scheme, the same function is used to encrypt and decrypt text. The ROT13 scheme performs alphabetic rotation of characters for 13 characters. Let's perform ROT13 using `tr` as follows:

```
$ echo "tr came, tr saw, tr conquered." | tr 'a-zA-Z' 'n-zA-M'
```

The output will be:

```
ge pnzr, ge fnj, ge pbadhrerq.
```

By sending the encrypted text again to the same ROT13 function, we get:

```
$ echo ge pnzr, ge fnj, ge pbadhrerq. | tr 'a-zA-Z' 'n-za-mN-ZA-M'
```

The output will be:

```
tr came, tr saw, tr conquered.
```

`tr` can be used to convert tab characters into space as follows:

```
$ tr '\t' ' ' < file.txt
```

There's more...

We saw some basic translations using the `tr` command. Let's see what else can `tr` help us achieve.

Deleting characters using `tr`

`tr` has an option `-d` to delete a set of characters that appear on `stdin` by using the specified set of characters to be deleted as follows:

```
$ cat file.txt | tr -d '[set1]'
#Only set1 is used, not set2
```

For example:

```
$ echo "Hello 123 world 456" | tr -d '0-9'
Hello world
# Removes the numbers from stdin and print
```

Complementing character set

We can use a set to complement `set1` by using the `-c` flag. `set2` is optional in the following command:

```
tr -c [set1] [set2]
```

The complement of `set1` means that it is the set having all the characters except characters in `set1`.

The best usage example is to delete all the characters from the input text except the ones specified in the complement set. For example:

```
$ echo hello 1 char 2 next 4 | tr -d -c '0-9 \n'
1 2 4
```

Here, the complement set is the set containing all numerals, space characters, and newline characters. All other characters are removed since `-d` is used with `tr`.

Squeezing characters with tr

The `tr` command is very helpful in many text-processing contexts. Repeated continuous characters should be squeezed to a single character in many circumstances. Squeezing of whitespace is a frequently occurring task.

`tr` provides the `-s` option to squeeze repeating characters from the input. It can be performed as follows:

```
$ echo "GNU is      not      UNIX. Recursive  right ?" | tr -s ' '
GNU is not UNIX. Recursive right ?
# tr -s '[set]'
```

Let's use `tr` in a tricky way to add a given list of numbers from a file as follows:

```
$ cat sum.txt
1
2
3
4
5

$ cat sum.txt | echo $[ $(tr '\n' '+' ) 0 ]
15
```

How does this hack work?

Here, the `tr` command is used to replace `'\n'` with the `'+'` character, hence we form the string `"1+2+3+. . 5+"`, but at the end of the string we have an extra `+` operator. In order to nullify the effect of the `+` operator, `0` is appended.

`$[operation]` performs a numeric operation. Hence, it forms the string as follows:

```
echo $[ 1+2+3+4+5+0 ]
```

If we use a loop to perform the addition by reading numbers from a file, it would take a few lines of code. Here a one-liner does the trick.

`tr` can also be used in this way to get rid of extra newlines as follows:

```
$ cat multi_blanks.txt | tr -s '\n'
line 1
line2
line3
line4
```

In the preceding usage of `tr`, it removes the extra '`\n`' characters into a single '`\n`' (newline character).

Character classes

`tr` can use different character classes as sets. The different classes are as follows:

- ▶ `alnum`: Alphanumeric characters
- ▶ `alpha`: Alphabetic characters
- ▶ `cntrl`: Control (nonprinting) characters
- ▶ `digit`: Numeric characters
- ▶ `graph`: Graphic characters
- ▶ `lower`: Lowercase alphabetic characters
- ▶ `print`: Printable characters
- ▶ `punct`: Punctuation characters
- ▶ `space`: Whitespace characters
- ▶ `upper`: Uppercase characters
- ▶ `xdigit`: Hexadecimal characters

We can select the required classes and use them as follows:

```
tr [:class:] [:class:]
```

For example:

```
tr '[:lower:]' '[:upper:]'
```

Checksum and verification

Checksum programs are used to generate checksum key strings from the files and verify the integrity of the files later by using that checksum string. A file might be distributed over the network or any storage media to different destinations. Due to many reasons, there are chances of the file being corrupted due to a few bits missing during the data transfer by different reasons. These errors happen most often while downloading the files from the Internet, transferring through a network, CD-ROM damage, and so on.

Hence, we need to know whether the received file is the correct one or not by applying some kind of test. The special key string that is used for this file integrity test is known as a **checksum**.

We calculate the checksum for the original file as well as the received file. By comparing both of the checksums, we can verify whether the received file is the correct one or not. If the checksums (calculated from the original file at the source location and the one calculated from the destination) are equal, it means that we have received the correct file without causing any erroneous data loss during the data transfer. Otherwise, the user has to repeat the data transfer and try the checksum comparison again.

Checksums are crucial while writing backup scripts or maintenance scripts that transfer files through the network. By using checksum verification, files corrupted during the data transfer over the network can be identified and those files can be resent again from the source to the destination.

In this recipe we will see how to compute checksums to verify integrity of data.

Getting ready

The most famous and widely used checksum techniques are **md5sum** and **SHA-1**. They generate checksum strings by applying the corresponding algorithm to the file content. Let's see how we can generate a checksum from a file and verify the integrity of that file.

How to do it...

To compute the md5sum, use the following command:

```
$ md5sum filename
68b329da9893e34099c7d8ad5cb9c940 filename
```

md5sum is a 32-character hexadecimal string as given.

We redirect the checksum output into a file and use that MD5 file for verification as follows:

```
$ md5sum filename > file_sum.md5
```

How it works...

The syntax for the md5sum checksum calculation is as follows:

```
$ md5sum file1 file2 file3 ..
```

When multiple files are used, the output will contain a checksum for each of the files having one checksum string per line, as follows:

```
[checksum1]  file1
[checksum1]  file2
[checksum1]  file3
```

The integrity of a file can be verified by using the generated file as follows:

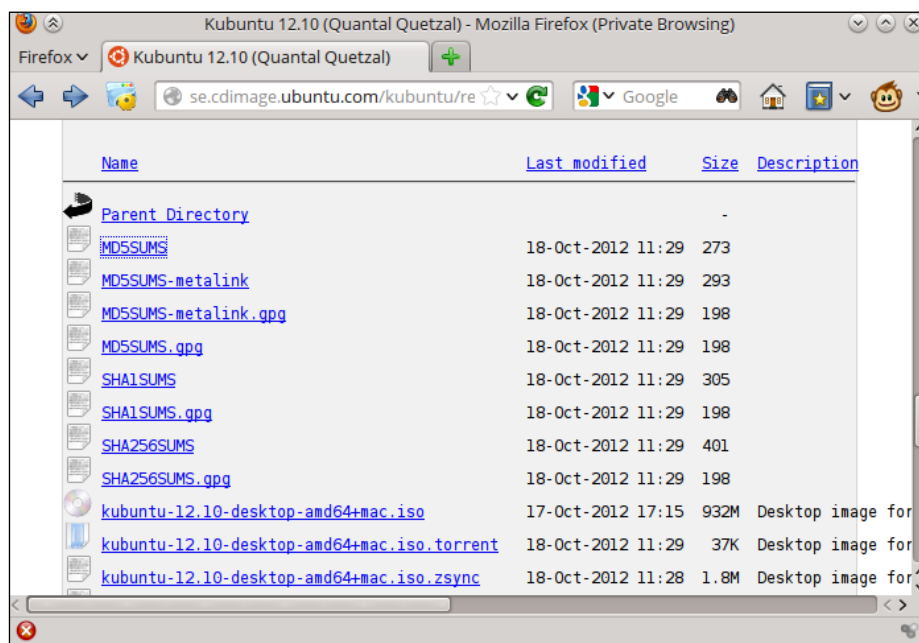
```
$ md5sum -c file_sum.md5
# It will output a message whether checksum matches or not
```

Or, alternately, if we need to check all the files using all .md5 information available, use:

```
$ md5sum -c *.md5
```

SHA-1 is another commonly used checksum algorithm like md5sum. It generates a 40-character hex code from a given input file. The command used for calculating an SHA-1 string is sha1sum. Its usage is very similar to that of md5sum. Simply replace md5sum with sha1sum in all the commands previously mentioned. Instead of file_sum.md5, change the output filename to file_sum.shal.

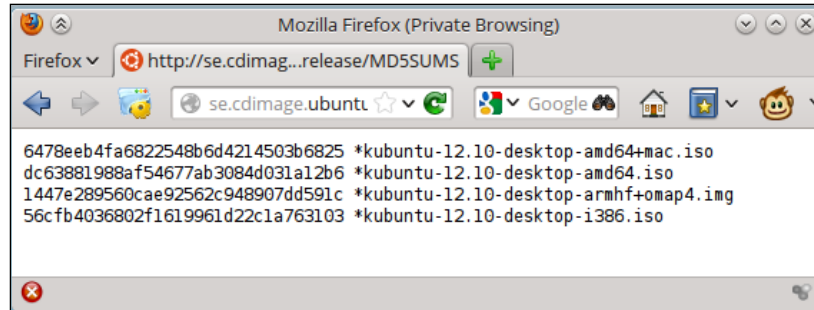
Checksum verification is very useful to verify the integrity of files that we download from the Internet. For example, ISO images are very susceptible to erroneous bits. A few wrong bits in the wrong location and the ISO may not be useable. Therefore, to check whether we received the file correctly, checksums are widely used. For the same file data the checksum program will always produce the same checksum string:



The screenshot shows a web browser window titled 'Kubuntu 12.10 (Quantal Quetzal) - Mozilla Firefox (Private Browsing)'. The address bar shows 'se.cdimage.ubuntu.com/kubuntu/re'. The main content area displays a directory listing with columns for Name, Last modified, Size, and Description. The listing includes files for MD5SUMS, SHA1SUMS, and SHA256SUMS, as well as ISO images and their torrents and zsync files.

Name	Last modified	Size	Description
Parent Directory	-	-	-
MD5SUMS	18-Oct-2012 11:29	273	
MD5SUMS-metalink	18-Oct-2012 11:29	293	
MD5SUMS-metalink.gpg	18-Oct-2012 11:29	198	
MD5SUMS.gpg	18-Oct-2012 11:29	198	
SHA1SUMS	18-Oct-2012 11:29	305	
SHA1SUMS.gpg	18-Oct-2012 11:29	198	
SHA256SUMS	18-Oct-2012 11:29	401	
SHA256SUMS.gpg	18-Oct-2012 11:29	198	
kubuntu-12.10-desktop-amd64+mac.iso	17-Oct-2012 17:15	932M	Desktop image for
kubuntu-12.10-desktop-amd64+mac.iso.torrent	18-Oct-2012 11:29	37K	Desktop image for
kubuntu-12.10-desktop-amd64+mac.iso.zsync	18-Oct-2012 11:28	1.8M	Desktop image for

This is the md5sum checksum that is created:



There's more...

Checksums are also useful when used with a number of files. Let us see how to apply checksums to a collection of files and verify correctness.

Checksum for directories

Checksums are calculated for files. Calculating the checksum for a directory would mean that we would need to calculate the checksums for all the files in the directory, recursively.

It can be achieved by the `md5deep` or `sha1deep` command. Install the `md5deep` package to make these commands available. An example of this command is as follows:

```
$ md5deep -rl directory_path > directory.md5
# -r to enable recursive traversal
# -l for using relative path. By default it writes absolute file path in
output
```

Alternately, use a combination of `find` to calculate checksums recursively:

```
$ find directory_path -type f -print0 | xargs -0 md5sum >> directory.md5
```

To verify, use the following command:

```
$ md5sum -c directory.md5
```

Cryptographic tools and hashes

Encryption techniques are used mainly to protect data from unauthorized access. There are many algorithms available and we have discussed the most commonly used ones. There are a few tools available in a Linux environment for performing encryption and decryption. Sometimes we use encryption algorithm hashes for verifying data integrity. This section will introduce a few commonly used cryptographic tools and a general set of algorithms that these tools can handle.

How to do it...

Let us see how to use tools such as `crypt`, `gpg`, `base64`, `md5sum`, `sha1sum`, and `openssl`:

- ▶ The `crypt` command is a simple and relatively insecure cryptographic utility that takes a file from `stdin` and a passphrase as input and output encrypted data into `stdout` (and, hence, we use redirection for the input and output files):

```
$ crypt <input_file >output_file
```

Enter passphrase:


It will interactively ask for a passphrase. We can also provide a passphrase through command-line arguments:

```
$ crypt PASSPHRASE <input_file >encrypted_file
```

In order to decrypt the file, use:

```
$ crypt PASSPHRASE -d <encrypted_file >output_file
```

- ▶ `gpg` (GNU privacy guard) is a widely used tool for protecting files with encryption that ensures that data is not read until it reaches its intended destination. Here we discuss how to encrypt and decrypt a file.

 `gpg` signatures are also widely used in e-mail communications to "sign" e-mail messages, proving the authenticity of the sender.


In order to encrypt a file with `gpg` use:

```
$ gpg -c filename
```

This command reads the passphrase interactively and generates `filename.gpg`. In order to decrypt a `gpg` file use:

```
$ gpg filename.gpg
```

This command reads a passphrase and decrypts the file.

 We don't cover `gpg` in much detail in this book. If you're interested in more information, please see http://en.wikipedia.org/wiki/GNU_Privacy_Guard.

- ▶ Base64 is a group of similar encoding schemes that represents binary data in an ASCII string format by translating it into a radix-64 representation. The `base64` command can be used to encode and decode the Base64 string. In order to encode a binary file into the Base64 format, use:

```
$ base64 filename > outputfile
```

Or:

```
$ cat file | base64 > outputfile
```

It can read from `stdin`.

Decode Base64 data as follows:

```
$ base64 -d file > outputfile
```

Or:

```
$ cat base64_file | base64 -d > outputfile
```

- ▶ **md5sum** and **SHA-1** are unidirectional hash algorithms, which cannot be reversed to form the original data. These are usually used to verify the integrity of data or for generating a unique key from a given data:

```
$ md5sum file
```

```
8503063d5488c3080d4800ff50850dc9  file
```

```
$ shasum file
```

```
1ba02b66e2e557fede8f61b7df282cd0a27b816b  file
```

These types of hashes are commonly used for storing passwords. Passwords are stored as their hashes and when a user wants to authenticate, the password is read and converted to the hash. Then, this hash is compared to the one that is stored already. If they are the same, the password is authenticated and access is provided, otherwise it is denied. Storing plain text password strings is risky and poses a security risk.



Although commonly used, `md5sum` and `SHA-1` are no longer considered secure. This is because of the rise of computing power in recent times that makes it easier to crack them. It is recommended to use tools such as `bcrypt` or `sha512sum` instead. Read more about this at <http://codahale.com/how-to-safely-store-a-password/>.

► Shadow-like hash (salted hash)

Let us see how to generate a shadow-like salted hash for passwords. The user passwords in Linux are stored as their hashes in the `/etc/shadow` file. A typical line in `/etc/shadow` will look like this:

```
test:$6$fG4eWdUi$ohTK0lEUzNk77.4S8MrYe07NTRV4M3LrJnZP9p.qc1bR5c.
EcOruzPXfEu1uloBFUa18ENRH7F70zhodas3cR.:14790:0:99999:7:::
```

`6fG4eWdUi$ohTK0lEUzNk77.4S8MrYe07NTRV4M3LrJnZP9p.qc1bR5c. EcOruzPXfEu1uloBFUa18ENRH7F70zhodas3cR` is the shadow hash corresponding to its password.

In some situations, we may need to write critical administration scripts that may need to edit passwords or add users manually using a shell script. In that case we have to generate a shadow password string and write a similar line as the preceding one to the shadow file. Let's see how to generate a shadow password using `openssl`.

Shadow passwords are usually salted passwords. `SALT` is an extra string used to obfuscate and make the encryption stronger. The salt consists of random bits that are used as one of the inputs to a key derivation function that generates the salted hash for the password.



For more details on salt, see the Wikipedia page [http://en.wikipedia.org/wiki/Salt_\(cryptography\)](http://en.wikipedia.org/wiki/Salt_(cryptography)).

```
$ opensslpasswd -1 -salt SALT_STRING PASSWORD
$1$SALT_STRING$323VkWkSLHuhbt1zkSsUG.
```

Replace `SALT_STRING` with a random string and `PASSWORD` with the password you want to use.

Sorting unique and duplicates

Sorting is a common task that we can encounter with text files. The `sort` command helps us to perform sort operations over text files and `stdin`. Most often, it can also be coupled with many other commands to produce the required output. `uniq` is another command that is often used along with a `sort` command. It helps to extract unique (or duplicate) lines from a text or `stdin`. This recipe illustrates most of the use cases with `sort` and `uniq` commands.