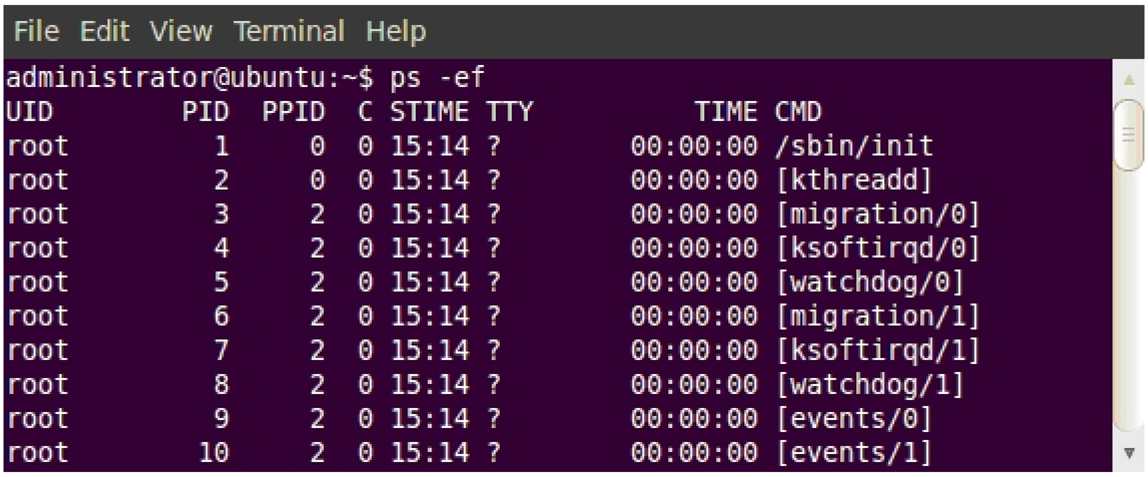
8 Advanced Scripting

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.



```
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.

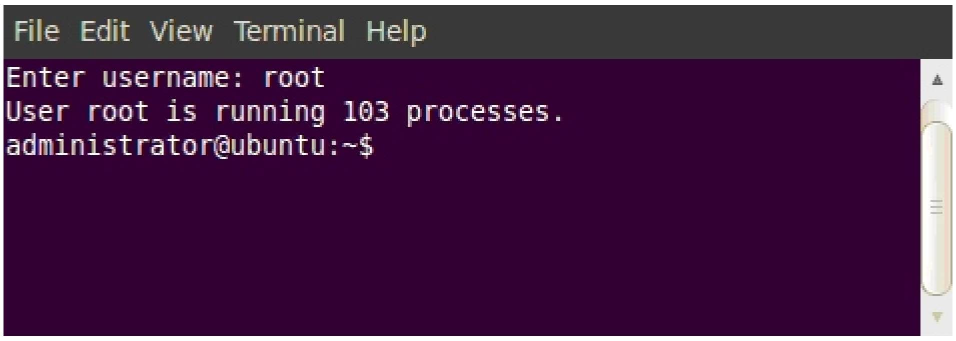
Column Name	Description
UID	Name or number of the user who owns the process.
PID	A unique numeric process identifier assigned to each process.
PPID	Identifies the parent process id, the process that created the current process.
STIME	The start time for the current process.
TTY	Identifies the terminal 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.



```
Enter username: root
User root is running 103 processes.
administrator@ubuntu:~$
```

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

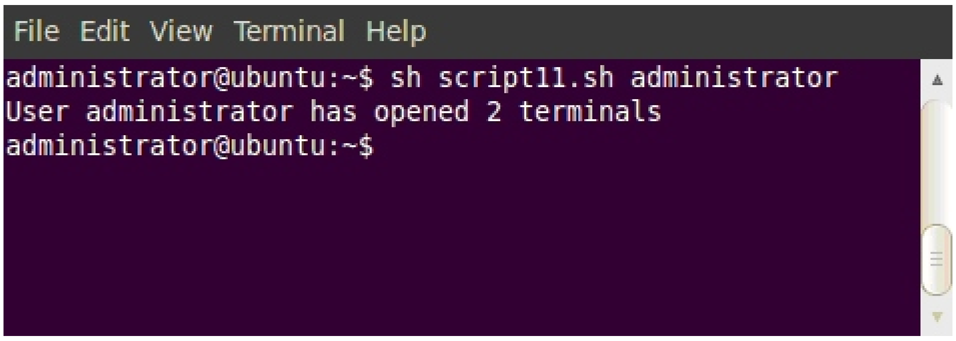
Then the process with PID=101 will be forcibly removed from the memory. Let us have a look at script similar to script10.sh that accepts user name as a command line argument and tells us how many terminals that user is using. The code of the script is given in the box below:

```
#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.



```
administrator@ubuntu:~$ sh script11.sh administrator
User administrator has opened 2 terminals
administrator@ubuntu:~$
```

Figure 8.3 : Output of Script 11

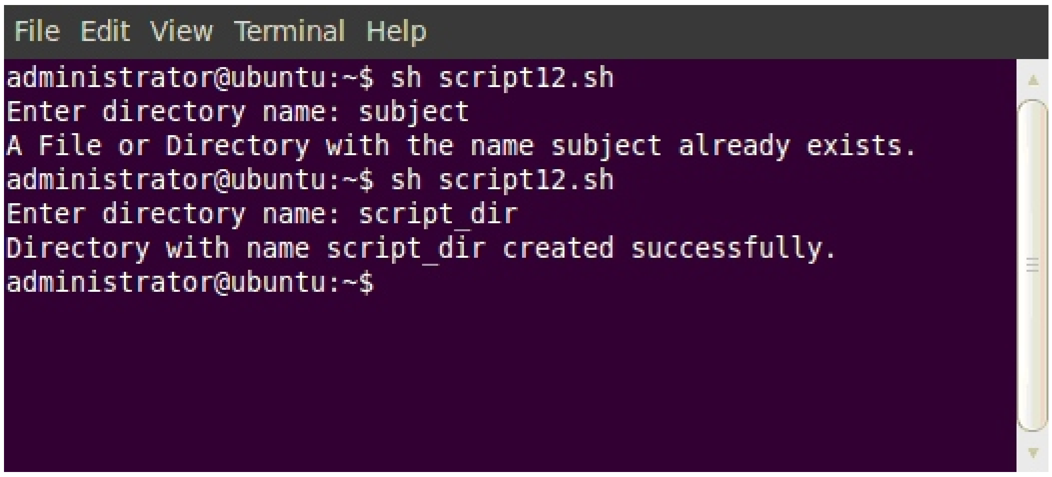
Let us try to understand the working of this script. In the first statement after comment we have combined four commands namely who, cut, grep and wc using pipe. The *who* command displays list of all users that have logged into the system. Its output is then given to the cut command. The cut command 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.



```
administrator@ubuntu:~$ sh script12.sh
Enter directory name: subject
A File or Directory with the name subject already exists.
administrator@ubuntu:~$ sh script12.sh
Enter directory name: script_dir
Directory with name script_dir created successfully.
administrator@ubuntu:~$
```

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

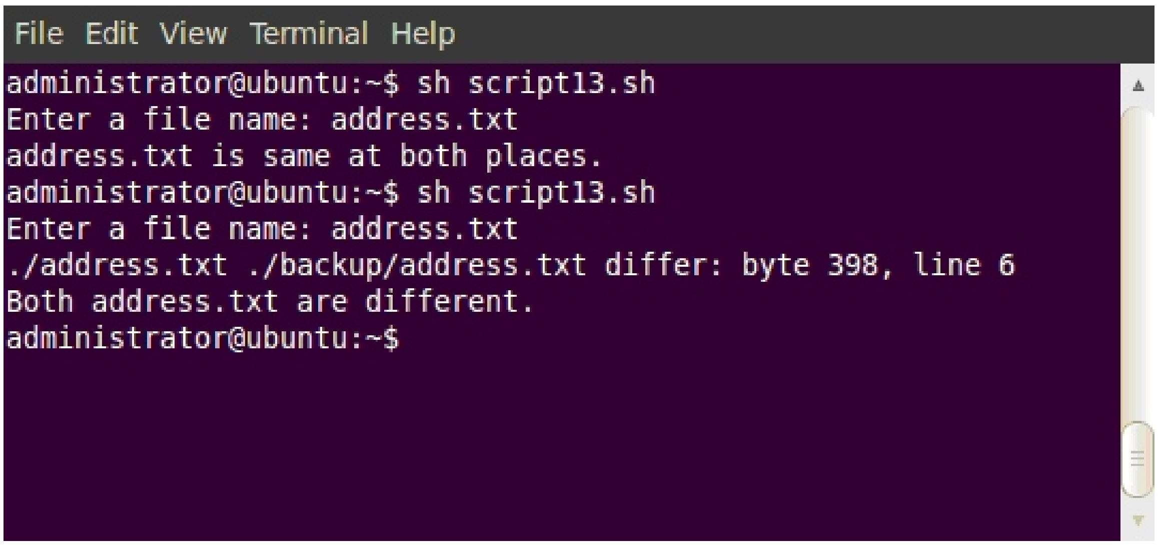
if-then-elif-then-else-fi

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.



```
administrator@ubuntu:~$ sh script13.sh
Enter a file name: address.txt
address.txt is same at both places.
administrator@ubuntu:~$ sh script13.sh
Enter a file name: address.txt
./address.txt ./backup/address.txt differ: byte 398, line 6
Both address.txt are different.
administrator@ubuntu:~$
```

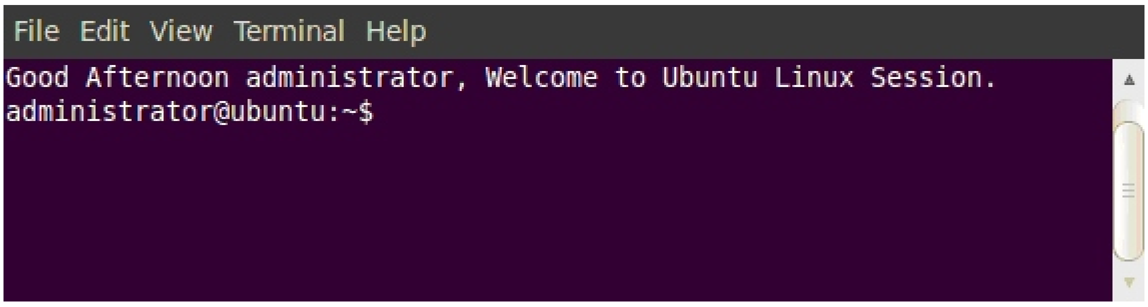
Figure 8.5 : Output of Script 13

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.



```
Good Afternoon administrator, Welcome to Ubuntu Linux Session.
administrator@ubuntu:~$
```

Figure 8.6 : Output of Script 14

The test command

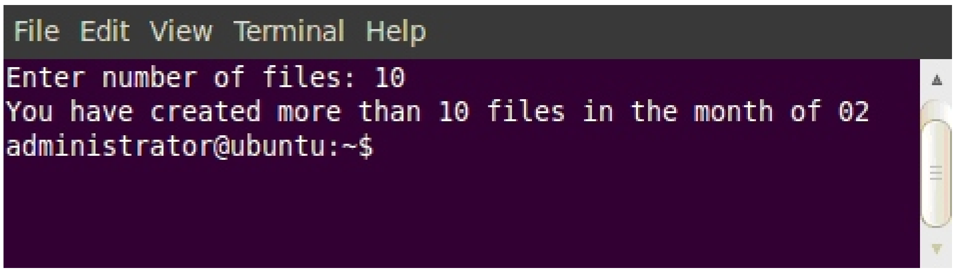
It is possible to use different forms of if statements. Linux also provides test command which can be used in place of square brackets used in previous scripts. Let us write script to check whether a user has created more than some specified files in a given month or not.

```
#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



```
Enter number of files: 10
You have created more than 10 files in the month of 02
administrator@ubuntu:~$
```

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

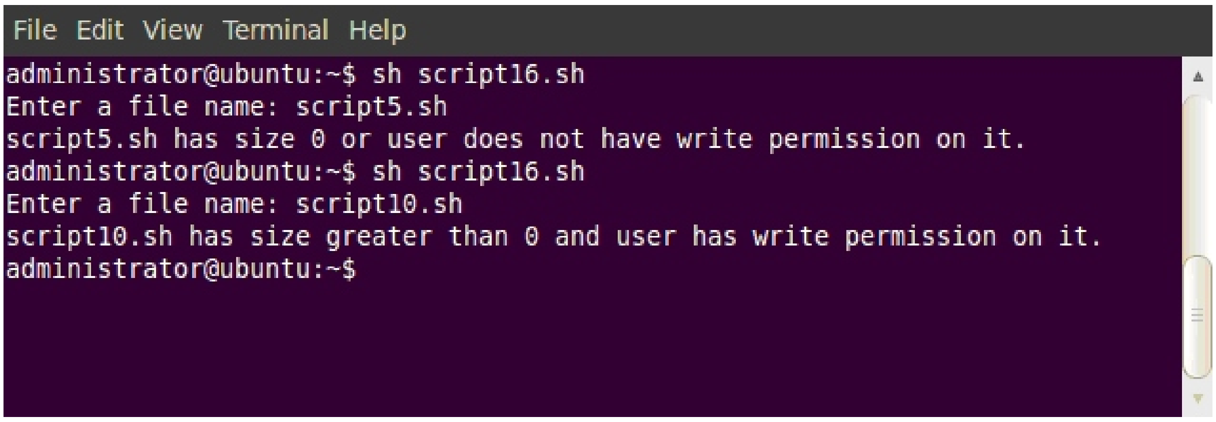
Many times administrator needs to find whether a specified file has size equal to zero or not. He can then perform maintenance operations like delete the file in case its size is zero. He may additionally need to find whether write permissions on the file is set or not. Let us write a script that allows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



```
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:~$
```

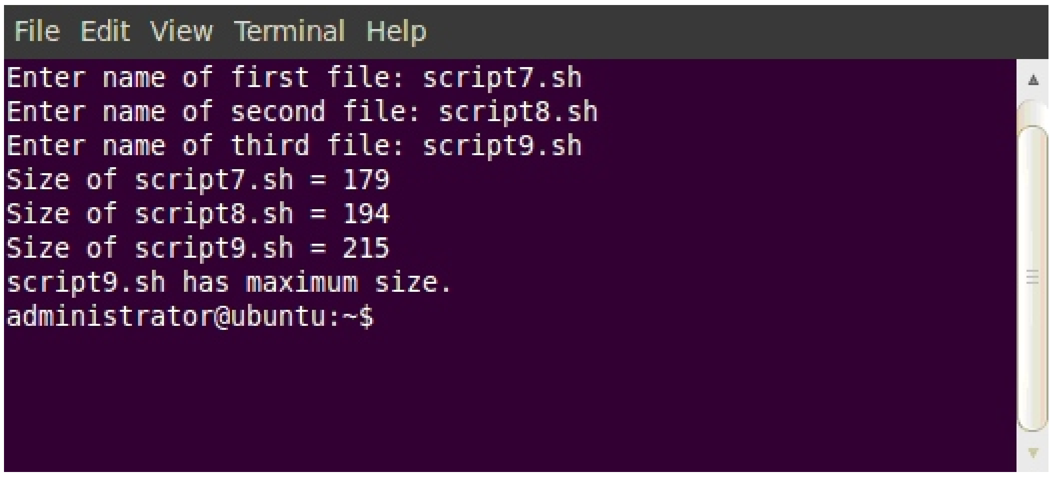
Figure 8.8 : Output of Script 16

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.



```
Enter name of first file: script7.sh
Enter name of second file: script8.sh
Enter name of third file: script9.sh
Size of script7.sh = 179
Size of script8.sh = 194
Size of script9.sh = 215
script9.sh has maximum size.
administrator@ubuntu:~$
```

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.

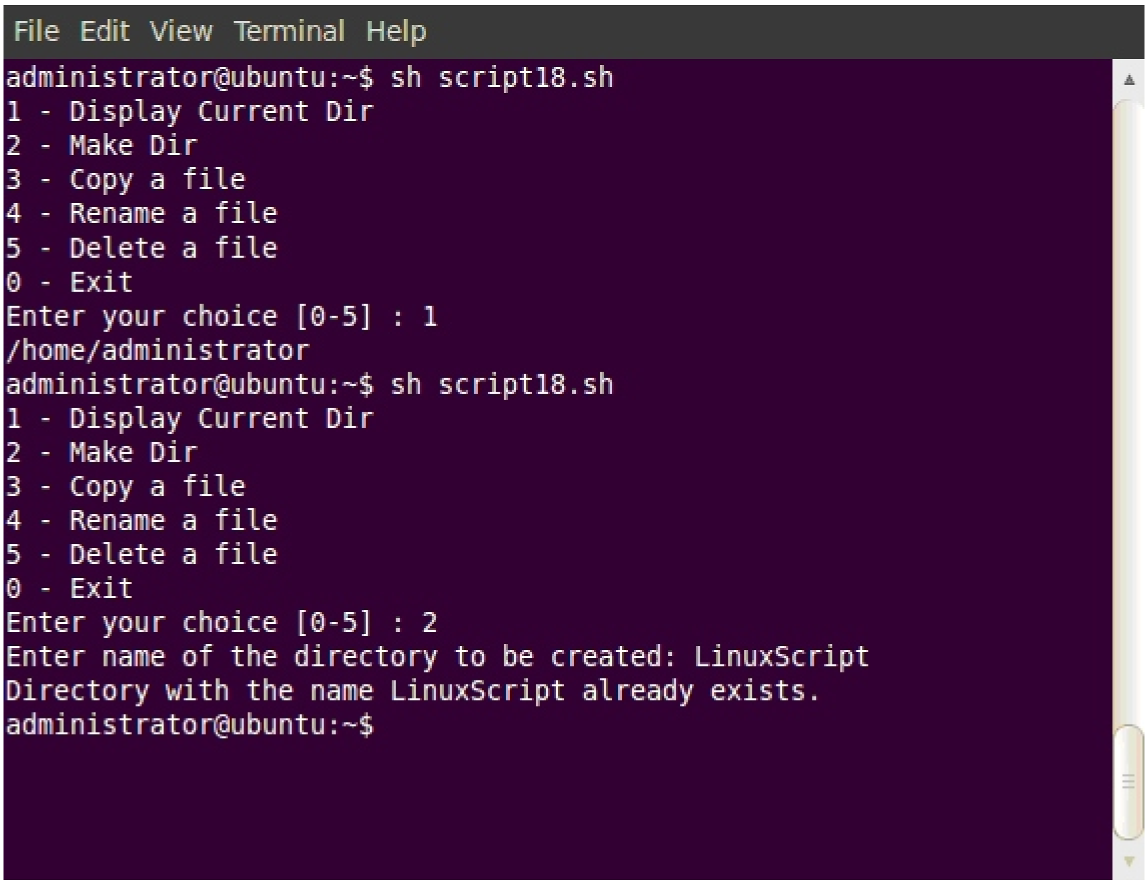
```
# Script 18: Script to perform various file and directory operations.
echo "1 - Display Current Dir "
echo "2 - Make Dir "
echo "3 - Copy a file "
echo "4 - Rename a file "
echo "5 - Delete a file "
echo "0 - Exit "
echo -n "Enter your choice [0-5] : "
read choice
case $choice in
1)
    echo $PWD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

```
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.



```
administrator@ubuntu:~$ sh script18.sh
1 - Display Current Dir
2 - Make Dir
3 - Copy a file
4 - Rename a file
5 - Delete a file
0 - Exit
Enter your choice [0-5] : 1
/home/administrator
administrator@ubuntu:~$ sh script18.sh
1 - Display Current Dir
2 - Make Dir
3 - Copy a file
4 - Rename a file
5 - Delete a file
0 - Exit
Enter your choice [0-5] : 2
Enter name of the directory to be created: LinuxScript
Directory with the name LinuxScript already exists.
administrator@ubuntu:~$
```

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’.

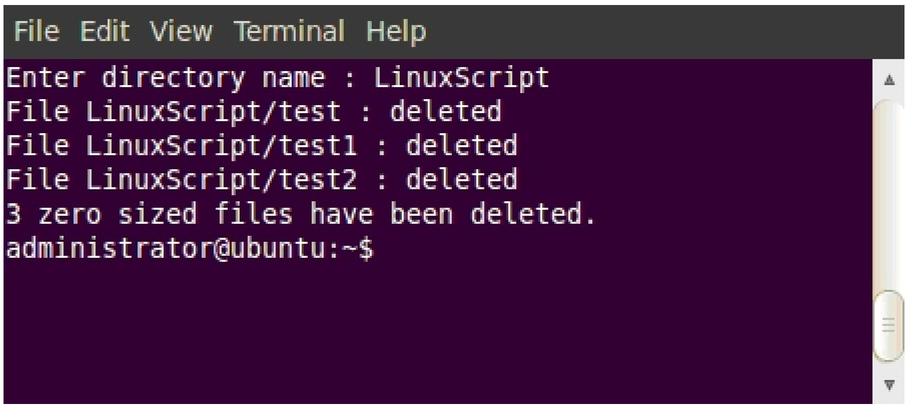
Handling Repetition

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.



```
Enter directory name : LinuxScript
File LinuxScript/test : deleted
File LinuxScript/test1 : deleted
File LinuxScript/test2 : deleted
3 zero sized files have been deleted.
administrator@ubuntu:~$
```

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

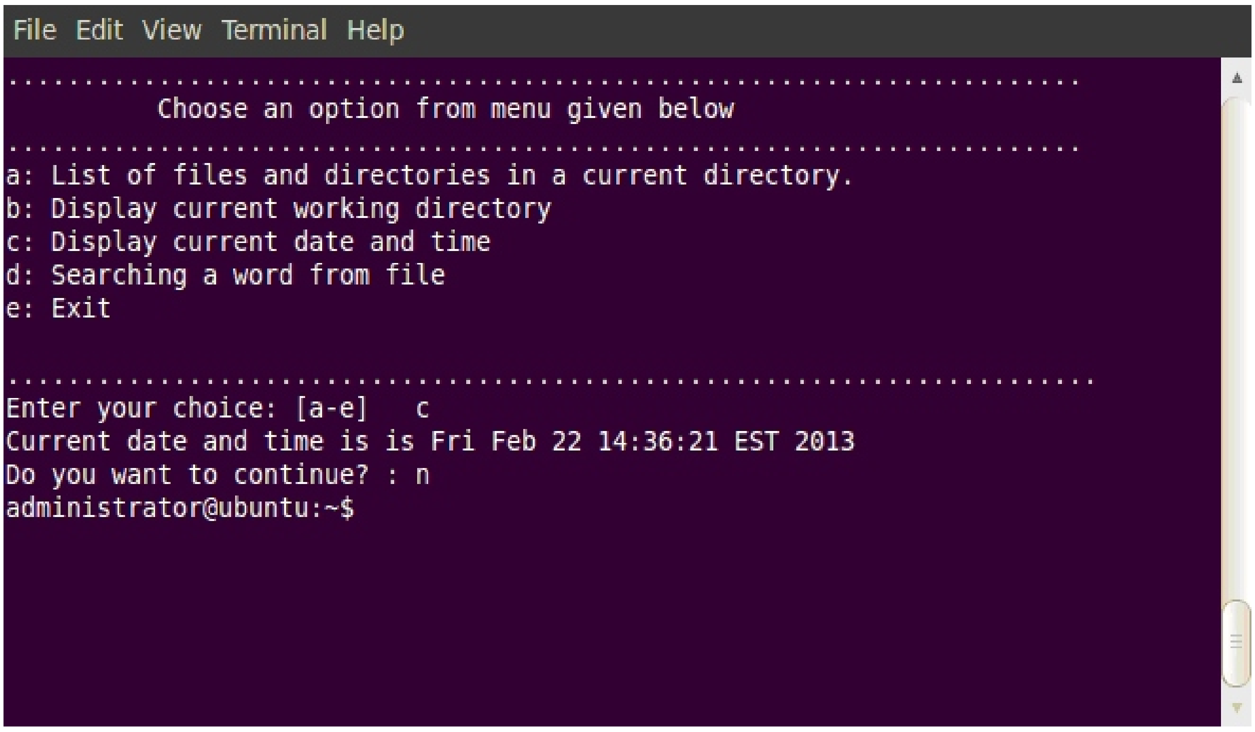
Another method to execute repetitive statements is to make use of the *until* statement. The until loop is similar to the while loop. However, the *until* loop executes till the condition is false and the while loop executes till the condition is true.

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.



```
.....................................................................
         Choose an option from menu given below
.....................................................................
a: List of files and directories in a current directory.
b: Display current working directory
c: Display current date and time
d: Searching a word from file
e: Exit

......................................................................
Enter your choice: [a-e]   c
Current date and time is is Fri Feb 22 14:36:21 EST 2013
Do you want to continue? : n
administrator@ubuntu:~$
```

Figure 8.12 : Output of Script 22

Functions in script

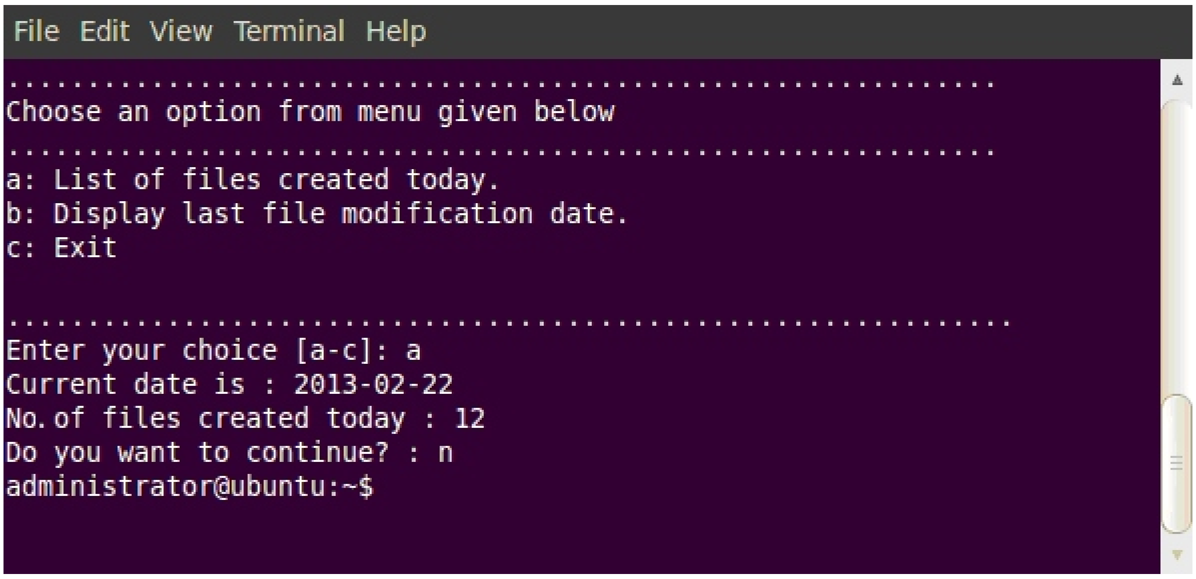
Linux shell script also provides us the feature of creating functions. Functions are small subscripts within a shell script. They are used make the scripting more modular. Using functions we can improve the overall readability of the script. The function used in shell script do not return a value, they 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.



```
..............................................................
Choose an option from menu given below
..............................................................
a: List of files created today.
b: Display last file modification date.
c: Exit

...............................................................
Enter your choice [a-c]: a
Current date is : 2013-02-22
No.of files created today : 12
Do you want to continue? : n
administrator@ubuntu:~$
```

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.