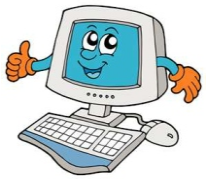
6 Basic Ubuntu Linux Commands



In standard 9 we have studied in depth the working of an operating system called Ubuntu. During the study we learned that once a user logs into the computer system having Ubuntu Linux, he/she can interact with the computer using command line interface or graphical interface. Both these interfaces are important and have their own uses. In this chapter we will learn how to use the command line interface in detail. The command line interface allows us to access the real power of Linux with greater efficiency. It is the most influential parts of any Ubuntu system.

Starting Up the Terminal

To open a command-line console in a graphical interface, a window named terminal window is provided in Linux. To open a terminal window, click on Applications àAccessories à Terminal alternatively you can use CTRL+ ALT + t keys together. A terminal window similar to the one shown in figure 6.1 will be seen on the screen. Note that the look of your actual window may be different from the one shown in figure 6.1 as the user might be different.

Figure 6.1 : Terminal Window

Once the window is clearly visible, you will see a blinking cursor preceded by some letters, and perhaps numbers and symbols, ending with a $. The first word in that string of characters is username, followed by the symbol @. The symbol @ is followed by the name of the computer that is being used. Finally you will find a colon and the name of the directory you are working in (Generally you start working in your home directory, which is represented by a ~ symbol).

The command prompt indicates that the interface is ready to interact with the user in the form of commands. A command is basically a program that accomplishes certain task. Once the prompt is displayed we can issue commands as described in this chapter. Before discussing various commands, let us revise our knowledge about the term shell. A shell is the command-line interface. Shell is a user program or an environment provided for user interaction. It is a command language interpreter that accepts or issues commands, understands it, interacts with the kernel to execute it and displays results as per given instruction. Numerous shells are available to work on the Ubuntu Linux systems, but the shells available on a particular system may vary.

Some popular shells provided with Linux are Bourne shell (sh), C shell (csh and tcsh), Korn shell (ksh) and bash (sh) shell. Bourne shell with sh as its acronym is the earliest Unix shell used as command line interface. Bourne shell provides basic mechanisms for shell script programming, which allows us to write a program based solely on commands. C shell identified as csh is another shell commonly found on Linux systems. Shell programming can be done using C programming syntax in this shell. The newer version of csh is tcsh. It provides additional shell script programming features to address the limitations of csh. The Korn shell or ksh was developed to combine the features of both sh and csh. Bash shell is a newer version of Bourne shell. Thus it contains same syntax and functions as sh. Nowadays bash is considered standard shell for Linux systems and is thus commonly used and available on all Linux operating systems.

Listing the shells available on the system

To find all available shells in your system you can use the cat command. Type the command as shown in figure 6.2 on the command prompt. You will get list of all the available shells in your computer system. The list of shell that you get as an output may be different from the one shown in figure 6.2 depending on your system configurations. The cat command is discussed later in this chapter.

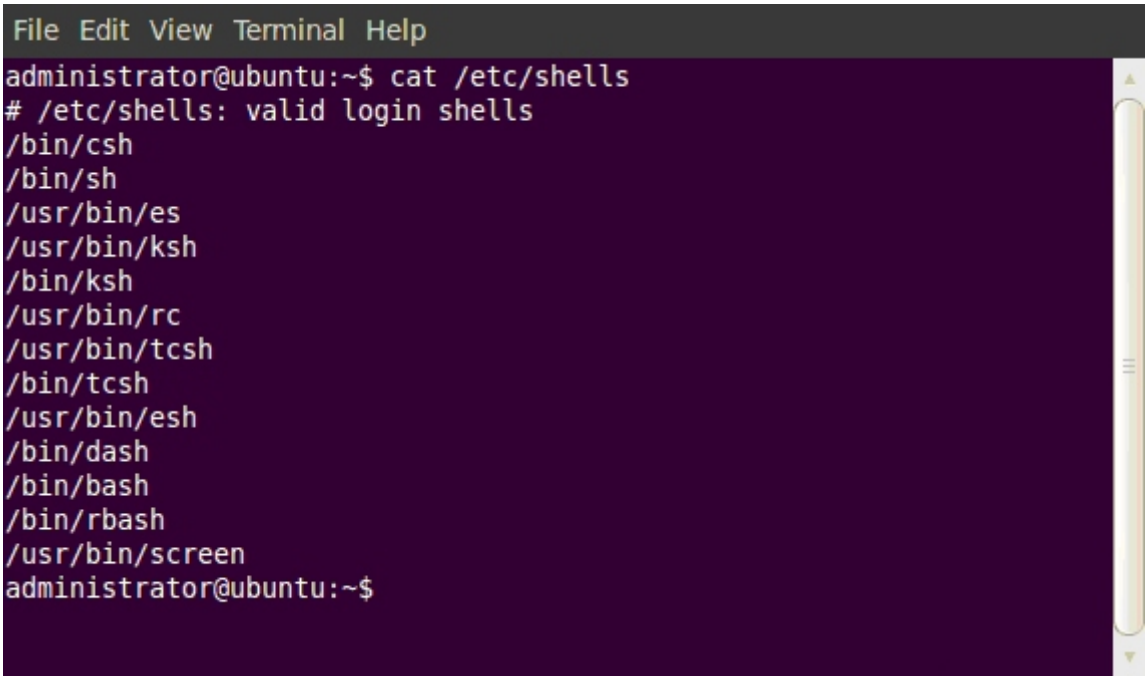


Figure 6.2 : Different available shells

Determining the default shell

Each Ubuntu Linux account is configured with a certain shell as its default command line interface. Each time you log on, this default shell is utilized for working within the system. Linux operating system comes with certain variables that contain current environment settings as its values and thus are known as environment variables.

The value of default shell is stored in one such environment variable named SHELL. Thus by displaying the value of the variable SHELL, we can come to know which our default shell is. To display the value of any variable, *echo* command can be used. Type echo $SHELL on the command prompt and press Enter key. Default shell will be displayed on the screen as shown in figure 6.3. Note that Linux commands are case sensitive hence SHELL, Shell and shell are not same.

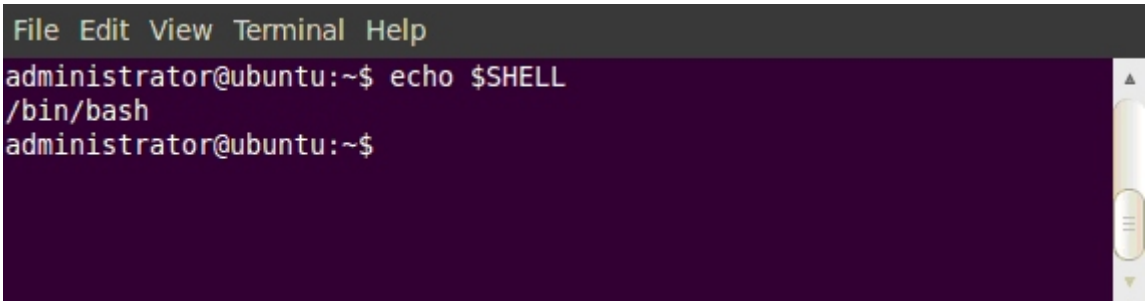


Figure 6.3 : Default Shell

Changing the current shell

As discussed earlier we have different shells available with Linux. To change your default shell, type the name of the shell you want to use on the command line. For example, if you want to use the C shell (provided it is available on the system), type csh at a command prompt. Then the command prompt will provide a csh interface.

Note :
A shell change is temporary and will last only as long as you are logged on that command line.

To return to default shell, type exit or press CTRL + d at the command prompt of the new shell.

Command Syntax

The syntax of Linux commands is uniform. It consists of three parts, in the order specified below :

- **Name :** It is the name of the command, for example ls, echo etc.
- **Options :** It is possible to alter the behavior of the commands by specifying additional options. A command may have zero or more options. Options when present starts with a hyphen symbol (-) and are usually a single letter or a digit. Some commands may have options with double hyphen and/or sequence of letters or digits. Depending on the command, the number and meaning of the options will vary.
- **Arguments :** Along with options user can also provide arguments. A command may take zero or more arguments to do its work. The number and expected meaning of the arguments vary from command to command. Some commands may take no arguments; others may take an exact number, while other commands may take any number of arguments.

Linux commands can be classified as internal or external based on whether its binary file exists or not. The commands that have a binary file explicitly stored in either /sbin, /usr/sbin, /usr/bin, /bin, or /usr/local/bin directories are called external commands. They are generally executed by the kernel and will generate a process id at the time of execution. Most of the commands that we use in

Linux are external commands. On the other hand the commands directly executed by the shell are called internal commands. Internal commands do not generate a new process.

To know whether a command is internal or external we can use the *type* command. The syntax of type command is shown below :

$type command

For example if we execute a command

$type info

we will get the output as shown below :

info is /usr/bin/info

As can be observed this refers to a binary file *info* stored in /usr/bin/, this indicates that info is an external command.

Issuing General Purpose Commands

As you are now familiar with the syntax of commands, let us see how to issue commands through command line interface. The best way to start with learning Linux commands is to try working with some general purpose Linux commands. You can issue a command by typing a command name followed by necessary options and arguments. Other way is you can type in first few letters of a command, press the tab key and the shell automatically provides the remaining information. For example, to display a calendar, type ca on the prompt and press tab key. Linux shell will automatically display list of all commands starting with alphabets ca including the calendar command *cal* on the screen. If you get more than one command in the list, then type the desired command on the prompt and press Enter key to execute the command.

Calendar (*cal*)

The *cal* command is used to display a calendar of any specific month or entire year. The default output of *cal* command is calendar of the current month. See figure 6.4.

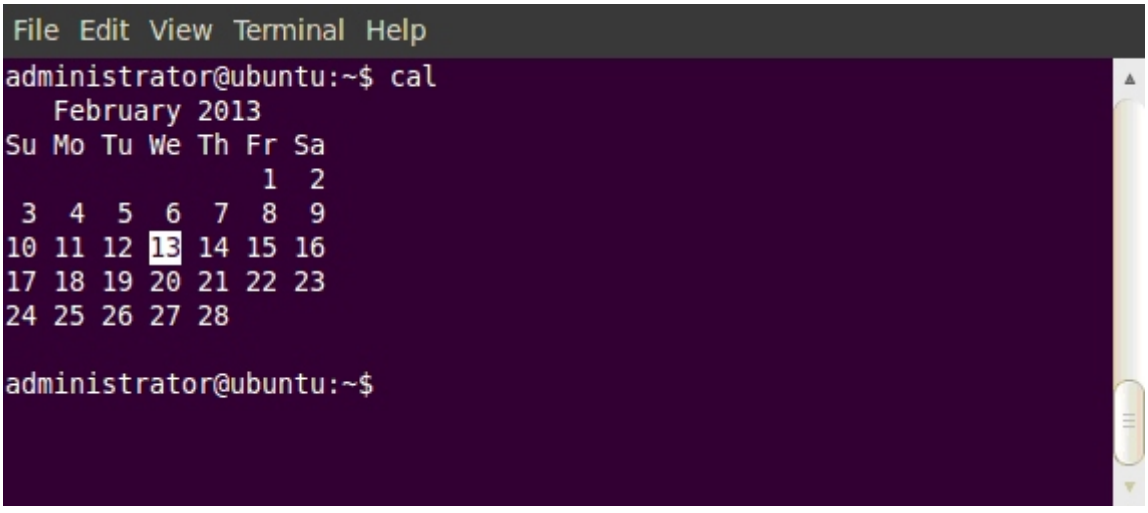


Figure 6.4 : Output of cal command

We can change the calendar as per our requirement, for example to display the calendar of January, 2013 type the following command on the prompt and press Enter key.

$cal 01 2013

The output of this command will be similar to the one shown in figure 6.4 except that the month would be January 2013. Similarly if we want to display the calendar of the entire year 2013, we simply have to type cal followed by the year as shown below :

$cal 2013

Note that the calendar of entire year may not be displayed on the entire monitor screen; hence we will have to use a pipe operator as shown below :

$cal 2013 | more

In the above command we have concatenated two commands, here *more* is also a command which takes input from the *cal* command. The pipe (|) symbol used in between the two commands is discussed in detail later in this chapter.

Date (*date*)

Another utility command is *date;* it is used to display the system date.

$date

The output of the command is shown in figure 6.5.

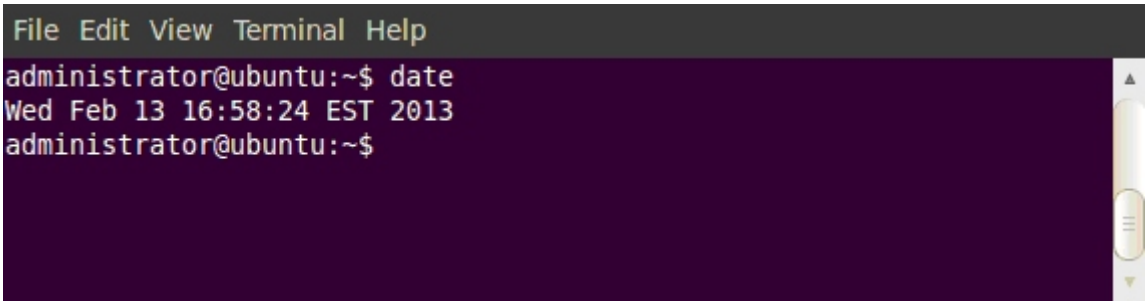


Figure 6.5 : Output of date command

Observe that the output displays both date as well as time. The date command can also be used with suitable format specification as arguments. Each format is preceded by + symbol, followed by % operator and a single character describing the format. For example, to display only the current date in mm/dd/yy format use the command shown below :

$date +%D

Figure 6.6 shows the output of the command.

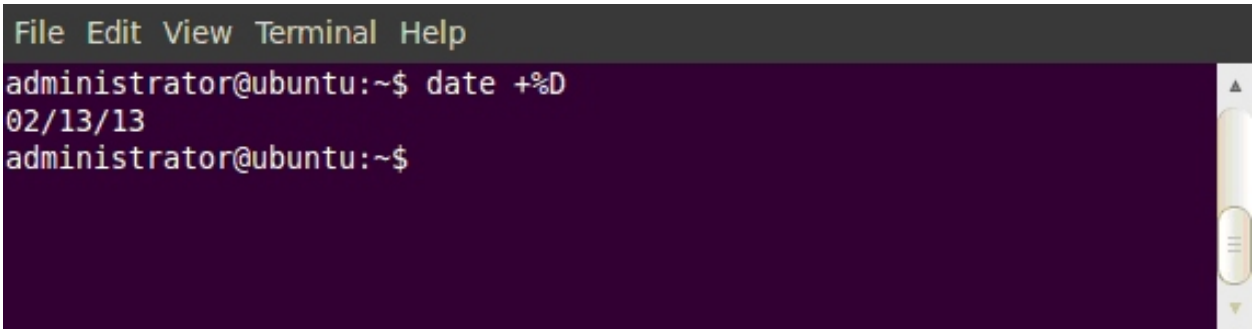


Figure 6.6 : Output of formatted date command

The command line calculator (*bc*)

The *bc* command in Linux is a command line calculator. In addition to performing simple mathematical functions, it can also perform conversions between different number systems, as well as allows us to use some scientific functions. To work with this command use the syntax given below :

$bc -l

A screen similar to the one shown in figure 6.7 will be displayed. Notice that the dollar prompt is not visible on the screen; this indicates that the bc command is now ready to take input from you. The -l switch is used to include the standard math library.

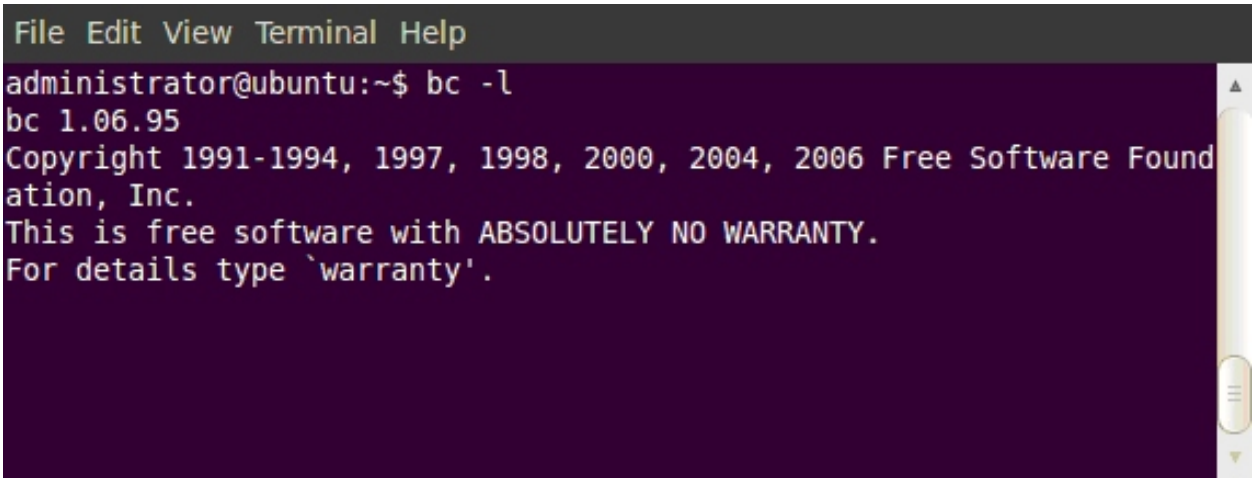


Figure 6.7 : Initiation of bc command

Now, just type the formula that you want to evaluate at the blinking cursor and the press Enter key. You can type a simple expression like 5 * 5, or you may type a complex expression with grouped operators. Let us type the expression **((99.1 / 5.15) * 99.9)**, now press the Enter key. The command will display the output in the next line as shown in figure 6.8.

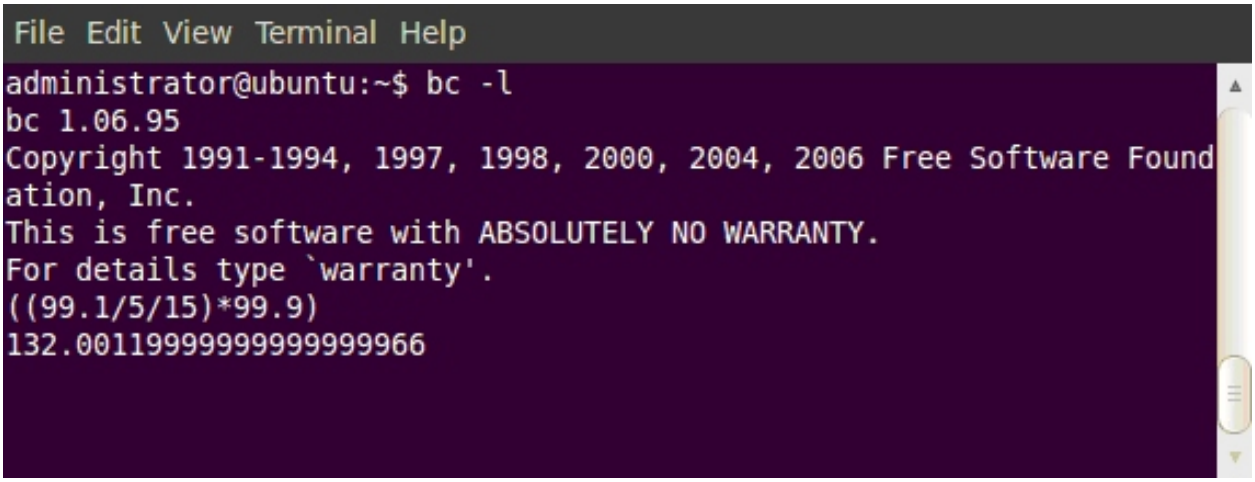


Figure 6.8 : Output of bc command

Observe that we still are not able to see the prompt; this simply means that we can continue working in command line calculator mode.

In addition to the normal mathematical functions like addition, subtraction, multiplication, division, modulus, and exponents, we can also use trigonometric or logarithmic functions like sine, cosine, arctangent, and log. For example, if you need to find the natural logarithm of value 2013, use the command **l(2013)** and you will get output **7.607381425639791484 20.**

The *ibase* function allows us to set the numbering system that we want to use for input. Similarly the *obase* function allows us to set what numbering system to use for output. Let us try to convert numbers from decimal number system to hexadecimal number system. First, we need to set *obase* as shown below :

obase = 16

Now, type the number you want to convert to hexadecimal as shown in the example and press the Enter key.

256

100

Here 100 is the hexadecimal equivalent of decimal number 256.

Similarly if you want convert this result to binary number system then just change the obase again as shown here.

obase=2

Now type **100** and the output will be 1100100, observe that this is binary equivalent of decimal 100 and not 256. The reason for this is very simple, we have not changed ibase, and hence all entries are considered to be decimal entries.

To convert hexadecimal value 100 to its binary equivalent, set ibase and obase as shown below :

ibase=16

obase=2

Now type a hexadecimal number that you need to convert to binary. For example, type **100** and you will get **100000000** as a result.

To return back to decimal mode set ibase to 10. Execute the following to find out the square root of a number using sqrt function available in math library.

sqrt(256)

16.00000000000000000000

As compared to graphical calculator the command line bc calculator is more faster and flexible. To return to the Linux command prompt, press CTRL+ d.

Displaying a message (*echo*)

We need to display message very frequently when using command prompt. The *echo* command is used to display a message on the terminal. For example, type the following command and press the Enter key. The string written after echo will be displayed on your monitor screen.

$echo Hi, I am learning Ubuntu Linux

Hi, I am learning Ubuntu Linux

It is also possible to enclose the string within double quotes. The output will not contain the double quotes. The echo command can also be used to display values of variable. For example, define a variable named cost and assign it value 10 as shown below :

$cost=10

Once you press the Enter key you will be returned to prompt. Now type the command given below :

$echo The cost of product is Rs. $cost

The cost of product is Rs. 10

To display the value of cost on the screen it is passed to the echo command. Notice that in the string we have written cost twice. The one that is prefixed with $ symbol represents a variable, while the other is a normal string. When the echo command finds any string prefixed with a $ symbol will consider it to be a variable. It will then try to print the value of variable.

The echo command can be used along with other commands to give meaningful output. For example we may combine the echo and the date command in the manner shown below :

$echo Current time is ‘ date +%T ‘

Current time is 14 :55 :04

Observe that the command to be executed is placed within back quotes (quotes available on key with ~ sign), thus first the date command will be executed and then its result will be displayed using echo command.

Changing password (*passwd*)

A user needs to change password very often due to various reasons. The *passwd* command helps us perform this operation. It is used to change the password of the current login account by default. The following command allows you to change the password.

$passwd

Once you press the Enter key a message similar to the one below will be shown, along with the blinking cursor.

Changing password for administrator

(current) UNIX password :

Type your current password and press the Enter key, Linux will check whether you have entered a valid password, and if you have then it will prompt for the new password. You will be asked to enter new password and retype it again as shown below :

Enter new UNIX password :

Retype new UNIX password :

passwd : password updated successfully

If you have typed the new password correctly and it does not conflict with any guidelines decided for password, your new password will be registered by the system. In case of any problem you may get an error message. It is also possible to change the password of other user of the system by specifying the username after the passwd command. For example, if we have a user named harshal, to change its password we may type the following command.

$passwd harshal

In case of genuine user you will be allowed to change its password.

Clearing the Screen (*clear*)

While working on command prompt you must have observed that the screen often gets full. At times it also becomes difficult to see the output clearly, we have one simple solution for this problem, use the *clear* command to remove data on the screen.

$clear

Getting Help on the Linux Commands

Before looking at any other commands first let us learn how to get help when using commands on Linux platform. Linux provides two inbuilt commands namely *help* and *man* to assist the user while working on the command line interface. All the commands that we use in Linux supports the -h (or -help) option. This option generates a small description of how to use the command. Figure 6.9 shows the use of help command.

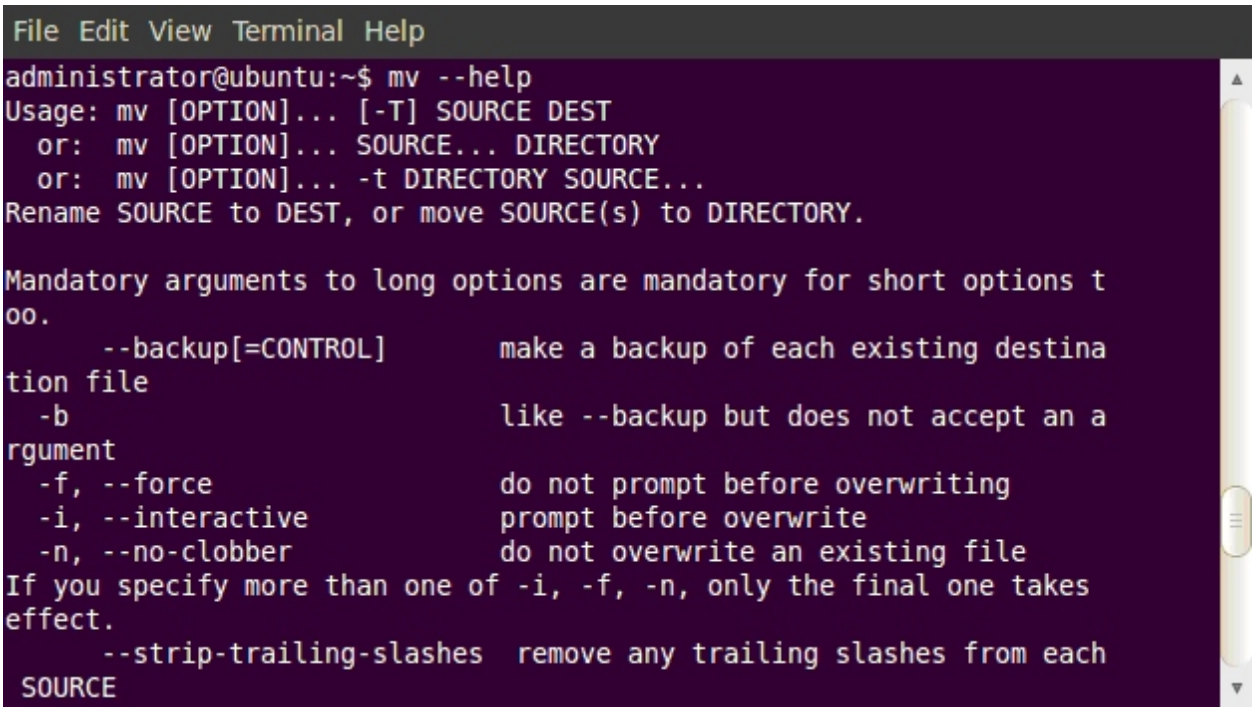


Figure 6.9 : Use of help command

Observe that the command used to display the help in figure 6.9 is *mv - - help*. The alternate mechanism to get help on the command is to use Linux online manuals. The man command activates a manual corresponding to a specific command that we need to look at. For example, the command *man mv* will show us the manual for the move command. Figure 6.10 shows the output of man command.

Observe that man command gives us exhaustive information of a command and may run into multiple screens. It generally displays one page at a time, as we press the Enter key the contents scroll down. To come out of the manual screen type alphabet ‘q’, this will take you back to the command prompt.

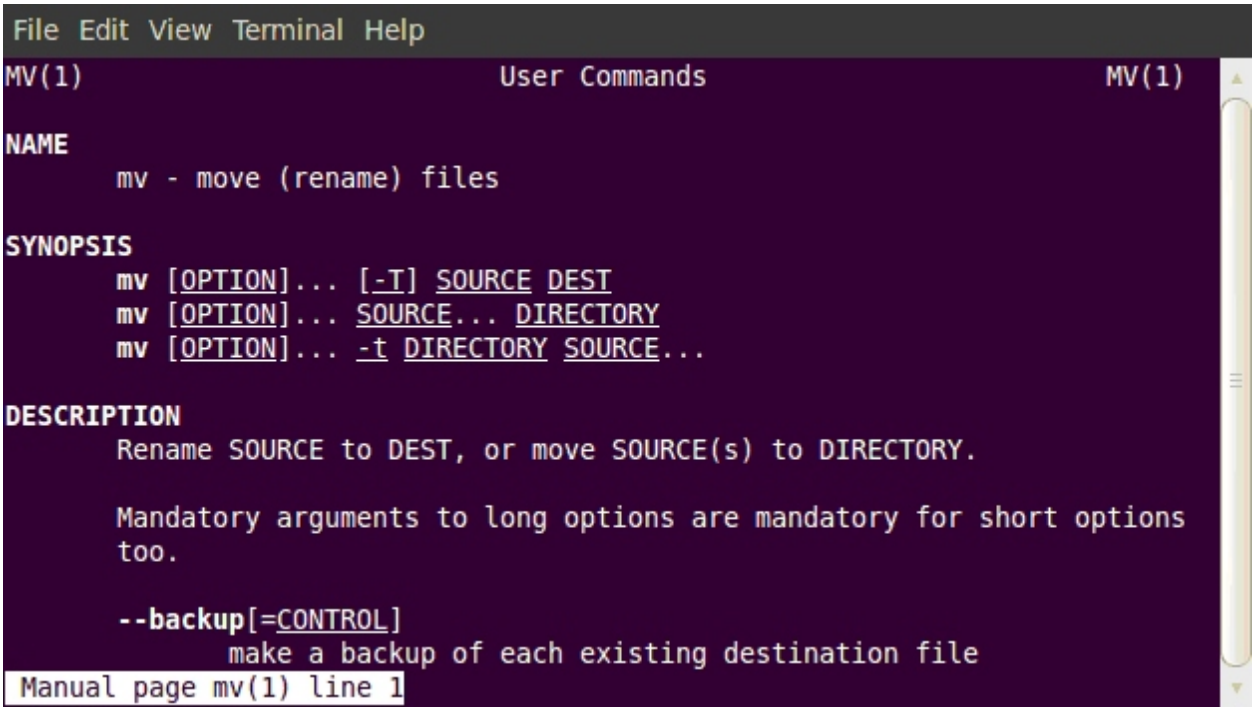


Figure 6.10 : Use of man command

In case we want only small description of a command then we may use the *whatis* command. It gives us one line explanation of the command, but omits any additional information about options. Figure 6.11 shows the sample output of whatis command.

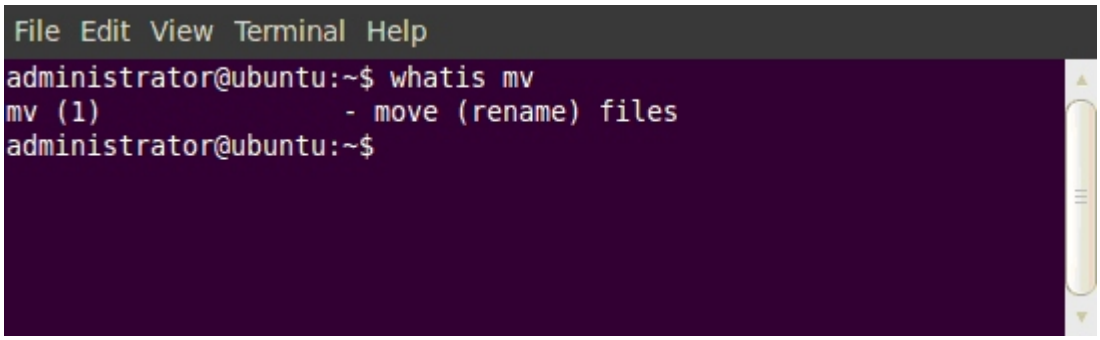


Figure 6.11 : Sample output of whatis command

Many times it may happen that we may not know which command to look for exactly. In such situations we may use the *apropos* command. The syntax of the command is mentioned below :

$apropos string

When we execute this command we will get a list of all the commands that has the string as the part of the command or command description. For example if we type *apropos copy* on the command prompt and try to execute it, then we may get screen full of commands that have copy as a string within the command or its description. A user must take caution while using this command. Sample output of apropos command is shown in figure 6.12.

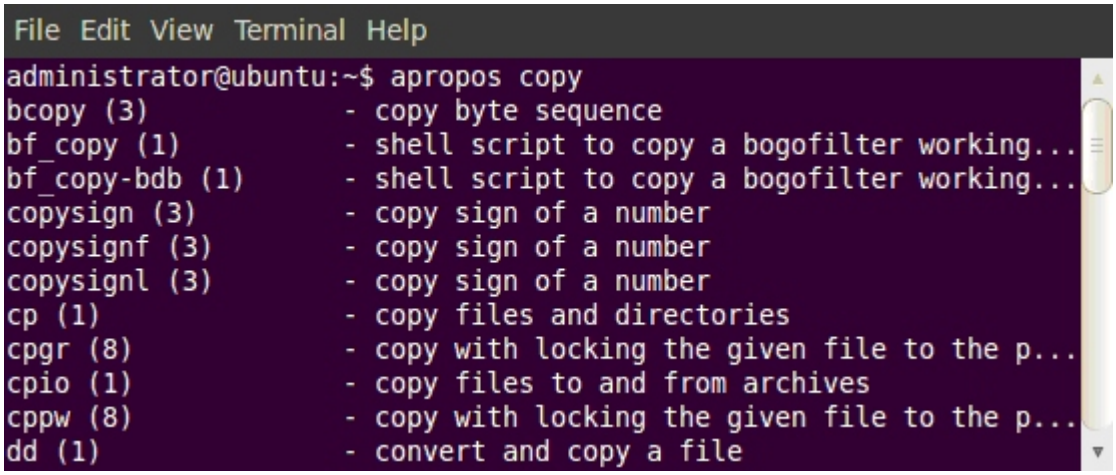


Figure 6.12 : Sample output of apropos command

Working with Directories

In Linux a directory is a special type of file that is used to store files and other directories. Here ' / ' symbol represents the root directory. All other directories come under root directory. Let us learn how to work with directories using terminal window.

Home directory

When a user logs on to the system, Linux automatically places the user in the directory called the home directory. It is created by the system at the time when a user account is created and generally will have a path /home/username. Here the username refers to the login name. For example if the username is harshal, then the home directory will be /home/harshal. It is possible to change this path if needed. The default working directory path is stored in system variable named HOME. We can cross check the directory by using the echo command.

$echo $HOME

/home/harshal

Note that the path displayed using this command is an absolute path name. Absolute path name is a sequence of directory names separated by / (slashes). An absolute path name shows a location in reference to the root directory. The first slash (/) is synonymous to root directory while the other slashes act as delimiters to other directory names. Thus the directory harshal is located within directory home which further is located in the root directory. Similarly the directory **/home/administrator** refers to home directory of username administrator.

It is possible to change the default path of the home directory. Suppose that you are able to see the output as shown below :

/home/its/ug1/svics

Here *svics* (home directory of user svics) is within the sub directory *ug1* (a directory that represents a sub group), which further is within a directory *its* (a directory that represents a group). Directory home and '/' have their default meaning.

Present Working Directory (*pwd*)

After we log into a system we can move around from one directory to another. But at any given point of time we will be located only in one directory. The directory where we are located at that moment is known as *current directory* or *present working directory*. To know the current directory that we are working in we can use the *pwd* command.

$pwd

Creating a Directory (*mkdir*)

A directory in Linux can be created using the *mkdir* command. The command takes the name of the directory to be created as its argument. Let us create a directory named *subject*.

$mkdir subject

The power of command line over GUI lies in its flexibility. If we create a directory using GUI we will be able to create one directory at a time, while it is possible to create multiple directories using a single mkdir command. The following command syntax illustrates the same.

$mkdir animals birds vehicles plants

The command when executed will create four directories named animals, birds, vehicles and plants in the current directory.

Change Directory (*cd*)

In the case when we need to store any data within a directory, first we need to make it our current directory. We can change (go within) a specific directory using the *cd* command. Let us try to create a directory named *math, science* and *economics* within a directory *subject.* To create these directories first we need to be in the *subject* directory. The command sequence shown below allows us to perform the said operation.

$cd subject

$pwd

/home/administrator/subject

$mkdir math science economics

Observe that the user name here is administrator. To again come back to the *administrator* directory, simply type the command below :

$cd ..

In the above command double dots (..) refer to the parent directory. Note that there should be one space between the cd command and the double dot.

Assume that you are in some internal directory that has path /home/administrator/subject/economics and you need to come back to the users home directory then again the cd command comes in handy. To come back to home directory we can issue the cd command twice as shown below :

$cd ..

$cd ..

The command sequence here will take us back one level at time. If we are say *M* levels down within the home directory, then we will have to execute the cd command *M* times. An alternative approach is to use a single cd command as shown below :

$cd ../../

Some example usage of the cd command along with its description is given in table 6.1.

Command Issued	Action Performed
cd ~/Desktop	Changes directory to /home/username/Desktop, from any current path. Here the symbol ~ refers to home directory of the user.
cd /	Changes directory to the root directory from any current path.
cd	Changes directory to the home directory from any current path.
cd -	Changes directory to the previously changed directory.
cd /var/www	Changes directory directly to the www sub-directory with directory var. It is useful when we know the path explicitly.

Table 6.1 : Sample cd commands

Remove Directory (*rmdir*)

An empty directory can be deleted by using the command *rmdir*.

$rmdir science

Here science is a directory name, and it will be removed using the above command only if it is empty. In case it is not empty we will get a message 'rmdir : failed to remove 'science' : Directory not empty'. In this case we have to first delete all the contents within the directory and then reissue this command. Note that it is also possible to delete multiple empty directories in the same way we created them.

To delete a non-empty directory with all its contents we can use the *rm* command as shown below :

$rm -r science

The rm command is discussed in detail in the later part of chapter.

Naming Conventions in Linux

We have created directories with different names using the mkdir command. While creating any object like a directory or a file in Linux we have to follow certain rules. On most of the Linux systems today, a name (of directory or file) can consist of up to 255 characters. Unlike Windows OS, names in Linux can practically consist of any ASCII character except for the slash (/) and the NULL character. Any other control characters or nonprintable characters are permitted. Examples of some valid names are .name, ^myname^-++, -{}(), test$#, xy.ab.ef etc.

However, it is recommended that a name should be relevant and contain only alphabetic characters, numerals, period (.), hyphen (-) and underscore (_). Linux strictly adheres to case sensitivity, thus math, Math and MATH are three different names. It is possible for the directories with these names to coexist at same level. If these names refer to a file then they can again coexist in the same directory.

Working with Files

Directory generally works as a container. The data is normally stored in files, which further may be stored in directory for proper arrangement and easy access. Text editors like nano, pico, vi or vim, ed and others are generally best suited for creating text file. However, many times the user needs to create a file quickly. The *cat* command comes in handy in such scenario. It is mainly used to display the contents of a small file on the terminal. But it can also be used to create a file, concatenate two files and append contents into a file.

Create a file using *cat* command

Let us create a file named *introduction* within the directory *economics*. To create this file you will have to first make the directory *economics* as your current directory. You can use the cd command here. Now type the command cat followed by a greater than (>) symbol and name of the file as shown below :

$cat > introduction

When we execute this command the cursor will be positioned in the next line, waiting for us to type the contents of the file. Type the text that you want to store in file and press CTRL + d. This will take you back to the command prompt. The combination CTRL + d in Linux indicate the end of file character. The greater than (>) symbol used in the above command is known as

redirection operator. It is used to instruct the shell that a redirection is required i.e., input should go to the specified file. The cat command when used without the greater than (>) symbol, displays the contents of the filename specified in argument. Figure 6.13 shows the process of creating and then displaying a file.

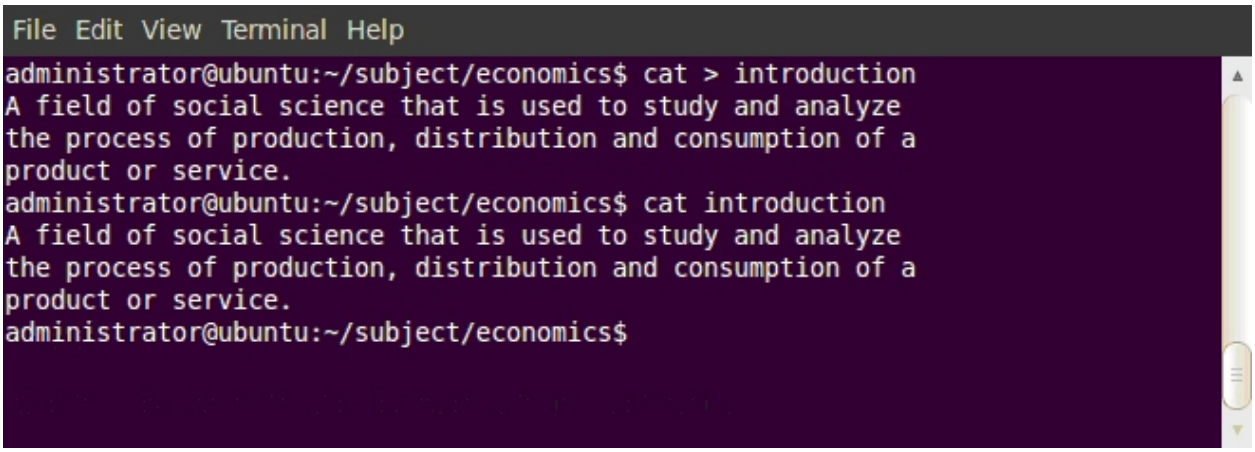


Figure 6.13 : Creating and displaying file using cat command

Appending contents using cat command

Assume that you have an existing file and you want to add some more content in the file. The cat command can be used again here with one simple change. The redirection operator used previously is to be replaced by append output (>>) redirection operator. The command to append data in the *introduction* file is shown below :

$cat >> introduction

An alternate definition states that Economics is a science which studies human behaviour as a relationship between ends and scarce means which have alternative uses.

[CTRL+d]

Note that if the file already exists and we use the command *cat > filename* then, the existing contents will be overwritten with the new one. So it is necessary to be careful while opening a file that already has some contents.

Concatenating multiple files using cat command

The cat command can also be used to concatenate the contents of multiple files and store it in another file. The syntax of using the concatenation is shown below :

$cat file1 file2 > file3

The above command will create *file3* that contains the text of both the files, namely file1 and file2. The new file created will have contents based on the sequence of filenames. Here the intial contents will be of file1, after which the contents of file2 will be appended.

Deleting a File (*rm*)

The *rm* command is used to delete/remove one or more files. For example to delete the file *introduction*, execute the following command :

$rm introduction

We can also delete multiple files using a single rm command. For instance the command

$rm file1 file2 file3

will delete all the three files supplied to it as argument. Table 6.2 gives some options that can be used along with the rm command.

Option	Usage
rm -i Filename	Deletes file using interactive mode. The user will be asked to verify the delete operation.
rm -r Directoryname	Deletes directory and along with all its contents.
rm -r *	Deletes all contents (file and/or directory) within the current directory. Here the symbol * is known as wildcard character. This is a very dangerous command as it will delete all the files and directories within the current directory, hence such command should be used only if you certain about the action and the result.
rm -rf *	Same as rm -r, but also deletes the contents even if it is write protected.

Table 6.2: Some options of rm command

Moving around the File system

So far we have learned how to create and delete a directory or file. Now let us see how to view the contents that are part of our file system. The *ls* command gives us the list of the contents in the current or a specified directory. The ls command can be used with different options to change the output. Let us begin with the plain and simple ls command without any options. Figure 6.14 shows the output of simple ls command. Note that the output on your computer may vary.

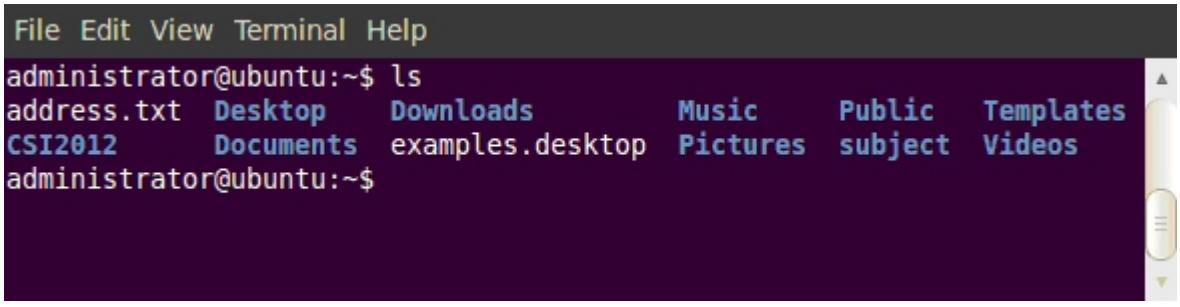


Figure 6.14 : Output of the ls command

Let us now create a file named *.introduction* and then check whether we are able to list it or not. Type the command shown below to create the file.

$cat > .introduction

Learning Ubuntu Linux is fun….

[CTRL + d]

Now use the ls command again to list the contents of a file. You must have observed that the file recently created is not visible on the screen. Note that in Linux any filename that is preceded by a '.' is treated as a hidden file. To list hidden files in the current directory we need to use -a option of the ls command. Figure 6.15 shows how to list hidden files.

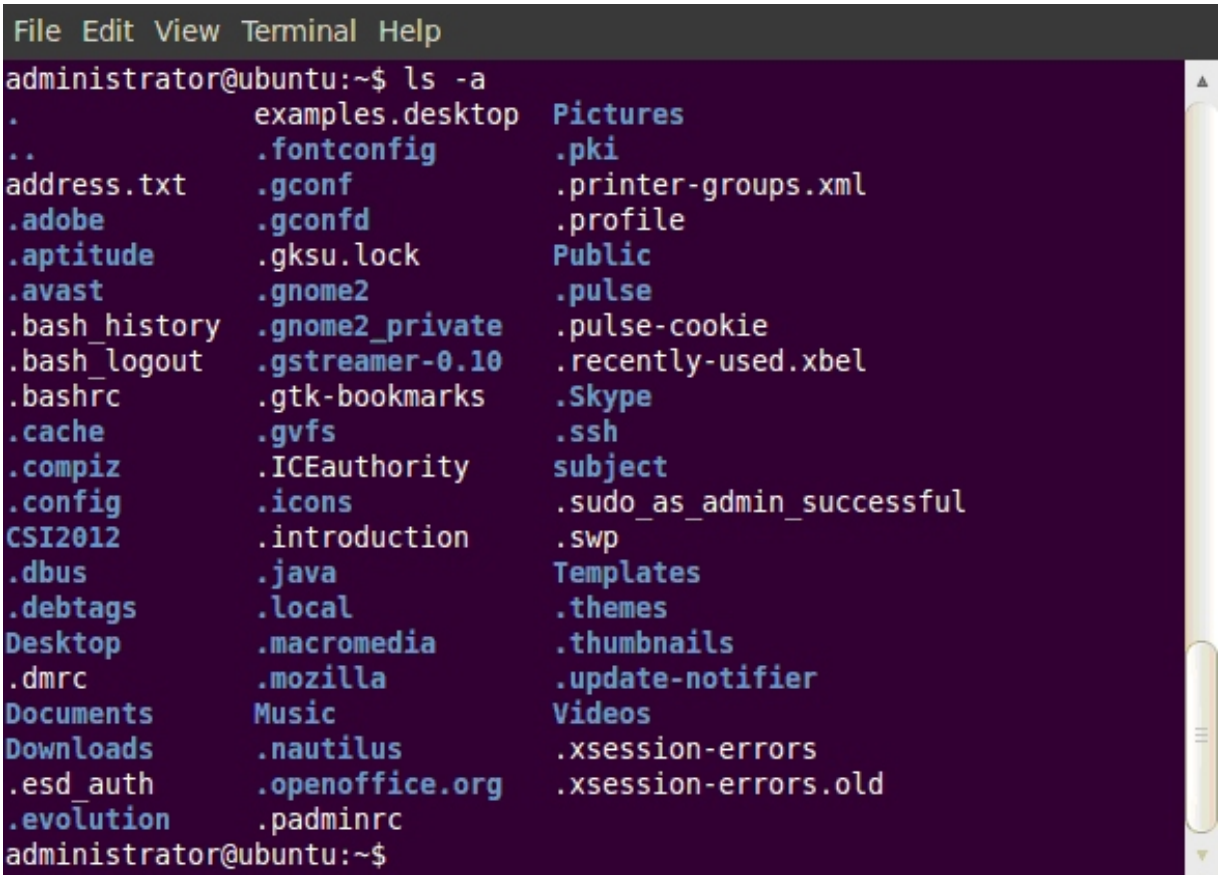


Figure 6.15: Listing hidden files

Observe the difference in the output of figure 6.14 and figure 6.15. Note that figure 6.15 shows more number of files than shown in figure 6.14. The two entries '.' and '..' visible in figure 6.15 are of importance, these two entries are automatically created in the directory whenever the directory is created. Table 6.3 gives some options that can be used along with the ls command.

Option	Usage
ls ~	Lists the files that are in user's home directory.
ls [svics]*	Lists all the files in which the first character of the filename matches with any of the given alphabets within the square brackets. The remaining part of filename can contain and valid ASCII character.
ls [n-s][5-7]??	Lists all files with 4 character filename. With the condition that the first character is in the range n to s, second character is in the range of 5 to 7, whereas the third and fourth characters are any valid ASCII character.
ls -r	List the files by sorting them in reverse order.
ls -t	List the files by sorting them based on their modification time.
ls -F	List the files and mark all executable files with * and directories with / symbol.
ls -1	List one file per line.

Table 6.3: Some options of ls command

Pattern Matching – The wildcards

In the above discussion you have already seen the usage of characters asterisk (*) and question mark (?). These characters are known as wildcard characters used for matching a pattern as required by the user. Table 6.4 summarizes the working of wildcards used by shell.

Wildcard	Pattern to be matched
*	Any number of characters including none
?	A single character
[abc]	A single character – either a, b or c (user can use other characters also).
[!abc]	A single character *other than* a, b or c (user can use other characters also).
[p-s]	A single character within the ASCII range of the characters p to s (user can use other characters also).
[!p-s]	A single character that is not within the ASCII range of the characters p to s (user can use other characters also).

Table 6.4: The wildcard characters

Manipulating Files and Directories

In the previous section we learned how to create and delete a file or a directory. Let us now see how to perform operations like copy, move, and assign permission on them.

Copying a file (*cp*)

Very often we need to create a replica of the data that we have generated, the *cp* command copies a file or group of files specified as an argument to it. It creates an exact replica of a file on the disk at the location specified by the user. The cp command needs at least two arguments. The first argument refers to a source file while the second argument refers to a destination file. Let us create a copy of file *introduction* using the following command :

$cp introduction new_introduction

After execution of the above command, an exact copy of file *introduction* will be created with the name *new_introduction*. If a file with the name *new_introduction* already exists, it will simply be overwritten without any warning from the system. In case no such file exists a new file will be first created and then the contents of the file *introduction* will be copied in it.

The cp command can also be used to copy more than one file into a specified directory. For instance, the following command :

$cp file1 file2 my_dir

will copy two files named *file1* and *file2* in a directory named *my_dir*. It is necessary that the directory *my_dir* already exists, or else we will get an error message. Table 6.5 gives some example usage of the cp command.

Command	Description
cp /vol/examples/tutorial/science.txt .	Copies the file *science.txt* to current directory. The dot (.) at the end refers to the current directory.
cp chap01 progs/unit1	A file named chap01 is copied within the directory progs with name unit1. (no directory with name unti1 should exist in progs directory)
cp chap01 progs	A file named chap01 is copied within the directory progs with same name. (because progs is a directory).
cp -r progs newprogs	The directory named progs along with all its contents is copied and stored as a directory newprogs.

Table 6.5 : Sample cp commands

Renaming files and/or moving files (*mv*)

Changing the name of a file or directory is another operation that user performs regularly. The *mv* command is used for renaming a file or directory. For example, to rename the file *introduction* to *introduction.txt*, execute the following command :

$mv introduction introduction.txt

The command will rename the file and store it at the same location as that of the file *introduction*. Thus no additional space is consumed on disk during renaming.

The *mv* command can also be used to move a file or group of files to a different directory. For example, the command

$mv file1 file2 my_dir

will move the files named *file1*and *file2* to the directory named *my_dir*.

This command is also used to rename a directory. For instance, the execution of the following command :

$mv math mathematics

will rename an existing directory *math* to *mathematics*.

Paging output (*more*)

The *more* command is used to view one page of content on the screen at a time. For instance if the file introduction.txt contains text that can not fit in a single screen, then reading its contents would become difficult. The *more* command when used displays the contents of file one page at a time. To view the next page we may press any key. Generally we may press character 'b' to view previous page and character 'f' to view next page. The sample usage of the command is shown below :

$more introduction.txt

Compare two files (*cmp*)

The *cmp* command compares two files of any type and writes the results to the standard output. If the two files compared differ in contents then the byte and line number at which the first dissimilarity occurred is reported. In case there is no difference between the contents of the files we simply see the command prompt again. The sample usage of the command is shown below :

$cmp introduction introduction.txt

Difference (*Diff*)

An extension of the *cmp* command is the *diff* command. The diff command compares two files and displays the contents of both files indicating where the difference lies. To understand the working of the *diff* command we have created a copy of the file introduction.txt and named it new_intro.txt. We have also removed some lines from the new file. Figure 6.16 shows the output of the command shown below :

$diff introduction.txt new_intro.txt

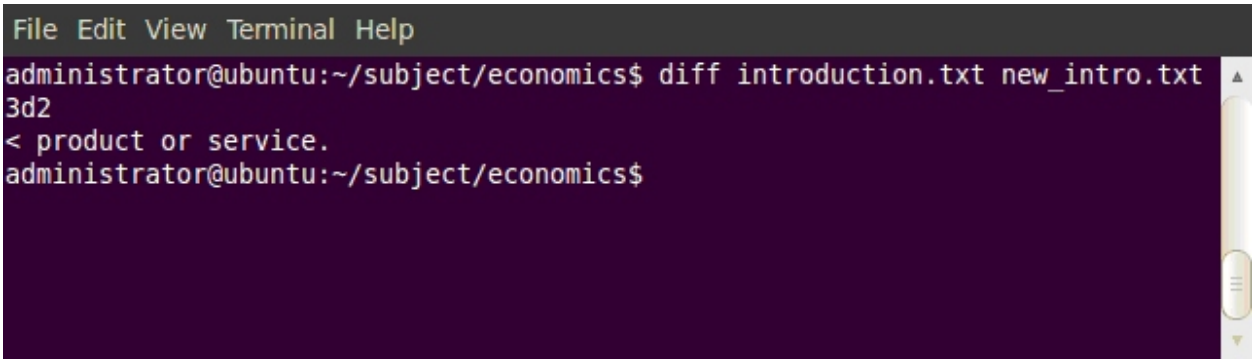


Figure 6.16 : Output of diff command

In figure 6.16 the lines beginning with a < indicates that file introduction.txt contains the text shown but file new_intro.txt does not contain the line. Any changes in the file new_intro.txt would be shown with the lines beginning with a > sign.

Counting lines, words and characters in a file (*wc*)

The *wc* command is used to count the number of lines, words, and characters in the specified file or files. The *wc* command can be used along with three options -l, -w and –c for counting lines, words and characters respectively. For instance, execute the following command to count the number of lines in the file introduction.txt.

$wc -l introduction.txt

4 introduction.txt

Similarly the commands **wc –w introduction.txt** and **wc –c introduction.txt** will give us the count of number of words and characters in the file. To get all the information together we can use the command as shown below :

$wc -l -w -c introduction.txt

4 49 307 introduction.txt

File permissions

In the earlier section, we have seen options which can be used with ls command. The ls command has several other options also. For example the following command when executed may result in the output similar to the one shown here.

$ls -l

total 6

-rw-r--r-- 1 administrator administrator 313 2013-02-15 18 :04 about_Gandhiji.txt

-rw-r--r-- 1 administrator administrator 444 2013-02-15 18 :19 introduction.txt

-rw-r--r-- 1 administrator administrator 401 2013-02-20 16 :43 address.txt

drwxr-xr-x 1 administrator administrator 4096 2013-02-21 18 :15 backup

-rw-r--r-- 1 administrator administrator 144 2013-02-13 18 :49 city.txt

-rw-r--r-- 1 administrator administrator 226 2013-02-20 14 :11 script10.sh

Observe the output; it gives us a clear idea about an object in our file system. An object can be categorized as a regular file, a directory or a process. It shows us the owner of the object, size of the object, date and time on which the object was created along with the name of the object. Let us try to understand the file permission in detail. Figure 6.17 shows the relation of different permissions w.r.t. owner, group and other users.

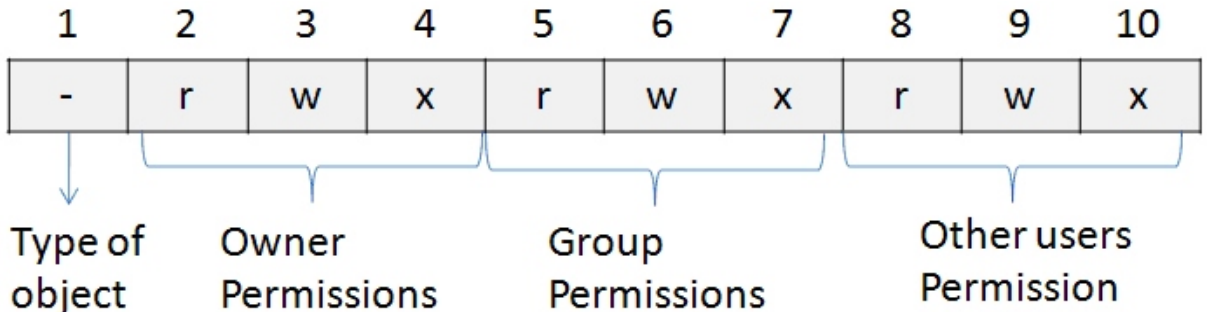


Figure 6.17 : File permissions

As can be observed in figure 6.17 the first column refers to type of object. The character '–' in the first column refers to a file, character 'd' refers to a directory and character 'p' refers to a process. The next nine characters tell the system what access is permitted for this object; hence the name "permissions". An object in Linux has three permissions namely, read (r), write (w) and execute (x).

The set of three characters after the file type shown in column 2 to 4 tells which permission the owner of the file has. Owner of the file is the user who created the file (administrator in our case). The character **r** in the first position means you are permitted to read the file. A **w** in the second position means you may write in the file. This includes the ability to delete a file. An **x** in the third position means you may execute the file. A hyphen '–' in any position means that you don't have that particular permission. As you can in the output of the ls -l command, administrator the user who owns the file can read and write files like about_Gandhiji.txt, introduction.txt and address.txt. The characters in column 5 to 7 denotes permission given to the group, to which user belongs. Similarly the characters in column 8 to 10 denote permission given to any other user or group. Generally the term others refer to the users of the system that do not belong to group to which the owner of the file belongs. A user if wishes can change the permission of an object that he/she owns.

The file permissions can also be given as numeric representation. We use octal (base of 8) number system for representing permissions as numeric values. Every octal digit combines read, write and execute permissions together. For example, in permission 644, "6" refers to the rights of the owner, "4" refers to rights of the group and "4" refers to rights of others. The permission 0644 when assigned is interpreted as read and write permission to owner, only read permission to group and others. Table 6.6 shows the interpretation of the octal numbers when used as permission.

Permission in text mode	Permission in octal mode	Meaning
---	0	No permission assigned
--x	1	Only execute access is allowed
-w-	2	Only write access is allowed
-wx	3	Write and execute access are allowed
r--	4	Only read access is allowed
r-x	5	Read and execute access are allowed
rw-	6	Read and write access are allowed
rwx	7	Everything is allowed

Table 6.6 : Octal numbers and permission

Changing Permissions (*chmod*)

To change the permission we use the *chmod* command. The operation of changing the permission is also known as change mode operation. For instance, in the above example we have seen that user (owner) has read and write permissions on the file. To make the file read only file the following command can be used :

$chmod ugo-w introduction.txt

The character 'u' in the above chmod command stands for user, 'g' for group and 'o' for other. After executing the command if we again list the file, then the output will be similar to the one shown below.

$ls -l introduction.txt

-r--r--r-- 1 administrator administrator 307 2013-02-11 14 :19 introduction.txt

Please note that write operation will not be permitted on this file. Additionally it also prevents user from deleting the file intentionally or unintentionally.

To assign write and execute permission to the owner of a file, execute the following command.

$chmod u+wx script10.sh

$ls -l script10.sh

-rwxr--r-- 1 administrator administrator 226 2013-02-20 16 :05 script10.sh

Here the file script10.sh is known as a script file. We are going to learn about shell scripting in the next chapters.

Table 6.7 shows some abbreviations and its meaning when used with the chmod command.

Category	Operation	Permission
u-user	+ assign permission	r- read permission
g-group	– remove permission	w - write permission
o-other	= assign absolute permission	x - execute permission
a-all		

Table 6.7 : Abbreviation used by chmod

I/O Redirection

A user interacts with the Operating System using a standard input device (keyboard). The Operating system displays the output on standard output device (monitor). Thus if any command is executed its input will be taken from the keyboard and output will be displayed on the monitor.

Sometimes it is useful to redirect the input or output to a file or a printer. Linux provides redirection symbols to change the standard input flow. The greater than symbol ' > ' implies redirection of output. It instructs the OS to put the output in the destination (file) specified by the user instead of displaying on the monitor screen. Similarly the less than symbol ' < ' implies redirection of input, it instructs the OS to accept the input from the specified source (file) instead of keyboard.

Assume that we issue a command **wc –l < introduction.txt**, here we are instructing OS to accept the input from a file named introduction.txt instead of the keyboard. Similarly the command **ls > list.txt** when executed will transfer the output of ls command to a file named list.txt instead of the monitor. When output redirection is used the output will not be displayed on the monitor hence to see the output we will have to use the command **cat list.txt**.

Piping

Redirection facility discussed above helps in associating the Linux commands to files. Many times we need to use multiple commands to perform a single operation. The piping facility of Linux helps in such cases. The pipe symbol (|) is used to provide the output of one command as an input to another command. The process of converting output of one command into input of another command is known as piping. Let us see an example of piping.

$ls | wc -l

When we execute the above command, the output of *ls* command becomes the input to the *wc* command. Thus we will get information of total number of files in a current directory. Real power of pipe facility can be availed when we use it along with filters. Filters have been discussed in the next section.

Filters

Filters are commands that accept data from the standard input, process or manipulate it and then write the results to the standard output. Various filters like head, tail, cut, paste, sort and uniq are available in Ubuntu Linux. Let us see the working of these filters.

Displaying lines from top of the file (*head*)

The head command is used to display the required number of lines at the beginning of the file based on user's requirement. When used without any option it displays first 10 lines of the file. To display the lines as per users requirement we need to pass an argument to the head command. For example to display first 2 lines of the file *introduction.txt*, execute the following command :

$head -2 introduction.txt

Displaying lines from bottom of the file (*tail*)

The tail command works exactly opposite of the head command. It displays specified number of lines from the end of the file. To display last 2 lines of the file *introduction.txt*, execute the following command :

$tail -2 introduction.txt

We can use the tail command to display lines from n^{th} line within the files. For example if we execute the following command :

$tail +5 introduction.txt will display lines from 5^{th} line onwards from the file introduction.txt.

Slicing a file vertically (*cut*)

The head and tail command discussed in the above sections are used to slice the file horizontally. We can slice the data within the file vertically using the *cut* command. The cut command gives exact and precise outputs if the file has specific delimiters. Let us create one such delimited file to understand the working of the cut command. Create a file named address.txt using cat command that stores the data as shown :

$cat address.txt

20013, Vaidehi, Sanjay, Shah, Sector-23, GH-6, Gandhinagar, 382023

20014, Dhrumil, Ajay, Patel, Yesh Enclave, Mota Bazar, Vidyanagar, 388120

20015, Harshit, Amit, Jain, 58, Jaldeep I, Ahmedabad, 380058

20016, Abdul, Shamsher, Khan, Khan Villa, M G Road, Nadiad, 388011

20017, Nirav, Jose, Mackwan, Jose House, M G Road, Nadiad, 388011

20018, Vidita, Harshal, Arolkar, 17, Jaldeep I, Ahmedabad, 380058

Let us see how to use the cut command along with its various options.

Cutting Characters (-c)

To extract specific characters from each line of the file, cut command with -c option is used. For instance, to extract the roll numbers and first names from the file address.txt execute the following command :

$cut -c 1-15 address.txt
20013, Vaidehi,
20014, Dhrumil,
20015, Harshit,
20016, Abdul, S
20017, Nirav, J
20018, Vidita,

Though the output looks fine to certain extent, it is not exactly the same as we would have expected. Look at the data of Abdul and Nirav it has additional characters. The -c option is useful for fixed length fields, we had problem in the output as the first names were not stored using a fixed length.

Cutting fields (-f)

To overcome the problem mentioned in the -c option we may use a delimiter. The cut command can treat values separated by the delimiter as separate field values. Observe that we have used comma ',' as a delimiter in the address.txt file. Thus, to extract roll number and first name only we may execute the following command :

$cut -d "," -f 1,2 address.txt
20013, Vaidehi
20014, Dhrumil
20015, Harshit
20016, Abdul
20017, Nirav
20018, Vidita

Observe that we have got the desired output now. In this command the –d is option used to specify delimiter appearing in a file (comma in our case) and –f option is used to specify field numbers to be displayed (roll number (1) and first name (2) in our case).

It is also possible to slice the file vertically from fields in between. For example assume that we want to display the first name, city and pin-code then we need to cut field numbered 2 and 7 onwards. The said operation can be performed by executing the following command :

$cut -d "," -f 2,7- address.txt
Vaidehi, Gandhinagar, 382023
Dhrumil, Vidyanagar, 388120
Harshit, Ahmedabad, 380058
Abdul, Nadiad, 388011
Nirav, Nadiad, 388011
Vidita, Ahmedabad, 380058

Here 7- in the command imply that we need to display all fields after field number seven (including seven) from the file address.txt. It is also possible to redirect the output to a file. For example if we execute the following command :

$cut -d "," -f 2,7- address.txt > output_cut.txt

the output instead of being displayed on monitor will be transferred to the file output_cut.txt.

Joining Contents (*paste*)

Two files can be pasted together using the *paste* command. For the paste command to work properly we need to ensure that both the files have exactly the same number of lines. If the number of lines is not same then the command may not result into expected output as it pastes from top of the files.

We will first create two different files named part1.txt and part2.txt using cut command and then join them using the paste command. Figure 6.18 shows the process of performing this operation.

```
File  Edit  View  Terminal  Help
administrator@ubuntu:~$ cut -d "," -f 1-4 address.txt > part1.txt
administrator@ubuntu:~$ cut -d "," -f 5-8 address.txt > part2.txt
administrator@ubuntu:~$ cat part1.txt
20013, Vaidehi, Sanjay, Shah
20014, Dhrumil, Ajay, Patel
20015, Harshit, Amit, Jain
20016, Abdul, Shamsher, Khan
20017, Nirav, Jose, Mackwan
20018, Vidita, Harshal, Arolkar

administrator@ubuntu:~$ cat part2.txt
 Sector-23, GH-6, Gandhinagar, 382023
 Yesh Enclave, Mota Bazar, Vidyanagar, 388120
 58, Jaldeep I, Ahmedabad, 380058
 Khan Villa, M G Road, Nadiad, 388011
Jose House, M G Road, Nadiad, 388011
 17, Jaldeep I, Ahmedabad, 380058

administrator@ubuntu:~$ paste part1.txt part2.txt
20013, Vaidehi, Sanjay, Shah     Sector-23, GH-6, Gandhinagar, 382023
20014, Dhrumil, Ajay, Patel      Yesh Enclave, Mota Bazar, Vidyanagar, 388120
20015, Harshit, Amit, Jain       58, Jaldeep I, Ahmedabad, 380058
20016, Abdul, Shamsher, Khan     Khan Villa, M G Road, Nadiad, 388011
20017, Nirav, Jose, Mackwan     Jose House, M G Road, Nadiad, 388011
20018, Vidita, Harshal, Arolkar  17, Jaldeep I, Ahmedabad, 380058

administrator@ubuntu:~$
```

Figure 6.18 : Example of paste command

Ordering Output (*sort*)

The *sort* command is used to order the data stored within a file in ascending or descending sequence at the time of display. Like the cut command, it also identifies fields and can sort on specified fields. When the sort command is used without any options, it sorts the file based on entire line. It reorders the lines based on ASCII sequence. The sorting is first applied on white spaces, followed by numerals, uppercase letters and finally lowercase letters.

Let us try to arrange the file address.txt in descending order of roll numbers (you must have observed that it is already arranged in ascending order). To display contents of file sorted in reverse order, execute the following command :

$sort -r address.txt

20018, Vidita, Harshal, Arolkar, 17, Jaldeep I, Ahmedabad, 380058

20017, Nirav, Jose, Mackwan,Jose House, M G Road, Nadiad, 388011

20016, Abdul, Shamsher, Khan, Khan Villa, M G Road, Nadiad, 388011

20015, Harshit, Amit, Jain, 58, Jaldeep I, Ahmedabad, 380058

20014, Dhrumil, Ajay, Patel, Yesh Enclave, Mota Bazar, Vidyanagar, 388120

20013, Vaidehi, Sanjay, Shah, Sector-23, GH-6, Gandhinagar, 382023

Note :

The execution of sort command does not modify the actual file. The records in the actual file will remain in the same position. The sort order is applied only at the time of displaying the output.

The sort command is mostly used in conjunction with other commands. For example, we can use it with the cut command to order the output of the cut command. Execute the following command to see the output of both the cut and sort commands when combined.

$cut -d "," -f 2-4 address.txt | sort

Abdul, Shamsher, Khan

Dhrumil, Ajay, Patel

Harshit, Amit, Jain

Nirav, Jose, Mackwan

Vaidehi, Sanjay, Shah

Vidita, Harshal, Arolkar

The cut command in the above example extracts second, third and fourth column from the file address.txt. The extracted output is then given as input to the sort command. The sort command then sorts the contents and displays it on the screen.

Character Conversion (*tr*)

The command used as filters work with a line or column. The *tr* (translate) command allows us to work with individual characters within a line. It is used to translate (convert) strings or patterns from one set of characters to another.

Working with the address.txt file you must have observed that it contains "," as delimiter. Assume that we do not want to show the delimiter at the time of the display, instead we would like to show a blank space. The tr command allows us to perform this operation. Execute the command shown below :

$cat address.txt | tr -s '[,]' '[ ]'

20013 Vaidehi Sanjay Shah Sector-23 GH-6 Gandhinagar 382023

20014 Dhrumil Ajay Patel Yesh Enclave Mota Bazar Vidyanagar 388120

20015 Harshit Amit Jain 58 Jaldeep I Ahmedabad 380058

20016 Abdul Shamsher Khan Khan Villa M G Road Nadiad 388011

20017 Nirav Jose Mackwan Jose House M G Road Nadiad 388011

20018 Vidita Harshal Arolkar 17 Jaldeep I Ahmedabad 380058

The translation is only applicable at the display it will not permanently replace the actual delimiter. The -s option squeezes the additional space visible in the actual file. In case we need to save the translations visible we can redirect the output to a new file.

Pattern matching (*grep*)

The find operation is one of the most widely used operation in GUI applications. We must have used the CTRL + f keys to find a keyword within files. The *grep* command performs similar operation from the command line interface. The command is based on a fundamental idea of search globally for a regular expression and display lines where instances are found (g/re/p).

Let us make use of the grep command to find a name within the file address.txt and display its record. Figure 6.19 shows the working of the grep command.

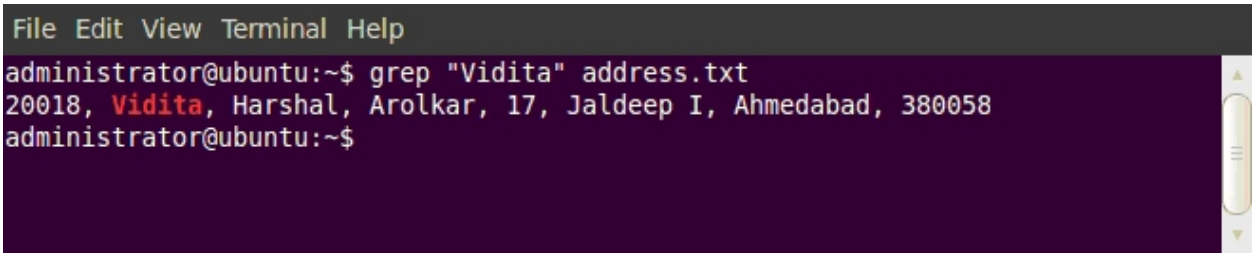


Figure 6.19 : Working of grep command

Observe that the string that we are looking for is shown in red colour, also we have enclosed the keyword in double quote. It is not compulsory to enclose the keyword in double quotes hence the command **grep Vidita address.txt** will also give same output. The string used in the grep command is case sensitive hence the strings "Vidita" and "vidita" are different. We can use different options along with grep command that will help us refine our search in a better way. Table 6.8 lists the options and their usage.

Option	Usage
-c	Return only the number of matches, without quoting the text
-i	Ignore case while searching
-l	Return only file names containing a match, without quoting the text.
-n	Return the line number of matched text, as well as the text itself.
-v	Returns all the lines that do not match the text.
-w	Return lines which display only whole words
-o	Shows only the matched string

Table 6.8: Options of grep command

One very powerful feature of grep command is to use a regular expression as a keyword. Say for example we want details about the persons who are staying in society that has 'Jal' as it's starting and 'I' as its end, then regular expression can be used. Let us execute the command shown here :

$grep "Jal.*I" address.txt

The output of this command is shown in figure 6.20. Here "Jal.*I" is a regular expression.

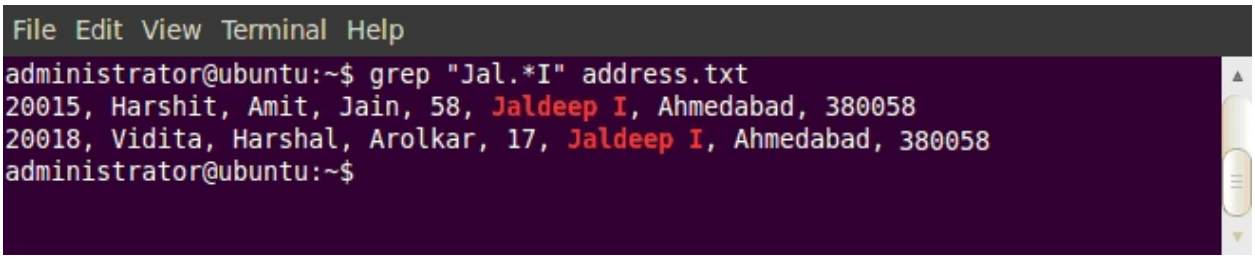


Figure 6.20 : Using regular expressions in grep command

A regular expression is normally followed by one of several repetition operators shown in table 6.9.

Repetition Operator	Meaning
?	The preceding item is optional and matched at most once.
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.
{n}	The preceding item is matched exactly n number of times.
{n,}	The preceding item is matched n or more number of times.
{,m}	The preceding item is matched at most m number of times.
{n,m}	The preceding item is matched at least n number of times, but not more than m number of times.

Table 6.9 : Repetition operator

Searching a file or Directory (*find*)

Many times we forget the location of the file or a directory that we have created. The *find* command helps us look for such forgotten objects. The find command looks for the search criteria (file or directory or both) that you have specified starting from the directory you specify within all its sub directories. We can also search for the object based on its name, owner, group, type, permissions, date, and other criteria. Note that the find command when used without any other arguments displays the pathname of all the files and directories in the present directory and all its subdirectories.

Assume that we want to look for the location of file introduction.txt that we had created earlier, the command would be

$find -name introduction.txt

If the file exists then its path will be given as an output. Otherwise we may either get an error or a prompt will be visible if no such file exists. Note that we know the name of the file here, what if we only remembered first few characters of the file name. The wildcard characters can be used in such cases. The example shown below helps in finding all files that start with string "intro".

$find -name intro*
./subject/economics/my1_dir/intro1
./subject/economics/my1_dir/intro
./subject/economics/introduction.txt
./subject/economics/introduction
./subject/economics/my_dir/intro1
./subject/economics/my_dir/intro
./subject/economics/intro1

The output of this command may vary on your screens depending on the number of files that you have which start with string "intro". Also in Linux we may not assign a file extension, hence we may not be able to know whether intro1, intro and introduction in the above outputs are files or directories. We may refine the search if needed by using the *type* option as shown below :

$find -name intro* -type f

Table 6.10 shows some example of find and its expected output description.

Command	Description
find / -type d	Search all directory and sub directory available on root only.
find . -mtime -1	Search objects modified within the past 24 hours.
find . -mtime +1	Search objects modified more than 48 hours ago.
find ./dir1 ./dir2 -name script.sh	Search directories "./dir1" and "./dir2" for a file "script.sh".
find -size 0 -delete	Search for files of zero byte and delete them from the disk.
find -executable	Search for the executable file in current directory.
find /home -user jagat	Search for the object whose owner is jagat within the home directory and its sub directory.
find . -perm 664	Search for object that has read and write authorization for their owner, and group but which other users can only read.

Table 6.10: Example of find command

Running Commands as the Superuser

When you log in to your computer, the account you use is a regular user account. This account has a limited right. The security model of Ubuntu generally allows you to work as a normal user. Not providing administrative rights prevents any accidental changes or installation of malicious programs that may disturb functioning of the system. But many times the user may need to have administrative privileges. The administrative privileges are available to only a user known as superuser. To use the superuser account when using the terminal, we need to add *sudo* as a prefix to the commands that we want to execute. For example, execute the following command to install a new program called skype from command line.

$sudo apt-get install skype

When you execute the command it will ask for password, provide the password of superuser (generally it will be different from the normal user). This password is the password of the first user that you

added when you installed Ubuntu Linux on the computer. Once the user is authenticated as sudo by means of the terminal, the software will start installing. Once the installation is over we can start using the software.

The super user can also perform the operations of adding, deleting or updating a user, group or object in the system. Some of the commands listed below are used for such purposes.

adduser : The *adduser* command creates a new user on the system. Figure 6.20 shows the process of creating a user. The command when executed will ask for password and some additional details as shown in figure 6.21. Once all the details are provided a new user along with its home directory will be created in the system.

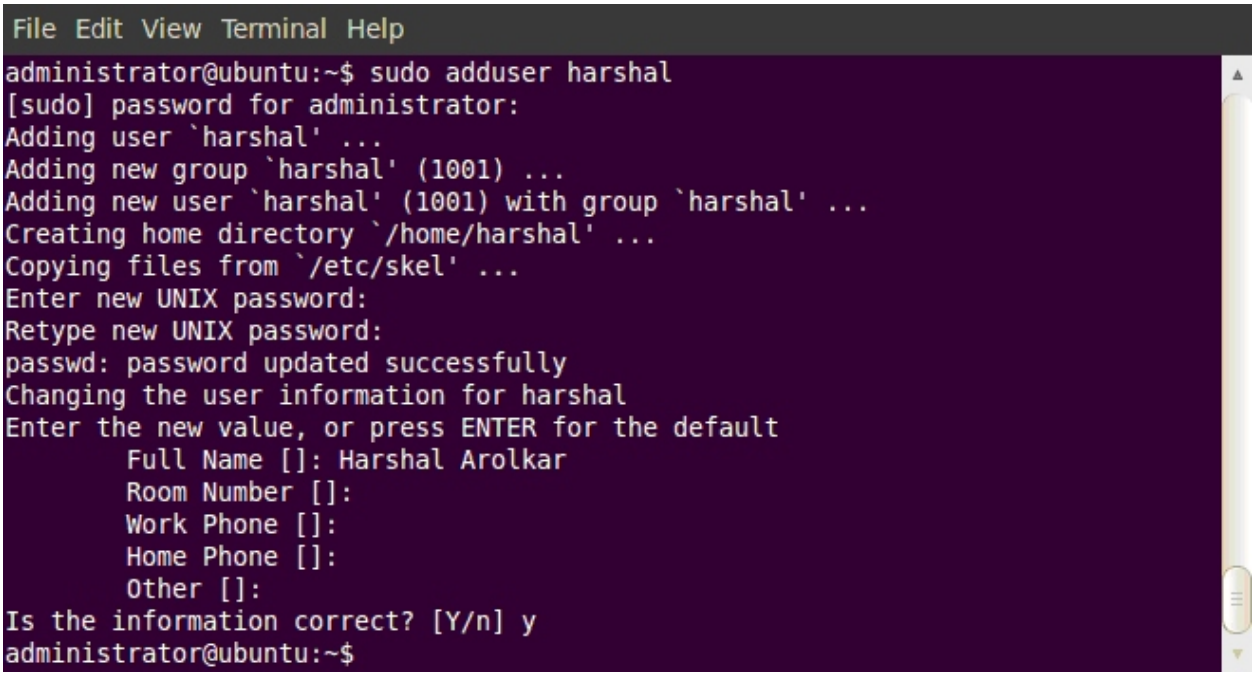


Figure 6.21 : Adding a user

passwd : The *passwd* command when executed as a super user do, allows us to change the password of any valid user of the system.

who : The *who* command when executed displays the list of all the users that are presently logged into the machine.

addgroup : The *addgroup* command adds a new group. The users are normally divided into groups so that they can be better controlled.

deluser : The *deluser* command is used to delete a user from the system. Note that we need to explicitly remove the user's files and home directory, by using the -remove -home option.

delgroup : The *delgroup* command deletes a group from the system. To perform this operation we must first make sure that no user is associated with the group that we are going to delete.

Summary

In this chapter we learned how to use the Ubuntu Linux command line interface. The CLI (Command Line Interface) when used allows us to perform all operations that we perform using the GUI in efficient and fast manner. We saw how to initiate the CLI using the Linux terminal. We also learned how to create, rename and delete a file or a directory, find out the directory that we are working in, change the directory if required. Later we saw how to create a copy of the file as well as directory. An access right is one of the ways to make sure that our data is not misused; we saw how to assign or change access rights. We saw how to increase the effectiveness of commands by joining multiple commands using the pipes. Further we looked at some features like counting the words or lines within the file, slicing the file horizontally and vertically, joining the file, searching for a file, directory or a string pattern within the files, arranging the display in ascending or descending order. Finally we saw how normal user can perform administrative tasks of installing new software, adding or deleting a user or group, checking who is using the system or change the password of some user.

EXERCISE

1. What is command prompt?

2. Describe shell. Name any three Linux shells.

3. How shells interpret command? Explain with suitable figure.

4. Write the steps used to start a terminal in Ubuntu Linux.

5. Explain following command in detail :

ls, cat, wc, chmod

6. What is the meaning of internal commands in Ubuntu Linux?

7. How can you find help for any command in Ubuntu Linux?

8. Explain different wildcard characters, giving suitable example.

9. Explain the use of pipes, giving suitable example.

10. What is redirection? Explain giving suitable example.

11. List the filter commands used in Linux.

12. **Choose the most appropriate option from those given below :**

(1) Which of the following command is used to count the total number of lines, words, and characters contained in a file?

(a) countw (b) wcount

(c) wc (d) wordcount

(2) Which of the following command is used to remove files?

(a) dm (b) rm

(c) delete (d) erase

(3) Which of the following command is used to remove the directory?

(a) rdir (b) remove

(c) rd (d) rmdir

(4) Which of the following command is used to count just the number of lines contained in a file?

(a) wc - r (b) wc - w

(c) wc - c (d) wc - l

(5) The command chmod 761 letter is equivalent to which of the following access rights?

(a) chmod u=7, g = 6, o = 1

(b) chmod a = 761

(c) schmod u = rws, g = rw, o = x

(d) chmod 167

(6) Which of the following refers to the maximum length of a filename in Linux?

(a) 8 (b) 10

(c) 200 (d) 255

(7) The hierarchy of a series of directories branching in a user system starts from which of the following directories?

(a) \home (b) \root

(c) /home (d) /root

(8) Which of the following command is used to copy a file?

(a) tar (b) cpio

(c) cp (d) copy

(9) Which of the following command is used to display your current working directory?

(a) path (b) pwd

(c) prompt pg (d) dir

(10) Which of the following command is used for searching a pattern in a file?

(a) grep (b) find

(c) lookup (d) All of the above

(11) Which of the following is not a redirection symbol?

(a) > (b) <

(c) * (d) >>

(12) Which of the following syntax is correct to assign a read permission on a user file?

(a) chmod r filename
(b) chmod u+r filename
(c) chmod filename r
(d) chmod filename u+r

(13) Which of the following refers to the minimum arguments of cp command?

(a) One
(b) Two
(c) Three
(d) None

(14) The mv command in Linux is used for which of the following purpose?

(a) To rename a directory
(b) To move a file
(c) To copy a file
(d) All of the above

(15) Which of the following command is used to view one page content on the screen at a time?

(a) More
(b) more
(c) PAGE
(d) page

Laboratory Exercises

1. Perform the following using Linux commands :

(a) Print the calendar of December 2012.

(b) Execute the command which displays login name, the name of your terminal and date and time since user logged in.

(c) List all files starting with character 'n' or 'N'.

(d) Display the current working directory.

(e) Prepare two files named class11_A.txt and class11_B.txt containing details of students of eleventh standard. The file should contain names of the students. Now merge these two files in a single file and name it class11_.txt. (Use cat command).

(f) Hide the three files created in question 'e'.

(g) List only directories.

(h) Get help on the use of the cat command.

(i) List all files whose fourth character is 'g' and sixth character is digit.

(j) Using terminal calculator perform the following operation :

(1) Calculate 2500/7

(2) Convert decimal 50 to its binary equivalent

(3) Convert decimal 25 to its hexadecimal equivalent

(4) Find square root of 36

(5) Convert hexadecimal 25 to its decimal equivalent

2. Show the output of following Linux commands :

(a) cat f1 >> f2

(b) echo $SHELL

(c) mkdir d1 d2 d3

(d) ls s*t

(e) ls [a-f]*

(f) ls -R

(g) ls -xR

(h) cp f1 f2

(i) ls | sort

(j) ls | tr -s " " | cut -d " " -f 5 | sort

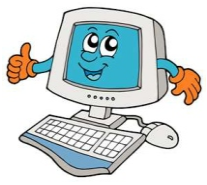
(k) ls -l | grep -c "address.txt"

(l) grep "Harshit Jain" address.txt

(m) chmod u-w address.txt

(n) wc -l address.txt > totalstudents

7 Vim Editor and Basic Scripting



In the previous chapter, we have discussed commands that can be used to work with Ubuntu Linux. The commands were executed one at a time. Though we could execute multiple commands by using the pipes, the process would become tedious as number of commands increases. A better way of executing multiple commands at one go is to type the sequence of commands in a text file. Then give this file to Linux shell for execution. The Linux shell will execute all the commands available within the text file in the specified sequence. This text file is known as shell script. A shell script can be defined as series of commands written in a plain text file. The shell scripts are commonly used by the users to perform routine individual tasks and system administration. In this chapter we will look at an editor that assists us in writing the shell script along with some sample shell scripts.

Working with Vim Editor

We have learned how to create a file using the cat command. The cat command although it allows us to create a file is not a good option to use when creating a shell script. We need a good text editor to perform such operations. Text editors like nano, pico, vi or Vim, ed and others are generally best suited for creating text file. Gedit is a graphical editor available with GNOME desktop environment. Kwrite is a graphical editor available with KDE desktop environment. We will use the Vim editor which is a visual display editor to write the shell script. This editor is available with almost all Unix and Linux flavors.

The Vim (Vi Improved) is a text editor written by Bram Moolenaar and first released publicly in 1991. It is an enhanced version of the vi editor distributed with most UNIX systems. Vim is a highly configurable text editor built to enable efficient text editing. The Vim editor can be used from both a command line interface and as a standalone application in a graphical user interface.

To work with the Vim editor we will have to initiate it first. Open a new Terminal Window. We can open the Vim editor using two ways, first type ***vi*** at the prompt and press Enter key or type ***vi*** followed by a file name and press Enter key. Figure 7.1 shows the Vim editor interface when we don't specify a file name.

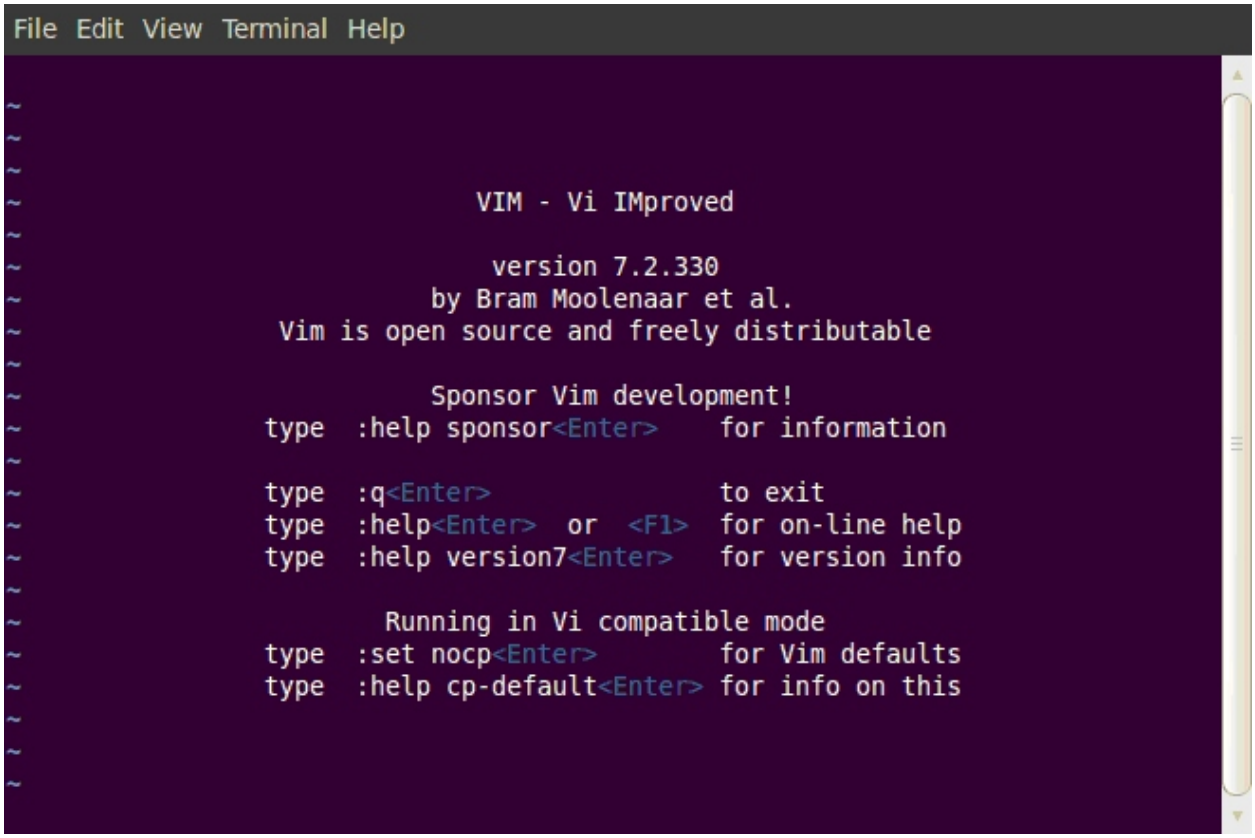


Figure 7.1 : Vim editor interface

It is a good option to start the Vim editor by specifying a file name. Type the command given below:

$vi first_vim_file

and press Enter key, the command when executed will open the editor as shown in figure 7.2.

Figure 7.2 : Creating file using Vim editor

Observe that the screen in figure 7.2 is filled with tildes (~) on the left side of the screen. The tilde (~) symbol indicates that the lines are yet to be used by the editor. Notice that the cursor appears in the top left corner of the screen and some text is visible on the last line. The last line is known as command line and it displays the name of the file along with the information about the total number of lines and columns within the file.

Vim Modes

The Vim editor functions in three different modes namely, a command mode, an insert mode and a last line mode.

The command mode

When we first start editing a file using the Vim editor, the editor will be opened in a command mode. We can issue many commands that allow us to insert, append, delete text, or search and navigate within our file. Note that when using command mode we can't insert text immediately. We first need to issue an insert (i), append (a), or open (o) command to insert the text in the file.

An extension to the command mode is a visual mode. Visual mode is a flexible and easy way to select a piece of text from the file. It is the only way to select a block of text that needs to be modified. Table 7.1 shows us the characters that assist us in the visual mode.

Command	Usage
v	Switch to the visual mode (allows us to manipulate characters)
V	Switch to the visual mode (allows us to manipulate lines)
CTRL + v	Switch to the block-visual mode (allows us to manipulate rectangular blocks of text)

Table 7.1 : Characters that are used in visual mode

The insert mode

When we issue an insert, append, or open command, we will be in the insert mode. The current text editors show us the current mode of operation. Once in an insert mode we can type text into our file or navigate within the file.

We can toggle between the command mode and the insert mode by pressing the ESC key. This operation will be performed often when using the Vim editor. Table 7.2 shows us the characters that assist us in the insert mode.

Command	Usage
a	To insert text after the current cursor position.
i	To insert text before the current cursor position.
A	To append text at the end of the current line.
I	To insert text from the beginning of a line.
O	To insert in a new line above the current cursor position.
o	To insert in a new line below the current cursor position.

Table 7.2: Characters that are used in insert mode

The last line mode

The last line mode normally is used to perform operations like quitting the Vim session or saving a file. To go to the last line mode we first need to be in the command mode. From command mode we can go to last line mode by pressing the colon (:) key. After pressing this key, we will see a colon character at the beginning of the last line of our editor window with a cursor blinking near it. This indicates that the editor is ready for accepting a "last line command".

It is possible to toggle back to the command mode from the last line mode by pressing the ESC key twice or by pressing the [Backspace] key until the initial ":" character is gone along with all the characters that we had typed or by simply pressing the ENTER key.

Creating a file in Vim

Let us now learn how to create a simple text file using the Vim editor. Execute the command given below to open the editor interface.

$vi about_Gandhiji

To enter text within the file the editor needs to be in the insert mode. By default, the editor will start in the command mode. As seen in table 7.2 there are several commands that put the editor into the insert mode. The most commonly used commands to get into insert mode are 'a' and 'i'.

Press 'i' and the editor will now be in the insert mode. Now type the contents as given in the box below.

> Mahatma Gandhi was born on 2nd October 1869 in Porbandar, Gujarat.
> His name was Mohandas Karamchand Gandhi.

Saving the file

Once we have written the contents shown we need to save this file. To save the file we need to switch to the last line mode from the insert mode. Press the ESC key and type colon (**:**), you will notice that colon is displayed in the bottom of the screen. Type ***wq*** (write and quit) as shown in figure 7.3 and press the Enter key. The Vim editor will now be closed and we will be able to see the shell prompt. To see the contents of the file we can use the cat command.

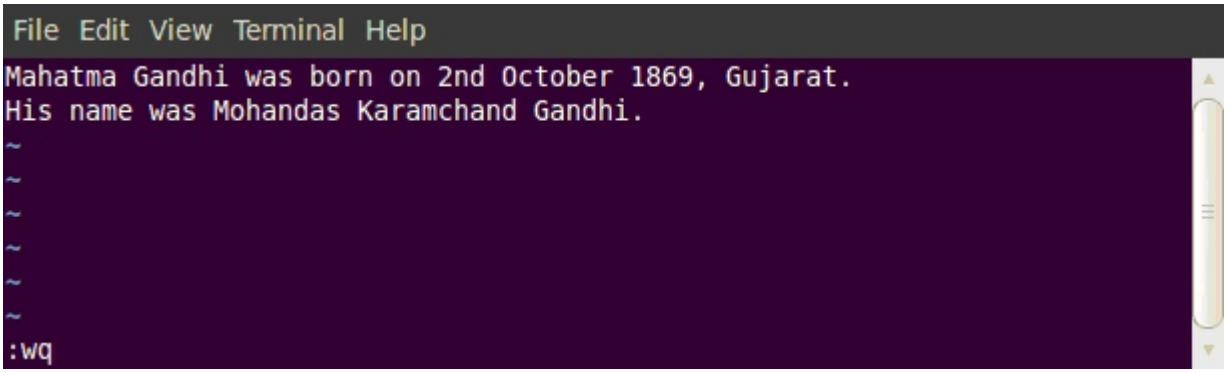


Figure 7.3: Saving a file and quitting the editor

There are several other commands available to save a file depending on the current status and usage. Table 7.3 shows commands that can be used in the last line mode along with their usage.

Command	Usage
:w	To save file and remain in editing mode
:wq	To save file and quit editing mode
:x	To save file and quit editing mode (same as above)
:q	To quit editing mode when no changes are made
:q!	To quit editing mode without saving changes made in the file
:saveas FILENAME	To save existing file with new name and continue editing it under the new file name.

Table 7.3 : The last line mode commands to save the file

Note that if we open the Vim editor without typing file name initially the text will be directly stored in the system buffer (main memory). To transfer the contents from the buffer to a hard disk we need to type the file name along with the *wq* command.

Moving around in the document

You must have observed when creating the file about_Gandhiji the arrow keys (by any chance if you used them) were not working as per our expectations. Normally the arrow keys are used to move the cursor in up, down, left and right directions. In insert mode we cannot do anything except for typing the text. When using the Vim editor we need to use some special keystrokes to move within the document after going to the command mode. Once in command mode we can also use the arrow keys to move within the document. Table 7.4 lists the keystrokes that are used to navigate within the documents.

Command	Usage
h	Moves cursor left
l	Moves cursor right
j	Moves cursor down
k	Moves cursor up
Spacebar	Move cursor right one space

-/+ Keys	Move cursor down/up in first column
CTRL + d	Scroll down one half of a page
CTRL + u	Scroll up one half of a page
CTRL + f	Scroll forward one page
CTRL + b	Scroll back one page
M	Move the cursor middle of the page
H	Move the cursor to top of the page
L	Move the cursor to bottom of page
$	Move the cursor to end of line
)	Move the cursor to beginning of next sentence
(	Move the cursor to beginning of current sentence
G	Move the cursor to end of file
W	Move the cursor one word at a time
Nw	Move the cursor ahead by N number of words
B	Move the cursor back a word at a time.
b	Move the cursor back a word at a time.
Nb	Move the cursor back by N number of words
e	Move the cursor to end of word
gg	Move to first line of file
0	Move to the beginning of the line.

Table 7.4 : Keys to navigate in file

Try and work on these keystrokes to become familiar with them.

Editing the Document

Editing the document is one of the most common operations that a user would perform once the document is created. It is possible to insert or delete any data at a specific position as per our

needs. We can also replace contents or change the case of individual characters if required. A user needs to toggle between the command and the insert mode when we edit a document. The characters in table 7.2 showed us how to initiate different insert options. Let us try to add the contents given in the box below to the file about_Gandhiji.

> His mother, Putlibai, was a very religious lady and used to tell him stories from the scriptures and mythology.
>
> Little Gandhi grew up to be an honest and a decent student. At the age of 13 he was married to Kasturba.

To edit the document we need to again open it using the vi command, hence execute the command **vi about_Gandhiji** again. You will observe that the blinking cursor is visible on the first character at top left. In normal cases we would have used the 'I' option to enter the insert mode, but we need to append the contents at the end.

Type G and you will see that the cursor gets positioned at last line. The position of the cursor will depend upon how the file was initially saved. Typing G may place the cursor in a new line after the last line (case when Enter key was pressed before saving the file) or the cursor will be placed on the first character of the last line (case when Enter key was not pressed before saving the file).

If the cursor is placed at new line we press ESC and then 'I' and start typing the contents. In case we are at the first character of the last line, then press 'o'. This action will take you to a new line. Now start typing the contents, in case of any errors in typing we may use the Backspace key and go back one cursor position at a time to correct the error. Once the editing is over press ESC key and type *:wq* to go the last line mode, save the file and end the Vim session. Figure 7.4 shows the look of the editor after the new contents have been added.

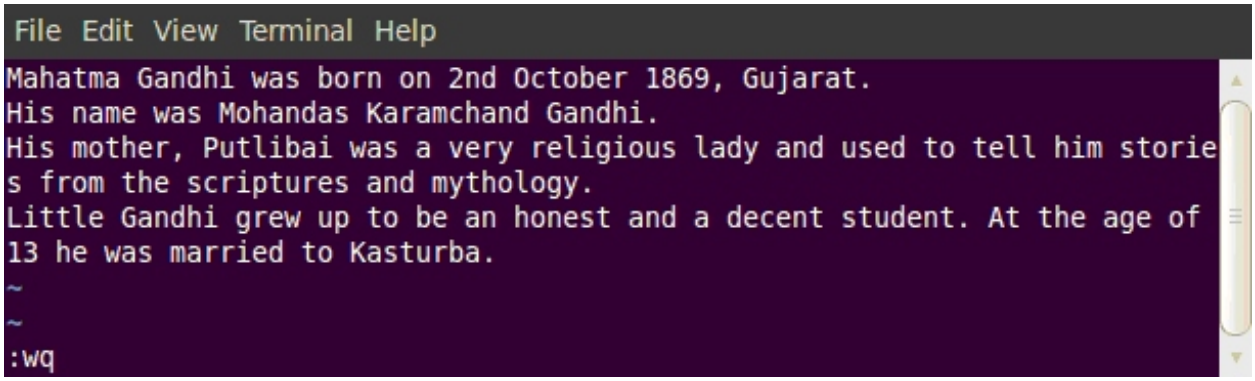


Figure 7.4 : Appending data in existing file

We may use the commands shown in table 7.5 to perform various editing operations on any document.

Command	Usage
u	Undo last change
U	Undo all changes to entire line
dd	Delete single line
Ndd	Delete N number of lines
D	Delete contents of line after cursor
C	Delete contents of line after cursor and insert new text. Press ESC key to end the insertion
dw	Delete one word
Ndw	Delete N number of words
cw	Change word
x	Delete the character under the cursor.
X	Delete the character before the cursor (Backspace).
r	Replace single character
R	Overwrite characters from cursor onward
s	Substitute one character under cursor and continue to insert
S	Substitute entire line and begin to insert at beginning of line
~	Change case of individual character
.	Repeat last command action.

Table 7.5 : Commands to perform editing

In addition to the commands given in table 7.5 the Vim editor also allows us to copy text from our file into temporary buffers and vice-versa. Each buffer acts like temporary memory, more commonly known as "clipboard". Table 7.6 lists some of the commands that are used for capturing and pasting data.

Command	Usage
yy	Copy single line (defined by current cursor position) into the buffer
Nyy	Copy N lines from current cursor position into the buffer
p	Place (paste) contents of buffer after current line defined by current cursor position.

Table 7.6 : Commands to capture and paste

Searching and replacing text

Searching for content and replacing it is another common operation performed by users. The Vim editor allows us to use special commands to search text or a regular expression within the file. We can also substitute a word in place of another using command. Table 7.7 lists various commands that are useful for performing search or replace operation within a file.

Command	Usage
/	Search for text in a forward direction
?	Search for text in a backwards direction
n	Search again in the same direction
SHIFT + n	Search again in the opposite direction
f	Press *f* and type the character to be searched. The cursor will move to that character on the current line.
SHIFT + f	Similar to *f* but searches in backward direction
t	Similar to *f* except that it moves the cursor one character before the specified character.
SHIFT + t	Similar to *t* but searches in backward direction
:s/old_string/new_string	Substitutes *new_string* for the first occurrence of the old_string in the current line
:s/old_string/new_string/g	Substitutes *new_string* for all the occurrences of the old_string in the current line
:%s/old_string/new_string/g	Substitutes *new_string* for all the occurrences of the old_string in the whole file
:%s/old_string/new_string/gc	Substitutes *new_string* for all the occurrences of the old_string in the file, but asks for confirmation before substituting the new_string

Table 7.7 : Commands to perform search and replace operation

Let us try to use some of these commands in the file about_Gandhiji. Assume that we wanted to replace all the occurrence of the word "Gandhi" with word "Gandhiji". To perform this operation we need to first open the file using the command *vi about_Gandhiji*, then go to the last line mode by pressing the *ESC* key and execute the command given below:

:%s/Gandhi/Gandhiji/g

Here %s indicates we are trying to replace a string. The term "Gandhi" refers to the old string that is to be replaced while the term "Gandhiji" refers to the new string. The option "g" indicates that we have to substitute all the occurrences of the term "Gandhi" with the term "Gandhiji" in the whole file. The output of this command is shown in figure 7.5. Observe that it also shows how many occurrences have been replaced. We need to save the file if we need this change to be reflected in it. If we quit without saving then the changes will not be reflected in the file.

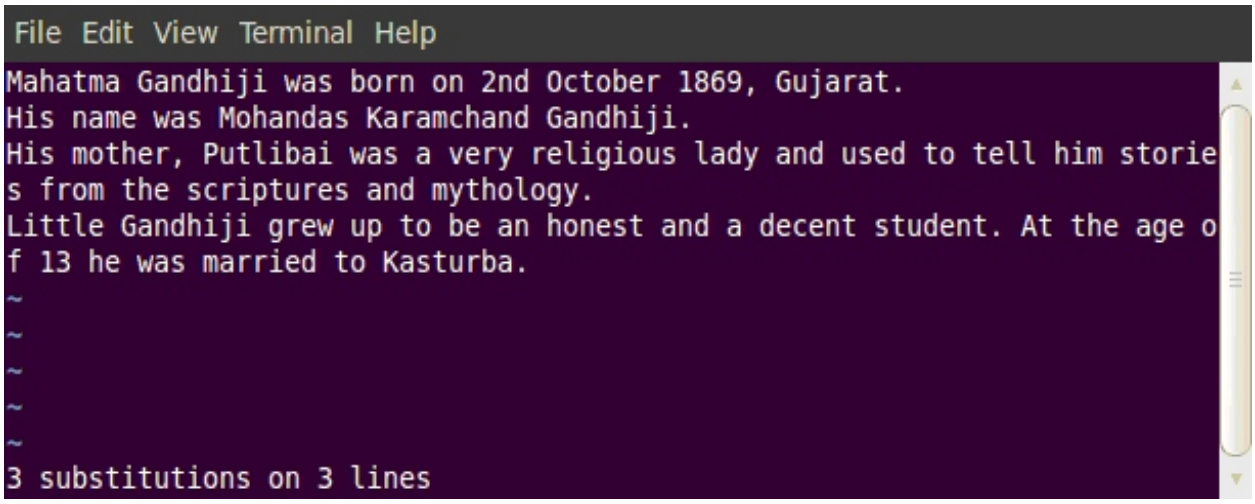


Figure 7.5 : Search and replace operation

Executing Linux commands through Vim

It is also possible to execute the Linux commands from within the Vim editor. To execute any Linux command we need to type the exclamation (!) symbol before the command.

For example if we want to see the current working directory then perform the following steps :

- Open the Vim editor by typing vi on the command prompt.
- Go to the last line mode by pressing 'ESC :'
- Type !pwd
- Press Enter key.

We will be able to see the current working directory. Figure 7.6 shows the operation and its output.

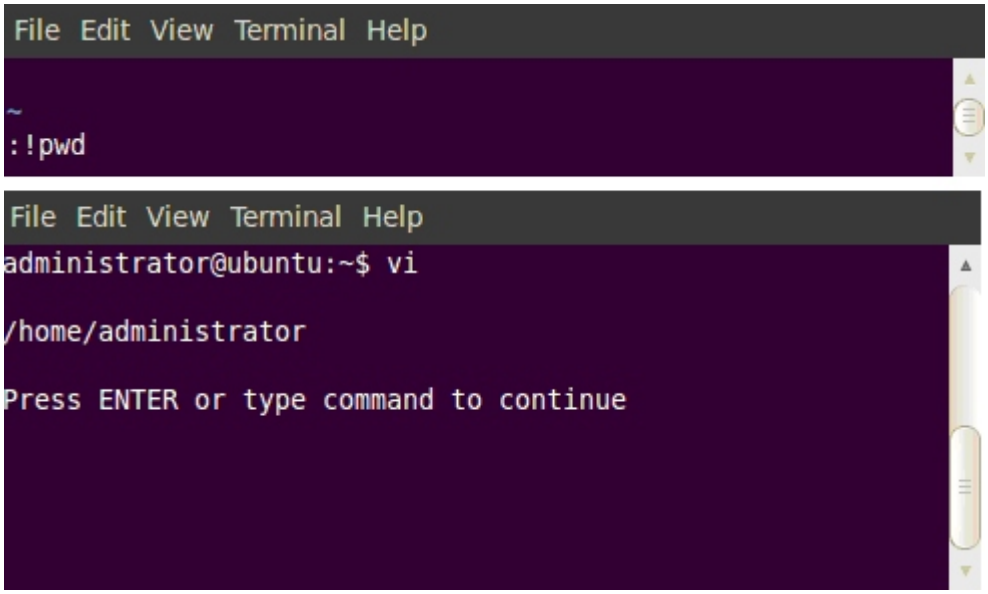


Figure 7.6 : Linux command in Vim editor

Similarly if we want to add the current date in a new line from the current cursor position within a file we may execute the following steps:

- Open the file using Vim editor.
- Go to the last line mode by pressing 'ESC :'
- Type r !date
- Press Enter key.

The r option allows us to insert data in the file or buffer.

Shell Script

We saw how a command can be executed from a command prompt as well as using the Vim editor. Both these mechanisms allow us to execute commands one at a time. A shell script allows us to execute more than one command at one go in a better way. Thus instead of spending time in typing the commands on the prompt every time a task needs to be performed, we can create a shell script and execute the given set of commands by typing a single line command.

Shell script can be defined as "Set of commands written in plain text file that performs a designated task in a controlled order." Shell scripts can be designed to be interactive; such a script may accept input from the user and perform different tasks based on the input provided.

Creating and Executing a Shell Script

We can create the shell script using any text editor. As we have learned to work with the Vim editor we will create scripts in it. Let us create a small shell script that welcomes a user. Type the script shown in the box below in Vim editor and save it with the name *script1.sh*. The extension

"sh" is basically used to specify that the file is a shell script. Note that it is not mandatory to create a file with an extension "sh", a file without extension can also be used as a shell script. But it is always a good practice to give extension as it will help us differentiate between normal files and shell script files.

```
#Script 1: Script to welcome the user who has logged into the system
clear
echo Hello
who am i
echo  Welcome to Ubuntu Linux
```

Observe the first line in the script; it begins with symbol '#'. Any line preceded by the '#' symbol is considered as a comment. The comments when part of the script are not executed; they are messages that help user understand the usage or meaning of the script. The second line is a command that clears the screen contents before giving the output of the script. The third line displays a message "Hello", the fourth line executes a command "*who am i*" that gives us the name and some additional details of the user currently logged into the system. The last line again displays a message "Welcome to Ubuntu Linux".

To execute the script we need to use *sh* or *bash* command. If the script is stored in current directory then type the command mentioned below:

$bash script1.sh or **$sh script1.sh**

Sometimes we might come across issues related to file privileges. For a script to be executed it needs to have execute permission explicitly set. By any chance if such a problem occurs we will have to use the chmod command to set the desired privileges. For Example issuing the command ***chmod + x script1.sh*** will make the file executable. If everything goes fine we will get the output similar to the one shown in figure 7.7.

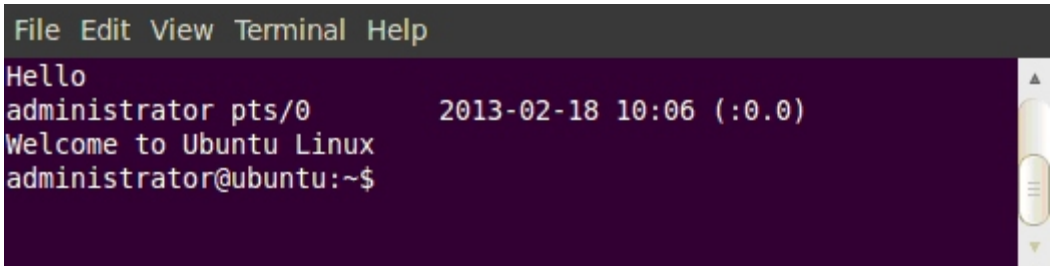


Figure 7.7 : Output of Script 1

Observe that in figure 7.7 we are getting some additional contents along with the user name. Let us use our knowledge of filters and try to remove the additional contents. The modified script is shown below:

```
#Script 2: Modified script to welcome the user who has logged into the system
clear
echo Hello
echo "`who am i | cut -d " " -f 1`"
echo Welcome to Ubuntu Linux
```

Observe that we have used the filter *cut* along with the command *who am i*. We further have joined the two commands using the pipe. To make sure that the contents within the double quotes after the echo command are not treated as message we enclose them in back quotes (‘ ‘). The back quotes are printed on the key with ~ sign on the keyboard. Type the modified script using Vim editor and save it as *script2.sh*. Execute this script and you will observe that we are only able to see the contents that we want. Figure 7.8 shows the output of the modified script.

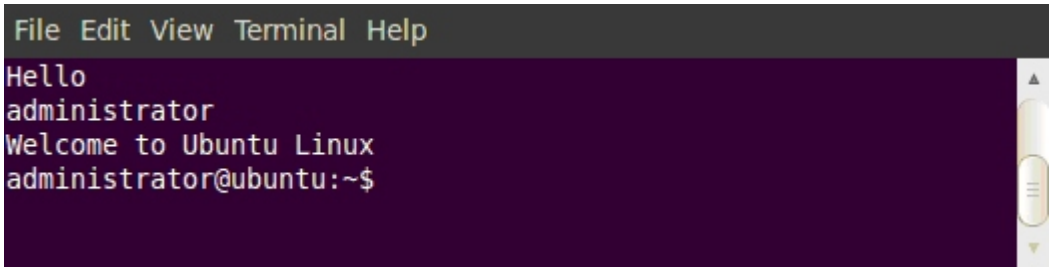


Figure 7.8 : Output of Script 2

Let us further modify the script to display current date and time. The script is given in the box below .

```
#Script 3: Script to welcome the user and display login date and time
clear
echo Hello
echo "`who am i | cut -d " " -f 1`"
echo Welcome to Ubuntu Linux
echo The current date and time is
date
```

Create a file named *script3.sh* and type the contents of the script 3 in it. Execute it and observe the output. Once we are comfortable using the shell scripts we will find them very helpful in performing repetitive tasks.

Shell Script Variables

The process of shell scripting is almost similar to the process of writing programs in a higher level language. One of the most common features of higher level programming is provision of variables.

As the name indicates variables are entities wherein we can store or edit a value. The value stored in the variable can also be reused or changed as per users need. Shell script variables like any other programming language variables are integral part of shell scripting. A variable when used in a shell script allows us to assign a value to it or accept its value from the user. We can also display the value assigned to this on the screen using echo command. Let us write a small script that shows the use of variable. The code of the script is given in the box below:

```
# Script 4: Shell script to show use of variables
clear
subject="Computer Science"
echo $subject is easy to learn.
```

Type the contents of the script 4 in a file named *script4.sh*. Let us try to understand the script just saved. Similar to all the other scripts the first line indicates a comment. The second line clears the screen of any previous contents. In the third statement we have defined a variable called subject and assigned it value "Computer Science" using a simple assignment operator. As the string contains white space in between two words we need to enclose it within double quotes. The fourth line displays the message on the screen. The '$' symbol preceding the variable name *subject* instructs the shell to extract the value stored or assigned in the variable. Figure 7.9 displays the output of the script when it is executed.

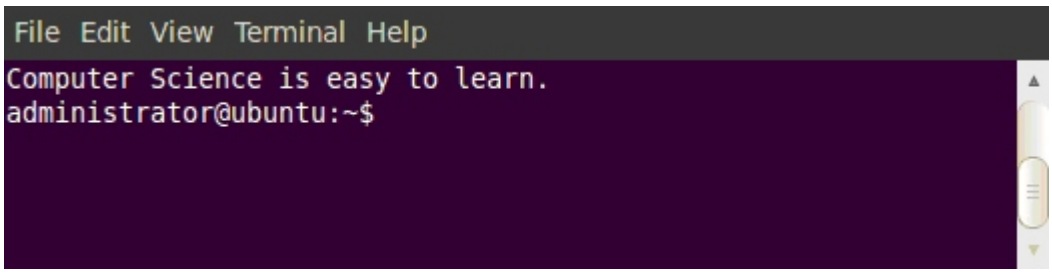


Figure 7.9 : Output of Script 4

A user needs to take care that there should not be any space on either side of the equal to (=) symbol at the time of assigning a value. If due to some reasons a space occurs at either side, the shell will interpret the string after the space as a command. This may give unexpected outputs. The statement subject="Computer Science" first creates a variable named *subject* and then assigns it a value "Computer Science". If we reuse this variable again, the old value stored in it will be overwritten. Let us try to understand the last statement by writing a simple script given in the box:

```
# Script 5: Shell script to show use of variables
clear
subject="Computer Science"
echo $subject is easy to learn.
subject="Economics"
echo $subject is easy to learn.
```

Save the script as script5.sh and observe its output after executing it.

We need to follow the rules mentioned below when defining a variable in a shell script.

- A variable name can consist of alphabets, digits or an underscore (_).
- No special character other than underscore allowed as part of variable name.
- The first character of a variable name must either be an alphabet or an underscore.

Note :

If the shell is unable to understand a word as a variable it will interpret it as a Linux command.

User Interaction and Shell Script

You must have observed in the previous example that we have assigned a value to the variable within the script itself. A variable when used in such a manner does not have much significance. A variable is generally used so that it can be further used in some operations with change in its value if needed. This property can be achieved only if we are able to accept the value of the variable from the user. It is possible to assign value to variables defined in the shell script using the *read* command. The *read* command expects the user to key in the data on the standard input device, it then takes all the contents that we type and stores it in the variable name supplied to it as an argument. Let us rewrite a script 5 to accept the subject names from the user. Save the code in the box with a file name *script6.sh*.

```
# Script 6: Shell script to accept value of variable from user
clear
echo -n "Enter your name: "
read name
echo -n "Enter name of a subject: "
read subject
echo Hello I am $name and I am studying $subject.
```

When we execute the script, the first executable statement will first clear the screen contents. Then it will display a message “Enter your name:”, the next statement waits for the user to enter its name. Pressing of Enter key indicates end of entry, so be careful that it is pressed only after the name has been typed. In case we press Enter key without typing anything the script will assign NULL value to the variable and move to next command. The third and fourth

statement also does the same task of displaying message and waiting for the user to key in a value for subject. The last statement displays the values of both the variables along with an appropriate message.

Observe that we have used -n option along with the echo command. This option instructs the echo command not to print a new line after the message is displayed. The echo command by default inserts a new line after displaying the message passed to it as an argument. Figure 7.10 shows output of the script.

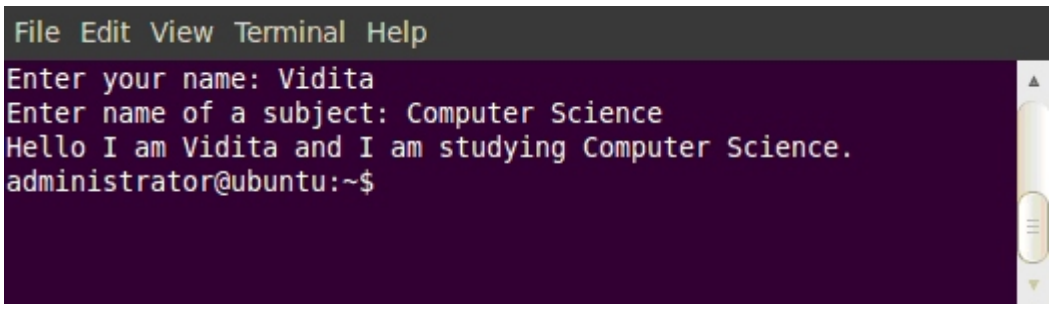


Figure 7.10 : Output of Script 6

As we are accepting the values of both the variables from the user, the output will change according to what user enters every time we execute the script. Let us write one more script that accepts a file name from the user and display the total number of lines in that file. The code of the script is shown herewith save it as *script7.sh*.

```
#Script 7: Shell script to display total number of lines in a file
clear
echo  -n "Enter a file name:  "
read fname
echo "There are `cat $fname | wc -l ` lines in a file $fname"
```

Figure 7.11 shows the output of the script when it is executed.

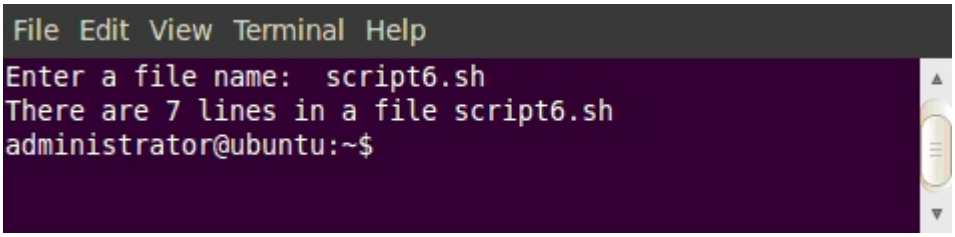


Figure 7.11 : Output of Script 7

Shell Arithmetic

In the previous section we saw how to define a variable, assign value to it and how to retrieve the stored value. The value assigned so far were all strings (set of alphabet, digit or special characters). We can also assign only numeric values to the variable and perform operation with them. Let us write a shell script that accept two numbers and perform addition of these numbers. The code of the script is shown in the box below:

```
# Script 8: Script to add two numbers
echo -n "Enter first number: "
read num1
echo -n "Enter second number: "
read num2
sum=`expr $num1 + $num2`
echo "The addition of $num1 and $num2 is $sum"
```

Type the script and save it as *script8.sh*. In this script the term *expr* means expression, the contents written after expr are assumed to be operands and operators of an expression. Note that there should be one space between operator (+) and operands ($num1, $num2). Additionally, there should be no space before or after the assignment operator (=).

Figure 7.12 shows the different output of the script when it is executed twice. We are able to see the different outputs in one screen as we have not used the clear command in this script.

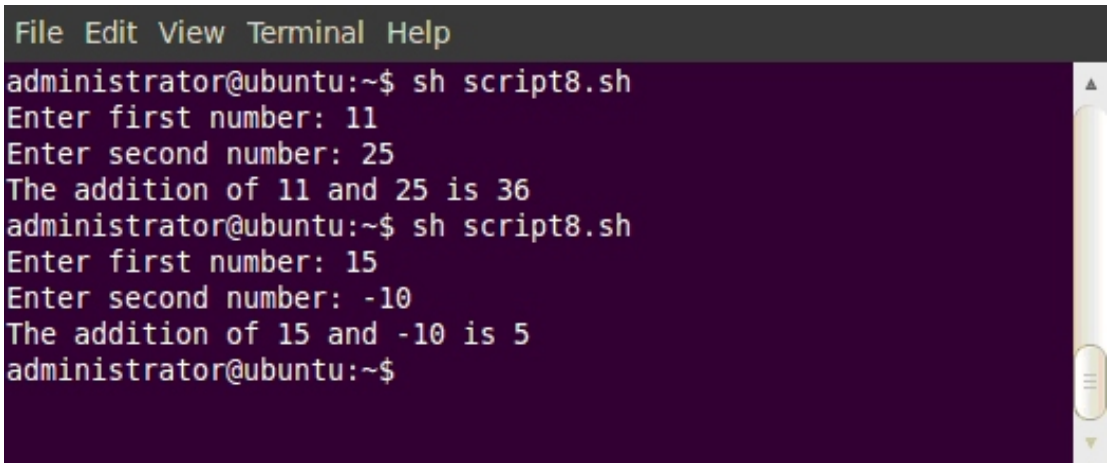


Figure 7.12 : Output of Script 8

We can also perform subtraction, multiplication, division and modular division by using the -, *, / and % operators respectively. The expressions are evaluated as per the general norms of mathematics.

In case of tie between operators of same priority, preference is given to the operator which occurs first. To force one operation to be performed earlier than the other, we can enclose the operation in parenthesis.

For example, in the expression **$num1 * \($num2 + $num3 \) / $num4**, the operation **$num2 + $num3** will be evaluated first as it is enclosed within parentheses. Observe that we have preceded the '*' symbol as well as the left and right parentheses by a back slash character (\).

Note :
We need to prefix the multiplication (*) symbol with backslash (\) character when finding product of two numbers. Otherwise the shell will treat the (*) symbol as a wildcard character.

Let us create one more script that will accept a birth year from the user and display users current age in years. The code of the script is shown in the box below.

```
# Script 9: Script to calculate age of user in years
echo -n "Enter year of your birth: "
read byear
cyear=`date | tr -s ' ' | cut -d " " -f 6`
age=`expr $cyear - $byear`
echo "You are $age years old as of today."
```

Observe that in script 9 to make sure that all the multiple spaces in output of date command are squeezed to single space we have used the *tr* command with *–s* option. As we want only the year value which appears in the 6th column of the output when the date command is executed we have used the *cut* filter. Figure 7.13 shows the different output of the script when it is executed.

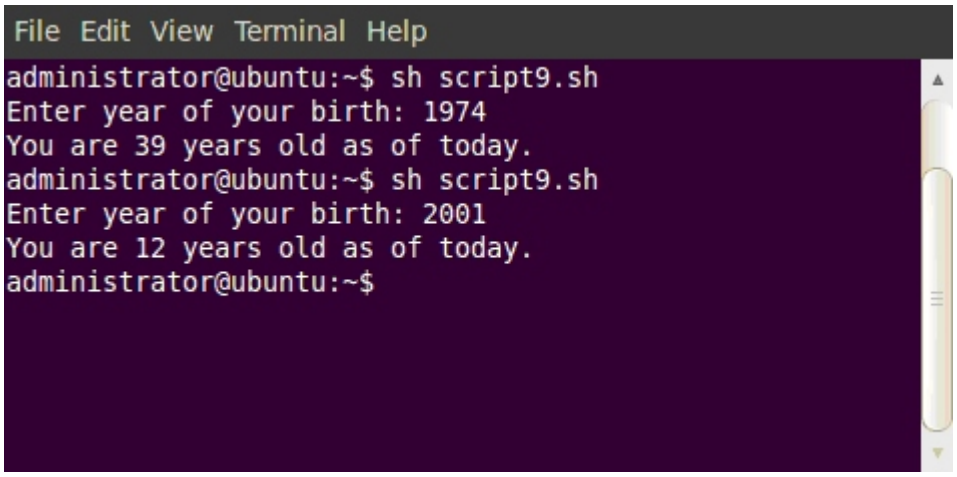


Figure 7.13 : Output of Script 9

Use of Shell Scripts

The shell script is a very powerful tool of Linux. It has almost all the capabilities of any higher level programming language. Once familiar with it we can perform and automate many tasks easily. Generally the repetitive task should be done using the shell scripts. Some uses of shell scripts are mentioned below:

- Create a new command using multiple set of commands.
- For automating many aspects of computer maintenance, for example create 1000 user accounts; delete all size 0 files, installation of new software etc.
- Data backup

As such there are no limitations on its usage; a user may use it for any purpose that he wants.

Summary

In this chapter we learned how to use the Vim test editor provided with Ubuntu Linux. The text editor is very powerful; it allows creating, updating and deleting contents of a file. Further we can search for required contents within a file. We also learned how to create a simple text file as well as a shell script using this editor. The shell script is a text file that contains sequence of commands that can be executed by simply using the shell scripts file name. Finally we saw how a shell script can be made equivalent to a high level program by making use of a variable and then using it in an expression.

EXERCISE

1. Explain different modes available in Vim editor.

2. List and explain the working of different save options of Vim editor.

3. Explain the difference between using dd and 2dd command.

4. What is a shell script?

6. List at least three uses of shell script.

7. **Choose the most appropriate option from those given below :**

(1) In how many modes Vim editor works?

(a) One (b) Two

(c) Three (d) Four

(2) Which of the following statement is true for Gedit?

(a) It is a Command line editor.

(b) It is a Graphical editor.

(c) It is not an editor.

(d) It is available with KDE Desktop environment.

(3) :wq in Vim editor is used for which of the following activities?

(a) To save file and remain in editing mode

(b) To save file and quit editing mode

(c) To quite editing mode without saving changes made in the file

(d) All of the above

(4) Which of following keys is not used to go into insert mode of the Vim editor?

(a) o (b) i

(c) a (d) cw

(5) Which of the following keys are used to delete a line?

(a) ce (b) ge

(c) dd (d) d$

(6) Which of the following statements is used to search for phrase in the file?

(a) :set is (b) :help cmd

(c) :!cmd (d) /phrase<ENTER>

(7) Which of the following syntax is used to substitute all occurrences of phrase1 with phrase2 in the entire file without asking for user confirmation?

(a) :%s/phrase1/phrase2/g (b) :%s/phrase1/phrase2/gc

(c) :s/phrase1/phrase2/g (d) :s/phrase1/phrase2/gc

(8) Which of the following character is used for commenting a line in a shell script?

(a) * (b) %

(c) $ (d) #

(9) Which of the following symbol instructs a shell script to extract the value of a variable?

(a) * (b) %

(c) $ (d) #

Laboratory Exercises

Write a script to perform following operations :

(a) To display the date and time in the given format:

"Today is February 15, 2013 and current time is 12:10:23"

(b) To display the login details of current user in the following format:

Name of the user:

Login date:

Login time:

(c) To display the date, time, username and current directory.

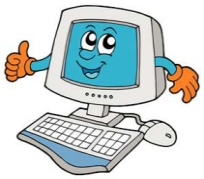
(d) To accept a string and a filename from the user. Search all occurrences of the string inside a given file.

(e) To accept a file name from the user and count number of lines in it.

(f) To accept two file names from the user and creates a new file containing the contents of both the files provided as input.

To accept two file names from the user and compare them.

8 Advanced Scripting



In chapter 7 we learned how to use the Vim editor and also saw how to write the basic shell scripts. We mentioned that the shell scripts have features similar to a higher level language. Are we then learning a new language? No, we are not learning any new language at all. We are learning one of the best feature that an open source OS provides. The shell scripts are used for routine system administration tasks. They are the best tools an administrator can get to easily monitor and control his systems even if he is at remote location. The shell scripts designed so far were sequential in nature; the commands were executed in the same order in which they appeared in the script. While performing administrative tasks, we may need to perform execution of some statements repeatedly. We may also need to skip execution of some statements based on predefined conditions. In this chapter we will see some scripts related to system administration and discuss how to use decision statements and looping constructs in shell script.

Finding Process Id

In Linux all programs (executables stored on hard disk) are executed as processes (a program loaded into memory and running). Each process when started has a unique number associated with it known as process id (PID). We can perform operations like view or stop a process. To see the processes associated with the current shell we can issue the *ps* command without any parameters. We can view the process of all the users by using the *ps -ef* command. Figure 8.1 shows the processes running on our system.

```
File  Edit  View  Terminal  Help
administrator@ubuntu:~$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 15:14 ?        00:00:00 /sbin/init
root         2     0  0 15:14 ?        00:00:00 [kthreadd]
root         3     2  0 15:14 ?        00:00:00 [migration/0]
root         4     2  0 15:14 ?        00:00:00 [ksoftirqd/0]
root         5     2  0 15:14 ?        00:00:00 [watchdog/0]
root         6     2  0 15:14 ?        00:00:00 [migration/1]
root         7     2  0 15:14 ?        00:00:00 [ksoftirqd/1]
root         8     2  0 15:14 ?        00:00:00 [watchdog/1]
root         9     2  0 15:14 ?        00:00:00 [events/0]
root        10     2  0 15:14 ?        00:00:00 [events/1]
```

Figure 8.1 : List of processes

Table 8.1 gives the meaning of some of the columns listed in figure 8.1.

Column Name	Description
UID	Name or number of the user who owns the process.
PID	A unique numeric process identifier assigned to each process.
PPID	Identifies the parent process id, the process that created the current process.
STIME	The start time for the current process.
TTY	Identifies the terminal that controls the current process.
TIME	Identifies the amount of CPU time accumulated by the current process.
CMD	Identifies the command used to invoke the process.

Table 8.1 : Explanation of columns displayed in ps –ef command

Many times an administrator needs to find how many processes a particular user is executing. Let us write a script that helps administrator find number of process run by a particular user.

```
#Script 10: Script to find out how many processes a user is running.
clear
echo -n "Enter username: "
read usrname
cnt=`ps -ef  | cut -d " " -f 1 | grep -o $usrname | wc -w`
echo "User $usrname is running $cnt processes."
```

Save the script as *script10.sh*. Let us try to understand the working of this script. The first command clears the content on the screen. Then a message is displayed for the user to enter a user name. The read command then assigns the string read from the keyboard to variable *usrname*. Then we have combined four commands namely ps, cut, grep and wc using pipe. The ps *-ef* command displays list of processes being run by all the users of the system. Its output is then given to the cut command. The cut command extracts the first field (username) from this output. The extracted list of first field is then given to the grep command. The grep command finds all the users that match with the value that is extracted from variable *usrname*. This matched list is then given to the wc command that counts the occurrence of the given word (username). Finally this word count is assigned to variable cnt. The last command then displays the actual output needed. Figure 8.2 shows the sample output of the script.

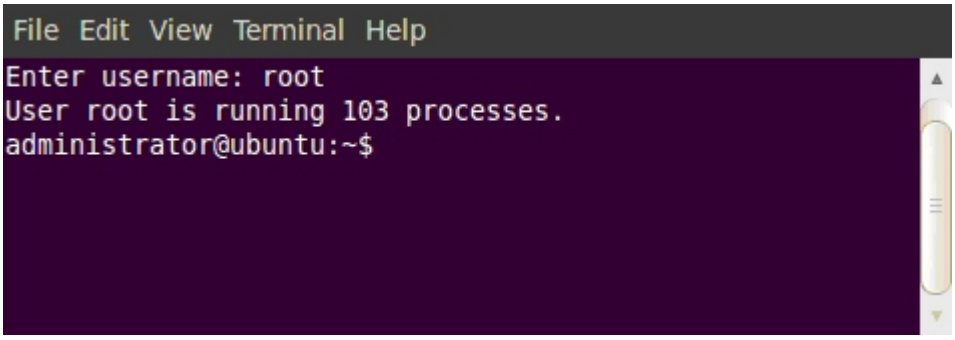


Figure 8.2 : Output of Script 10

As mentioned earlier we may remove a process and release some memory space if so required. To remove the process from memory we use the *kill* command. For example if we issue a command

$kill -9 101

Then the process with PID=101 will be forcibly removed from the memory. Let us have a look at script similar to script10.sh that accepts user name as a command line argument and tells us how many terminals that user is using. The code of the script is given in the box below:

```
#Script 11: Script to find out how many terminals a user has opened.
cnt=`who | cut -d " " -f 1 | grep -o $1 | wc -w`
echo "User $1 has opened $cnt terminals"
```

Save the script as *script11.sh*. Observe that the script we created in the previous example used variables. In this script we have made use of a command line argument. The entity $1 here refers to a command line argument. To execute this script type the command as mentioned below:

$sh script11.sh administrator

You must have observed that the script is executed in the similar manner as we have executed the previous script. But here we have specified additional value “administrator” (readers may specify any name of their choice). Linux stores the values provided through command line in dollar variables, named $1, $2, $3 and so on. First argument will be stored in $1, second in $2, third in $3 and so on till $9. These arguments are known as command line arguments. The output of the script is shown in figure 8.3.

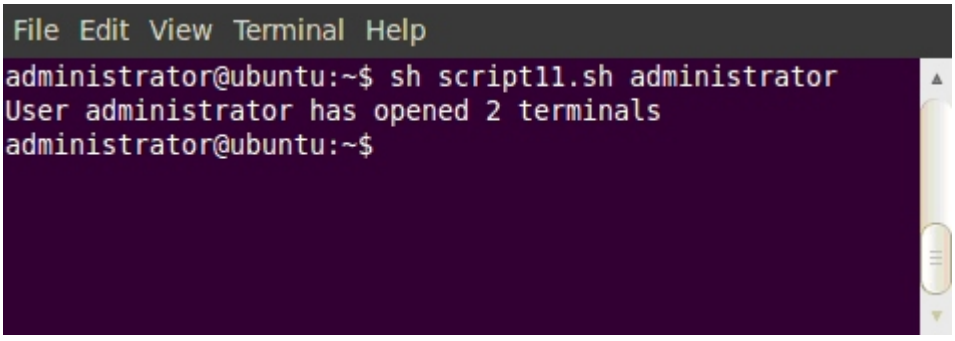


Figure 8.3 : Output of Script 11

Let us try to understand the working of this script. In the first statement after comment we have combined four commands namely who, cut, grep and wc using pipe. The *who* command displays list of all users that have logged into the system. Its output is then given to the cut command. The cut command extracts the first field from this output. The extracted list of first field is then given to the grep command. The grep command then finds out all the users that match with the entered command line argument value ($1 = administrator). This matched list is then given to the wc command that counts the occurrence of the given word (username). Finally this word count is assigned to variable cnt. The last command then displays the actual output needed.

Decision Making Tasks

Let us say an administrator wants to create a directory, he can do it using an mkdir command. But if he uses a script for creating a directory he can generate appropriate messages also. Let us write a script that allows an administrator to create a directory.

```
#Script 12: Script to create a directory with appropriate message.
echo -n "Enter directory name: "
read mydir
if  [ -d $mydir -o -f $mydir ]
then
 echo "A File or Directory with the name $mydir already exists".
 exit 0
fi
mkdir $mydir
echo "Directory with name $mydir created successfully."
```

Save the script as *script12.sh*. Observe that in this script we have used an if-then-fi construct. This construct in shell scripts allows us to perform decision making. The *if* statement of Linux is concerned with the exit status of a test expression. The exit status indicates whether the command was successfully executed or not. The exit status of command is 0 if it has been executed successfully; otherwise it is set to 1. Figure 8.4 shows the output of the script.

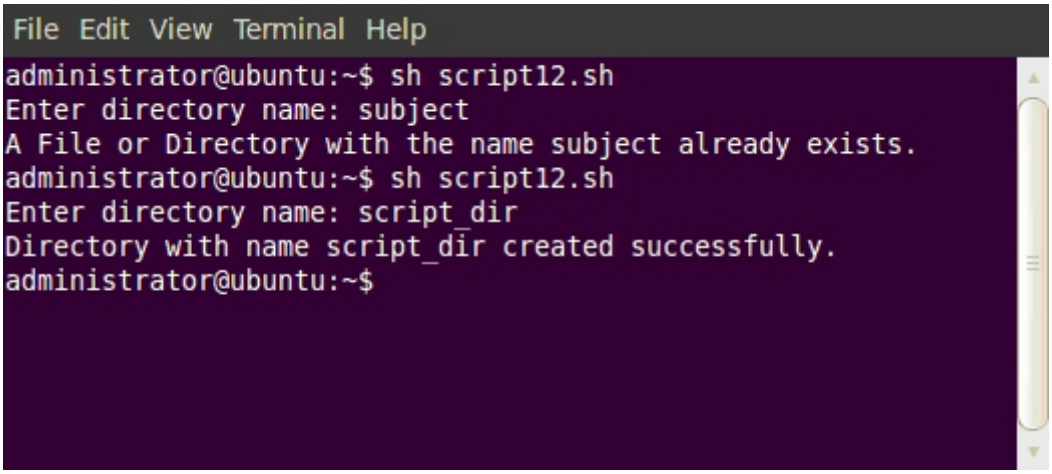


Figure 8.4 : Output of Script 12

Observe the output of figure 8.4 carefully. In one case we get message indicating that the directory already exists and in second case we are able to create a directory with the specified name. Note that the condition in the above script is enclosed in a square bracket. There should be one space after opening square bracket and one before closing square bracket. If the condition is evaluated to true then statements typed inside *then* block will be executed otherwise not. The end of the *if* statement is indicated by *fi* statement. Also note that the *then* keyword should be typed below if statement else we will get error. The -d, -f -o options used in the script will be discussed later in the chapter.

We can use the following four decision making instructions while creating a shell script in Linux:

if-then-fi

if-then-else-fi

if-then-elif-then-else-fi

case-esac

It is a normal practice to copy a file and keep in it another directory (maybe for the purpose of backup). The user many times gets confused whether both the files are same or different. Let us write a script that helps user compares such files. The script when executed compares both the files using the *cmp* command. Based on the output of the cmp command it then displays appropriate messages.

```
#Script 13: Script to compare files.
echo -n "Enter a file name: "
read fname
if cmp ./$fname  ./backup/$fname
then
     echo "$fname is same at both places."
else
     echo "Both $fname are different."
fi
```

Save the script as *script13.sh*. Here we first accept a file name from the user. To keep the script simple as of now we have used absolute paths for directory (user can convert it relative path). We have also assumed that the file names at both the locations are same. Figure 8.5 shows the output of the script.

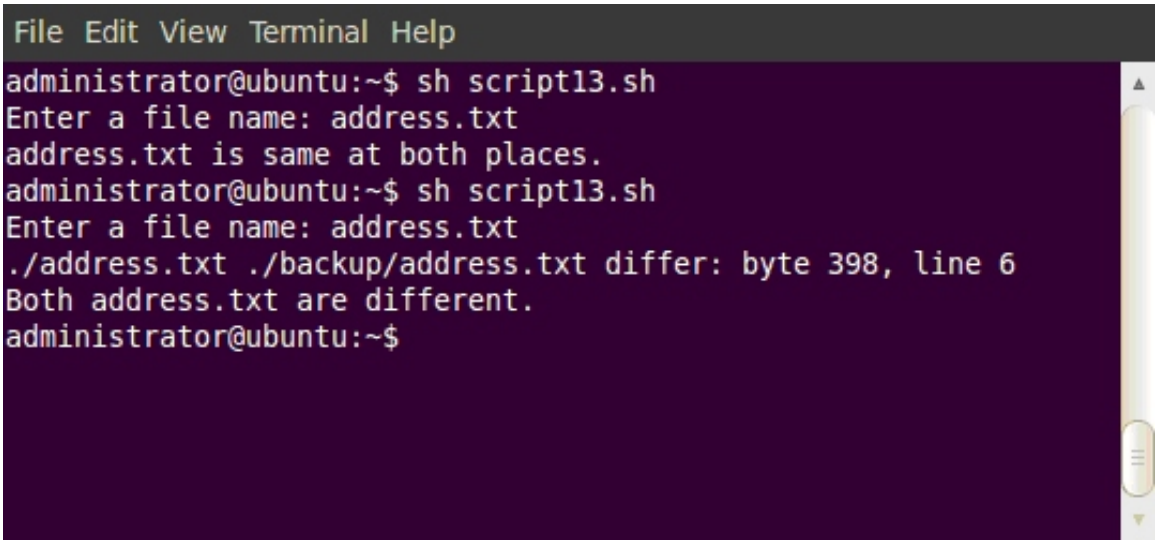


Figure 8.5 : Output of Script 13

Here we have executed the script twice. In the first run both the files contents are same hence we get the message that both files are same. Before second run we have modified the file in the current directory hence we are getting the message that files are different.

In previous chapter we have written a small script to welcome the user that has logged in the system. Let us modify it further so that we display a proper welcome message (Good morning, Good afternoon or Good evening depending on the time the user has logged in).

```
#Script 14: Script to display welcome message to the user.
clear
hour=` date +"%H"`
usrname=`who am i | cut -d " " -f 1`
if [ $hour -ge 0 -a $hour -lt 12 ]
then
    echo "Good Morning $usrname, Welcome to Ubuntu Linux Session."
elif [ $hour -ge 12 -a $hour -lt 18 ]
then
echo "Good Afternoon $usrname, Welcome to Ubuntu Linux Session."
else
echo "Good Evening $usrname, Welcome to Ubuntu Linux Session."
fi
```

Figure 8.6 shows the output of the script. The output will vary depending on when the user has logged in.

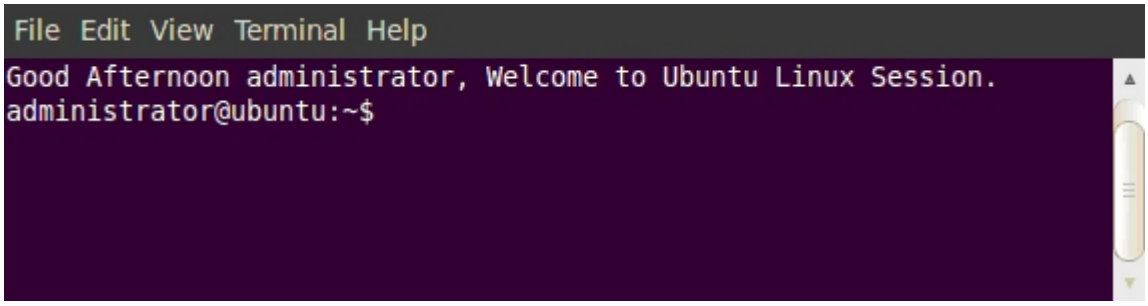


Figure 8.6 : Output of Script 14

The test command

It is possible to use different forms of if statements. Linux also provides test command which can be used in place of square brackets used in previous scripts. Let us write script to check whether a user has created more than some specified files in a given month or not.

```
#Script 15: Script to see whether user has created more than specified files in a month.
clear
cnt=`ls -l | grep -c [-]"$1"`
echo -n "Enter number of files: "
read nfile
if test $cnt -gt $nfile
then
    echo "You have created more than $nfile files in the month of $1"
else
 echo "You have not created more than $nfile files in the month of $1"
fi
```

Let us try to understand the script. Here we have defined a variable named *cnt*. This variable is assigned the total count of the files created in a specified month. To find out the number of files we have used two commands namely *ls* and *grep*. The ls -l command is used to display details of files and directories. Its output is then given to the grep command that matches regular expression [-]"$1". The month is specified as two digit numeric value and accepted through command line argument assigned to $1. Then we have defined a variable *nfile* that stores the value of number of files that we want to compare with. The -gt option in the test indicated greater than comparison. Here we are checking whether the value of cnt is greater than the value of nfile or not. If the value of cnt is greater than the value of nfile we print the message "You have created more than $nfile files in the month of $1", where $nfile and $1 are replaced with appropriate values. Otherwise we print message "You have not created more than $nfile files in the month of $1".

Figure 8.7 shows the output of the script when we issue a command shown below on the command prompt.

$sh script15.sh 02

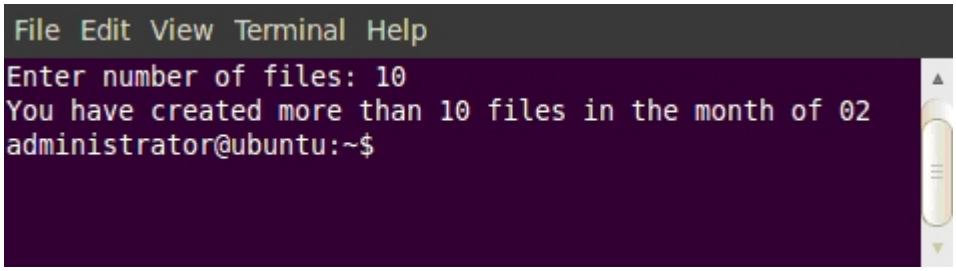


Figure 8.7 : Output of Script 15

The if statement can work with numerical values, strings and files. In turn the tests performed are known as numerical test, string test and files test respectively. Observe that we have used options like -d, -f, -o, -a, -ge, -lt and -gt in the scripts created so far. All these options allow us to perform various types of condition matching.

Relational Operators

The numerical test is performed using relational operators. The options -ge, -lt and -gt refers to relational operators. These operators are used to compare values of two numeric operands. Table 8.2 lists the relational operators that can be used in shell scripts along with their usage.

Operator	Usage
-gt	greater than
-lt	less than
-ge	greater than or equal to
-le	less than or equal to
-ne	not equal to
-eq	equal to

Table 8.2 : Relational operators

Logical Operators

For taking precise and appropriate decisions many times a user needs to combine one or more conditions. To combine conditions we make use of logical operators. Table 8.3 lists the logical operators along with their usage.

Operator	Usage	Minimum conditions that can be combined	Output
-a	AND	Two	True if both conditions are true, false otherwise
-o	OR	Two	True if any one condition is true, false only if both conditions are false
!	NOT	One	Converts true to false and vice versa

Table 8.3 : Logical Operators

File Operators

It is also possible to use *if* statement to check the status of file or a directory. Similar to the relational operators we have file operators that allows us to check the status of a file. These operators are used as a condition within the if statement. By using file operators we can come to know whether a specified name is an ordinary file or a directory. We can also find out the status of file permissions using them. Table 8.4 lists usage of these options.

Condition Tested	Output
-s name	True if a file with the specified name exists and has size greater than 0.
-f name	True if a file with the specified name exists and is not a directory.
-d name	True if a directory with the specified name exists.
-r name	True if a file with the specified name exists and the user has read permission on it.
-w name	True if a file with the specified name exists and the user has write permission on it.
-x name	True if a file with the specified name exists and user has execute permission on it.

Table 8.4 : File test conditions

Many times administrator needs to find whether a specified file has size equal to zero or not. He can then perform maintenance operations like delete the file in case its size is zero. He may additionally need to find whether write permissions on the file is set or not. Let us write a script that allows administrator to check the file size and know what permissions are allocated to the file.

```
#Script 16: Script to check file size.
echo -n "Enter a file name: "
read fname
if [ -s $fname -a -w $fname ]
then
 echo $fname has size greater than 0 and user has write permission on it.
else
 echo $fname has size 0 or user does not have write permission on it.
fi
```

Save the script as *script16.sh*. Here the statement **if [-s $fname -a -w $fname]** has multiple conditions. The result of the *if* statement is evaluated when both the conditions give us some output. Table 8.5 lists the value that can be generated as output when the if statement is evaluated and figure 8.8 shows different output of the script.

-s $fname	Reason	-w $fname	Reason	if [-s $fname -a -w $fname]
False	File size = 0 or File does not exists	False	Write permission not set	False
False	File size = 0 or File does not exists	True	Write permission set	False
True	File size > 0	False	Write permission not set	False
True	File size > 0	True	Write permission set	True

Table 8.5 : Outputs of if [-s $fname -a -w $fname]

```
File  Edit  View  Terminal  Help
administrator@ubuntu:~$ sh script16.sh
Enter a file name: script5.sh
script5.sh has size 0 or user does not have write permission on it.
administrator@ubuntu:~$ sh script16.sh
Enter a file name: script10.sh
script10.sh has size greater than 0 and user has write permission on it.
administrator@ubuntu:~$
```

Figure 8.8 : Output of Script 16

The if-then-fi and if-then-else-fi statements used so far allow us to test limited set of conditions. In case a user needs to perform more number of tests these statements may not be of much help. In such cases we may use the if-then-elif-then-else-fi or the case statement.

Let us write a script that accepts three files from user and displays the file which has maximum size.

```
#Script 17: Script to find the file with the maximum size.
clear
echo -n "Enter name of first file: "
read fname1
echo -n "Enter name of second file: "
read fname2
echo -n "Enter name of third file: "
read fname3
fsize1=`wc -c $fname1 | cut -d " " -f 1`
fsize2=`wc -c $fname2 | cut -d " " -f 1`
fsize3=`wc -c $fname3 | cut -d " " -f 1`
echo Size of $fname1 = $fsize1
echo Size of $fname2 = $fsize2
echo Size of $fname3 = $fsize3
if [ $fsize1 -eq $fsize2 -a $fsize1 -eq $fsize3 ]
 then
   echo "All files have same size"
elif [ $fsize1 -gt $fsize2 -a $fsize1 -gt $fsize3 ]
 then
        echo "$fname1 has maximum size."
elif [ $fsize2 -gt $fsize1 -a $fsize2 -gt $fsize3 ]
then
        echo "$fname2 has maximum size."
else
        echo "$fname3 has maximum size."
fi
```

Save the script as *script17.sh*. The six statements after the clear command are used to accept the file names from the user. The next three statements calculate size of the files, later these sizes are displayed to the user. Finally using the if condition, the script finds out the file that has maximum size. Figure 8.9 shows the output of the script.

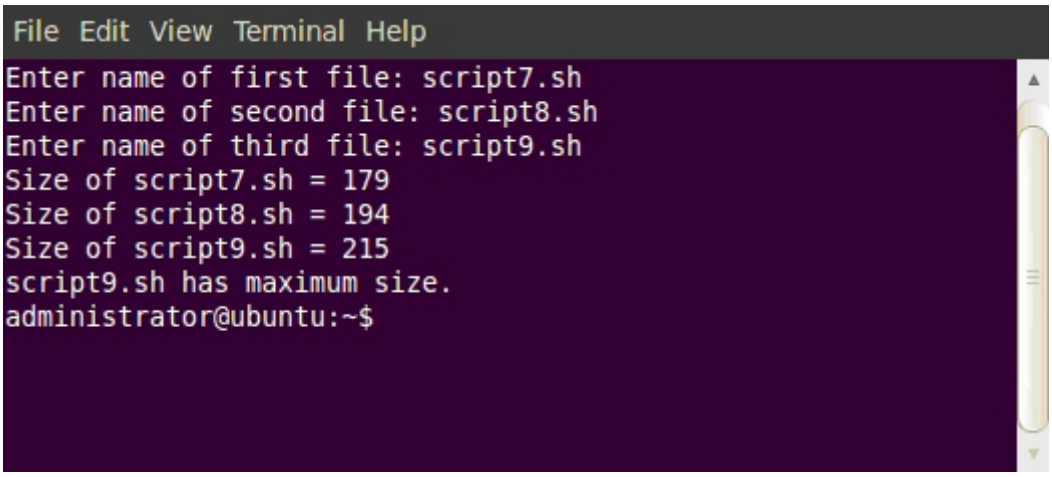


Figure 8.9 : Output of Script 17

The case statement

The if-then-elif-then-else-fi statement looks clumsy as number of comparison grows. The alternate option for checking such conditions is to use a case statement. Let us write a script that allows us to accept a choice from the user and perform different file operations based on the entered choice.

```
# Script 18: Script to perform various file and directory operations.
echo "1 - Display Current Dir "
echo "2 - Make Dir "
echo "3 - Copy a file "
echo "4 - Rename a file "
echo "5 - Delete a file "
echo "0 - Exit "
echo -n "Enter your choice [0-5] : "
read choice
case $choice in
1)
    echo $PWD
    ;;
```

```
2)
 echo -n "Enter name of the directory to be created: "
 read dname
 if [ -d $dname ]
 then
     echo "Directory with the name $dname already exists."
      exit 0
   else
     mkdir $dname
      echo "Directory $dname created successfully."
 fi
 ;;
3)
   echo -n "Enter source file name : "
   read sfile
   echo -n "Enter destination file name : "
   read dfile
   cp -u $sfile $dfile
   ;;
4)
    echo -n "Enter old file name : "
    read oldf
    echo -n "Enter new file name : "
    read newf
    mv $oldf $newf
    ;;
5)
    echo -n "Enter file name to delete : "
    read fname
    rm $fname
      ;;
```

```
0)
    exit 0
     ;;
*)
   echo "Incorrect choice exiting script."
esac
```

Save the script as *script18.sh*. Observe that for each operation that we need to perform we have written different section. When a user enters a numeric value between 0 and 5, it is assigned to the variable *choice.* The case statement extracts the value of variable *choice,* the control is transferred to the section with a matching value specified before the closing round brackets. All the statements written within that section are executed till two semicolons (;;) are encountered. Once these semicolons are encountered the control is transferred to the line after the end of the case statement. The end of case statement is specified by esac keyword. The shell then starts executing statements written after the end of case statement.

If user enters any value that does not match any of the case value specified, then the control is transferred to the section that has asterisk (*) as its value. If specified, this section allows a user to exit the script or perform additional processing after displaying an appropriate message. Figure 8.10 shows us different output of script 18.

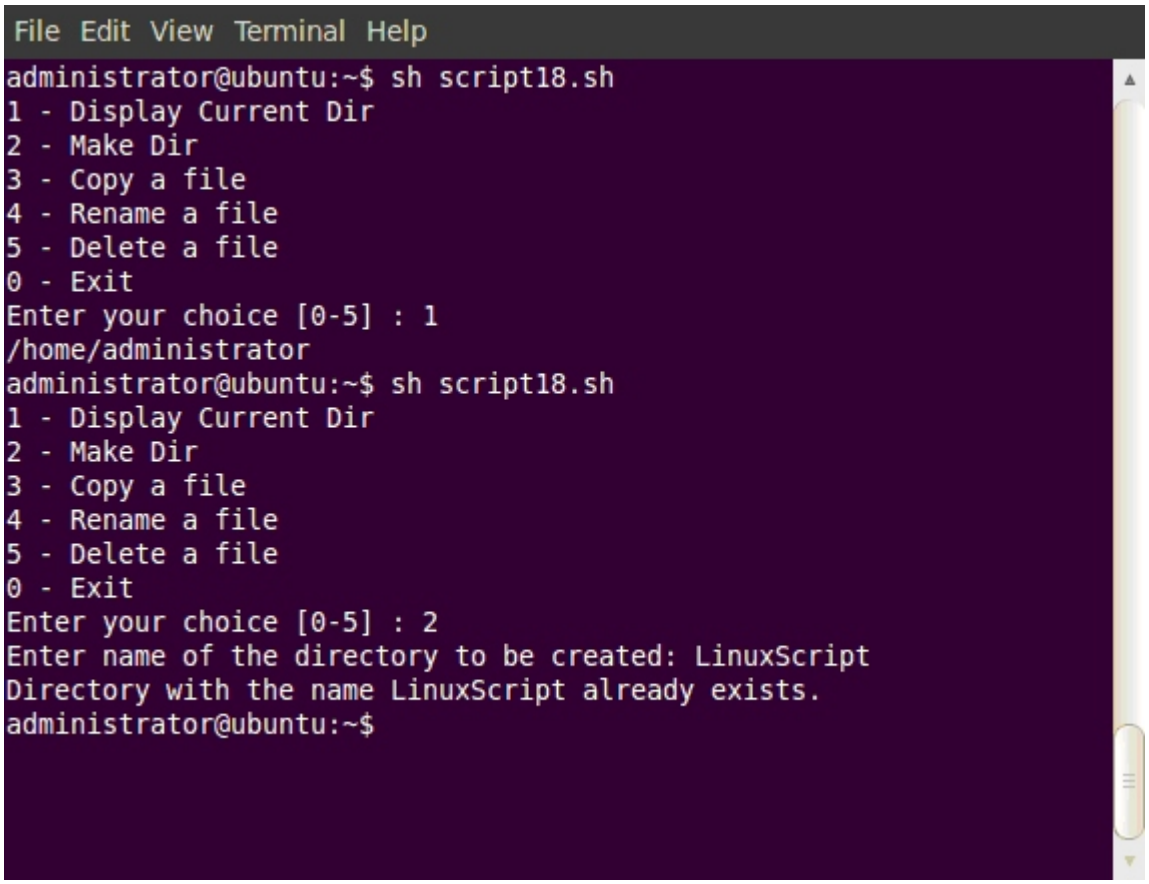


Figure 8.10 : Output of Script 18

The syntax of case statement is:

```
case variable_name in
value1)
        Command1
        Command 2
        ….
        ;;
value 2)
        Command 1
        Command 2
        ….
        ;;
        *)
Command 1
        Command 2
        ….
        ;;
esac
```

Note :

We can assign numeric, character or string values to the variable that accepts the choice. In case we assign string values then within the case it should be enclosed between single quotes. For example if we accept string *abc* then within the case statement it should be mentioned as 'abc'.

Handling Repetition

Cleaning of disk space is a normal operation that the administrator needs to perform. Let us write a simple script that assists the administrator in finding zero sized file and delete it. The script to perform the operation is given below:

```
#Script 19: Script to delete zero sized files.
echo -n "Enter directory name : "
read dname
if [ ! -d $dname ]
then
 echo Directory $dname does not exist.
```

```
else
ctr=0
for i in `find "$dname/" -type f -size 0c`
do
 rm $i
 echo File $i" : deleted"
 ctr=`expr $ctr + 1`
done
if [ $ctr -gt 0 ]
then
 echo "$ctr zero sized files have been deleted."
else
 echo "No zero sized files present in directory."
fi
fi
```

Observe that in this script we have used a statement ***for i in 'find "$dname/" -type f -size 0c'***. This statement is used to repeat some actions again and again. Figure 8.11 shows the output of script 19.

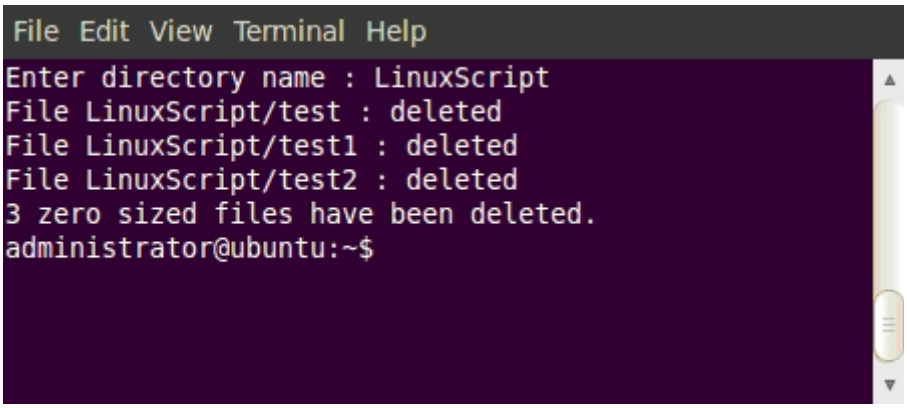


Figure 8.11 : Output of Script 19

While writing scripts for certain tasks we may require performing an action multiple times. The process of repeating the same commands number of times is known as looping. Linux provides three keywords namely *for*, *while* and *until* that can be used to perform repetitive actions.

In script 19 we have used *for* statement. The *for* loop allows us to specify a list of values in its statement. The loop is then executed for each value mentioned within the list. The general syntax of *for* statement is shown below:

```
for control-variable in value1, value2, value3…..
do
        command 1
        command 2
        command 3
done
```

Another activity that administrator regularly performs is taking backup of files. Let us say he needs to take backup of particular type of files. In such a case, taking backup of one file at a time does not make sense. Creating an exact copy at another location will also waste storage space. In such cases an administrator can use a script that first creates a backup directory in the folder where the files are located. Then the files which needs backup are copied into it. The directory is then compressed and finally moved to a new location. The script written below performs this action.

```
#Script 20: Script to backup and compress desired files from current location.
clear
dat=`date +"%d_%m_%Y"`
bdir=backup_$dat
if [ ! -d $bdir ]
then
 mkdir $bdir
else
 echo "Directory with name $bdir already exist."
 exit 0
fi
echo -n "Enter the extension of the files to backup: "
read fextn
ctr=0
for i in `ls -1 *.$fextn`
 do
  cp $i ./$bdir
  ctr=`expr $ctr + 1`
 done
```

```
if [ $ctr -gt 0 ]
then
 tar -czf $bdir.tar $bdir
 cd $bdir
 rm -r *.*
 cd ..
 rmdir $bdir
 echo "All files with extension .$fextn stored in $bdir.tar"
else
 rmdir $bdir
 echo "No files with the extension found."
fi
```

Save the script as *script20.sh*. Let us understand how the script works. Initially we have defined two variables namely *dat* and *bdir*. The *dat* variable is assigned the value of current date in the specified format. For example if the current date is 21 February 2013, then variable *dat* will be assigned value 21_02_2013. The variable *bdir* is then assigned value backup_21_02_2013. Then we check whether such a directory exists or not. If it does not exist we create this directory otherwise we exit with the message saying that the directory already exists. If we create a directory then we ask the user to enter a file extension. The script looks for the files with specified extension in the current directory and if found copies them in the backup directory. Once all files are copied, the backup directory is compressed (packed) using the tar command. The tar ***-czf $bdir.tar $bdir*** statement performs this operation. Here we create a tar file named backup_currentdate.tar. Then we empty the contents of the backup directory and delete it. In case we do not find any files with the extension specified we display appropriate message. The administrator if he wants now can move the compressed tar file to the location he desires. We can uncompress the tar file by using the command ***tar -xvf filename***.

Repetition: while statement

We can also use the *while* statement for looping. It repeats the set of commands specified between keywords *do* and *done* statements as long as the condition specified as an expression is true. Let us write a script that allows administrator to remove a specified number of files from a directory.

```
#Script 21: Script to delete specified number of files from a directory.
clear
echo -n "Enter the name of directory from where you want to delete: "
```

```
read dname
if [ -d $dname ]
then
cd $dname
echo -n "Enter the number of files you want to delete: "
read fdel
ctr=1
while [ $ctr -le $fdel ]
do
 echo -n  "Enter the name of the file to be deleted: "
 read fname
 if [ -f $fname ]
  then
   rm $fname
   echo "$fname deleted successfully."
  else
   echo "File with name $fname not found."
 fi
 ctr=`expr $ctr + 1`
 done
else
echo "Directory $dname does not exist."
fi
cd ..
```

Save the script as *script21.sh.* Let us understand how the script works. Initially the user is prompted to enter a directory name. The *dname* variable is assigned this value. Then we check whether such a directory exists or not. If it exists we change to that directory and ask user the number of files he wants to delete. Then we start a while loop that finds the files to be deleted. If the file is found we delete it else we display a message indicating file not found. The loop is continued till the value of variable ctr is less than or equal to the number of files specified by the user. Once the operation is over we go back to the parent directory. The syntax of while loop is shown below:

```
while [ test_condition ]
do
        command or set of commands
done
```

Repetition: until statement

Another method to execute repetitive statements is to make use of the *until* statement. The until loop is similar to the while loop. However, the *until* loop executes till the condition is false and the while loop executes till the condition is true.

So far, we have seen how we can use decision-making and looping constructs to write shell scripts. Script 19 is an example of shell script which uses some of the constructs discussed above. It is a menu driven script demonstrating until-loop, to display list of files in a current directory, changing password, displaying current date and time and searching a word from a file.

```
#Script 22: Script to perform operations till user decides to exit.
choice=y
until [ $choice = n ]
do
clear
echo "....................................................................."
echo "       Choose an option from menu given below          "
echo "....................................................................."
echo "a: List of files and directories in a current directory."
echo "b: Display current working directory"
echo "c: Display current date and time"
echo "d: Searching a word from file"
echo "e: Exit"
echo "   "
echo "....................................................................."
echo -n "Enter your choice [a-e]: "
read ch
case $ch in
a)
```

```
ls -l
;;
b)
echo "You are working in `pwd`"
;;
c)
echo "Current date and time is is `date`"
;;
d)
echo -n "Enter the word to be searched: "
read word
echo -n "Enter the file name from which the word is to be searched: "
read file
if [ -f $file ]
then
grep $word $file
else
echo -n "File with name $file does not exist."
fi
;;
e)
exit
;;
*)
echo "Incorrect choice, try again.."
;;
esac
echo -n "Do you want to continue? : "
read choice
done
```

Save the script as *script22.sh*. When user executes the script he will be shown a menu and asked to enter a choice. Depending on the choice entered an action from the case will be executed. Enter different choice each time and see the output. The script will keep on executing till user enters *e* as a choice in which case the exit statement within the case is executed or he enters *n* when the question "Do you want to continue?" is asked. Figure 8.12 shows the output of script 22.

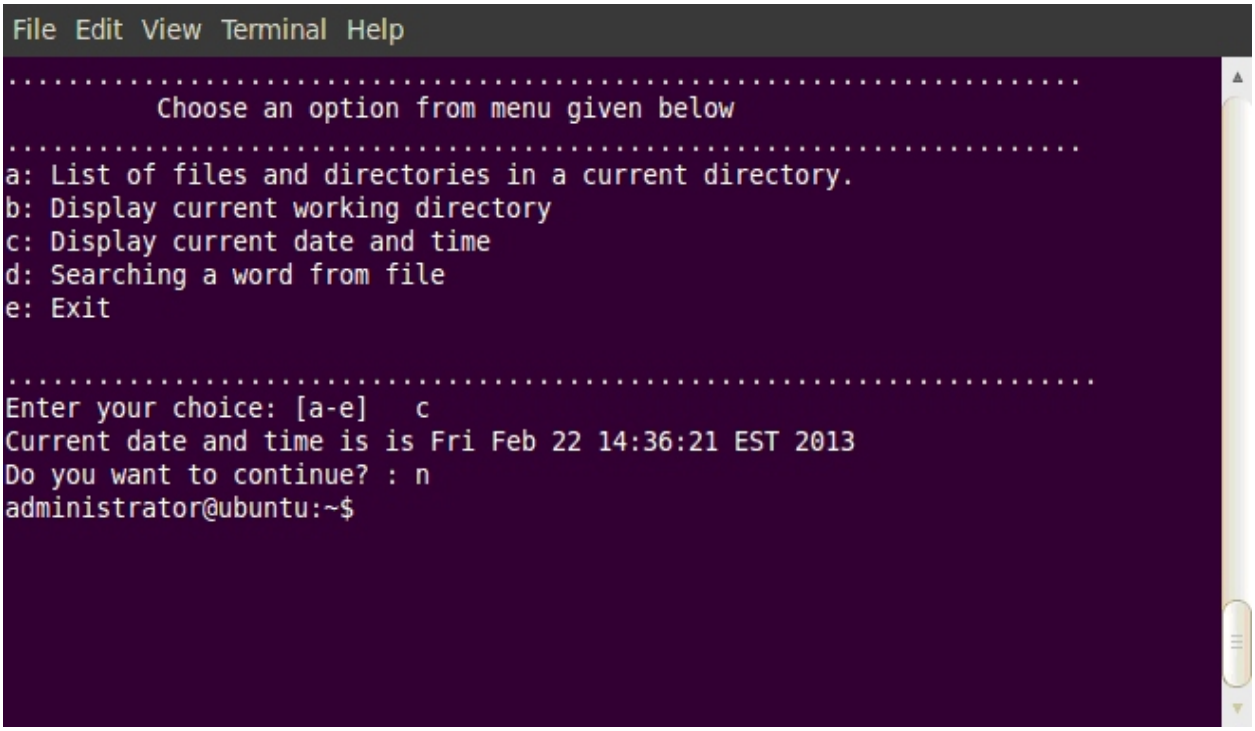


Figure 8.12 : Output of Script 22

Functions in script

Linux shell script also provides us the feature of creating functions. Functions are small subscripts within a shell script. They are used make the scripting more modular. Using functions we can improve the overall readability of the script. The function used in shell script do not return a value, they return a status code. Let us see one script that assists the user in finding out how many files were created on current date or when a particular file was last modified.

```
#Script 23: Script to show use of function.
file_today(){
  cur_date=`date +'%Y-%m-%d'`
   cnt=`ls -l tr | grep "$cur_date" | wc -l`
   echo "Current date is : "$cur_date
   echo "No. of files created today : "$cnt
}
```

```
modified_today(){
if [ -f "$1" ]
then
stat -c %y "$1"
else
echo ""$1" does not exist"
fi
}
choice=y
until [ $choice = n ]
do
clear
echo "..........."
echo "       Choose an option from menu given below          "
echo "..........."
echo "a: List of files created today."
echo "b: Display last file modification date."
echo "c: Exit"
echo "   "
echo "..........."
echo -n "Enter your choice [a-c]: "
read ch
case $ch in
a)
file_today
;;
b)
echo -n "Enter a file name: "
read fname
modified_today $fname
;;
```

```
c)
exit
;;
*)
echo "Incorrect choice, try again.."
;;
esac
echo -n "Do you want to continue? : "
read choice
done
```

Save the script as *script23.sh*. Observe that in script 23 we have used two functions namely file_today() and modified_today(). The opening and closing parenthesis after a variable name indicates that it is a function. When user enters *a,* function file_today() that contains code for finding the files created on a current date is called and executed. Similarly when user enters *b* he is prompted to enter a file name. This name is then passed to function modified_today() that checks if the files exists or not. If the file exists its last modification date is displayed otherwise appropriate message is displayed. Figure 8.13 shows the output of script 23.

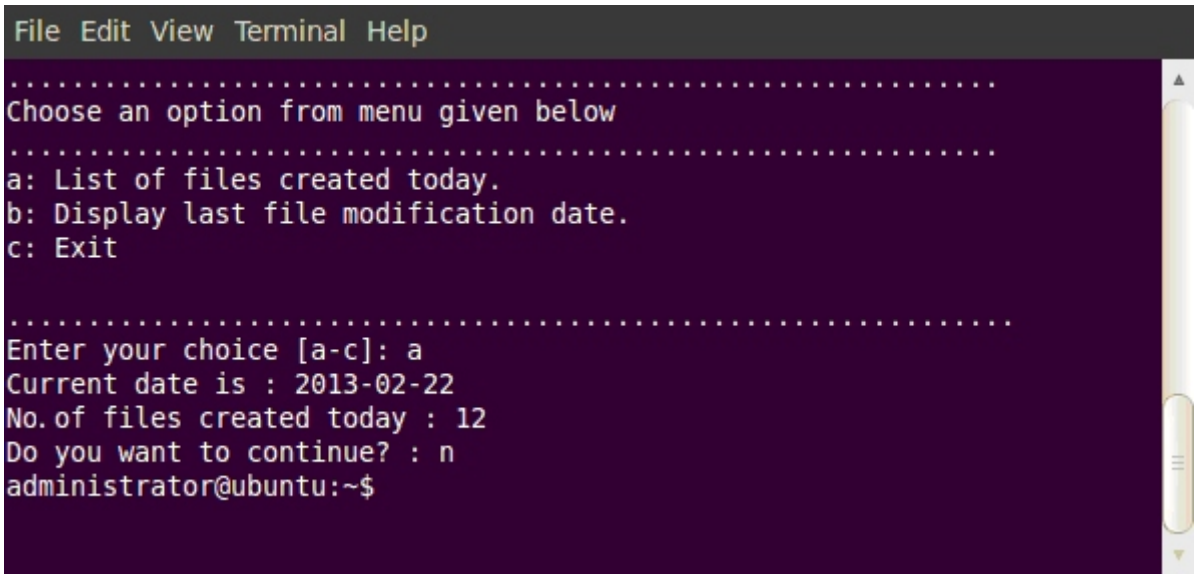


Figure 8.13 : Output of Script 23

Summary

In this chapter we have seen how a shell script can be used for several tasks of system administration. We saw how decision making and looping constructs can be used in shell scripts. We also saw how we can write a shell script in the form of functions. The shell script thus offers the facility to combine the power of various inbuilt commands. This makes it almost equivalent to a higher level programming language.

EXERCISE

1. Explain conditional execution in shell script with proper example.
2. Explain case statement of shell script with the option of pattern list.
3. Explain the use of while loop.
4. Explain the use of until loop.
5. **Choose the most appropriate option from those given below :**

(1) Which of the following command is used to set the file permission as executable?

(a) grep (b) chmod

(c) ls (d) x

(2) Which of the following symbol is used to break the flow of control in the case statement?

(a) ** (b) ;;

(c) ++ (d) >>

(3) Which of the following keyword specifies the end of the case statement?

(a) end-case (b) end case

(c) esac (d) stop-case

(4) Which loop iterates till the condition evaluates to true?

(a) while (b) until

(c) for (d) case

(5) Which of the following allows us to specify a list of values in its statement?

(a) while (b) until

(c) for (d) if

(6) In case structure, which of the following character denotes default case?

(a) * (b) +

(c) d (d) All of the above

(7) Which of the following statement is true for testing whether the file is read only or not?

(a) test –read filename (b) check –read filename

(c) test –r filename (d) check –r filename

(8) Which of the following indicate the end of if condition in shell script?

(a) end-if (b) fi

(c) } (closing curly bracket) (d) It does not have any end statement

(9) Which of the following operator is used for less than comparison in Linux?

(a) < (b) lessthan

(c) lt (d) -lt

(10) Which of the following can be used in place of square brackets used in if condition?

(a) Curly braces (b) test command

(c) check command (d) All of the above

Laboratory Exercises

Write a shell script to perform the following operations:

(a) To accept two file names from user. The script should check whether the two file's contents are same or not. If they are same then second file should be deleted.

(b) To count and report the number of entries present in each subdirectory mentioned in the path, which is supplied as a command-line argument.

(c) To list name and size of all files in a directory whose size is exceeding 1000 bytes (directory name is to be supplied as an argument to the shell script).

(d) To rename a file.

(e) To convert all file contents to lower case or upper case as specified by user.

(f) To find out available shells in your system and in which shell are you working.

(g) To find out the file that has minimum size from the current directory.