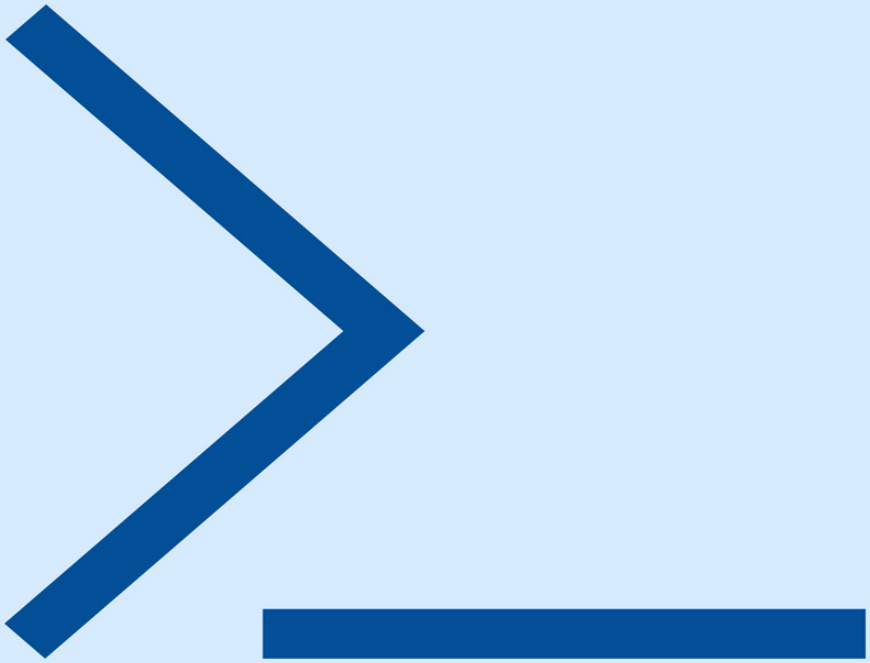
LINUX

— COMMAND LINE —

GETTING STARTED WITH
BASH AND SHELL SCRIPTING

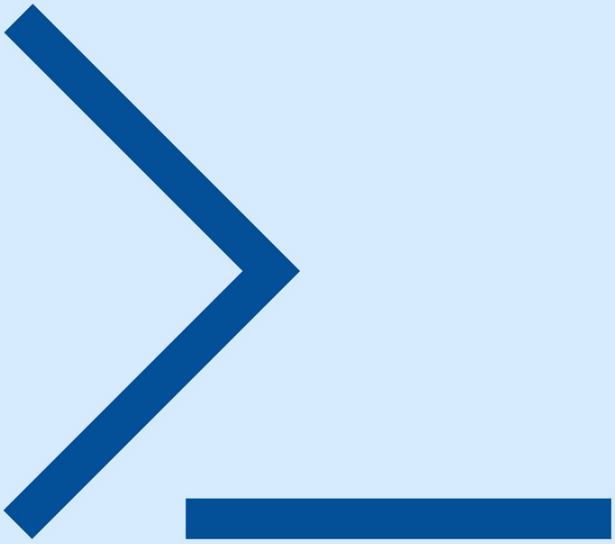


TRAVIS BOOTH

LINUX

— COMMAND LINE —

GETTING STARTED WITH
BASH AND SHELL SCRIPTING



TRAVIS BOOTH

The Linux Command Line

Getting Started with Bash and Shell Scripting

Travis Booth

Other Books by Travis Booth

Deep Learning Series

Deep Learning With Python: A Hands-On Guide for Beginners

Deep Learning With Python: A Comprehensive Guide Beyond the Basics

Python Data Analytics Series

Python Data Analytics: The Beginner's Real World Crash Course

Python Data Analytics: A Hands-On Guide Beyond the Basics

Python Data Science Series

Python Data Science: Hands-On Learning for Beginners

Python Data Science: A Hands-On Guide Beyond the Basics

Click Here to Explore Further!

Bonus Offer: Get the Ebook absolutely free when you purchase the paperback via Kindle Matchbook!

Table of Contents

Preface

Linux is one of the most dominant and adaptable operating systems globally. Linux operating system is utilized in many areas of Computer Science, ranging from embedded to smart devices. This Operating system is utilized in High-Performance Computers, servers that are running the global Internet infrastructure services like DNS, smartphones, android tablets, and televisions.

The Linux Operating System is distributed with intuitive and modern graphical user interfaces, which are customized to the distribution of Linux in use ranging from GNOME to KDE to Cinnamon. On any Linux computer, the shell is the most powerful and pliant way of interacting with the operating system kernel and attendant services. A Linux shell is used to execute individual commands. Commands are a set of instructions sent to the Linux kernel to tell the computer what to do. The Linux shell can also execute and interpret commands from a written text file called a bash script, which enables easy automation of tasks.

Many System administration tasks are routine and repetitive that need to be automated for example, automatically like sending alerts, e-mails, and performing routine maintenance. This book contains beginner information about the Linux command line, shell and bash scripting. The book introduces the novice Linux user to basic Linux scripting skills to automate and perform a number of Systems Administration tasks. The System Administration tasks in Linux range from parsing and interpreting text, to networking services operation automation. The Linux shell is only as powerful as the user intends it to be. Linux experts feel the power of shell scripting since they use the shell to do large automation. Linux Command Line gives you these magical capabilities!

What This Book Covers

This book is an immersive introduction into the Linux Operating system shell, how to configure the shell and the Linux environment and to write basic bash/shell scripts).

Chapter 1 : Introduces the reader to Linux Command Line concepts. The concepts will detail the different types of shells that exist and mainly focus

on BASH as the shell of choice. The chapter will also feature an immersive rundown of Linux Commands, BASH commands and how to start with BASH.

Chapter 2: Details different usage scenarios of BASH that users may meet in Systems Administration. This chapter will look at Redirection Basics, shell environment basics with an introduction to the VI text-editing tool.

Chapter 3: Introduces File searching and archiving. This chapter will also discuss Linux operations to find and remove duplicate files and enumerating words in a file.

Chapter 4: Looks at introducing our users to the basics of writing scripts and starting out with writing their first Bash script.

How to Use This Book

This book is written as a Linux Command Line and Bash scripting Primer, teaching the user how to handle BASH shell scripting. You should go through the chapters in sequence. You may choose to skip some chapters if you are reasonably comfortable with the concepts covered in it.

Prerequisites

To comfortably use this book, the main expectation is that you have a working Linux Server installation in either Centos 7 or Ubuntu 18.4 operating systems. You can get this in one of two ways:

1. **Install Linux on an (Oracle Virtual Box) virtual machine.**

You are expected to install your distribution of choice, either CENTOS 7 or Ubuntu 18.4. I will explain the steps to install Centos 7 in this book. The rest of the book will be dealing with commands, which work on all Linux variants. Setup your Virtual Machine with 2GB RAM and 20GB Hard drive.

2. Using a **Live CD** on your Laptop/Desktop

Linux distributions like Centos/Ubuntucan can be run directly from a USB flash drive or CDROM/DVDROMwith no installations on your machine. To enable you to use **a LIVE CD** , you have to change your BOOT Order in BIOS settings to allow you to BOOT from either CDROM or USB Flash

drive first. After booting using your Live CD, go through the rest of the book and follow along with the commands. Much of the material in this book is "practical," so jump in and get your hands dirty.

Introduction

The Linux operating system's history can be traced from early LINUX systems that were freely distributed to educational institutions like colleges and universities in the USA. Initiatives like the Berkeley Software Distribution (BSD) put in some work on the operating systems. The Free Software Foundation helped make many of the parts required to fully develop a LINUX-like operating system that is free of charge. The last major component to be delivered was the Linux kernel developed by a Finnish student called Linus Torvalds around 1990 and with online collaboration from many Linux enthusiasts the Linux Kernel is what it is today.

This is a beginner Linux bash-scripting book that is going to introduce the Linux Operating System and explain why people use the Operating System over other proprietary systems. In this section, I will start by introducing our reader to what an Operating system is. I will detail the functions of an operating system, before introducing Linux and putting together the advantages and disadvantages of using Linux.

What is an Operating System?

An operating system is a computer hardware and software management program for users. Originally, operating systems were built to conduct repetitive hardware duties that focused on file management, running programs, and receiving user instructions. Using a user interface, you communicate with an operating system's kernel that enables the operating system to obtain and interpret user-sent instructions. To execute a job, you only need to submit an instruction to the operating system, such as reading a file or printing a document and the operating system will coordinate with the hardware to execute that instruction.

Operating system functions

There are many functions that are performed by the operating system, but the main purpose of the operating system is to provide an interface between a User and the Computer System's hardware. The different functions that the operating system performs are explained below.

Resource Manager

The Linux Operating System, like any other OS, is a Resource Manager. This means that it manages all Resources that are attached to the system such as Memory and Central Processing Unit and all Input Devices. The operating system will determine when the CPU will perform which operation.

Process Management

The operating system always keeps track of processes simultaneously running on your computer. The OS keeps track of various procedures running and decides which process and when it has access to the CPU. The system must also provide methods to start, stop, and change process statuses.

Hardware Management

During Linux System Boot up process, the Computer performs POST (Power-On Self-Test) which is a process that This operator finds out the hardware components currently attached to the Computer (CPU, hard drive, USB devices, and network cards) and loads the device drivers and modules required to access those particular hardware devices.

Memory Management

Memory management is the function where the operating system handles and manages the primary memory and extended memory of the Computer System. Memory management entails movement of system and custom processes back and forth between the main memory and the secondary memory during execution, such as the hard drive. Memory management also keeps track of all volatile memory places on the computer, irrespective of whether they are freely assigned to some processor. This operator finds out the quantity of memory to be assigned to specific procedures. Applications in Linux require RAM (random access memory) and swap space (extended memory) to function. The operating system is responsible for memory management to ensure optimal sharing of this resource by various programs running on the system.

File System Management

File system constructs or components are either built-in into the Linux operating system or loaded as modules. In Linux, everything is viewed as a file from directories, processes, and devices. The Linux operating system plays a major part in controlling who owns and who has access to processes, files, and folders, also known as directories that the file systems contain.

User and Group Management

System Administrators in all major operating systems add, update, delete, and manage user and group accounts on their systems. The Linux operating system facilitates the addition, deletion, and updating of user and group accounts and managing related user and group access permissions and level of control a user has on the system. Individual users and group accounts in Linux facilitate user access to their files and processes they own or are owned by other users of the System.

Programming Tools

The Linux operating system offers a number of programming tools for developing programs and libraries to implement Application Programming Interfaces that allow Linux to connect with many other Programs.

What is Linux?

Linux is a solid, lightweight and sturdy open-source operating system that has enterprise-level Internet services, extensive development tools, fully functional graphical user interfaces (GUIs), and a massive number of applications ranging from office suites to multimedia applications (Petersen, 2008).

Linux is one of the twenty-first century's most eminent technological advances. In addition to its effect on huge Internet growth and its position as an enabling technology for a range of computer-driven systems, Linux development has been a model of how cooperative initiatives can surpass what people and businesses can do alone (Negus, 2015).

Linux operating system like any other operating system is composed of the Kernel, GNU utilities, a graphical user interface (GUI) and application

programs. The detailed components above each play a crucial role in the Linux operating system's operations.

The Linux Kernel

The Linux system's heart is the kernel. The kernel regulates all computer system hardware and software, allocating hardware if needed, and implementing software if needed.

The kernel is mainly accountable for four primary tasks, namely the management of system memory, software program management, hardware management, and file system management.

As an operating system, Linux comprises of the OS kernel, and it's components that oversee the management of your software and hardware and allows you to run applications on your System.

The OS of Choice

Linux is an operating system that has been developed through collaboration between communities of developers worldwide. Linux has been tested rigorously in the IT industry and was found to be stable, easy to use, and highly secure with a very powerful command-line interface. These and many other advantages stated below are the major reason why people choose Linux to power their Server infrastructure.

Linux OS Advantages

Linux operating system has many advantages that make it a solid choice for server and other operating system functions.

Powerful Command Line Interface

Linux distributions have a powerful bash shell. The shell is also called the terminal, and it affords users a text-only interface that gives them great power.

High Security

Linux operating system is known to be very secure when it comes to avoiding viruses and malware. The developers of Linux had security in mind when they were developing the Operating System. The major reason

for this security comes from the fact users are expected to create and login into Linux systems as a regular user. By default, most Linux distributions like Debian and Ubuntu do not permit Root Login. Most people access their Systems using limited regular accounts; which mean they have no privileges that enable them to cause damage to the system. Linux regular users can only cause damage to their own files.

The other major reason for Linux's greater level of security is that the Operating System's source code is publicly accessible for review since Linux is open-source software. The source code is accessed and reviewed by a huge community of enthusiastic developers all over the globe, which implies that most of the bugs and vulnerabilities have already been found and are being continually found.

Greater System Stability

The Linux operating system is very sturdy and crashes less frequently. Linux installations continue functioning well even after many years running continuously. Indeed, some users have setups that were installed once and had been running without incident for many years.

Linux servers in many data centers have high uptime, and their overall availability nears 99.9 percent. Linux servers do not require reboots after every system and program updates or patches. This sturdiness and high uptime have resulted in the bulk of the Internet running Linux servers in their infrastructure. ZDNET website estimates that a whopping 96.3 percent of the best Web servers are running on Linux.

Easy to Maintain

Maintaining Linux involves regular updates to the programs running on the system. Systems are regularly monitored and checked for bugs and vulnerabilities. Updating programs ensures that bugs are fixed, new features are added, and vulnerabilities are plugged. Linux is very easy to maintain even for beginner users. Experience Linux users and administrators are able to make central updates and program patches. Operating systems need a way to check for vulnerabilities continuously. All Linux distributions have their own central program repositories, where program code is audited and tested regularly for bugs and any other reported issues, the software repositories are used to keep the system up to date and safe. Linux

distributions offer regular system updates and no reboots are done after system updates. As a Linux user, it is possible for you to schedule system updates at regular times. Bash scripts and Linux Cron jobs are written to automate the processes of scheduling system updates.

Can be Run on any Piece of Hardware

Linux operating systems have efficient use of system resources, which include CPU and RAM. Linux uses the computer system's hardware resources effectively and efficiently. This efficiency means that you can run your Linux installation on any piece of decent hardware. Linux systems are running on a variety of architectures from low-specked computers to mobile phones and smaller gadgets.

Free to Use

Getting a Linux operating system is 100 percent free except for a few Enterprise Distributions, which charge for Support like Red Hat and Canonical. Linux is totally free of charge; users can download Linux and be good to go with decent skills. Linux has all the basic programs required by a typical and expert user available. Businesses and Governments can leverage the cost of the Operating system and build robust infrastructure at a significantly lower budget. Whole Internet ecosystems from DNS, DHCP, Email, and Communication are setup on Free Linux for free.

100 Percent Open Source

Linux is an operating system developed by a community of developers, and the source code of this operating system is freely available for users to make contributions through documentation, finding and fixing bugs. This community involvement means that there are many eyes that are auditing and working on the system to make it stable and error-free. Users are free to download the source code and modify or even distribute it.

Easy to Use

Linux operating system has increasingly become the mainstream. Linux is now user-friendlier and has a simple and intuitive graphical user interface (GUI) that makes it useable even by novice users. Contrary to the general view that Linux is only for geeks, it is now being used on laptops and other

gadgets. Someone used to Windows can now easily move and use Linux on their Personal Computers. The GUI has evolved to the extent that, without understanding any shell scripts or code, most of what typical users want can be accomplished on Linux as readily as is done on Windows.

Linux is Flexible for Customization

Linux has gifted all its users, from beginners to advanced administrators, the flexibility to change the look and feel of their environment to suit their requirements. We have more than three desktop environments such as GNOME, KDE, Unity, Cinnamon, and MATE.

Linux has a large pool of alternative system and application programs that users may select to install on their distributions.

Global System Support Available

Over the Internet, there is a powerful Linux Open source community that provides support for Linux through different forums. Any question posed in forums will generally be answered quickly as many enthusiastic Linux volunteers are online and working to fix the issues. Also accessible for business enterprises is the paid support option, with businesses such as Red Hat and Canonical providing 24-7 assistance for critical installations and services.

Linux Disadvantages

Linux operating system has a number of disadvantages of which the most mentioned is that it is difficult for most people to learn to use the operating system. The biggest disadvantage in Linux is that most hardware manufacturers build their device drivers for Windows, so the Linux user has to handle the programming of hardware drivers. Linux has an inherent lack of driver support.

Linux infrastructure setups require that your administrators have a high-level understanding of the operations of the operating system. Any organization needs to ensure that they have the requisite Linux expertise before they start using it.

The other disadvantage of Linux is the lack of a standard universally accepted distribution across the globe. There are varieties of Linux

distributions that operate differently and manage things uniquely. This may be daunting for most users

.

Chapter One: Linux Command Line

In this chapter, we are going to dive deep into the Linux Command Line Interpreter. We are going to concentrate on the Bash shell and its environment. We will have a brief introduction into the command line Interpreter, Shell, and Bash. I will guide you with regards to how to begin Bash programming.

The Command-Line Interpreter

Linux operating system can be accessed by users through the Graphical User Interface, also known as the GUI and through the command-line interface, also known as the CLI.Command-line interface also referred to as the terminal or console, is Linux's text-based interface into which you enter commands, which are basically instructions to your Computer. Commands are made up of options and arguments. These commands are read and interpreted by the command-line interface. If you are accessing your Computer using the GNOME or KDE graphical user interface, the command line interface is accessed by opening a terminal. When a user accesses the text-based login into the computer, the command line interface will be presented as the BASH shell.

What is a Shell?

The shell has an eminent function in a Linux system that is to run commands. The Linux shell is simply a program that reads user commands entered from the keyboard and passes them to the kernel for execution or processing. The shell can actually be considered as a full programming language that has functions and variables, and data structures such as arrays. The shell is directly connected to the kernel, and thus it has native File Input and Output primitives built into its syntax. The shell has a native process and job control function in a Linux operating system.

The shell enables interactive access to the Linux kernel and programs running on it. The shell executes and functions as a regular program, and it initializes or starts whenever a user logs in to a console. A shell is a macro processor that interprets and executes user instructions to the system through commands entered through the keyboard. The Linux bash shell is touted as both a command-line interpreter and a traditional programming language.

The shell offers the user interface a collection of GNU utilities as a command interpreter. It is possible to create files that contain lines of commands as code, and these files become commands themselves. These newly written commands known as shell scripts kept in directories such as **/bin** have the same status as system commands, enabling system administrators to create custom environments to automate their routine tasks. Linux shells are used interactively ornon-interactively depending on the method used to make them. Shell recognizes keyboard-typed input in the interactive mode. Shells also read commands from a file called bash script when executing non-interactively.

The shell is available to you in any of the many Linux distributions available. The shell enables us to write commands and package them as executable script files to run shell and all user programs, work with Linux file systems, compile computer program code, and administer the system. The shell gives many Linux experts much more power to control how a system operates and function. It is through the shell that users can customize and automate their Linux infrastructure.

A shell enables both synchronous and asynchronous execution of bash commands. The shell waits for concurrent commands to finish before accepting more input; while reading and executing extra commands, asynchronous commands continue to perform in conjunction with the shell. The constructs of redirection allow fine-grained control of these commands' input and output. In addition, the shell enables contents of command settings to be controlled. Shells also provide a small set of

built-in commands that can't or can't get functionality via distinct services. A user may check the types of shells **/etc/shells** that their system accepts:

```
[ppeters@rad-srv ~]$ cat /etc/shells
/bin/bash
/bin/sh
/sbin/nologin
/bin/dash
/bin/tcsh
/bin/csh
```

Every user created on the Linux system has a default shell set for them. The default shell settings are kept in the **/etc/passwd** file. Below is an example**/etc/passwd** entry for user *ppeters* :

```
ppeters:x:500:500::/home/ppeters:/bin/bash
```

Example Commands

To check the default shell for a USER in the Computer please type the following command in your terminal and check if your result resembles one below:

```
[ppeters@rad-srv ~]$ cat /etc/passwd
ppeters:x:500:500::/home/ppeters:/bin/bash
```

In Linux users are able to switch from one shell to the other. The user switches their shell by just typing the name of the shell of choice. If you just type the name, your System will locate the folder where the shell you have chosen is located using PATH settings. After typing the name of shell and pressing enter, a new prompt is usually shown, since each shell has its own appearance:

```
[ppeters@rad-srv ~]$ sh
sh-4.1$
```

What is Bash?

BASH, also known as ‘Bourne-Again SHell' is the default user shell on Linux systems. Bash is primarily a command-line interpreter who has traditional programming language constructs. In many circles Bash is treated as a programming language albeit with logic peculiar to Bash. Bash as a programming language has to support for variable constructs, functions and with program decision flow control constructs like conditional statements and loops.

BASH shell prompts are depended on the privileges of the logged-in user. There is a dollar sign ($) prompt for all regular users. Linux operating system has many other types of shells, each with its own prompt, for example, csh has **%** as the prompt. All privileged users like the root user have unique prompts; they have the**#** as the prompt. (Petersen, 2008).

Bash as the command line interpreter keeps all commands that are entered into the terminal as strings and has no concept of numbers unless a user has instructed it to do arithmetic calculations. Bash operates in a simplified way as follows:

1. Bash takes the user's keyboard input and separates it into words basing on white spaces (space or tab).

2. Bash then assumes that the **initial word** from the user input is always a **command** . Bash then takes everything that follows after the first word as **arguments** to be passed to the command.

3. After parsing and splitting the user input Bash then tries to process the command and add arguments if they are available.

Getting Started with Bash

Linux has a number of Distributions that are available for selection to users. All these Linux distributions have the bash as the default shell. Before working through this book, it is important to make a choice of your Distribution. I encourage the reader to choose Centos 7 Operating system. After choosing your Linux Distribution, it is important to know which command line-based text editor you are going to use.

The writer recommends installation of VIM (VI Modified) editor on your Linux Server. Installation is different in Centos and Ubuntu.

Vi is a popular and extensively used text manipulation editor among Linux system administrators. Although **Vi** is widely used, it is tough to learn for novice Linux users primarily because it operates in two different modes, namely the **insert mode** and the **command mode** . The insert mode is where users normally type text in a file, and command mode is where your keyboard input is interpreted as commands to perform on the text.

Setting up the Environment

Linux is a very flexible and customizable system. You can set the environment settings and variables that define how your shell is going to function. You can allow the system to automate something that you find yourself repeatedly setting or changing.

The Shell Profile

For any Linux user, there's a **~/.profile** file in the user's home directory. It is in this **.profile** file that you can personalize user settings attached to their profile. When the command-line interface is initialized the following steps happen,

- ***/etc/profile*** is executed first
- Then the ***/etc/bash.bashrc*** file if your default shell is bash is executed, then
- ***~/.profile*** , file is processed and
- Finally**, *~/.bashrc*** file is executed lastly.

The***~/.profile*** file is executed by any of the available shells, so it is important to put generic settings in this file. All bash-specific settings should go in the ***~/.bashrc*** file.

System administrators can set bash environment variables and aliases in the ***.bashrc*** file and even run commands. All the files listed under the user's home directory that are prefixed with a dot are called "**dotfiles** ."

Environment Variables

Linux allows the use of environment variables to change the way the Linux system functions. In the command line, you can set Environment variables on the terminal, or statically by adding an entry

into the**~/.bashrc** file. The bash shell has four customizable prompts, namely PS1, PS2, PS3, and PS4.

PS1 Prompt

PS1, also known as **Prompt String 1** , is the basic in-built customizable bash shell variable displayed before each bash command in Linux. Customizations done to the prompt include adding and modifying elements, changing background colors. Below is the default bash prompt.

Figure 1- PS1 Prompt

Figure 1 above shows PS1 prompt with **username** ppeters, **hostnamerad-srv.**

The most usual setting for PS1 is ***\u@\h:\w$*** whichis a display of the **username** , the **hostname** , and the **current directory** the user is working in.

The following line is another example of the PS1:

ppeters@rad-srv:/var/log$

The prompt above is displaying that the user has **username** ppeters and is logged into a server with hostname "**rad-srv** ," and the user is currently working in the **/var/log** folder.

PATH

The bash**PATH** environment variable is configured to direct the shell to the location of the most common programs, and user-created scripts that need to be executed. The default paths for most of the common Linux system commands is**/bin** , **/sbin** ,**/usr/bin** , and **/usr/sbin** . Users can build their own custom scripts in**/usr/local/bin** ,**$HOME/scripts** , and **$HOME/bin** . All user custom scripts' directory path has to be appended to the PATH so that it will be easy to locate them through the shell even when you are in a different folder: **PATH=${PATH}:${HOME}/bin**

Whenever you do not append the custom script path to the PATH environment variable, you may have to provide either an **absolute** or **relative** path to the command. As an example:

```
[ppeters@rad-srv ~]$ myscript.sh
-bash: myscript.sh: command not found
[ppeters@rad-srv ~]$ /home/ppeters/bin/myscript.sh
Hello World
[ppeters@rad-srv ~]$ cd /home/ppeters/bin
```

```
[ppeters@rad-srv bin]$ ./myscript.sh
```

Linux security best practices do not recommend putting a dot (.) in your PATH, especially prefixing your PATH with a **dot** .

Users are encouraged to avoid appending a colon at the beginning or end of the PATH, or a pair of colons separated by nothing, as that has the same effect as a single dot (.). Find below the example

PATH=$PATH :${ HOME}/bin

rather than:

PATH=${HOME}/bin:$PATH

File System Navigation

In Linux, everything is a file from programs, processes, devices, and directories. One of the most important command-line functions is navigating the Linux file and directory structure. We have a number of inbuilt bash commands which can be used to navigate through a Linux directory. Linux uses a Hierarchical File Structure which starts at the root directory **(/)** moving down to files or leaves. We basically have three major commands utilized in file navigation, which are

- **pwd** : This acronym stands for Print Working Directory. This command displays the current working directory that a user is in.

```
[root@rad-srv ~]# pwd
/root
```

cd: CD is a command to move or jump from one directory to the next. CD Stands for Change Directory.

```
[root@rad-srv ~]# cd /var/log
[root@rad-srv log]#
```

ls : The LS command is used to display contents of a directory.

```
[root@rad-srv sysconfig]# ls
anaconda console grub kdump network rdisc rsyslog snmptrapd sysstat.ioconf atd cpu power init
```

Listing Directory Contents

To list the files and directories in the current working directory, we use the **ls** command.

```
[ppeters@rad-srv ~]$ ls
Desktop Documents Music Pictures Public Templates Videos
```

Introduction to Directories

A directory is a file whose major function is to store file names and their related details. Directories or Folders in Linux, whether they are ordinary, special, or directory, contain all files.

Linux operating system uses the hierarchical structure to organize files and directories. The Directory tree is the name given to this structure in Linux. The directory tree starts with the single root node, the slash character (/), and every other folder falls under the root directory.

Linux operating system has three basic types of files, namely regular, directories, and special files. **Regular Files** are those files that contain text, data, or program code/scripts. **Directories** are a type of file also called folders that contain or store both special and regular files. Linux directories are the exact equivalent to folders in Windows. We also have what are called **Special Files** in Linux. These special files give us access to a number of hardware devices such as hard drives, keyboards, monitors, CD-ROM drives, USB flash devices, and Ethernet adapters.

Listing Files in Directories

System administrators in Linux do a lot of troubleshooting and use a number of tools. We have a tool to display a list of folders and the files stored in the current directory; we use the ls (listing) command:

```
[ppeters@rad-srv ~]$ ls
```

This ls command is used with a number of options such as the **-l** option, which gives more details about the listed files:

```
[ppeters@rad-srv ~]$ ls -l
total 8578999
drwxrwxr-x  2 ppetersppeters4698 Sept 2 09:59
-rw-rw-r--  1 ppetersppeters45371Sept 2 08:38 12.jpg
drwxr-xr-x  2 ppetersppeters5659 Sept 1  2019folder1
drwxr-xr-x  2 root   root        4096 Sept 1  2019folder2
[ppeters@rad-srv ~]$
```

The ls –l command gives us the long listing of the contents of the file or directory. The ls –l command displays more information, andso let's break it down more.

The first character to the far left represents the type of the file. Reading from the output above, we have character **d,** which represents a directory and – (hyphen) represents a regular file. Just next to the first character in the output we have the File Permissions column. There are nine characters in this column broken down into 3by3 fields, which represent user, group, and global file permissions. The nine characters are broken down into three character batches with each three-character batch having read, write, and execute permissions. In the output above the first **rwx** represents read, write, and execute permissions assigned to the owner of the file. The second **r-x** for folder 1 means that group ppeters has read and executed permissions but no write permissions to folder1.

The next or second column in the output is Number of Links field. For folder1, there are two links. The third column or field displays the owner of the displayed file. In our example above, the owner is ppeters. The fourth column is a display of the group to which the owner of the file belongs to. The fifth field is the Size column. This column specifies the size of the file in bytes. The sixth field or column is the **Last modified date & time. This field** specifies the date and time of the last modifications done to the file. The last column is the name of the file.

The output of the ls –l command shows the type of the file in the first character of the output result. From the example above, we have – and the d characters. The table below details all the types of files that we have in Linux and their associated symbols.

Serial.	Character& Description

1	- Ordinary file, such as a text file, binary file, or hard link.
2	**B** Special Block file. Represents a Block I/O device file such as a hard drive.
3	**C** Special Character file. Represents a raw file such as a physical hard drive.
4	**D** Represents a Directory file that displays a list ofcontained files and directories.
5	**L** This symbol represents a soft or Symbolic link file. These are Links on any ordinary file.
6	**P** This symbol represents named pipe. This is a mechanism for interprocess communications.
7	**s** This symbol is for Sockets used in interprocess communication.

Meta-Characters or Wildcards

We have characters in Linux that are useful when we are dealing with regular expressions. The following characters; * (star) and? (Question mark) are called wild card operators or Meta characters. The***** symbol is used to match zero (0) or more characters, and the question mark (**?**) is used to match with a single character.

Illustration

```
[ppeters@rad-srv ~]$ ls pp*.doc
```

Displays all the files, the names of which start with **pp** and end with **.doc** :

```
ppeters01.doc        ppeters02.doc        ppeters03.docppeters04.doc        ppeters05.doc
```

The example above shows that, ***** works as wildcard character that matches any character. To display all the files that end with just **.doc** , users can use the following command:

```
[ppeters@rad-srv ~]$ ls *.doc
```

Linux Hidden Files

We have invisible files in Linux, and they are known by the first character, which is the period character (.). In Linux, most programs like the shell use dot files to keep configuration details.

Examples of the common hidden files include:

- **.profile** - The Bourne Again Shell (Bash) start-up script

We can list all the invisible files, using the **ls** command by just specifying the **-a** option for example

```
[  ppeters@rad-srv ~]$ ls -a
.       .profile      docs     lib     peters_file
..      .rhosts       hosts    pub     users
[ppeters@rad-srv ~]$
```

- **The Single dot (.)** is a representation for the current working directory.
- **The Double dot (..)** is the representation for the parent folder.

File Creation using VI Command

When working on the command line interface, we can use the **vi** editor to create regular files on any Linux file system. Example command to create a file called ppeters_file:

```
[ppeters@rad-srv ~]$ vi ppeters_file
```

The command given above opens a file with the given filename ppeters_file. To get into **insert mode** , the **vi** mode in which to edit the file you press key **i** . Once the user is in edit mode, they can start typing their data or script in the file as in the following program:

```
This is P Peters's file. It was created today at 12 noon.
```

```
We are going to store this text in this file.
```

Once a user is done entering their text in the file, they follow the steps below:

- Enter the **esc** key to exit the edit mode.
- Press **Shift + ZZ** keys simultaneously to exit the file completely.

You will now have a file created with **filename ppeters-file** in the current working directory.

File modification in Linux

Vi editor is a tool that is utilized to modify existing files in a bash shell. To edit an existing file in Linux type the following command:

```
[ppeters@rad-srv ~]$ vi ppeters-file
```

Once this **ppeters_file** file is opened, you can get into insert mode by pressing the key **i** , and this enables you to modify the file. To move around the open file, the user needs to exit insert mode first by pressing **the esc** key. Having exited insert mode, the keystrokes below allows one to move up, down, left and right in a vi file.

- **l** this key allows one to move right.
- **h** this key gets you to move to the left.
- **k** this key allows one to move upside in the open file.
- **j** this key is fordownward movement in the **vi** document.
- The above keys are used to position the cursor on any part of the file or document you want to edit. When you get to the part of the file you want to work on, you press the **i** key to get into insert mode. When you finish editing the file, the user should press **Esc key** and then press the following keys simultaneously **Shift + ZZ** to save and exit the file completely.

Displaying File Contents using the Cat Command

Using the **cat command, which is** short for "concatenate" command allows the Linux user to view or display the contents of a file. Below we have an example to display or show the contents of the file ppeters_file we created above:

```
[ppeters@rad-srv ~]$ cat ppeters_file
```

```
This is P Peters’s file. It was created today at 12 noon.
We are going to store this text in this file.
[ppeters@rad-srv ~]$
```

The concatenate or cat command works with a number of options to modify our view. We can display line numbers in the file we are viewing by appending our cat command with**-b** command option as shown below:

```
[ppeters@rad-srv ~]$ cat -b ppeters_file
1 This is P Peters’s file. It was created today at 12 noon.
2 We are going to store this text in this file.
[ppeters@rad-srv ~]$
```

Counting Words in a File

In Linux, we have a command to count the number of words in a file. This command is called **wc** short for "word count." To enumerate the number of lines, characters, and words in a file, we use the **wc** command with a number of options. The example below illustrates the usage of the **wc** command:

```
[ppeters@rad-srv ~]$ wc /var/log/messages
2 14 1013 /var/log/messages
[ppeters@rad-srv ~]$
```

After running the **wc** command, the output has a number of columns. **The First Column** in the result is the representation of the total count of lines in the file ppeters_file. **The following Column2** is the display of the words counting the file. We then have **Column Three** , which is a representation of the total number of bytes in the file. This column displays the file's actual size. The **Fourth and last Column in the results** is the representation of the name of the file.

The wc command can be utilized to count the contents of a number of files. This is done through appending all the names of files to the wc command.

We have an example below to show the structure of the wc (word count) command on multiple files:

```
[ppeters@rad-srv ~]$ wc file_name1 file_name2 file_name3
```

Using the CP Command to Copy Files

Our operating system has a number of command-line tools to manipulate files and directories. To create a copy of a file or directory we the **cp** command. The cp abbreviation stands for copy and is used to make a copy of a file. The structure of the command is as follows:

```
[ppeters@rad-srv ~]$ cp source_file destination_file
[ppeters@rad-srv ~]$ cp source_file directory_name
[ppeters@rad-srv ~]$ cp source_file1 source_file2 source_fileN directory_name
[ppeters@rad-srv ~]$ cp [option] source_file destination_file
[ppeters@rad-srv ~]$ cp source_name destination_name
```

The bash statement below is illustratingthe cp command copying file **ppeters_file** to **ppeters_copy.**

```
[ppeters@rad-srv ~]$cp ppeters_file ppeters_copy
```

After copying the command creates an additional file, there is going to be one more file **ppeters_copy** in your current working directory. This file is going to bathe exact replica of the original file **ppeters_file** .

File Renaming using the MV Command

There is built-in functionality in Linux to change the name and location of files and directories. This is called the renaming function; we utilize the **mv** (move) command to rename files and also change their location in the file system. The illustration of the mv command syntax is highlighted below:

```
[ppeters@rad-srv ~]$ mv old_name_file new_name_file
```

The following mv command will be executed to rename the existing file **ppeters_file** to **newppeters_file** .

```
[ppeters@rad-srv ~]$ mv ppeters_file newppeters_file
[ppeters@rad-srv ~]$
```

The **mv** command will rename an existing file and completely change it into the new file. In this case, you will find the only **newppeters_file** in your current directory.

File Deletion using the RM Command

The Linux command-line interface has the **rm** command, which is the command-line program to delete a file. The command rm has the following syntax:

```
[ppeters@rad-srv ~]$ rm ppeters_file
```

Generally, Linux files may contain some important data so we should always follow best practices in erasing them. The best practice is to use the interactive mode while using this **rm** command. It is recommended that the **rm** command be used with the**–i option.**

Below is the illustration of the **rm command** to remove a file completely. We are removing an existing file called **filename** .

```
[ppeters@rad-srv ~]$ rm filename
[ppeters@rad-srv ~]$
```

Through the rm command, we are capable of deleting multiple files at once by just appending the files that need to be deleted to the rm command.

```
[ppeters@rad-srv ~]$ rm file_name01 file_name02 file_name03
[ppeters@rad-srv ~]$
```

Absolute and Relative Paths in Linux

Files in a Linux operating system are organized within a hierarchical file system tree with the root directory (/) at the top of the inverted tree. The path to any file within this Linux directory hierarchy is described by file's path.

Components of the file path are separated by / (forward slash). We have absolute and relative path names in Linux. Absolute paths begin with the forward-slash / in its name.

Below we have some examples of absolute paths.

```
[ppeters@rad-srv ~]$ /etc/systemd
[ppeters@rad-srv ~]$ /var/log/messages
[ppeters@rad-srv ~]$ /usr/local
```

Linux paths can also be relative to the user's current working directory. All Relative pathnames do not begin with /. Relative to a user with username **ppeters'** home directory, example pathnames may be

```
[ppeters@rad-srv ~]$ ppeters_files/newfile
```

When a user wants to determine their location within the filesystem hierarchy at any time, they enter the command **pwd** to print their current working directory or location:

```
[ppeters@rad-srv ~]$pwd
/home/ppeters
[ppeters@rad-srv ~]$
```

Listing Contents of Directories

The bash ls command is used to display all the files in a directory. The illustrationbelow shows the ls command syntax and structure:

```
[ppeters@rad-srv ~]$ ls directory_name
```

The illustration below is a listing of all the files contained under the **/usr/bin/** directory:

```
[ppeters@rad-srv ~]$ ls /usr/bin
wc
[ppeters@rad-srv ~]$
```

Creating Directories using mkdir

Linux command line has built-in bash commands that allow users to create file directories. The mkdir command is the tool used to create these folders, as illustrated below:

```
[ppeters@rad-srv ~]$ mkdir directory_name
```

Looking at the command above the directory directory_name should be the absolute or relative path of the directory to be created. Mkdir illustration:

```
[ppeters@rad-srv ~]$ mkdir mydir_name
```

The bash command above creates the folder called **mydir_name** in the user's current working directory. This is creating a directory using the relative path. Absolute pathname example:

```
[ppeters@rad-srv ~]$ mkdir /tmp/mydir_peters
```

The mkdir command above creates the directory **mydir_peters** in the **/tmp** directory. The command **mkdir** displays no output for successful creation or execution of the command.

You may create multiple directories on the command line, using **mkdir** by just listing each of the directories. For example:

```
[ppeters@rad-srv ~]$ mkdir docs_dir public_dir
```

The statement above creates the docs_dir and public_dir directories under the current working directory.

Creating Parent Directories

Linux command line gives system administrators the ability to create different directories from parent directories to the child folders. When you want to create a child directory, but the parent directory is not there, you use mkdir with –p option or switch to first create the parent folder before creating the ensuing child directory. When the –p option is not specified the **mkdir command** displays the following error message:

```
[ppeters@rad-srv ~]$ mkdir /tmp/ppeters/test
mkdir: Failed to make directory "/tmp/ppeters/test";
No such file or directory
```

Below is an example illustrating how to create parent directories in Linux:

```
[ppeters@rad-srv ~]$ mkdir -p /tmp/ppeters/test
```

The above command creates all the required directories, including the parent.

Removing Directories

In Linux you do not delete directories that contain files and other directories using the **rm** command but can be deleted using the **rmdir** command as shown below:

```
[ppeters@rad-srv ~]$ rmdir dir_name
```

Note : Whenever you want to delete a directory, you have to ensure that it is empty, meaning there are no other folders and files inside.

It is possible to delete many directories in one go as follows:

```
[ppeters@rad-srv~]$rmdir dir_name1 dir_name2 dir_name3
[ppeters@rad-srv~]$
```

The command deletes the directories dir_name1, dir_name2, and dir_name3, only if they don't contain any files or folders. There is no console output when the **rmdir** command executes successfully.

Renaming Directories

The bash **mv (move)** program is used to rename directories. The syntax is shown below:

```
[ppeters@rad-srv ~]$ mv olddirname newdirname
[ppeters@rad-srv ~]$
```

You can change a directory name from **mydirname** to **yourdirname** as follows:

```
[ppeters@rad-srv ~]$ mv mydirname yourdirname
[ppeters@rad-srv ~]$
```

Linux Shell Variables

In this section, I will be discussing Bash parameters, also known as variables in Linux. A bash parameter is a string of characters assigned a value. These parameters are assigned either a number, text, filename as values.

A parameter essentially is just pointer or reference to the actual data stored in the system. Through the bash, shell users are able to create, delete, and assign variables.

Linux Parameter/Variable Names

In Linux variable names contain letters of the alphabet (a to z or A to Z), positive digits (0 to 9) and the special character accepted for variable names is the underscore character (_).

By convention, bash shell variables are declared in UPPERCASE.

For example:

```
_JOHN
TOKEN_A
VAR_NAME01
VAR_NAME02
```

Below are examples of invalid variable names:

```
2_VAR
-VARIABLE_NAME
VARNAME1-VARNAME2
VARNAME_A!
```

We cannot use special characters such as **!** , ***** , or **–** since these characters have a special meaning for the shell.

Defining Shell Variables

Example bash variable definition:

var_name=var_value

Example

```
USER_NAME="James Jones"
```

The above example declares the variable USER_NAME and assigns the value "James Jones" to it. Variables of this type are called **scalar variables** . Scalar variables hold only one value at a time.

Bash Shell variables allow users to store any value. For example:

```
VARIABLE1="James Jones"

VARIABLE2=100
```

Accessing Variable Values

Prefixing the variable name with the dollar sign (**$**) is the only way to access variable values.

The bash script below will access the value of declared variable NAME and print it on STDOUT −

```
#!/bin/sh
```

```
NAME="James Jones"

echo $NAME
```

The above script will produce the following value −

James Jones

Variable Types

Bash programming introduces us to three types of variables. In Bash, we have **local, environment, and shell variables.** These variables are essentially a temporary store of data or value. A local variable is a store of value that is present within the current instance of the bash shell. This variable and its values are unavailable to any other programs running in the operating system. The local variable setting is done at the command prompt.

The next type of variable we have is the **Environment Variable** . This is a variable or parameter that is available to all child processes of the bash shell. Most programs require environment parameters for them to work correctly. Shell scripts define only the parameters necessary for that instance.

Lastly, we have **Linux Shell Parameters or Variables** . Shell variables are special bash shell parameters set by the shell and are required by bash for it to function correctly. These parameters are either environment variables or are local variables.

As an example, the process ID number, also known as the **PID** , is represented by the **$** character:

```
[ppeters@rad-srv~]$ echo $$
13457
```

As illustrated above, the **$$** command displays the PID of the current bash shell: 13457

Below is a table displaying the special variables that can be used in bash shell scripts:

#	The Variable &the Description
1	**$0** This variable displays the current script's filename.
2	**$n** These parametersare dependent on thenumber of arguments with which a script was called. From the variable **n** is a non-negative number that represents the current argument's position (the initial argument is always $1, the argument that follows is $2 and so on).
3	**$#** This displays the argument count supplied to the bash script.
4	**$*** In linux we double quote all arguments. If we input two arguments to a bash script, then $* is equal to $1 $2.
5	**$@** We individually double quote all arguments. If a bash script gets two arguments, $@ is results in $1 $2.

6	**$?** The above variable is the exit status of the command executed last.
7	**$$** The variable above shows the process ID number of the current bash shell. For all bash scripts, $$ details the process ID under which they execute.
8	**$!** This variable displays the processID number of the background command that was lastly executed.

Linux Terminal Arguments

The Linux command-line interface arguments like $1, $2,...$9 are called positional variables, the $0 variable points to the command we are executing, the program, the bash shell script, or the bash function and the special variables here $1, $2, ...$9 are passed as the arguments to the bash command.

The script below uses a number of special variables that are related to the command line interface:

```
#!/bin/sh



echo "File Name: $0"

echo "Initial Argument : $1"

echo "Second Argument : $2"

echo "Double Quoted Values: $@"

echo "Double Quoted Values: $*"

echo "The Total Count of our Arguments is: $#"
```

Displayed below is an example run for the above script −

```
[ppeters@rad-srv ~]$./peters_test.sh James Jones

File Name : ./peters_test.sh

Initial Argument : James

Second Argument : Jones

Double Quoted Values: James Jones

Double Quoted Values: James Jones

Total Count of our Arguments is: 2
```

Bash Special Variables $* and $@

Linux bash has some special variables that are used to access all the terminal arguments at once. The variables **$*** and **$@** will output the same result same unless you enclose them in"" double quotation marks.

These two variables specify terminal parameters. The special variable "$*" takes the whole set of arguments as a single argument with spaces in between whilst the special variable "$@" takes the whole list and splits it into disparate arguments.

To process an unknown count of command-line arguments with either the $* or $@ special variables we can use bash script below:

```
#!/bin/sh


for TOKEN in $*
do
    echo $TOKEN
done
```

Below is the example run of the bash script above:

```
[ppeters@rad-srv ~]$./peters_test.sh James Jonesis Old
James
Jones
is
Old
```

Special Note : The **do...done** is a looping structure that will be detailed later on.

Bash Shell Exit Status

The special variable, **$?** represents the exit status of a previously executed bash command.

The numerical value returned by all completed commands is termed the Exit status. All successful commands return 0 exit status as a rule, and one exit status is returned for all unsuccessful commands.

For specific reasons, some commands return extra exit statuses. Some commands, for instance, distinguish between types of errors and return different exit statuses based on the particular type of failure.

Below is an example of a successful command:

```
[ppeters@rad-srv ~]$./peters_test.sh James Jones
File Name : ./peters_test.sh
Initial Parameter : James
Second Parameter : Jones
Double Quoted Values: James Jones
Double Quoted Values: James Jones
Total Count of Parameters : 2
[ppeters@rad-srv ~]$./echo $?
0
```

```
[ppeters@rad-srv ~]$
```

Bash Array Values

In this part of the book, I will be discussing how to use bash shell arrays. There are bash shell variables that are capable of holding single values. These bash shell variables are termed scalars.

The bash shell promotes a distinctive variable type known as an array variable. This can hold various values simultaneously. Arrays provide a grouping technique for a set of variables. You can use a single array variable rather than creating new names for each and every variable that is needed to store all the other variables.

While naming arrays, all the naming laws mentioned for Shell variables would apply.

Defining Array Values

The distinction between an array variable and a scalar variable can be described as follows. Suppose you are attempting to portray as a collection of variables the names of different learners. Each of the names is a scalar variable.

```
ARR_NAME01="James”

ARR_NAME02="Peter"

ARR_NAME03="Martin"

ARR_NAME04="Samson"

ARR_NAME05="Linda"
```

To store all the names listed above, we can use a single array. The easiest way to create an array variable is to follow the example below. This enables one of its indices to be assigned a value.

```
arrayname[index]=value
```

From above the name of the array is *arrayname* , and the *index* is the index of an item in the array that has to be set, and value is what you want to set for that item. The example below:

```
ARR_NAME[0]="James "

ARR_NAME[1]="Peter"

ARR_NAME[2]="Martin"

ARR_NAME[3]="Samson"

ARR_NAME[4]="Linda"
```

The syntax of an array initialization, using the **bash** shell, is shown below:

```
arrayname=(value01 ... valueN)
```

How to Access an Array's Values

After setting an array variable, you can access it, as shown below:

```
${arrayname[index]}
```

The name of the array above is *arrayname* , and the *index* is the indice of the value to be processed. Below is the example to access array values:

```
#!/bin/sh


ARR_NAME[0]="James "
ARR_NAME[1]="Peter"
ARR_NAME[2]="Martin"
ARR_NAME[3]="Samson"
ARR_NAME[4]="Linda"
echo "Indice One: ${ARR_NAME[0]}"
echo "Indice Two: ${ARR_NAME[1]}"
```

Below is the result of executing the bash script above:

```
[ppeters@rad-srv ~]$./peters_test.sh
Index One: James
Index Two: Peter
```

Below are the alternatives ways to access arrays:

```
${arrayname[*]}
${arrayname[@]}
```

From above the name of the array is **arrayname** . Below is the bash script to illustrate how to access arrays:

```
#!/bin/sh


ARR_NAME[0]="James "
ARR_NAME[1]="Peter"
ARR_NAME[2]="Martin"
ARR_NAME[3]="Samson"
ARR_NAME[4]="Linda"
echo "Our Method One: ${ARR_NAME[*]}"
echo "Our Method Two: ${ARR_NAME[@]}"
```

The result of the bash script:

```
[ppeters@rad-srv ~]$./peters_test.sh
Our Method One: James PeterMartinSamsonLinda
```

```
Our Method Two: James PeterMartinSamsonLinda
```

Shells in Linux particularly bash supports a number of operators. This section will discuss in detail the different operators.

In Linux bash shell there are arithmetic, Relational, Boolean, String s and File Test Operators.

Originally the shell did not have a mechanism to perform arithmetic operations; it used external programs, like **awk** or **expr** .

The example below illustrates how to add two digits:

```
#!/bin/sh


val_add=`expr 5 + 9`

echo "The Total added value is: $val_add"
```

The result of the script above:

```
The Total added value is: 14
```

There are a number of points to be considered while performing addition −

- We should have spaces between the operators and the expressions. For instance, 6+9 is incorrect; it should be written as 6 + 9.
- The arithmetic addition expression should be put between back ticks**‘.‘**

Bash Arithmetic Operators

The table below shows all the arithmetic operators.

Assuming that the variable **x** holds eight and variable **y** holds 15:

The table below shows all the bash arithmetic operators.

Operator	Description	Example
+ (Addition)	This operator Adds the values on both side of the operator together	`expr $x + $y` results in23
- (Subtraction)	This operator minuses the right hand value from the left hand value	`expr $x - $y` results in-7
*** (Multiplication)**	This operator finds the product of the operands on either side of the * operator	`expr $x \* $y` results in 120
/ (Division)	This operator Divides the left hand value by the right hand value	`expr $y / $x` results in 1
% (Modulus)	This operator Divides left hand value by the right hand value and returns the remainder	`expr $y % $x` will give 7
= (Assignment)	This operator sets the value of the right side operand to the value of the left side operand	y = $b would set value of y into x
== (Equality)	This operator Compares the value of two digits, and if	[$x == $y] gives

	both numbers have the same value then returns result true.	result false.
!=(Not Equality)	This operator Compares just two numbers, and if both numbers are different then returns true.	[$x != $y] gives result as true.

All conditional expressions should be enclosed in square braces as shown below

[$x == $y] is correct whereas, **[$x==$y]** is incorrect.

Long integers are used in all arithmetic calculations.

Bash Relational Operators

Relational operators are supported in a bash shell, especially those that are specific to numbers. Relational operators do not work on strings.

As an illustration, the following operators function to check the relation between 15 and 40 as well as in-between "15" and "40" but not in-between "fifteen" and "forty."

Assuming that variable **x** holds 15 and variable **y** holds 40:

Operator	Description	Example
-eq	This operator determinesif the givenvalues are equal or not; if they are equal, then the condition is true.	[$x -eq $y] is false.
-ne	This operator determines if the values of the two operands areequivalent; if the operands are not equivalent, then the condition is true.	[$x -ne $y] is true.
-gt	This operator is used to determine if the value in the left operand is bigger than the value of operand to the right; if value is greater, then our condition returns true.	[$x -gt $y] is false.
-lt	This operator Determines if the value of the left operand is less than the value of the right operand; if true, then the condition is true.	[$x -lt $y] is true.
-ge	This operator determines if the value of the left operand is greater than or equal to the value of the right operand; if affirmative, then the condition becomes true.	[$x -ge $y] is false.
-le	This operator tries to find out if the value of the left operand is less than or equal to the value of the right operand; if affirmative, then the condition becomes true.	[$x -le $y] is true.

In bash scripting all the conditional statements should be put inside spaced square brackets. For instance, this illustration [**$x <= $y]** is the correct format whereas, the following illustration **[$x <= $y]** is wrong.

Bash Boolean Operators

Bash shell supports a number of Boolean operators as shown in the table below.

Assuming variable **x** holds 20 and variable **y** holds 40:

Boolean Operators Table

Operator	Description	Example
!	This operator is called the logical negation. It inverts a condition into its opposite value.	[! false] is true.
-o	This operator is called the logical **OR** . If one of the two operands' value is true, then the condition is true.	[$x -lt 40 -o $y -gt 100] is true.
-a	This operator is called the logical **AND** . If both operands are true, then the condition returns true otherwise it is false.	[$x -lt 40 -a $y -gt 100] is false.

Bash String Operators

The Bash shell supports the following String Operators.

Assuming variable **x** holds "def" and variable **y** holds "hij":

Operator	Description	Example
=	This is the equality operator used to check if the two operands' values are the same that is equal; if they are equal, then the condition returns true.	[$x = $y] is false.
!=	This NOT EQUAL operator is used to determine whether the value of our left operand is equal or not to the right operand; if the two operands' values are not equal then the condition returns true.	[$x != $y] is true.
-z	This ZERO operator is used to check if the given string operand's size is zero; if the string has zero length, then it the conditional returns true.	[-z $x] is false.
-n	This NON-ZERO operator checks if the input string's operand size is non-zero; if the string is non-zero length, then the condition returns true.	[-n $x] is true.
Str	This STR operator is used to determine if the input string **str** is not an empty string; if string entered or read is empty, then the condition returns false.	[$x] is true.

Bash File Test Operators

There are a number of operators that are used to test various properties associated with files in Linux.

Assuming a variable **filename** holds an existing file name "peters_test" which is 200 bytes in size and has the following permissions **read** , **write** and **execute** :

Operator	Description	Example
-b filename	This block file operator is used to findout if thefilename is a special block file; if it is, then our condition returns true.	[-b $filename] is not true.
-c filename	This character operator is used to findout ifthe filename is a special character file; if it is true, then the condition returns true.	[-c $filename] is not true.
-d filename	This directory operator is used to determine if filename is a directory or not; if it is a directory, then our condition returns true.	[-d $filename] is false.
-f filename	This regular file operator is used to find out if filename is our regular linux file as opposed to a regular directory or another special file; if the filename is a regular file, then our condition returns true.	[-f $filename] is not false.
-g filename	This SGID operator is used to find out if our filename has the SGID bit turned on. SGIDstands for set group ID (SGID); if the bit is set, then our condition returns true.	[-g $filename] is not true.
-k filename	This sticky bit operator is used to findout ifour file called filename has their sticky bit set; if sticky bit is set, then our condition returns true.	[-k $filename] is not true.
-p filename	This pipe operator is used to find out if the file with the name filename is a named pipe; if the file called filename is indeed a pipe, then our condition returns true.	[-p $filename] is not true.
-t filename	This operator is used to findout ifour file descriptor is open and also if it is associated with a linux terminal; if answer is yes, then our condition returns true.	[-t $filename] is not true.
-u filename	This SUID operator is used to findout ifthe file called filename has the SUID bit set – SUID stands for set User ID; if bit is set, then our condition returns true.	[-u $filename] is not true.
-r filename	This readable operator is used to findout ifthe file called filename is readable; if it readable, then our condition returns true.	[-r $filename] is not false.
-w filename	This writable operator is used to findout ifthe file named filename is writable; if it is writable, then our condition returns true.	[-w $filename] is not false.
-x filename	This executable operator is used to findout ifthe file named filename is an executable file; if it is, then our condition returns true.	[-x $filename] is not false.

-s filename	This operator finds out iffilename has size greater than 0; if yes, then condition becomes true.	[-s $filename] is not false.
-e filename	This operator finds out iffilename exists; is true even if file is a directory but exists.	[-e $filename] is not false.

Bash Shell Decision Making

In this section, I am going to discuss bash shell decision-making in Linux. Given two paths in bash shell scripting, there are situations where you need to adopt a single path out of the choice of two. Conditional statements are used that allow your program to make the right decisions and perform the correct actions.

Linux Bash Shell promotes conditional statements based on distinct actions that are used to conduct distinct actions. We have mainly two bash statements used in making decisions:

- The bash **if...else** decision statement
- The bash **case...esac** decision statement

The Bash if...else Decision Statements

For decision making in bash, we use If else which can be used to choose an option from a number of given options.

Linux Bash Shell uses the following forms of **if…else** statement:

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

In bash, a number of the**if statements** use relational operators to check relations.

Bash Case Statement

The bash case statement, also known as the case…esac statement in bash scripting is a conditional flow statement used to replace nested if-elif statements.

The case-esac statement is a solid alternative to the multi-level if-then-else-fi conditional flow statements. This method of conditional flow enables the user to match a number of values against one variable. Case statements handle multi-branch conditions more efficiently than nested if…elif statements.

We have the case…esac illustration below:

```
case $var_name in
condition01)
    command01
    …
    …
    command0N
    ;;
```

```
condition02)
    command01
    …
    …
    command
    ;;
    *)
esac
```

The **case...esac** statement in bash shell looks very similar to the **switch...case** statement used in traditional programming languages like **Java** or **C++** and **PERL** .

Bash Loop Constructs

Bash scripting utilizes similar programming constructs used in our traditional programming languages like Java and C++. Bash uses Loops to take a series of commands and keep executing them until a certain condition. Looping constructs are utilized mainly to automate repetitive and routine tasks in Linux. There are mainly three loop structures used in bash scripting namely; while, until and for loops.

Nesting Loops in Bash

Bash allows the use of loops inside other loops, a process called nesting. In essence, a user can put a while loop inside another while loop.

While Loop Nesting

We can place a while loop inside another while loop when we are creating our bash scripts as illustrated below.

Syntax

```
while commandname01 ; # this is the outer loop
do
   Statement(s) to be executed if commandname01 holds true

while commandname02 ; # this is the inner loop
do
      Statement(s) to be executed if commandname02 holds true
done

Here put Statement(s) to be executed if commandname01 holds true
done
```

Example:

This is an example of nested loop. Let's add another countdown loop inside the loop that is to count up to seven:

```
#!/bin/sh
```

```
a=0
while["$x"-lt 8]# this is the outer loop loop1
do
   b="$x"
while["$y"-ge 0]# this is the inner loop loop2
do
       echo -n "$y "
       b=`expr $y - 1`
done
   echo
   a=`expr $x + 1`
done
```

This script will produce the result below. Please note how **echo -n** functions here. The **-n** option ensures that the echo command does not print the command line new line symbol.

Bash Loop Control

The upcoming section is a discussion of bash control structures for looping for example break and continue statements. We have looked at creating loops and working with loops to automate different jobs. There are instances when you should exit a loop or jump over some parts of the loop. At such instances, we use the break and continue statements to control looping.

The Break Statement

The Linux break statement is a bash shell construct that is used to stop an entire conditional loop from execution after finishing the execution of all code units leading up to the break statement. When the loop finishes, it steps through to the other sections of the code.

Break Statement Syntax

Break statement example

```
break
```

We use the example above to exit from nested loops.

```
break n
```

Above the argument, **n** represents the **n** th enclosing loop to be exited from.

Break Statement Example

Below is an example that shows a loop terminating as soon as **x** becomes 8 −

```
#!/bin/sh
x=0
while[ $x-lt 10]
do
   echo $x
if[ $x-eq 8]
then
break
fi
   a=`expr $x + 1`
done
```

We have the nested loop example below. The bash script below shows us that the program code exits out of the first and second loops if **varname1 equals 13** and **varname2 equals 10** −

```
#!/bin/sh


for varname1 in123 4 5 6 7
do
for varname2 in09
do
if[ $varname1 -eq 13-a $varname2 -eq 10]
then
break13
else
          echo "$varname1 $varname2"
fi
done
done
```

Bash Continue Statement

Bash looping control continue statement is the same as the break statement. The continue statement is used to exit the current code iteration of a for, until, while and select loop whereas the break statement

is used to exit the whole loop. When continue is used in a for loop, the variable gets the value of the next element.

The bash continue statement is utilized when an error has occurred within a loop, but the user still needs to execute the next code iteration.

The Continue Statement Syntax

continue

The continue command above just jumps the current iteration within the same loop and executes the next iteration.

We pass integer arguments to the continue statement to enable jumping of commands to the next iteration inside the nested loops just like with the break statement.

continue N

The continue command above shows **N** , which represents the **Nth** enclosing loop to continue from. The continue N exits all remaining iterations at the loop level and continues at the loop N levels above.

Example

The example below shows that the **continue** statement exits from the if…then statement and begins to process the next echo statement:

```
#!/bin/sh


NUMBERS="8 9 10 11 12 13 14"


for NUMBERSin $NUMBERS

do

Z=`expr $NUMBER % 2`

if[ $Z-eq 0]

then

      echo "Our result is an even number!!"

continue

fi

   echo "Our result isan odd number"

done
```

Bash Variable Substitution

Ser	Escape & Description

ial	
1	\\ The Backslash escape sequence
2	**\a** The alert (BEL) escape sequence
3	**\b** The Backspace escape sequence
4	**\c** This is the suppress trailing newline escape sequence
5	**\f** This is the form feed escape sequence
6	**\n** This is the new line escape sequence
7	**\r** This is the carriage return escape sequence
8	**\t** This is the horizontal tab escape sequence
9	**\v** This is the vertical tab escape sequence

When it meets an expression containing one or more special characters, the shell performs the substitution.

Substitution Example

The substitution illustration below shows that the variable is assigned a value, and the value is the one that is printed on the terminal. This symbol is called the new line character, which essentially places the cursor on the next line after the execution of the command **"\n."**

```
#!/bin/sh



x=20

echo -e "The variable X stores the value $x \n"
```

The script above will show us the result below. From the echo statement above we have **-e** switch which enables bash shell to read and understand the backslash escape symbol. After executing the script above, our result is as shown below.

```
root@bakyrie:~# vim variable.sh
```

```
root@bakyrie:~# chmod u+x variable.sh
```

```
root@bakyrie:~# bash variable.sh
```

```
The variable X stores the value 20


root@bakyrie:~#_
```

When we remove the –e switch from our echo command, we the \n backspace character interpreted as a string and printed with the rest of the statement.

```
root@bakyrie:~# bash variable.sh
The variable X stores the value 20 \n
root@bakyrie:~#
```

The echo printing command is used with a varied number of escape sequences:

We have the –e switch that is used to help bash shell in interpreting backspace escape characters. We also have the –n switch which is utilized to disable the insertion of the new line when a statement or string is printed to the terminal screen.

Bash Command Substitution

Bash shell Command substitution is the feature that enables the shell to process a command and has resulting output of that command to substitute the text of that command itself, or the output is sent back to the terminal as an argument to another command.

Bash Command substitution is the process by which bash executes a set of instructions and then substitutes the commands with that output from processing.

Substitution Structure

The structure of the substitution command is illustrated below.

```
`command`
```

The backquote character is used when performing command substitutions.

Example

Command substitution is usually used to assign command output to a variable. The examples below show the substitution of the command:

```
#!/bin/sh


DATE=`date`

echo "TheDate today is $DATE"


USERS=`who | wc -l`

echo "The number of users who are logged inis $USERS"
```

```
UP_TIME=`date ; uptime`

echo "The system Uptime is $UP_TIME"
```

The result of the execution of the above script:

```
Date is Thu Sep  5 03:59:57 CAT 2019

The number of users who are Logged in is 1

The system Uptime is Thu Sep  5 03:59:57 CAT 2019

03:59:57 up 10 days, 16:30,  2 user,  load avg: 0.43, 0.47, 0.55
```

Variable Substitution

Variable substitution allows the shell programmer to manipulate the state-based value of a variable.

Table of all the possible substitutions −

Serial.	Form & Description
1	**${var_sub}** This allows the Substitution of the value of *var_sub* .
2	**${var_sub:-word_sub}** If our *var_sub* is either null or unset, then the variable *word_sub* is substituted for variable **var_sub** . The value of *var_sub* does not change.
3	**${var_sub:=word_sub}** If variable *var_sub* is either null or unset, the *var_sub* is set to the value of **word_sub** .
4	**${var_sub:?message}** If variable *var_sub* is null or unset, *message* is printed to standard error. This operator finds out that variables are set correctly.
5	**${var_sub:+word_sub}** If *va_sub* is set, *word_sub* is substituted for var_sub. The value of *var_sub* does not change.

Example

Following is the example to show various states of the above substitution −

```
#!/bin/sh


echo ${var_sub:-"This var_sub Variable is unset"}
```

```
echo "1 –The Value of variable var_sub is ${var_sub}"


echo ${var_sub:="This var_sub Variable is unset"}
echo "2 –The Value of variable var_sub is ${var_sub}"


unset var_sub
echo ${var_sub:+"This is the default variable value"}
echo "3 –The Value of variable var_sub is $var_sub"


var_sub="The Prefix"
echo ${var_sub:+"This is the default variable value"}
echo "4 –The Value of the variable var_sub is $var_sub"


echo ${var_sub:?"May you Please Print this message"}
echo "5 –The Value of the variable var_sub is ${var_sub}"
```

Upon execution, you will receive the following result:

```
Variable is not set
1 –The Value of variable var_sub is
This var_sub Variable is unset
2 –The Value of the variable var_sub is This var_sub Variable is unset

3 –The Value of variable var_sub is
This is the default variable value
4 –The Value of the variable var_sub is The Prefix
The Prefix
5 –The Value of the variable var_sub is The Prefix
```

Understanding Shell Building Blocks

Introduction to Bash Shell syntax

When a user enters their input using the keyboard, the bash shell reads the input string and splits it down into words and operators, using the quoting rules to define each character of input.

When the shell finishes reading the string and breaking it down into words and operators, bash shell then transforms the broken down words and operators into commands and other bash constructs,

which display an exit status used for processing. The bash shell works through the input string as detailed below.

- Initially, the bash shell reads the input from the user custom created file called a bash script or the shell reads an input stream from the users' keyboard input.
- The shell breaks up the input stream into words and operators, using the quoting rules. The word*s and operators are split up using wild card operators* . We then have Alias expansion done.
- Simple and compound bash commands are created from the split up tokens after the shell analysis of the words and operators.
- The expanded tokens are split up into a list of filenames, commands, options, and arguments through a number of shell expansions.
- We then have input/output redirection. We then have redirection operands and their operators taken off the parameter list.
- The shell then executes the bash commands.
- The shell may wait for the completion of command executions before it collects its exit status.

Understanding Shell Commands

Bash shell commands such as *mkdir dir1 dir2* are made up of the shell command, and arguments, separated by spaces. In the mkdir example, there the command is mkdir, and the arguments are dir1 and dir2, respectively.

It is easy for users to create complex shell commands by simply adding together elementary commands organized in a number of ways including using a pipeline where the output of one basic command is directed to be the input of a second command, using conditional statements and loops. For example **ls | less**

Working with Bash Shell Functions

Bash shell functions are a number of commands grouped together processing and can be called a number of times. Shell functions are executed in a similar way to elementary commands. Functions help in making bash scripts more readable. Functions are executed by calling their name. When the function name of a shell is called as a basic command, the listing of commands in the set is processed.

A look at Shell Variables

The Bash shell variable or parameter is an entity that stores string, integer, and array values and is referenced by either a name, a number or a special character. All parameters that are referenced by name are called variables. We have three types of these parameters, namely positional, special parameters, and variables. A variable can be either a name, a digit, or any other special value.

Bash shell variables are parameters that store a name. All bash shell variables have a value, and they also have zero or more attributes. In bash scripting, we use the declare built-in command to create a variable. If we do not assign a value to the variable, bash assigns the null string

Shell Variable Expansions

Variable or parameter expansion is the process to get the value from the referenced entity, for example, expanding a variable to display its value. Expansion is the ability to access and manipulate stored values of variables and parameters. Elementary expansion is done by prefixing the variable name with the **$** character.

The shell variable expansion is performed after the command strings split into word and operator tokens.

Chapter Two: Shell Input and Output

In this chapter, I will be discussing the Shell input/output redirections in detail. Bash commands receive their input from the terminal in Linux systems and return the resulting output to your terminal. Usually, a command reads its input from the standard input, which is your default terminal. Similarly, a command types its output to normal output, which is your default terminal again.

What is Bash Redirection?

Redirection is the process of capturing output from a script file, command, and a regular file.

Redirection is perfumed through the use of some expressions which are regular or wild card symbols. To redirect the output into a normal **file,** we utilize shell Meta character called angle **brackets** '<', '>' or to capture output to a **program we use** the shell Meta character called the **pipe** symbol '|.'

Whenever bash shell commands are processed, there is always input or output redirection through reading and interpretation of special notations. Redirections enable the FD or file descriptors of the bash commands to open and close. Through redirections, file descriptors refer to various files and may change files from which we read the commands. Redirections in current bash shell processing settings are used to change file descriptors. Input and output redirections process from left to right in the order they appear.

Whenever you see the redirection statement with the **less than operator sign** "<" and no file descriptor number know that we are referring to the standard input (file descriptor 0). Whenever we use the **greater than sign** ">" and the file handler number is omitted, then we are referring standard output (file descriptor 1).

Users have to note that order is very important in a redirection.

Redirection Standard Output

Bash command-line interface, also known as the terminal has a built-in result console called the standard output. The term standard out abbreviated STDOUT is the regular console on the command line where results of command executions display. Linux bash shell regularly monitors this standard output location for any new output. Whenever the fresh output is found, the console prints it out to the console screen, for users to see the output.

Redirection Standard Input

The STDIN read standard input is the default place on the bash shell where commands listen for input streams of data. As an example, when a user types cd command without

parameters, the stdin will listen for input from the user, displaying what you punch in on your keyboard.

Redirection Using Pipe

In bash, shell the Pipes are setup to connect output result from one command to another command as input. Piping is done by separating the two statements with the pipe symbol (|). Below is the illustration of this piping:

```
[ppeters@rad-srv ~]$echo"They are all here"
They are all here
[ppeters@rad-srv ~]$echo"They are allhere" | sed"s/They/We/"
We are all here
```

The statement, echo "They are all here" displays the result. They are all here to the STDOUT screen. Now when we pipe the result to sed "s/They/We/," sed receives that output as its input and substitutes **"They"** with **"We,"** then prints out that result to stdout.

The bash shell only shows the final result after it's gone through and got processed by the sed utility, and prints the result from the piped statement to the STDOUT console screen.

If sed displays its result to STDOUT screen, it is possible to pipe one sed result to another sed easily.

```
[ppeters@rad-srv ~]$echo"Hello world"| sed"s/world/people/" | sed"s/Hello/Greetings/"
Greetings people
```

The above, example shows that echo is connected to sed, then the result of this is also connected to another sed. Pipes are good for making the output from one command and making it the input in another command. Pipes are a key part of the Linux philosophy of "miniature sharp tools": we can chain commands together with pipes, this means that each command just needs to do exactly one thing and then pass the result on to another command that receives it as input.

Standard Error

The standard error ("stderr") is similar to standard output and standard input, but the major difference is that stderr is where error messages are taken to. Try catching a file that doesn't exist to see some stderr output:

```
[ppeters@rad-srv ~]$ catfile-is-not-there
cat: file-is-not-there: No such file or directory
```

The output above looks just like normal stdout output.Let us try to change that output using pipes:

```
[ppeters@rad-srv ~]$cat file-is-not-there | sed’s/Non existant/Train Squash/'

cat: file-is-not-there: No such file or directory
```

Reading from the example above, we have not changed the output result even after introducing piping. With piping, the shell gets the standard output of the left side of the pipe and introduces it as the input of the bash command positioned to the right side of the pipe symbol. From the example above the error output from the cat command redirected to **standard error** , and not to **standard out** , meaning that nothing went through the pipe to sed.

Redirecting Command Output

After normal command execution the standard out and standard error, streams display on the console terminal. This makes these outputs visible to the user on the console. In normal Linux bash operations, we can *redirect* the stdout and stderr output streams to a custom file using the > "greater-than" operator:

```
[ppeters@rad-srv ~]$ echo “Greetings World”

Greetings World

[ppeters@rad-srv ~]$ echo“Greetings World”> new-msg-file

[ppeters@rad-srv ~]$ catnew-msg-file

Greetings World
```

The illustration above shows that the second echo statement didn't display the statement to the terminal console because the output was redirected to the file called new-msg-file. The statement > new-redirect-file has mainly two functions:

- The output redirector creates a file called new-redirect-file if the file doesn't exist; and
- it would replace new-redirect-file's contents with the new contents if the file was already there

Now if the new-redirect-file was already there, and we did echo greetings> new-redirect-file, it would now have only greetings in it. If you want to add more data to the file by appending, you are required to use the >> operator:

```
[ppeters@rad-srv ~]$ cat new-msg-file

Hello there

[ppeters@rad-srv ~]$ echo hello out there again >> new-msg-file

[ppeters@rad-srv ~]$ cat new-msg-file

Hello there
```

```
hello out there again
```

Linux File Handles

In Linux we have file handles also known as file descriptors. File descriptors are unique positive integers that uniquely identify open files in the bash shell. We have three main file descriptors, namely 0,1 and 2, that describe how data resources may be accessed. File handle 0 refers to data stream STDIN read standard input, whilst file handle one represents data stream STDOUT read the standard, and lastly file handle 2 refers to data stream STDERR read standard error.

Name	File handle	Description	Abbrev
Standard input	**0**	This is the default data stream for terminal input. In the bash shell this defaults to the user's keyboard input.	**Stdin**
Standard output	**1**	This is the default data stream for output, as an example when a command displays or prints text. The default on the terminal is the user screen or console.	**Stdout**
Standard error	**2**	This is the default data stream for output related to errors happening on the terminal. In the bash shell, it defaults to the user's console or screen.	**Stderr**

Stderr

We can use file descriptors to duplicate output. To duplicate terminal output to a file descriptor, we use the >& "greater than and ampersand" operator prefixed to the file descriptor number. Below is the example:

```
# Redirecting stdout result to the stdout (File Descriptor 1)
[ppeters@rad-srv ~]$ echo"hi there people">&1
hi there people
# Redirect stdout to stderr (File Descriptor 2)
[ppeters@rad-srv ~]$ echo"hi there people">&2
hi there people
```

The example above is similar to redirecting output to a file, as we did a couple of examples before, stdout and the other descriptors are known as special files that require us to use >& instead of >.

The output above looks similar, but the changes are clear when we start piping our output. Below is an example of what happens when redirections are done to stdout in comparison to when they are done to stderr:

```
# Redirect to the stdout, going through the redirection pipe
[ppeters@rad-srv ~]$echo"some are redirections">&1 | sed"s/some/all/"
all are redirections
# Redirect to the stderr
[ppeters@rad-srv ~]$ echo“some are redirections”>&2 | sed"s/some/all/"
all are redirections
```

Common Use Cases for stderr, stdout, and stdin

Example of redirecting combined output to a file:

```
[ppeters@rad-srv ~]$./command file_name1 file_name2 file_name3 > log-filename 2>&1
[ppeters@rad-srv ~]$ cat log-filename
stderr file_name2
stdout file_name1
stdout file_name3
```

Chapter Three: Files

We will start with an introduction to file searching and a detailed look at the various search tools available. I will then discuss file archiving and the various ways of creating archives.

There are six unique ways to search for files in the Linux command line, and each of the ways has its advantages. In this section of the book, we are going to look at the various search tools, namely; **locate, which, find, whatis, whereis, and apropos** . These tools have their unique features and merits; these search utilities are used for different purposes.

Linux commands for searching and finding files are varied, and each command performs well at other areas than the other tools. This section will detail the various search commands and look at their strengths in searching. Below we will start with a very brief introduction to these search tools before we go into full details of each tool.

A Basic Introduction to Search Tools

Linux administrators may need to know the location of a file or directory on the filesystem but may not know where to find it. There are six bash commands that can be used to search for it, including whatis, whereis, find, and locate, apropos and which.

Introduction to Find

Below is the syntax of the **find** command:

```
[ppeters@rad-srv ~]$find search_path search_pattern
```

If the path for the file is not specified, the search when using find begins in the user's current working directory and searches down into all the sub folders.

The find command has many options which are reviewable through use of the find manual pages. The manual pages for the find command are found by entering **man find** at the bash terminal. The most utilized option is the **-name** , which directs our find command to look for files and directories with the specified string in their name.

```
[ppeters@rad-srv ~]$ find / -name file_name
```

The illustration of the find command above is a search for files in the root directory "/" with the name "file_name" string in their search pattern.

The Locate Command

The command locate’s syntax is:

```
[ppeters@rad-srv ~]$locate search_pattern
```

The locate command enables us to see each file or folder that has the search_pattern string contained in the file or directory name. To locate a file or folder with search pattern "finger," enter the following statement:

```
[ppeters@rad-srv ~]$locate pwd
```

The bash locate command makes use of the slocate data store to search for files and folders with the search pattern string "finger" in their names.

The locate command searches for strings patterns faster when the locate database is recent. That locate database is scheduled for auto-updates by running a cron task every night. In Linux Cron job is a small utility that executes in the terminal background, executing a variety of scheduled responsibilities, for instance, making locate database updates at 6 am every morning.

The updated b command refreshes the slocate database on the command line interface.

Which, Whatis and Whereis Search Tools

The Which Command

Below is the syntax of the command:

```
[ppeters@rad-srv ~]$ which command_name
```

The above which statement returns the path of the binary, or executable, bash commands.

```
[ppeters@rad-srv ~]$which cd
/usr/bin/cd
[ppeters@rad-srv ~]$
```

The illustration of the which command above displays /usr/bin/cd.

The Whereis Command

Below is the syntax of the whereis command:

```
[ppeters@rad-srv ~]$whereis command_name
```

The whereis command illustration below returns the paths of the find command, the binary file of the find command, the location of the source code, and the location of the find man page.

```
[ppeters@rad-srv~]$whereis find
/usr/bin/find /usr/share/man/man1p/find.1p.gz /usr/share/man/man1/find.1.gz
[ppeters@rad-srv ~]$
```

The Whatis Command

The syntax of the whatis command:

```
[ppeters@rad-srv ~]$ whatis command_name
```

The whatis command above displays details about the command_name from the command's man pages.

```
[ppeters@rad-srv ~]$whatis lp
```

Search Tools in-Depth

Searching Using Find Command

One of the most eminent and regularly used search command-line tools is the Find Command. The Linux Find command is a search and locate utility that displays the path of files and directories based on specified search strings matching given parameters.

There are a number of conditions used to search for files in bash namely; find by name, find by permissions, user, group, file-type, modification and creation date and size of the file.

This book will give illustrations of the most common use cases of the find command.

The bash **find** command is a command-line tool for traversing a file hierarchy. This command is used to locate files and directories in the file system and perform subsequent operations on the discovered files. Find command supports searching by name, directory, file, owner, modification and creation dates, and file permissions. By using the ‘-exec' option with the find command, other Linux bash commands can be subsequently executed on files or folders found.

Find Command Syntax

```
[ppeters@rad-srv ~]$find [search_path] [expression] [-options] [search_item]
```

Find Files using Name or Extension

Within the Linux filesystem, we can search for a file using the file's name or extension.

The illustration below searches for *.star files in the /home/ppeters/ folder and all sub-folders:

```
[ppeters@rad-srv ~]$find /home/ppeters/ -name "*.star"
```

Common Linux Find Commands and Syntax

Bash find statements have the following syntax:

```
[ ppeters@rad-srv ~]$find options find_path statement
```

The options switch takes control of the actions and the find process' optimization technique.

- The find_path feature defines where find begins its search, especially at the Linux root directory for the search string.
- Statement from the syntax above takes control of all the tests that filter through the Linux directory tree to give an output.

Take a look at the illustration below:

```
[ppeters@rad-srv ~]$find –O2 -L /var/www/html -name "*.php"
```

This find command statement enables optimization at the maximum level (-O2) and lets the find command to follow soft links (-L) which are also called symbolic links. The find command traverses the whole /var/www/html directory tree for files that end with .php.

Find Command Examples

Statement	What it does
find / -name 1st_file.conf	Searches for a file called 1st_file.conf in root directory and and its sub-folders.
find / -name *.png	Search for all .png image or picture files inside the / directory and its sub-folders.
find /etc -type f –empty	Search for an blank file inside the /etc directory.
find / -user ppeters -mtime 3 -iname ".db"	Search for all .db records that were altered in the past 3 days by ppeters who is a user.

Find Options and Optimization

The find command' basic configuration ignores soft links popularly known symbolic links. Using the find command option's –L switch allows the result to be a symbolic link.

By optimizing its filtering technique, the find command improves its efficiency. Find command-line utility has three performance stages, namely: -O1 the first stage, -O2 or the second stage, and the third stage, which is -O3. The first optimization, by default, is-O1. This optimization forces the find command to search for outcomes-based firstly on the name of the file before all previous tests are executed.

The level of optimization called -O2 gives priority to searches for the file name, just like in level-O1. -O2 performs all searches of the file type prior to processing additional resource-hungry tasks. Optimization Level-O3 allows the most serious optimization to be found and reordering all trials based on their relative cost and the probability of success.

Find Optimization

Level	Detail
First level -O1	First optimization level, it is the default where the filtering is initially based on the name of the file.
Second Level -O2	This is the second level optimization for find command which also starts the initial searches with name of the file, before moving on to the type of file.
Third Level -O3	This is the third level of find command optimization which allows automatic re-arrangement of the searches basing on the efficient use of resources and the inherent probability of success

-maxdepth X	This switch performs a search of the user's current working directory and its sub-directories X levels deep.
-iname	This switch represents a filter that disregards sentence cases
-not	Using the find command with this –not switch option ensures that the output results that do not match the test case are displayed.
-type f	This option is used to do a search for all files only.
-type d	This switch is used to search for all directories only.

Using Modification Time with Find

Let's assume that you are using your Linux system and you seem not to remember where you saved some power-point and text files under. You only remember the time period you downloaded the files to your file system. How do you use the find command-line utility to locate your power-point and text files?

You need to use the find command with modification time switch enabled. In bash command line each and every file has three time stamps, which are records of the last time some operations were done on the file namely atime, ctime and mtime. The time stamp atime represents the time when the file was accessed, while ctime represents a time when the file had its status changed that is the file or its attributes were changed. Lastly, we have the mtime which represents the modify time or a time when the file's contents were changed.

In the bash shell, you can look for files with time stamps that are within a certain age range, or even make comparisons to other time stamps.

It is possible to use find command with the -mtime option. The filter or search returns a list of files that were last accessed N*24 hours ago. For instance, to search for a file that was last accessed in the last three months (90 days), we use the option -mtime +90.

- **-mtime +25** this means you are searching for a file altered 25 days ago.
- **-mtime -30** this means you are searching for a file less than 30 days old.
- **-mtime 40** If you do not include + sign or – sign it means the exact number of days specified for the example above means exactly 40 days ago.

The find bash command enables users to find or search for files based on the date the file was altered or modified:

```
[ppeters@rad-srv ~]$ find /var/named -name "*zone" -mtime 4

[ppeters@rad-srv ~]$ find /home/testuser/ -name "*conf" -mtime 6
```

\ The initial find command returns a listing of all files that have the search string zone at the end and were altered four days prior. The second find command filters the test user's home directory for all files that have names that end with the string conf and were altered in the previous six days.

Using Grep to Search

The find command can only filter the hierarchy of the directory depending on the name and metadata of such a file. We use a tool like grep if you need to search for text or content within a file. The

instance below illustrates the use of grep:

```
[ppeters@rad-srv ~]$ find /etc -type f -exec grep "conf" '{}' \; -print
```

The above statement searches each object that is of type file specified by the "-type f" switch in the /etc directory and then executes the grep "conf" command to find all the files that have "conf" text string in them for each object file that meets the criteria. The matching files will be displayed on the terminal screen using the (-print) switch. For finding match outcomes, the curly braces"{}" are used as a placeholder. The curly braces {} are contained in single quotes (') to prevent the handling of a malformed file name by grep.

To prevent shell interpretation, the -exec command is terminated with a semicolon (;), which should be fled (\ ;).

This type of command could have used the xargs command to produce a comparable output before the-exec option was implemented:

```
[ppeters@rad-srv ~]$ find /usr -type f -print | xargs grep "binary"
```

Finding and Executing Files Using Find

Basic usage of **-exec**

The -exec option or switch takes as its argument and implements an internal utility with optional statements.

If the empty curly braces {} are present anywhere in the given command, the current pathname (/the /path/file_name) will replace each of the instances. There is no need to quote the curly braces {} in the bash shell.

The command has to be prefixed with a semicolon; to check out where it ends (as there may be more options later). The shell must be quoted as \; or';' to protect the; from the shell, otherwise the shell will see it as the end of the find command.

The Example below (the \ at the end of the first two lines are just for line continuations):

```
find / -type f –name ‘*.ppt’ –exec grep –q ‘devops’ {} ‘;’ –exec cat {} ‘;’
```

This will locate in or below the current working folder all standard files (-type f) whose names match the pattern*.ppt. It will then check whether the string "devops" occurs in any of the files identified using grep-q (which produces no result, only an exit status). **a cat** will be implemented for those files that contain the string to display the file's contents to the terminal.

Each -exec also functions as a "test" on the searched pathnames, just as -type and -the name does. If the command returns a zero exit status (meaning "success"), it will consider the next part of the find command; otherwise, the find command will continue with the next pathname. In the above instance, this is used to locate files.

The find **-exec** option or switch executes a program that has a match with the find statement against any item. Take the specific example below:

```
[ppeters@rad-srv ~]$ find /etc -name "rc.local" -exec chmod u+r '{}' \;
```

The statement above searches in the /etc folder for files that have the name rc.local and then executes the chmod g+r program to change permissions of the file to give users' group the read rights to any instance of rc.conf found.

The bash commands that run with the -exec option are executed in the find process' folder. Using the -execdir ensures that the command is executed in the folder in which the match result resides. This solves the security issues and produces better operations performance.

The -exec and –execdir switches run without additional prompts in the bash shell. It is possible to replace -exec with the option -ok and replace the alternative -execdir with -okdir if you require prompts before you take any action.

Searching and Deleting Files

This -delete option in find should be utilized with extreme caution because the action is irreversible. In Linux, you may append the option **-delete** to the end of a match expression to delete all files that match the find pattern. The -delete option should only be used when you are very certain that the output results match *only* the files that you need to delete.

In the following example, the find command locates all files in the filesystem hierarchy beginning at the current working directory and traversing into the directory tree. In this example, find will delete all files that end with the characters .backup:

```
[ppeters@rad-srv ~]$ find . -name "*.backup" -delete
```

Searching Using the Locate

The Linux Locate command is used to search for files using the names of the files. There are two most popular user-accessible file search utilities called find and locate. The locate utility operates better and quicker than the find command because it looks through a database instead of searching the file system when a file search is started. This database includes bits and file components.

Locate Command

Locate Command Syntax

```
[ppeters@rad-srv ~]$ locate [OPTION] search_pattern
```

Locate Exit Status: The locate command will quit with status 0 whenever a match is found. After searching and there are no matches found the locate command will exit with status 1.

Options:

- **-b, –basename:** Match only the name of the base against the patterns specified.
- **-c, –count:** Write down the total number of matching entries instead of writing file names on the standard output.
- **-d, –database DBPATH :** The default database should be replaced with DBPATH. The: (colon) separated list of database file names is called DBPATH. If there is more than one *–database* options specified, the output path is an addition of the different paths. The default database replaces an empty database file name. The standard input is referred to as the database file name.
- **-e, –existing :** Displays only the command locate entries referring to files that exist during the locate command execution time.

- **-L, –follow:** When we are checking for file existence through the use of the *–existing* option, we also use -follow to trail all symbolic links. Through this option, all broken symbolic links are left out from the output.

- **-h, –help :** Specifying the –h option, will result in the summary of all the available options being displayed to standard output.

- **-i, –ignore-case :** this option ensures that case distinctions are ignored when matching search patterns.

- **-l, –limit, -n LIMIT :** this option ensures that the execution successfully exits after finding LIMIT entries. When the *–count* option is also specified; the output count is also limited to LIMIT.

- **-0, –null :** This option separates the entries on the result using the ASCII NULL character instead of writing each entry on a separate line.

- **-S, –statistics :** This option writes statistics about each database read to the terminal console instead of looking for files and exit successfully.

- **-q, –quiet :** This option ensures that we do not write error messages encountered whilst reading and running databases.

- **-r, –regexp REGEXP :** This option is to search for a pattern matching a basic regular expression REGEXP.

- **–regex :** Interpret all PATTERNs as extended regexps.

- **-V, –version :** Write information about the version and license of locating on standard output and exit successfully.

- **-w, –wholename :** Only match the entire path name against the patterns indicated. The default option is this choice. You can specify the reverse using -basename.

Below is the command to locate a file using the name.

```
[ppeters@rad-srv ~]$ locate search.txt
```

The example above searches for search.txt file .

Example to limit our search queries to a particular number.

```
[ppeters@rad-srv ~]$ locate "*.php" -n 25
```

The example above will show 25 files ending with .php results.

Example to display the otal number of matching entries .

```
[ppeters@rad-srv ~]$ locate -c [.yaml]*
```

The example above there will count files ending with *.yaml* .

Searching with Which Command

The Linux, **which** command is used to determine the path or location of a specified executable that is performed when you type that particular executable or command in the terminal prompt. The command searches for the executable defined as an argument in the PATH environment variable directories.

The command searches through your path's directories and attempts to find the command you're looking for. It enables you to determine which version of a program or command will be running when you enter the command line with its name.

Imagine that we were having a program called **ls** . We know the computer is installed, but we don't know where it is. It has to be somewhere in the path because it runs when we type its name. We can use this command to find it:

Using which Command

The following shows the syntax of which command:

```
[ppeters@rad-srv ~]$ which [options] command
```

If the filename was given as a command in the bash shell, it returns the pathnames of the files (or links) that would be executed in the current environment. It does this by searching the location for executable files matching the arguments ' names in the PATH environment variable. The, which command ignores all soft links.

Which Command Options

-a	Print all matching pathnames of each matching *filename* .

Exit Status for Which Command

The, which command returns the values below called an exit status, depending on what happened:

0	Status 0 means that all *filename* s in the search pattern were found, and all were executable.
1	Status 1 means that one or more *filename* s were not located, or were not executable.
2	Exit status 2 means that an invalid option was specified in the which command

Which Example

```
[ppeters@rad-srv ~]$ which bash
```

The command above searches the pathname of the file which would be run if the **bash** command were executed. The result of the command above is:

```
[ppeters@rad-srv ~]$ /bin/bash
```

Searching with whatis command

The whatis command in Linux displays the search results as one-line manual page descriptions.

Following is its syntax:

```
whatis  [-dlv?V]  [-r|-w]  [-s  list]  [-m  system[,...]] [-M path] [-L locale] [-C file] command_name
```

The basic usage of the whatis command is simple. The user passes the tool name as input, and whatis will search and display a one-line description of the command utility.

Here's an example:

```
[ppeters@rad-srv ~]$ whatis ls
```

The following output was produced by the aforementioned command:

```
[ppeters@rad-srv ~]$ ls (1)  - list directory contents
```

Searching with the Whereis Command

The Linux whereis command is used to find a command's binary, source, and manual page files. This command searches for files in a limited number of places (binary file directories, man page directories, and directories of libraries).

Usually, it is used to locate a program's executables, its man pages, and settings files. The command's syntax is straightforward: just type whereis, followed by the command or program name you'd like to learn more about.

When working on the bash command line, we desire to know the path to a specific binary file or a command quickly. Yes, in this case, the find command is an alternative, but it takes a lot of time and will probably also generate some unwanted outcomes. For this purpose, there is a particular command intended: whereis.

Whereis command is used to locate the source / binary command file and manual sections for the specified Linux system file. Making comparisons between the whereis command and the find command we find some similarities as both tools are used for the same jobs. The only major difference between the two is that the whereis command gives a more accurate result in a short space of time.

First, the names supplied are removed from the path or directory elements and any prefixed extensions of the.ext form, for instance, .c. prefixes of s. resulting from source control are also removed. The whereis command then tries to locate the command we are searching for in the standard Linux directory list.

Points to note about whereis command:

- The bash whereis program utilizes chdir version two utility to ensure quick results; we should make sure that the pathnames are full and well-formed. The given pathnames must begin with a forward slash '/' and should be a valid path in the system directory tree; otherwise, it will exit without any valid result.

- There is a fixed non-dynamic coded path for the whereis command, which means you may sometimes not find what you're looking for.

Command whereis Syntax

```
[ppeters@rad-srv ~]$  whereis [options] filename
```

Example whereis command: When we want to find the location of *apropos* command, we execute the following command in the terminal:

```
[ppeters@rad-srv ~]$  whereis apropos
apropos:/usr/bin/apropos /usr/share/man/man1/apropos.1.gz
[ppeters@rad-srv ~]$
```

Options

Symbol	Description
-b	Find only for program binaries.
-m	Find only for manual pages sections.
-s	Find only for command sources.
-u	This option helps in searching for all unusual entries. Linux files are said to be unusual if they do not have one entry of each requested type. As example, "**whereis -m -u *** " is searching for files in the current working directory which do have documentation.
-B	This option alters or otherwise puts a limit to the paths where **whereis** looks for binaries.
-M	This option alters or otherwise puts a limit to the locations where the command whereis searches for manual pages sections.
-S	This option changes or otherwise puts a limit to the places where whereis searches for command sources.
-f	This option terminates the past folder listing and informs of the beginning of file names, and this must be utilized only when any of the **-B** , **-M** , or **-S** options are in use.

Searching with Apropos Command

Linux operating comes with an enormous number of commands, so sometimes remembering each and every command is quite hard. In such instances, the apropos command becomes helpful. The apropos command helps the user who does not remember the precise command but recalls some keywords associated with the command that defines its uses or features. Using the keywords given by the user, the apropos command locates the command and its features, through searching the Linux man page.

Apropos Syntax

```
[ppeters@rad-srv ~]$ apropos [OPTIONS] keyword
```

Example apropos command: Let’s assume that you would like to archive or compress a file but you don’t know the command to use to archive a file then you could type the apropos command in

the terminal and it will display all the related commands and a short description of their functionality.

After running the **apropos archive** command below you will find a number of commands listed on the terminal that have something to do with file archiving.

```
[ppeters@rad-srv ~]$ apropos archive
apt-secure (8)       - Archive authentication support for APT
ar (1)               - create, modify, and extract from archives
Archive::Zip (3pm)   - Provide an interface to ZIP archive files.
Archive::Zip::FAQ (3pm) - Answers to a few frequently asked questions about A...
Archive::Zip::MemberRead (3pm) - A wrapper that lets you read Zip archive mem...
Archive::Zip::Tree (3pm) - (DEPRECATED) methods for adding/extracting trees u...
deb-extra-override (5) - Debian archive extra override file
deb-override (5)     - Debian archive override file
dpkg-deb (1)         - Debian package archive (.deb) manipulation tool
dpkg-split (1)       - Debian package archive split/join tool
funzip (1)           - filter for extracting from a ZIP archive in a pipe
gpg-zip (1)          - encrypt or sign files into an archive
llvm-ar (1)          - LLVM archiver
llvm-ar-3.8 (1)      - LLVM archiver
phar (1)             - PHAR (PHP archive) command line tool
phar.phar (1)        - PHAR (PHP archive) command line tool
phar.phar7.1 (1)     - PHAR (PHP archive) command line tool
phar7.1 (1)          - PHAR (PHP archive) command line tool
ptardiff (1)         - program that diffs an extracted archive against an une...
ptargrep (1)         - Apply pattern matching to the contents of files in a t...
ranlib (1)           - generate index to archive.
swipl-rc (1)         - SWI-Prolog resource archiver
tarcat (1)  - concatenates the pieces of a GNU tar multi-volume archive
unzip (1)   - list, test and extract compressed files in a ZIP archive
unzipsfx (1)       - self-extracting stub for prepending to ZIP archives
x86_64-linux-gnu-ar (1) - create, modify, and extract from archives
x86_64-linux-gnu-ranlib (1) - generate index to archive.
zip (1)      - package and compress (archive) files
zipgrep (1) - search files in a ZIP archive for lines matching a pat...
zipinfo (1)          - list detailed information about a ZIP archive
```

Options

Option	Description
-d**,** --debug	This option displays all the information about debugging.
-v**,** --verbose	This option displays on the terminal console the verbose warning messages.
-r**,** --regex	This option parses and Interprets each *keyword* as a regular expression. This is normally the default action. Each of the *keywords* will be matched agains page names and the manual page descriptions individually. The matches are not bound to word boundaries.
-w**,** --wildcard	This option parses and Interprets each of the *keywords* as a string pattern with all the shell-style wildcards or meta characters. Each of the *keywords* will be matched against the existing page names and the manual page descriptions individually. If the argument **--exact** is also utilised, then output result match will only be located if the expanded *keyword* string or pattern matches the whole description or Linux manual page name. Otherwise, the *keyword* is also allowed to match on word boundaries in the description.
-e**,** --exact	With this option each *keyword* will be matched exactly against the manual page names and the ensuing manual page descriptions.
-a**,** --and	This option only outputs the items that match all the given *keywords* . The default output result is to display all items matching any *keyword* .
-l**,** --long	This option ensures that we do not trim the result to fit the terminal console width. In most normal cases, the output will be trimmed to fit the terminal width to avoid malformed results from badly-written manual pages' NAME sections.
-s ***list* ,** --sections ***list* ,** --section ***list***	This option only searches the given manual pages sections. The given *list* is a colon- or comma-separated list of manual pages sections. If an entry in the manual pages *list* is a page section, for example "**4** ", then the listed output of descriptions will show all the pages in sections "**4** ", "**4python** ", "**4x** " etc; whilst if a *listing* shows an extension, for instance "**5perl** ", then the resulting list will include pages only in that exact part of the section of the manual.
-m ***system* [,...],** --systems=***system* [,...]**	This option is used to search in the other operating system that the user has access to. For instance, to filter the manual page descriptions of an operating system called "MacOS", we use the option **-m MacOS** . The specified *system* can be a combination of comma-separated names of operating systems.
-M ***path* ,** --manpath=***path***	This option is used to specify an alternate set of colon-

	separated manual page tree to filter. By default the **$MANPATH** environment variable is used by **apropos** , unless it's empty or not unset, in which case it will determine an appropriate "manpath" based on your **$PATH** environment variable. This –M option overrides the contents of **$MANPATH** .
-L ***locale ,*** --locale=***locale***	**The apropos command** normally determines the current system locale by making a call to the C procedure called the **setlocale** which searches various environment variables, possibly including **$LC_MESSAGES** and **$LANG** . We may temporarily override the determined value using the –locale option. This option is used to supply a locale string to **apropos command** . Note that locale option does not take effect until the search for the manual pages actually starts. Result such as the help message will always be output in the initially set locale.
-C ***file ,*** --config-file=***file***	Use this user configuration file rather than the default of **~/.manpath** .
-h**,** --help	This option helps in displaying a help message and exit.
-V**,** --version	This option shows the version information.

Environment

SYSTEM	Setting the **$SYSTEM** environment variable, has the same effect as if it this variable was specified as the argument to the **-m** option.
MANPATH	Setting the **$MANPATH** environment variable, will have its value interpreted as the colon-separated manual page tree search path.
MANWIDTH	Setting the **$MANWIDTH** environment variable, makes sure that the value is used as the terminal width (as discussed in the **--long** option). If this variable is not set, the terminal console width will be calculated using **ioctl** if it is available, the value of the **$COLUMNS environment variable** , or it falls back to the default 80 characters if all else fails.
POSIXLY_CORRECT	If this environment variable called the **$POSIXLY_CORRECT** is set, to a null value, the default **apropos** search will be an extended regular expression (**-r**).

Compressing is very helpful when storing valuable files and sending large files across the Internet. Compressing an already compressed file adds additional overhead, so you'll get a marginally larger file. Stop compressing a file that is already compressed. Linux has many programs for compressing and decompressing files. I'm now going to discuss the two most used utilities for compressing and decompressing files.

Compressing and Decompressing Files

The most common tools used to compress and decompress files in Linux operating systems are gzip and bzip2.

Compress and Decompress Files using gzip Program

The gzip is a utility to compress and decompress files making use of the Lempel-Ziv coding (LZ77) algorithm.

Compressing files using gzip

To compress a file named **firstfile.txt and** replace it with a gzipped compressed archive called firstfile.txt.gz, run the following command in your terminal:

```
[ppeters@rad-srv ~]$ gzip firstfile.txt
```

Gzip command line utility always replaces the original file you supply as the argument for instance above argument **firstfile.txt** is replaced with a gzipped compressed archive named **firstfile.txt.gz** .

The gzip compression utility is used in a number of ways. An example is to create a gzip-compressed archive of a bash command's output result. The following example statement shows.

```
[ppeters@rad-srv ~]$ ls -l Downloads/ | gzip > firstfile.txt.gz
```

The illustration above creates a gzipped compressed archive of the long directory listing of the Downloads folder in the user's home directory.

Compressing files and redirecting the output

Formally, the gzip utility compresses the given input files and directories and replaces them with the gzipped compressed archive. We have a way of making sure that we keep the original file and redirect the output to the standard output terminal. For instance, the following statement compresses **firstfile.txt** and redirects the resulting output to **redirect.txt.gz** .

```
[ppeters@rad-srv ~]$ gzip -c firstfile.txt > redirect.txt.gz
```

Similarly, we may also decompress an gzip archive file and specify the output file we want to be created:

```
[ppeters@rad-srv ~]$ gzip -c -d redirect.txt.gz > firstfile1.txt
```

The illustration command above decompressed the **redirect.txt.gz** file and redirects the resulting output to a file called **firstfile1.txt** . The redirection instance above the original file is not deleted.

Decompressing files with gzip tool

To uncompress the **firstfile.txt.gz file** , and subsequently substitute it with the original archive which is uncompressed, we follow the example below:

```
[ppeters@rad-srv ~]$ gzip -d firstfile.txt.gz
```

We have the gunzip decompression utility we can use to deflate a gzipped compressed file. We use gunzip, as shown below:

```
[ppeters@rad-srv ~]$ gunzip firstfile.txt.gz
```

Viewing contents of compressed files

It is possible to view the files and folders contained in an archive file through using the gunzip program with the –c switch without the need to decompress the archive as shown below:

```
[ppeters@rad-srv ~]$ gunzip -c firstfile1.txt.gz
```

Alternatively, we can view the contents of an archived file using the **zcat** utility, as shown below.

```
[ppeters@rad-srv ~]$ zcat firstfile.txt.gz
```

You may also redirect the output from gunzip and zcat through pipe "|" to the "less" bash command to view the output in a paginated way as shown below.

```
[ppeters@rad-srv ~]$ gunzip -c firstfile1.txt.gz | less
```

```
[ppeters@rad-srv ~]$ zcat firstfile.txt.gz | less
```

Alternatively, we can use the **zless** command program, which performs the same job as the pipelined zcat command.

```
[ppeters@rad-srv ~]$ zless firstfile1.txt.g
```

Setting the compression level with gzip

Another notable advantage of gzip is that it supports the use of compression levels. There are mainly three supported compression levels as given below.

- **1** – Fastest (lower compression)
- **9** – Slowest (Best compression level)
- **6** – Default level

To compress a file named **firstfile.txt** , replacing it with a gzipped compressed version with the **best** compression level, we specify the -9 switch with the gzip utility as shown below:

```
[ppeters@rad-srv ~]$ gzip -9 firstfile.txt
```

Concatenate multiple compressed files

It is also possible to join together or concatenate multiple compressed archive files into one archive. The example below shows how.

```
[ppeters@rad-srv ~]$ gzip -c firstfile01.txt > redirect.txt.gz
```

```
[ppeters@rad-srv ~]$ gzip -c firstfile02.txt >> redirect.txt.gz
```

The example above shows two commands compressing firstfile01.txt and firstfile02.txt and stores them in one archive file named **redirect.txt.gz** .

You can view the contents of both files (firstfile1.txt and firstfile2.txt) without extracting them using any one of the following commands:

```
[ppeters@rad-srv ~]$  gunzip -c redirect.txt.gz
```

```
[ppeters@rad-srv ~]$  gunzip -c redirect.txt
```

```
[ppeters@rad-srv ~]$ zcat redirect.txt.gz
```

```
[ppeters@rad-srv ~]$ zcat redirect.txt
```

To learn more about the gzip utility, you can visit the manual pages by typing the command below:

```
[ppeters@rad-srv ~]$ man gzip
```

Compressing and Decompressing with bzip2

The **bzip2** utility is similar to the gzip compression program. The main difference between the two is in the compression algorithms used. Bzip2 uses the compression algorithm called the Burrows-Wheeler block sorting text compression algorithm and Huffman coding. All the files bzipped using the bzip2 have the .**bz2** extension.

The usage of gzip and bzip2 is the same. We will just be substituting the **gzip** in the above examples with **bzip2** , **gunzip** with **bunzip2** , **zcat** with **bzcat** , and so on.

Bzip2 compression replaces the file we are compressing with the compressed version of the archive; this can be achieved by running the command below:

```
[ppeters@rad-srv ~]$ bzip2 firstfile.txt
```

If you are not willing to substitute the original file, use **-c** switch with the bzip2 tool and redirect the output result to a new file.

```
[ppeters@rad-srv ~]$ bzip2 -c firstfile.txt > output.txt.bz2
```

To decompress a compressed file:

```
[ppeters@rad-srv ~]$ bzip2 -d firstfile.txt.bz2
```

Or,

```
[ppeters@rad-srv ~]$ bunzip2 firstfile.txt.bz2
```

To view the contents of a compressed file without decompressing it:

```
[ppeters@rad-srv ~]$ bunzip2 -c firstfile.txt.bz2
```

Or,

```
[ppeters@rad-srv ~]$ bzcat firstfile.txt.bz2
```

```
[ppeters@rad-srv ~]$ man bzip2
```

Archiving Files and Directories

There are mainly two utilities, namely; tar and zip, that are used to archive files and directories. This section of the book is going to be split into two sections; the first section will detail usage of TAR utility, and the second section will discuss the ZIP utility.

The Tar Command

The Tape Archive utility commonly known as the Tar is a Linux utility for archiving files and directories on the bash command line. Tar is utilized to put together or store a number of files which can be the same or unique size into one archive file. The tar utility has four major working modes, namely c, x, r, and t.

1. **c** – C mode is for creating a tar archive from one or more files and from one or more folders.
2. **x** – This is the Extract mode for file archiving.
3. **r** – This is the operating mode to add new files to the end of a tar archive file.
4. **t** – This mode provides a listing of the entities inside of a tar archive file.

Creating New Tar Archives

For illustration purposes only, a directory called example_dir will be used. This example_dir contains three unique file types; a music file, a picture file, and a word document.

```
[ppeters@rad-srv ~]$ ls example_dir/
wordfile.odt imagefile.jpg musicfile.mp3
```

Below is the command for creating a new archive of the folder example_dir.

```
[ppeters@rad-srv ~]$ tar cf example_dir.tar example_dir/
```

From the example above, **c** flag is the create symbol, and the file name is represented by the **f flag** .

The illustration below is demonstrating the creation of a tar archive from a number of files in the user's present working directory:

```
[ppeters@rad-srv ~]$ tar cf files_arch.tar 1st_file 2nd_file 3rd_file
```

Extracting Archives

We can extract a tar archive whilst we are in the same directory it is located in by just executing the tar command below:

```
[ppeters@rad-srv ~]$  tar xf example_dir.tar
```

It is also possible to extract a tar archive file in a folder in a different location by using the capital C or the "**C"** flag in our options part of the command. For instance, the tar statement below extracts the specified tar archive in the **/usr/local** directory.

```
[ppeters@rad-srv ~]$  tar xf files_dir.tar -C /usr/local
```

Alternatively, go to the **/usr/local** directory and extract the archive, as shown below.

```
[ppeters@rad-srv ~]$ $ cd /usr/local
[ppeters@rad-srv local]$ $ tar xf ../example_dir.tar
```

There are instances when you may want to extract files of a specific type. For example, the following command extracts the ".jpg" type files.

```
[ppeters@rad-srv ~]$ tar xf example_dir.tar --wildcards "*.jpg"
```

Creation of compressed gzipped and bzipped archives

Tar command creates the .tar archive file by default. It is important to note that the tar command is useable with the file gzip and bzip2 compression tools. The archive file that ends with the **.tar** extension refers to the uncompressed tar archive. The .tgz and **tar.gz** extensions represent **gzip compressed** archive files, whilst the .tbz and the **file extensions** refer to **bzip compressed file** archive. The gzip and bzip2 archives are compressed archives.

Example **creation of a gzip compressed file** archive:

```
[ppeters@rad-srv ~]$ tar czf example_dir.tar.gz example_dir/
```

Or,

```
[ppeters@rad-srv ~]$ tar czf example_dir.tgz example_dir/
```

For all the illustrations above we compress the archives using **z** flag for the gzip compression method.

The **v** flag is used with the tar command to enable the user to view the progress while creating the archiv e.

```
[ppeters@rad-srv ~]$ tar czvf example_dir.tar.gz example_dir/
example_dir/
example_dir/wordfile.odt
example_dir/imagefile.jpg
example_dir/musicfile.jpg
```

The **v** flag above means verbose.

In Linux it is possible to create gzip compressed archive from a list of files:

```
[ppeters@rad-srv ~]$ tar czf files.tgz file_01 file_02 file_03
```

To extract the gzipped archive in the current directory, use:

```
[ppeters@rad-srv ~]$  tar xzf example_dir.tgz
```

To extract the compressed archive in a different directory, use -C flag.

```
[ppeters@rad-srv ~]$  tar xzf example_dir.tgz -C Downloads/
```

Example to create **bzipped archive** .

To create a bzip compressed archive file we use **j** flag as shown below.

Create an archive of a directory:

```
[ppeters@rad-srv ~]$ tar cjf example_dir.tar.bz2 example_dir/
```

Or,

```
[ppeters@rad-srv ~]$ tar cjf example_dir.tbz example_dir/
```

Creating a bzip archive from a list of files:

```
[ppeters@rad-srv ~]$ tar cjf files.tar.bz2 file01 file02 file03
```

Or,

```
[ppeters@rad-srv ~]$ tar cjf files.tbz file01 file02 file03
```

Use the v flag to display the progress.

We show the example to extract the bzipped archive into the current working directory:

```
[ppeters@rad-srv ~]$  tar xjf example_dir.tar.bz2
```

Or, extract the archive to some other directory:

```
[ppeters@rad-srv ~]$  tar xjf example_dir.tar.bz2 -C /tmp
```

Creating an archive of multiple files/folders

The tar command allows you to create an archive from multiple files and folders. The process of creating a gzip compressed archive of multiple directories or files at once, follow the example below:

```
[ppeters@rad-srv ~]$ tar czvf example_dir.tgz Downloads/ Documents/ example_dir/wordfile.odt
```

The above illustration creates a compressed bzip archive of **Downloads** , **Documents** directories and **wordfile.odt** file in **example_dir** directory and save the archive in the current working directory.

Exclude directories and/or files from while creating an archive

It is possible to exclude specific files or directories from your tar archive. This is a useful feature when backing up your data. You can choose which files or directories to back up. We use the **–exclude** switch to remove the files that we do not want appended to our compressed backup archive file. For instance, you may want to create an archive of your /home directory, but exclude Documents and Pictures folders.

Illustration is shown below

```
[ppeters@rad-srv ~]$ tar czvf example_dir.tgz /home/ppeters --exclude=/home/ppeters/Documents --exclude=/home/ppeters/Pictures
```

The tar command above creates a compressed gzip archive of my $HOME directory, excluding Documents and Pictures directories. To create a compressed bzip archive, replace **z** with **j** and use the extension .bz2 in the above example .

List contents of archive files without extracting them

To list the contents of an archive file, we use **t** flag.

```
[ppeters@rad-srv ~]$ tar tf example_dir.tar
example_dir/
example_dir/wordfile.odt
example_dir/imagefile.jpg
example_dir/musicfile.jpg
```

To view the verbose output, use **v** flag.

```
[ppeters@rad-srv ~]$ tar tvf example_dir.tar
drwxr-xr-x ppeters/ppeters 0 2019-09-11 19:52 example_dir/
-rw-r--r-- ppeters/ppeters 9942 2019-09-11 13:49 example_dir/wordfile.odt
-rw-r--r-- ppeters/ppeters 36013 2019-09-11 11:52 example_dir/imagefile.jpg
-rw-r--r-- ppeters/ppeters 112383 2019-08-22 14:35 example_dir/musicfile.jpg
```

Append Files to Existing Archives

Files or directories can be added/updated to the existing archives using **r** flag. Take a look at the following command.

```
[ ppeters@rad-srv ~]$ tar rf example_dir.tar example_dir/ ppeters/ example_txt.txt
```

The above command will add the directory named **ppeters** and a file named **example.txt** to example_dir.tar archive.

Linux users can validate if the files have been added to the:

```
[ppeters@rad-srv ~]$ tar tvf example_dir.tar
drwxr-xr-x ppeters/ppeters 0 2019-09-11 6:56 example_dir/
-rw-r--r-- ppeters/ppeters 9942 2019-09-11 6:49 example_dir/wordfile.doc
-rw-r--r-- ppeters/ppeters 36013 2019-09-11 6:54 example_dir/imagefile.jpg
-rw-r--r-- ppeters/ppeters 112383 2019-09-11 6:45 example_dir/musicfile.aac
drwxr-xr-x ppeters/ppeters 0 2019-09-11 6:32 ppeters/
-rw-r--r-- ppeters/ppeters 0 2019-09-11 6:49 ppeters/linux_text.txt
-rw-r--r-- ppeters/ppeters 0 2019-09-11 6:57 example_text.txt
```

Examples to create and extract

Create tar archives

- **Plain tar file:** tar -cf **files_archive.tar** file_01 file_02 file_03
- **Gzip compressed file:** tar -czf filesarchive **.tgz** file_01 file_02 file_03
- **Bzip compressed file:** tar -cjf filesarchive **.tbz** file_01 file_02 file_03

Extract tar archives

- **Plain tar archive: tar -xf** files_archive.tar
- **Gzip compressed file: tar -xzf** files_archive.tgz
- **Bzip compressed file: tar -xjf** file_sarchive.tbz

If the user wants to learn much more about the tar archive they may refer to the tar manual pages by typing the command below.

```
[ppeters@rad-srv ~]$ man tar.
```

Archiving Files Using ZIP

ZIP is a Linux file packaging and compression tool. Each file is stored with the.zip extension in a single .zip {.zip-filename} file.

Zip compresses files to shrink the size of the file and is also used as a file packaging tool. In many operating systems such as UNIX, Linux, and Windows zip is present.

If you have a restricted bandwidth between two Linux computers and want to transfer the files more quickly, use zip to compress the files and transfer them afterward.

The zip program places one or more compressed documents in a single zip archive along with file data (name, route, date, last modification time, security, and file integrity check data). You can pack a complete directory structure.

In the section above, we learned how to archive files and folders using the tar archiving command. In the coming section, we are going to have a deep dive into the use of Zip an **Unzip** programs to archive and compress files and directories. The zip command is an archiver, and compression tool rolled into, so we can use it for three functions namely file archiving, compression and decompression.

However, for compression, we have gzip and bzip2 tools which are the dominant programs.

Zip File and Directory Archiving Utility

Most Linux distributions come packaged with the zip utility. Just in case if it is not available, you can install it using the distribution's default package manager. Below we have different installations of zip and unzip on the various linux distributions.

On the RHEL, CentOS and Fedora variants:

```
[ppeters@rad-srv ~]$ sudo yum install zip unzip
```

On Debian and Ubuntu distributions

```
[ppeters@rad-srv ~]$ sudo apt-get install zip unzip
```

On distributions SUSE and openSUSE

```
[ppeters@rad-srv ~]$ sudo zypper install zip unzip
```

Creating A New Archive

To create zip archive with a group of files into one, the command used is below:

```
[ppeters@rad-srv ~]$ zip files.zip file_01 file_02 file_03
```

Below is the sample output displayed on the bash terminal:

```
adding: file_01 (stored 0%)
adding: file_02 (stored 0%)
adding: file_03 (stored 0%)
```

The above statement creates a zipped file archive called **files.zip** from **file_01** , **file_02** and **file_03** . Users do not have to append the **.zip** extension to the file name. It is added for clarity only there.

To create a zip archive of a folder, execute the following:

```
[ppeters@rad-srv ~]$ $ zip zip_dir.zip files_dir/
adding: files_dir/ (stored 0%)
[ppeters@rad-srv ~]$
```

The example above created a zip archive named **zip_dir.zip** of the **files_dir** folder. This statement only adds the parent directory to the zip archive. There are cases where you may need to archive all sub-folders recursively; we use **-r** flag as illustrated below.

```
[ppeters@rad-srv ~]$ zip -r zip_dir.zip files_dir/
 adding: files_dir/ (stored 0%)
adding: files_dir/text_file.txt (deflated 67%)
adding: files_dir/audio_file.mp3 (deflated 6%)
adding: files_dir/image_file.png (deflated 3%)
[ppeters@rad-srv ~]$
```

```
[ppeters@rad-srv ~]$ zip -r zip_dir.zip files_dir/ file_01 file_02 file_03
adding: example/ (stored 0%)
adding: example/Image.png  (deflated 3%)
adding: file1 (stored 0%)
adding: file2 (stored 0%)
adding: file3 (stored 0%)
[ppeters@rad-srv ~]$
```

Unlike the tar archiving utility, the zip archiving and compression tool builds a zip archive and compresses it when necessary. Tar utility only archives, and it does not compress.

Creating an archive with many directories and files

At times it is necessary to create an archive with more than one file and directories. We use the –r switch with the zip command to recursively create the zip archive containing all the specified folders and files:

```
[ppeters@rad-srv ~]$ zip zip_dir.zip file4
adding: file4 (stored 0%)
[ppeters@rad-srv ~]$
```

The illustration above has an existing archive and we are adding another file to the specified zip_dir.zip archive. Looking at the zip program example above; the statement only updated the contents of existing archive file without making any replacements. Whenever a user specifies an existing file or folder archive, that existing archive gets updated and not replaced. This action only preserves the existing archive. This capability to allow files to be added to a pre-existing archive is an advantage of the zip program over the tar archiving utility.

Extract Zip archive files

Extracting archives is as simple as creating archives. To extract an archive, you need to type unzip command and specifying the archive file you would like to extract:

```
[ppeters@rad-srv ~]$ unzip zip_dir.zip
```

Below is an example output result of the unzip command:

```
Archive: zip_dir.zip
creating: files_dir/
inflating: files_dir/text_file.txt
inflating: files_dir/audio_file.mp3
inflating: files_dir/image_file.png
extracting: file_01
extracting: file_02
extracting: file_03
```

```
extracting: file_04
```

With the unzip utility a user can extract a specific file or folder selectively from the zipped archive. The following statement below extracts **file_03** from the archive **zip_dir.zip.**

```
[ppeters@rad-srv ~]$ unzip zip_dir.zip file_03
Archive: zip_dir.zip
extracting: file_03
[ppeters@rad-srv ~]$
```

Listing Zip Archive Contents

The Unzip program or utility allows users to display a listing of the contents of a zip archive without file extraction through, use of the **-l** flag.

```
[ppeters@rad-srv ~]$ unzip -l zip_dir.zip
Archive: zip_dir.zip
Length Date Time Name
--------- ---------- ----- ----
0 2019-09-11 15:15 files_dir/
286 2019-09-11 15:16 files_dir/text_file.txt
8073033 2019-09-11 14:03 files_dir/audio.mp3
6799 2019-09-11 14:45 files_dir/image.jpg
10 2019-09-11 15:58 file_01
10 2019-09-11 15:58 file_02
10 2019-03-11 15:58 file_03
10 2019-03-11 16:37 file_04
--------- -------
6050289 8 files
[ppeters@rad-srv ~]$
```

Listing the contents of a zip archive is very useful when you want to determine the file(s) or folder(s) you would like to extract instead of just extracting the whole zip archive. There are instances where users need to determine which files they would like to extract or delete from an archive. It is at these instances it is necessary to view or list the contents of a zip archive.

Creating an Encrypted rchive

The noblest feature of the zip utility is that it allows users to create an encrypted or password-protected archive. With this encrypted or password-protected archive, users are required to enter their password for them to be able to extract or view the contents of the archive.

Encrypting an archive, is achieved through the use of the **-e** flag.

```
[ppeters@rad-srv ~]$ zip -e -r zip_dir.zip files_dir/ file_01 file_02 file_03 file_04
```

```
Enter password:
Verify password:
adding: files_dir/ (stored 0%)
  adding: files_dir/text_file.txt (deflated 67%)
adding: files_dir/audio_file.mp3 (deflated 6%)
adding: files_dir/image_file.png (deflated 3%)
adding: file1 (stored 0%)
adding: file2 (stored 0%)
adding: file3 (stored 0%)
adding: file4 (stored 0%
[ppeters@rad-srv ~]$
```

When an archive is created with encryption you need to enter your password when extracting or viewing the contents of the archive.

```
[ppeters@rad-srv ~]$ unzip zip_dir.zip
Archive: zip_dir.zip
[zip_dir.zip ] files_dir/text_file.txt password :
[ppeters@rad-srv ~]$
```

Creating a Multi-Part Zip Archive

There are times when you need to share a zipped archive over the internet. Most archives may be too big, and the solution is to create multi-part archive made up of sequential small-sized archives. These multi-part archives can be sent sequentially as small archives.

We can create a multi-part archive split into many smaller parts, for example, **3 MB** each, as follows:

```
[ppeters@rad-srv ~]$ zip -r -s 5m archive.zip files_dir/
```

The zip command above will create many zip archive files of size 5MB like **archivemp.z01** , **archivemp.z02** , **archivemp.z03** , and **archivemp.zip** . All these split files need to be sent to the recipient. When deflating and extracting these archive files, all the small parts that make up the contents will be extracted into a single folder called files**_dir** .

```
[ppeters@rad-srv ~]$ zip -r -s 2g archive.zip files_dir/
```

Create Archives of Other Programs

Zip command-line utility accepts standard input; this makes it feasible to create archives of other command results. The following example shows that we can pipe the output from **ls –l** command into the zip command utility as input to zip.

```
[ppeters@rad-srv ~]$ls -l Documents/|zip ls_documents.zip -
 adding: - (deflated 56%)
```

The **ls_documents.zip** is the archive of "ls " command. The command won't output the contents of the zip archive using **-l** switch.

Instead, you could view its contents using the command:

```
[ppeters@rad-srv ~]$ unzip -p ls-documents.zip | less
```

The example shows that **-p** switch refers to the pipe.

Compressing archives by stating the compression rate

The statement below shows how to archive a directory and its contents with the highest compression level [9] :

```
[ppeters@rad-srv ~]$ zip -r -9 archive.zip files_dir/
```

The Zip command-line utility has support for three compression levels, as shown below:

- **1** – Fastest compression speed but Worst
- **9** – Slowest compression speed but results in the Best or highest compression
- **6** – Is the Default level of compression

Exclude files or folders while creating archives

When creating zip archives, we can choose to exclude all unwanted files or sub-folders. We use **-x** flag, as shown below:

```
[ppeters@rad-srv ~]$ zip -r zip_dir.zip files_dir/ -x files_dir/image_file.png
```

The zip utility command above creates an archive of the **files_dir** directory, excluding **image_file.png** file from the archive.

With unzip utility a user can verify the contents of the zip archive without needing to extract the archive file

```
[ppeters@rad-srv ~]$
```

```
 [ppeters@rad-srv ~]$ unzip -l zip_dir.zip
```

Delete Files from an Existing Archive

With the zip utility, you are able to delete files inside an archive even after the archive was created.

We use the –d flag with the zip command to delete the unwanted files as illustrated below.

```
[ppeters@rad-srv ~]$zip -d zip_dir.zip "files_dir/text_file.txt"
```

```
deleting: files_dir/text_file.txt
```

```
[ppeters@rad-srv ~]$
```

Similarly, to delete a group of the same type of files, for example, **.txt** filesuse:

```
[ppeters@rad-srv ~]$ zip -d zip_dir.zip "files_dir/*.txt"
```

To find out more about the ZIP and unzip we use the Linux manual pages to learn more.

```
[ppeters@rad-srv ~]$ man zip
```

```
[ppeters@rad-srv ~]$ man unzip
```

Chapter Four: Writing Bash Scripts

Bash Shell scripts are just a collection of instructions you write and run together in a file. You're just putting and running a sequence of instructions in a text file. The distinction is that bash scripts can do much more than batch files.

"A bash shell script is a computer program intended to operate the Linux bash shell, an interpreter of the command line. Scripting languages are regarded to be the different dialects of bash shell scripts. Typical shell scripting tasks include file manipulation, program execution, and text printing.Linux has more than one shell, and scripting any of them is possible. In this chapter and in this book, we are going to focus on *bash shell scripting* .

Bash Shell scripts are used for automating system administration tasks in Linux, encapsulating complicated configuration information, and acquiring the complete power operating system. Combining commands enables you to generate fresh commands, adding value to your operating system.

Writing Your First Shell Script

The first bash scripting project is the Hello World example. Open your favorite editor and write a shell script file named as **firstscript.sh** containing following lines.

Type in your terminal:

i. **Vi** firstscript.sh

ii. Press letter "I" on your keyboard to enter into insert mode

iii. Type the following in the firstscript.sh

```
#!/bin/bash
echo "hello world" //display to the terminal screen
```

iv. After typing in the above then press **‘ESC'** key on the keyboard to leave insert mode and enter command mode

v. After leaving insert mode then type **‘:wq'**

vi. The **:wq** command above writes the script and quits the Vi editor.

The initial line above is called the **shebang.** It informs the Linux shell that this is a bash script and that it should be executed on the **/bin/bash** shell. The ensuing line is just an ***echo*** statement, which displays the string of words passed to it to the terminal console.

After typing the Vi command to create the firstscript.sh, there's a need to give the shell script execute permissions to make it runnable. You can set the execute permission, as shown below:

```
[ppeters@rad-srv ~]$ chmod +x firstscript.sh //add file execute permission
```

Executing the script has two options, as shown below. You can use either method to run the script after granting it execute permissions:

```
[ppeters@rad-srv ~]$ bash firstscript.sh
```

```
[ppeters@rad-srv ~]$ ./firstscript.sh
```

Output of the script

```
Hello world
```

Above is our own first and a very basic shell script that prints `*Hello world* ` to the terminal screen.

The common command-line interface language for Linux is the Bash shell script. Systems administrators may need to execute many routine commands daily for a variety of purposes. These daily routine tasks may be converted to bash scripts for automation. Any Linux user can learn bash scripting language so fast and with great ease.

Bash scripting makes use of comments alongside the code just like in any other programming language. These scripting comments are not processed with the rest of the bash script at the execution time. Well commented bash scripts are important to enable users to understand the script and its logic. There are many ways to add comments in bash script. The example script below shows how one can add single-line or multiple-line bash comments. Below is a bash file with bash script showing how to use bash comments. The ‘#' symbol is used to add single-line comments while a single quote (‘) with full colon ‘:' is used to add multi-lines comments.

comments_script.sh

```
#!/bin/bash
#Input a number
echo "Please enter a number"
read x
: '
Check if the entered number is
less than 12 or greater than 12 or equal to 12
'
if [[ $x -lt 12 ]]
then
echo "The number you entered is less than 12"
elif [[ $x -gt 12 ]]
then
echo "The number you entered is greater than 12"
else
echo "The number you entered is equal to 12"
fi
```

Sample output of executing the above script:

The example below shows that the script is executed three times with input values 10, 12, 15, and 23. The following output will appear.

```
root@BAKYRIE:~# bash comments_script.sh
Please enter a number:
10
The number you entered is less than 12
root@BAKYRIE:~# ./comments_script.sh
Please enter a number:
15
The number you entered is greater than 12
root@BAKYRIE:~# ./comments_script.sh
Please enter a number:
12
The number you entered is equal to 12
root@BAKYRIE:~# ./comments_script.sh
```

```
Please enter a number:
23
The number you entered is greater than 12
root@BAKYRIE:~#
```

The echo command

The`echo` program or command is used in a bash script to display scripting output to the terminal console. We can use numerous options with the echo command in bash to display the output in many ways. Below we are illustrating the echo command by creating a bash file with the following script to demonstrate the two simple uses of the `echo` command. From the example below, the first `echo` statement will display to the terminal screen a text data with a new line carriage return and the second command prints a simple text with no newline carriage return.

echo_commands.sh

```
#!/bin/bash
#Display  the first text string
echo "Print text with a carriage return new line"
#Display the second text string
echo -n "Print text without the carriage return new line"
```

Output result:

Execute the bash script.

```
root@BAKYRIE:~# chmod u+x echo_commands.sh
root@BAKYRIE:~# ./echo_commands.sh
Print text with a carriage retun new line
Print the text without the carriage return new lineroot@BAKYRIE:~#
```

The following output will appear after executing the above command.

Variables

Any programming language requires variable declaration. Variables of Bash can be declared in various ways. If a value is allocated in a variable then at

the start of the variable no symbol is used. The symbol'$' is used with the name of the variable when reading the variable value. Variable can be used on the shell terminal or may be used in a bash script.

The commands below are declaring a variable of type string called **myvariable** with a specified value, and then we print the value of the variable in the terminal.

```
$ myvariable="Bash is cool"
$ echo $myvariable
```

Output:

```
[root@BAKYRIE:~# ]myvariable="Bash is cool"
[root@BAKYRIE:~# ]echo $myvariable
Bash is cool
root@BAKYRIE:~#
```

Please create a bash script file with the following contents. Declare two variables here. These two variables are **$x** and **$y** . We need to ascertain if the input value, which is stored in **$x** is equal to the value stored in **$y.** Then the following message, "The two **Numbers are equal** " will be displayed otherwise "**The numbers are not equal** " will be displayed to the terminal screen.

var_numbers.sh

```
#!/bin/bash
echo "Please enter any number"
read x
y=23
if [[ $x -eq $y ]]
then
echo "The two numbers are equal here"
else
echo "The two numbers here are not equal"
fi
```

SampleOutput:

After running the script as shown below by line 3 we get the result below:

```
[root@BAKYRIE:~# ]vi var_numbers.sh
[root@BAKYRIE:~# ]chmod u+x var_numbers.sh
[root@BAKYRIE:~# ]./var_numbers.sh
Please enter a number:
16
The two numbers here are not equal
[root@BAKYRIE:~# ]./var_numbers.sh
Please enter a number:
13
The two numbers here are not equal
[root@BAKYRIE:~# ]./var_numbers.sh
Please enter a number:
23
The two numbers are equal
root@BAKYRIE:~#
```

The above var_numbers.sh script were executed three times with values 16, 13 and 23.

Conditional Statement

You can use conditional statement in bash like just another programming language. If-then-else' and' case' statements are primarily used in any programming language to implement condition statements. The use of a conditional statement is shown in this chapter of this tutorial by using the' if' statement. Create a bash file where a conditional statement is used with the following script. Here, the user will take two values as input and store $code and $age in the variables. If' is used to verify the $age value is higher than or equal to 18, and the $register value is 1100. If both circumstances are valid, another message will be printed, "You are qualified to see the film." will be printed otherwise "**You are not eligible to see the movie** " will be printed.

conditional.sh

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

```
echo "Please enter your code"
read code
echo "Please enter your age"
read age

if [[ $age -ge 20 && $code -eq '3300' ]]
then
echo "You are allowed to watch this movie"
else
echo "You are not allowed to watch this movie"
fi
```

Sample Output:

After taking distinct input values, the following output will appear. 1200 is provided as a code, and 19 is provided as the age for the first script run and if the condition returns completely untrue. 3300 is also provided as a code, and 23 is provided as the age of the second phase, which returns a true condition.

```
[root@BAKYRIE:~# ]vim conditional.sh
[root@BAKYRIE:~# ]chmod u+x conditional.sh
[root@BAKYRIE:~# ]bash conditional.sh
Please enter your code
1200
Please enter your age
19
You are not allowed to watch this movie
[root@BAKYRIE:~# ]./conditional.sh
Please enter your code
3300
Please enter your age
23
You are allowed to watch this movie
root@BAKYRIE:~#
```

Loops

When executing various parts of the script, the loop is used to do the job. Like other languages, Bash supports three kinds of loop statement. The three-loop statements are for loop, while loop, and until loop. A specific loop is used in the script based on the programming demands. In this chapter, the uses of these three loop types are shown using straightforward examples.

Using for loop

We will generate a bash script file with the bash script where for iteration we use the' for' loop. The' for' loop is used primarily to iterate an input list or array. A list of the name of the weekday is used here and every name of the weekday is iterated using the loop. The' If ' conditional statement is used to print a specific statement depending on the name of the weekday.

forloop.sh

```
#Read the name of a day of the week  in each iteration of the loop
for day in Monday Tuesday Wednesday Thursday Friday Saturday Sunday
do

#Check whether the day of the week is either a Monday or Thursday
if [[ $day == 'Monday'  || $day == 'Thursday' ]]
then
echo "Training on $weekday at 9:30 am"

#Check whether the day of the week is Tuesday or Wednesday or Friday
elif [[ $day == 'Tuesday'  || $day == 'Wednesday' || $day == 'Friday' ]]
then
echo "Seminar on $day at 11:00 am"
else

#Print ‘Holiday’ for other days
echo "This day $day is a Holiday"
fi
done
```

Output:

Run the script.

```
[root@BAKYRIE:~# ]chmod u+x forloop.sh
[root@BAKYRIE:~# ]./forloop.sh
Training on Monday at 9:30 am
Seminar on Tuesday at 11:00 am
Seminar on Wednesday at 11:00 am
Training on Thursday at 9:30 am
Seminar on Friday at 11:00 am
Saturday is a Holiday
Sunday is a Holiday
root@BAKYRIE:~#
```

Using while loop

Create a bash file with the following bash shell script where for program iteration, we will utilize the' while ' loop. This script will print figures that are even and divisible by five from the range 1 to 20. Here, the $counter variable is used to regulate the loop iteration, and each iteration increases the value of this variable by 1. When the' if' condition returns true, the value of $counter will be printed.

counter_script.sh

```
#!/bin/bash

#Display or echo the message
echo "Output numbers that are even and divisible by 4"

#Declare and give the counter its initial value
counter=1

#Repeat until the $counter value is less than or equal to 40
while [ $counter -le 40 ]
do

#Determine if the $counter is divisible by 2 and 6
if [[ $counter%2 -eq 0 && $counter%4 -eq 0 ]]
then
```

```
*#Print $counter without newline*
echo  "$counter"
fi

*#Increment $counter by 1*
((counter++))
done
echo "Done"
```

From the script above there are only 10 numbers within the range 1-40 that are even and divisible by 4. The following output will appear after executing the counter_script.

```
[root@BAKYRIE:~# ]chmod u+x counter_script.sh
[root@BAKYRIE:~# ]./counter_script.sh
Output numbers that are even and divisible by 4
4
8
12
16
20
24
28
32
36
40
Done
root@BAKYRIE:~#
```

Using the ‘until' loop construct example

We are creating a bash file with the following script where we are using the **‘until`** loop for iteration. This until_script.sh script will display all odd numbers from 10 to 50. The variable **$x** is used in this script for iteration.

until_script.sh

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

```
#Initialize the variable, x
x=50

#Repeat the loop until the value of $x is greater than 10
until [ $x -lt 10 ]
do

#Check whether the value of x is odd
if [[ $x%2 -gt 0 ]]
then
echo $x
fi

#Increment the value of x by 1
((x=$x-1))
done
```

The script will print all odd numbers between 50 to 10. The following output will appear after running the script.

```
[root@BAKYRIE:~# ]vim until_script.sh
[root@BAKYRIE:~# ]chmod u+x until_script.sh
[root@BAKYRIE:~# ]./until_script.sh
49
47
45
43
41
39
37
35
33
31
29
27
25
23
21
19
```

```
17
15
13
11
root@BAKYRIE:~#
```

Functions

In essence, a Bash function is a collection of commands that can be used and reused in scripts many times. The aim of a function is to assist you to increase the readability of your bash scripts and prevent writing the same code over and over again.

Bash functions are somewhat restricted compared to most programming languages. We'll explain the basics of Bash functions in this section and demonstrate you how to use them in your shell scripts.

A bash method or procedure is basically a number of instructions, which can be called repeatedly. The aim of a function is to make your bash scripts readable and to prevent the same program from being written over and over.

Bash methods are somewhat constrained in comparison to traditional programming languages. We present the basics of Bash functions in this section of the book and demonstrate their use in your bash shell scripts.

Function Declaration in Bash

The structure of declaring a bash function is intuitive. Bash functions are declared in two diverse ways:

1. The first format begins with the name of the function, followed by brackets () also known as parentheses. This is the format of choice and the most frequently used format.

```
function_name () {
        commands
        }
```

Function Single line format 1:

```
function_name () {commands;}
```

The second function syntax begins with the **function** reserved word followed by the name of the function.

```
function function_name {
commands
}
```

Function Single line format 2:

```
function function_name { commands; }
```

When a code block has to run numerous times in a script, the function is used to do the job. You will only need to call the function by name if you need to perform various times the script specified in the feature rather than adding various times the same script. First bracket start and end is used with the name of the feature to declare the feature in bash script. The method can be called in bash by just the name of the function. As another conventional programming language, Bash does not support function argument. But in this section, the value can be handed to bash function in a distinct manner. With the return statement or without using the return statement, the value can be returned from the function.

Create a bash file with the script below to learn how to declare and call the functions in the bash script. The basic_function.sh script below declares three different functions. The first procedure or function to print a basic message, "Bash scripting for beginners," is declared. The 2nd function is defined to assign a string value after calling the procedure in a variable, $return_str which will print, "Learn bash scripting here."

basic_function.sh

```
#!/bin/bash

#Declaring a basic procedure/method/function
function display_string()
{
echo "Bash scripting for beginners"
}
```

```
#Declaring a method/function to return a string value
function str_data()
{
#Initializing a variable with initial string value
str_data="Start bash scripting today"
}
#Declaring a procedure/method to read the entered argument
function circle_area()
{
#Read the input arguments
local radius=$1
circle_area=$(echo $radius*$radius*3.14 | bc)
#Display the area of a circle to screen
echo "The area of the circle given radius $radius above is $area"
}
#Invoke the function to display a string of characters
display_string

#Invoke the method to assign a string to a variable
string_data

#Display the variable string to screen
echo $string_data

#Read value of the radius entered
echo "Please enter your radius size"
read rad

#Invoke the function to create area with radius size
calculate_area $rad
```

The output below is a result of executing the script **basic_function.sh** created above. Initially, we have two calls to print_message and ret_strdata functions that will output string messages to the terminal screen. Lastly, we invoke the calculate_area function to display the area of a circle calculated from the input read from the standard input.

```
root@bakyrie:~# vim basic_function.sh
```

```
root@bakyrie:~# chmod u+x basic_function.sh
root@bakyrie:~# bash basic_function.sh
Bash scripting for beginners
Learn bash scripting today
Please enter your radius size
6
The area of the circle given radius 6 above is 113.04
root@bakyrie:~#
```

Calling programs in a script

In any bash script, you can utilize numerous types of programs like bash, source, exec, and eval to invoke other commands. As an illustration let us suppose that you have built four bash script files, sum_script.sh, deduct_script.sh, product_script.sh and division_script.sh to sum, deduct, find the product and division result. Below we have the arguments for the command line subtract_script.sh and division_script.sh. Below is the script for all these four files.

sum_script.sh

```
#!/bin/bash
x=60
y=40
((asnwer=$x+$y))
echo "The sum of x and y is $answer"
```

deduct_script.sh

```
#!/bin/bash
x=$1
y=$2
((answer=$x-$y))
echo "The result of the deduction of y from x is $answer"
```

product_script.sh

```
#!/bin/bash
((answer=$1*$2))
echo "The product of $1 and $2 is $answer"
```

divide_script.sh

```
#!/bin/bash
x=$1
y=2
((answer=$x/$y))
echo "The result of dividing $x by $y is $answer"
```

Use source, bash, eval, and exec commands to create a bash file named call_function.sh with the contents listed below. Before running the following script, you must set the permissions that allow this script to execute on the bash shell. The command ' source' is used to call the file add_script.sh. The command ' bash' is used to perform the file subtract_script.sh. The ‘eval' command is used to run the product_script.sh bash file. For the ‘eval' program, we capture the two operand values for the' eval' from the bash shell arguments. Lastly, we run the exce command, which only operates with an absolute path. To do this, divide_script.sh's the complete path name is provided in the bash file below.

call_function.sh

```
#!/bin/bash
1st_script="sum_script.sh"
2nd_script="deduct_script.sh"
3rd_script="product_script.sh"
4th_script="/home/ppeters/code/divide_script.sh"

source "$1st_script"

bash $2nd_script 50 20

echo "Enter the value of x"
read x
echo "Enter the value of y"
read y
```

```
eval bash $3rd_script $x $y
exec $4th_script 30
```

Creating Menus

Bash has a helpful command to generate a useful menu called the command' select.' Using this command, different menu kinds can be produced. This command uses a specific info list to generate a menu.

Use the vim command to create a menu_script script file with the example script code below. The menu_scipt.sh script file is an illustration of how to use the ‘select' command to create a menu of options in bash.

The menu_script script file shows a list of five menu items that will be displayed as a menu in this instance, prompting the user to select any language from the list. The selected value is stored in the $linguistic variable and subsequently printed by concatenating with another string. The bash program will frequently request that the language be selected awaiting the user to press the termination menu choice 6 to exit the menu_script.sh program.

menu_script.sh

```
#!/bin/bash

#Display this message for the user
echo "Please choose your favorite language"

# Make language list for the menu item
select language in Perl C++ Javascript Bash C# Exit
do
#Display the chosen value
if [[ $language == "Exit" ]]
then
exit 0
else
echo "Your chosen language is $language"
fi
done
```

According to the example output below, the user pressed four initially that printed Bash then pressed 5 for the second time which displayed the C#, then pressed 1 for Perl and lastly pressed 6 to exit the program.

```
[root@BAKYRIE:~]# chmod u+x menu_script.sh
[root@BAKYRIE:~# ]./menu_script.sh
Please choose your favorite language
1) Perl
2) C++
3) Javascript
4) Bash
5) C#
6) Exit
#? 4
Your chosen language is Bash
#? 5
Your chosen language is C#
#? 1
Your chosen language is Perl
#? 6
done
```

Command Line Processing

Sometimes when executing the script from the command line, we need to provide input values. In bash, this assignment can be performed in two respects. One way is to use variables of argument, and another is to use the function getopts. This segment shows the input of command-line information from the terminal using the argument variable.

Create a bash script file with the accompanying script to view the value of the argument reading from the command line. This script will read three command-line arguments, $operand1, $operand2 and $operator, which will be stored in the variables. The number and second argument value must be one of the four arithmetic operators (' +','-',' /','x') to perform the script correctly. If the declaration checks the $operator value and performs the operator-based procedure and the value is printed.

commandline.sh

```
#!/bin/bash

#Display the argument variable list
echo "Your argument values are: $1 $2 $3"

# Store the argument values in operands
operand01=$1
operand02=$3
operator=$2

#Do a Check of the second command argument value to do the arithmetic operation
if [[ $operator == '+' ]]
then
((result=$operand01+$operand02))
elif [[ $operator == '-' ]]
then
((result=$operand01-$operand02))
elif [[ $operator == 'x' ]]
then
((result=$operand01*$operand02))
elif [[ $operator == '/' ]]
then

((result=$operand01/$operand02))
fi

# print the result
echo -e "Your operation result is = $result"
```

The commandlinescript.sh script above is run for four times using four different types of arithmetic operators. The following output will appear for the argument values, **12 + 6, 12 – 6, 12 x 6 and 12 / 6** . The result output below shows all the inout and the results.

```
[root@bakyrie:~]# vim commandlinescript.sh
[root@bakyrie:~]# chmod u+x commandlinescript.sh
[root@bakyrie:~]# ./commandlinescript.sh 12 + 6
```

```
Your argument values are: 12 + 6
Your operation result is = 18
[root@bakyrie:~]# ./commandlinescript.sh 12 – 6
Your argument values are: 12 – 6
Your operation result is = 6
[root@bakyrie:~]# ./commandlinescript.sh 12 x 6
Your argument values are: 12 x 6
Your operation result is = 72
[root@bakyrie:~]# ./commandlinescript.sh 12 / 6
Your argument values are: 12 / 6
Your operation result is = 2
[root@bakyrie:~]#
```

Arguments are used in bash, and they are passed to the name-value pairs in bash.

WE would like you to create a commandline2.sh bash file, with the following script to demonstrate how to read argument values using their name. The command-line bash script will read two argument variables. The named argument values are printed in the script's first statement.

Next, a loop will be used to iterate the array containing the value of the command line argument. By using the' cut' command, each element in the array is separated into a key-value pair. Next, a case statement is used to print a particular message based on the key value.

commandlinescript2.sh

```
.# !/bin/bash

#Display the list of argument variables
echo "Your argument values are: $1 $2"

#Read each argument independantly using for loop
for arg in "$@"
do

#Separate the argument name and the value
key=$(echo $arg | cut -f1 -d=)
```

```
value=$(echo $arg | cut -f2 -d=)

#Display a message based on argument's name
case $key in
name) echo "The Student's name is $value";;
mark) echo "The obtained mark is $value" ;;
*)
esac
done
```

The result below shows the output from executing the commandlinescript.sh, the script is supplied with two command-line arguments which are; name= "James Jones" and mark=83. The two command-line arguments are split by the script, and the resulting two values are displayed on the screen.

```
root@bakyrie:~# vim commandlinescript2.sh

root@bakyrie:~# chmod u+x commandlinescript2.sh

root@bakyrie:~# ./commandlinescript2.sh name="James Jones" mark=83

Your argument values are: name=James Jones mark=83

The Student's name is James Jones

The obtained mark is 83

root@bakyrie:~#
```

Performing Arithmetic Operations

It is a common requirement for many programming languages to perform arithmetic operations. Bash performs the arithmetic operations differently from other standard languages of programming. Arithmetic operations in bash are performed in multiple ways. This section shows one of the simple ways to do arithmetic operations.

We are going to create an example bash file called arith_script.sh with the script below. This example script is a demonstration of how to perform the four basic types of arithmetic operations. We will use double brackets "(( ))" at the beginning of the script to show simple addition and division

operations. The addition and division operators are flanked by operands, which are number enclosed in double brackets. After the addition and division operations, we move next, to the pre-increment operation, as shown below. Finally, in the last part of the script, there's demonstration of the shorthand notation to perform increments.

arithmetic_script.sh

```
#!/bin/bash
# Do the sum operation
result=$((60+150))
# Display to screen the addition result
echo "Addition = $result"

# Divide the values given
result=$((150/25))
# Display to the screen the division result
echo "division = $result"

# Assign a value to X
X=10
# Doing pre-increment
((--X))
# Print the value of X
echo "Value after decrement = $X"

# Using shorthand operator
(( X += 10 ))
# Print the value of X
echo "Value after adding 10 = $X"
```

From above all the arithmetic operations are executed through the use of double brackets. We may use ‘let', ‘**expr** ' and ‘**bc** ' commands to do arithmetic operations in bash.

Manipulating Bash Strings

The string data in bash can be used to perform many types of tasks. Some are string concatenation, string comparison, string splitting, string case

change, etc. There are no bash built-in string functions to do the string operations like in other standard programming languages. The section of this book discusses some common string manipulations.

String Concatenation

String concatenation is the process of adding two or more strings together. Strings are joined together in bash by appending one string after another. We can create a bash file with the following script to show the use of string concatenation. We will initialize two string variables and display them after combining the two string variables. The example shows that the content of **$stringcat01** and **$stringcat02** will be merged and printed.

stringconcat.sh

```
#!/bin/bash

#Initialize first string variable
stringcat01="This user "
#Initialize second string variable
stringcat02="Loves Bash Scripting"
#Display the output after combining both string variables
echo "$string1$string2"
```

The script above produces the result below

```
[root@bakyrie:~]# vim stringconcat.sh
[root@bakyrie:~]# chmod u+x stringconcat.sh
[root@bakyrie:~]# ./stringconcat.sh
This user Loves Bash Scripting
[root@bakyrie:~]#
```

String Comparison

To make string data comparisons, bash uses a unique kind of comparison operators. We are going to create a bash file with the script below to demonstrate how two strings are compared. In the stringcompare.sh script, a

string value is read from the standard input as an input and is compared to the other string. If the first string value matches, then a message is displayed, "You can program in Python" or "You can program in PERL."

stringcompare.sh

```
#!/bin/bash
echo “Enter any string value”
read text
#Check the input data is equivalent to “Python”
if [ $text == "Python" ]; then
echo "You can program in Python."
else
echo "You can program in PERL"
fi
```

Bash String splitting

To divide string information, Bash has no built-in divided feature. The string information can be split in bash depending on distinct kinds of boundaries in various ways. Create a bash file with the script below to demonstrate how to split the string information into bash. As input, a string value is assumed. This script divides the space-based value of $text. Here, the delimiter is set using the IFS variable. To split the text value and store the values in an array, the' read' command is used here. For loop, the array is iterated, and the value of each element is printed.

stringsplit.sh

```
#!/bin/bash
#Enter a text string into the terminal
echo “Please input a text string here”
read str_text
# Set up the space delimiter
IFS=' '
#Split the text string $str_text into an array using delimiter
read -a str_array <<< "$str_text"
# Display each instance of the array str_array
for value in "${str_array[@]}";
```

```
do
printf "$value\n"
done
```

Below is the example output from the stringsplit.sh bash script

```
root@bakyrie:~# vim stringsplit.sh
root@bakyrie:~# chmod u+x stringsplit.sh
root@bakyrie:~# bash stringsplit.sh
“Please input a text string here”
How are you bash users
How
are
you
bash
users
root@bakyrie:~#
```

Changing Case of the String

Most languages of scripting have built-in features to alter the string data case. But using the' tr' command or using the keywords': upper' and': lower' can change the result of the string data in bash. Create a bash file with the script below to understand how to change the bash situation. Here, by using the symbol' ^^' the first string information is transformed to uppercase and the second string is transformed to lowercase by using the command' tr.' The command' tr' searches all the upper letters in the string and converts the characters to lower case.

stringcase.sh

```
#!/bin/bash
#Initialize the 1st text string
textstr1='james@company.com'
```

```
#Print the value of $textstr1 by converting all characters to uppercase
echo “${email^^} ”

#Initialize the 2nd text string
textstr2=’Bash scripting basics’

#Print the value of $textstr2 by converting all uppercase to lowercase
echo $textstr2 | tr [:upper:] [:lower:]
```

Reading string data through the loop

The string data functions for any programming language as a character array. In this section of the book, we discuss how the ' for ' loop is used to read text string data in bash. Create a bash script file with the content script below to use a loop to read each portion of the text string value.

readstring.sh

```
#!/bin/bash
# using for loop to read each string
for value in Bash Programming for the Beginners
do
echo $value
done
```

Bash Return Codes and Catching

In Linux, some bash script may capture the return codes from the next script file using the '$?' symbol. The ‘$?' symbol is used to call the second script so that it switches to read the returned code from a previous bash script file.

Our illustration below has two bash scripts, namely firstscript.sh bash file and secondscript.sh. From the example firstscript.sh bash file returns a code after execution.

The second bash script file secondscript.sh below has script to catch the return code from firstscript.sh and perform a specific task. The two scripts are shown below. The bash script firstscript.sh file is called at the beginning of the secondscript.sh file. Firstscript.sh returns the input value-based exit code. Secondscript.sh captures the '$?' code and compares it to 1. If both

values are equivalent, the terminal screen will be displayed with the message "The number you entered is greater than 100," and if not equal the terminal screen will have the message, "The number you entered is less than or equal to 100."

firstscript.sh

```
#!/bin/bash

echo "Please enter a number"
read x

# Checkif the number you entered is less than or equal to 100
if [[ $x -le 100 ]]
then
exit 0
else
exit 1
fi
```

secondscript.sh

```
#! /bin/bash

#Run the bash file , firstscript.sh
bash "first.sh"

#Check if the return code is equal to 1
if [ $? -eq 1 ]
then
echo "The number  you entered is greater than 100"
else
echo "The number you entered is less than or equal to 100"
fi
```

Output:

Run the script.

```
$ bash secondscript.sh
```

The following output will appear when the script is executed and passed the values 56 and 110 one at a time.

```
[root@bakyrie:~]# vim firstscript.sh
[root@bakyrie:~]# chmod u+x firstscript.sh
[root@bakyrie:~]# vim secondscript.sh
[root@bakyrie:~]# chmod u+x secondscript.sh
[root@bakyrie:~]# ./secondscript.sh
Please enter a number
56
The number you entered is less than or equal to 100
[root@bakyrie:~]#
root@bakyrie:~# ./secondscript.sh
Please enter a number
110
The number  you entered is greater than 100
root@bakyrie:~#
```

Bash reads and writes to files

Bash script should be able to read from and to write to files as a basic requirement. As with all other mainstream programming languages, Bash has no built-in function to read from and to write to files. There are several methods in bash to read from a file. One of the most popular methods to read from or write to a file in bash is using the bash 'cat' command, also known as the concatenate command. The cat command is primarily used to read the file's entire content. We can use the looping and' read ' command to read the bash file line-by-line. Using the redirect operator,' >' greater-than symbol, it is essential to write data to bash files. We can also use the double greater-than symbol ' > >' symbol to append more contents to any bash file.

Reading file in bash

We are going to create a bash file with the following bash script to read an existing file called ‘**hardwarefile.txt** .' The contents of this hardware file are given below. In the bash script, the entire contents of the file are read by the **`cat`** command first and next; the while loop is used to read the bash file line by line.

hardwarefile.txt

```
DvdRom
RAM
Monitor
Keyboard
Mouse
Scanner
Printer
```

readfilescript.sh

```
#!/bin/bash

echo "Reading the hardwarefile.txt file using cat command"

# Read the content of the file using `cat` command
content=`cat hardwarefile.txt`
echo $content

echo "Reading the file line by line using loop"

# Assign the filename
filename='hardwarefile.txt'

# Each line of the file will be read by each iteration of the loop
while read line;
do
# print the line
echo $line
done<$filename
```

Output:

Run the following commands.

```
$ cat hardwarefile.txt
$ bash readfilescript.sh
```

From the illustration above the first command will display the contents of the file, **hardwarefile.txt** without running any bash script and the second

command will run the script of **readfilescript.sh** and print the content of the file for two times by using **`cat`** command and **`read`** command with while loop.

```
root@bakyrie:/home/ppeterszw# cd
root@bakyrie:~# vim hardwarefile.txt
root@bakyrie:~# vim readfilescript.sh
root@bakyrie:~# cat hardwarefile.txt
DvdRom
RAM
Monitor
Keyboard
Mouse
Scanner
Printer
root@bakyrie:~# chmod u+x readfilescript.sh
root@bakyrie:~# ./readfilescript.sh
Reading the hardwarefile.txt file using cat command
DvdRom RAM Monitor Keyboard Mouse Scanner Printer
Reading the file line by line using loop
DvdRom
RAM
Monitor
Keyboard
Mouse
Scanner
Printer
root@bakyrie:~#
```

Writing File in Bash

The bash script created below in the bash file is used to write new content in a new empty file and append data in the file if it has content already.

writefilescript.sh

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

```
echo "Please enter any text string"
#Read string data
read string01
#Add input data into the file for the first time
echo $string01 > testfile.txt

echo "Please enter some other text string"
#Read another text string
read string02
#Append some input data at the end of the file
echo $string02 >> testfile.txt

#Show the full content of the file
echo `cat testfile.txt`
```

The following output result will be displayed after running the script writescript.sh.

```
root@bakyrie:~# vim writescript.sh
root@bakyrie:~# chmod u+x writescript.sh
root@bakyrie:~# ./writescript.sh
Please enter any text string
Phew eish
Please enter some other text string
Can be tough
Phew eish Can be tough
root@bakyrie:~#
```

Piping in Bash

A bash shell pipe is a way to connect the output of one command or program to the input of yet another command without using any temporary files to store the preliminary results of the previous command.

Syntax

```
1st_cmnd | 2nd_cmnd

1st_cmnd | 2nd_cmnd | nth_cmnd
```

```
1st_cmnd arg01 | 2nd _cmnd arg01 arg02

get_info_cmd | check_info_cmd | execute_info_cmd | build_info_cmd
> result_data.file

get_data_cmd < input_data.file | verify_data_cmd | process_data_cmd |
format_data_cmd > output_data.file
```

You can pipe or link two or more commands concurrently.

- The data path, which links the two commands, is called a *pipe* .
- The bash vertical bar (|) is called the pipe symbol.
- A shell pipe supports the Linux philosophy of chaining commands together to complete complex jobs.
- Redirection in pipes is also allowed.
- The data path only works in one direction:

In bash scripting Pipes "|" are utilized to redirect bash command output from one program to the input into the other command. Pipes are used with several types of bash programs to particular output results. The bash command-line interface input is linked to the command output result through the use of the bash pipe operator. There is a need to write the bash commands in sequence using the pipe (|) symbol to get the expected output. With the pipe utility, we can execute two or more commands jointly in a single command by using a pipe. Without the pipe operator, users have to carry out compound commands in several lines to do the same straightforward job. Piping is advantageous for making sure that multiple lines of commands are shortened to ensure the Linux job is done in simple.

Pipe syntax

```
1st _cmnd | 2nd _cmnd| … | nth_cmnd
```

The example above shows that the output of program first-cmnd will be passed as the input of 2nd_cmnd.

A similar type of undertaking that is done with several lines of instructions can be executed with the pipe operator. For instance, **student_marks.txt**

text file has the following output below.

student_marks.txt

```
James   BSE-109      55
David   BSE-104      65
Sarah   BSE-101      87
Susan   BSE-104      58
Daina   BSE-109      80
Susan   BSE-101      72
```

We are going to demonstrate a bash script that initially sorts or arranges the contents of the students_marks.txt file. The bash script then runs the grep command to look for and print all the entries of the string representing course ‘BSE-101' . Normally this can be done by using three separate commands. In this instance, we are going to use piping to join together the three commands into one. The first command to the left in the script sorts the file alphabetically. The second command will search the course code text string ‘BSE-101' using the **`grep** ` command. The results of the grep command are stored in **a tempfile.txt** file. The third command executes a word count command that displays the total count of lines, number of words, and the total count of characters in the **tempfile.txt** file by using **`wc`** command. Below is the rudimentary way of sorting, searching, and redirecting using separate multiple commands on the terminal.

```
[root@bakyrie:~]# sort student_marks.txt
[root@bakyrie:~]# grep 'BSE-101' student_marks.txt > tempfile.txt
[root@bakyrie:~]# wc tempfile.txt
```

Example Output Result:

There are three entries of the course, with name ‘BSE-101' that exist in the file student_marks.txt. There are nine different words that are also present and 79 characters altogether. Below is the result of running the three separate commands.

```
root@bakyrie:~# vim student_marks.txt
root@bakyrie:~# sort student_marks.txt
```

```
Daina   BSE-101        80
David   BSE-104        65
Sarah   BSE-101        87
Susan   BSE-101        72
Susan   BSE-104        58
Susan   BSE-109        55
```

```
root@bakyrie:~# grep 'Susan' student_marks.txt
Susan   BSE-109        55
Susan   BSE-104        58
Susan   BSE-101        72
root@bakyrie:~# wc -l tempfile.txt
3 tempfile.txt
root@bakyrie:~#
```

From the illustration above, we can easily join together all the three lines of commands above and produce the same output result by running a single command with pipes, as shown in the following example command. We do not use a temporary file here to get the output result. The illustration below shows the chaining of output from the sort command into the input of the grep command and lastly the word count of the result from grep.

```
[root@bakyrie:~]# sort student_marks.txt | grep 'BSE-101' | wc
```

Output

The results of running the above-piped statement are the word count of anything that contains the string "BSE-101" from the student_marks.txt file. The word count firsts lists the number of occurrences of the string being grepped "BSE-101" and also outputs the number of words in this instance we have nine words and lastly displays the total number of characters in the results which is seventy-nine in our example above. The output of the command will be 3 9 79. Three entries of output with string "BSE-101", 9 words and 79 characters counted.

```
root@bakyrie:~# sort student_marks.txt | grep 'BSE-101'
```

```
Daina   BSE-101        80
```

```
Sarah   BSE-101        87
```

```
Susan   BSE-101        72
root@bakyrie:~# sort student_marks.txt | grep 'BSE-101' | wc
      3       9      79
root@bakyrie:~#
```

Bash Scripting Exercises

First Exercise:

In your first exercise, you will be creating a bash script to display on the terminal screen the message, "Having fun with Shell Scripting!"

Create exercise01.sh

```
#!/bin/bash
echo “Having fun with shell Scripting!”
```

Second Exercise:

Modify the bash script you did from the first exercise above to introduce a variable. The variable will store the contents of the message "Having fun with shell Scripting!"

Create exercise02.sh

```
#!/bin/bash
str_info= “Having fun with shell Scripting!”
echo $str_info
```

Third Exercise:

This exercise is about using variables to store the output of the command "hostname." Print "This script is running on the machine called _." where "_" is the output of the "hostname" command.

Create exercise03.sh

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

```
HOSTNAME=$(hostname)
echo “This script is running on the machine called $HOSTNAME”
```

Fourth Exercise:

This exercise is about creating a bash file used to determine whether a file's path is in existence. When the script finds that the file path does exist, it will display a message that the "file's path does exist." The next part of the script checks to determine if you are able to write to the file. If the user has permissions to write to the file the terminal screen will have a message, "You have rights to alter the file's path." Otherwise, if you have no permissions, display on the terminal console the message that "You do NOT have permissions to edit file's path."

Create file exercise04.sh and input the script below

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

```
USER_FILE=”/home/ppeters/exercises/test_file.txt”
```

```
if [ -e “$USER_FILE” ]
  then
     echo “The $USER_FILE path does exist”
fi
```

```
if [ -x “$USER_FILE” ]
  then
    echo “You have permissions to process $USER_FILE”  else
    echo “No permissions to process $USER_FILE”
fi
```

Fifth Exercise:

In this exercise the learner is expected to create a Linux bash script file called exercise05.sh that displays on the terminal screen "chihuahua","doberman","hound","basset","pitbull"and "corgie" on the terminal screen with each one of the dog breeds appearing on a separate line. The learner should do this in as few lines as possible.

Create exercise05.sh

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

```
DOGS=”chihuahua doberman hound basset pitbull corgie”
```

```
for DOG in $DOGS
  do
    echo $DOG
  done
```

Conclusion

Linux is a wonderfully diverse language and you should take the time to explore it. Granted it might seem a bit ocmplex at first but with constant practice, you'll being to see why the mouse is overrated as a companion!

Jokes apart, my aim with this book has been to give you an introduction and walk you through some of the most basic linux commands and features. The idea is for you to work your way through the sample code at your own pace.

There are many applications Linux lends itself to and this is precisely what my other books on the topic cover. For example, how would you like to be an ethical hacker and start a career in network security and administration? Linux truly is a programmer's dream.

Hopefully you've enjoyed this under the hood look at what goes on with Linux. I hope this book has been an enjoyable and informative erad for you. Do let me know what you think and be on the lookout for the next book in this series in which I'll take you deeper into the world of Linux and bash scripting.

I wish you the best of luck in your Linux learning adventures!

References

Negus, C. (2015). *Linux ® Bible Ninth Edition* (9th ed.). Wiley.

Petersen, R. (2008). *Linux: The Complete Reference, Sixth Edition* . McGraw-Hill.