20CS55 – SHELL SCRIPTING

LAB MANUAL

B.Tech – II Semester

1. Vision of the Department

The Computer Science & Engineering aims at providing a continuously stimulating educational environment to its students to attain their professional goals and meet global challenges.

2. Mission of the Department

- **DM1:** To develop a strong theoretical and practical background across the computer science discipline with an emphasis on problem solving.
- **DM2:** To inculcate professional behavior with strong ethical values, leadership qualities, innovative thinking and analytical abilities into the student.
- **DM3:** Expose the students to cutting-edge technologies which enhance their employability and knowledge.
- **DM4:** Facilitate the faculty to keep track of latest developments in their research areas and encourage the faculty to foster healthy interaction with industry.

3. Program Educational Objectives (PEOs)

- **PEO1:** Pursue higher education, entrepreneurship and research to compete at global level.
- **PEO2:** Design and develop products innovatively in computer science and engineering and in other allied fields.
- **PEO3:** Function effectively as individuals and as members of a team in the conduct of interdisciplinary projects; and even at all the levels with ethics and necessary attitude.
- **PEO4:** Serve ever-changing needs of society with a pragmatic perception.

4. PROGRAMME OUTCOMES (POs):

PO 1	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO 2	**Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO 3	**Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO 4	**Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO 5	**Modern tool usage**: Create, select, and apply appropriate techniques, resources

	and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO 6	**The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO 7	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO 8	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO 9	**Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO 10	**Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO 11	**Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO 12	**Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

5. PROGRAMME SPECIFIC OUTCOMES (PSOs):

PSO 1	The ability to apply Software Engineering practices and strategies in software project development using open-source programming environment for the success oforganization.
PSO 2	The ability to design and develop computer programs in networking, web applications andIoT as per the society needs.
PSO 3	To inculcate an ability to analyze, design and implement database applications.

6. Pre-requisites: Nil

7. Course Educational Objectives (CEOs):

The main objective of this course is to familiarize with the Unix/Linux command line and running simple commands and concept of environment variables and with the simple use of environment variables.

8. Course Outcomes (COs):

At the end of the course, the student will be able to:

CO 1: Understand the basic unix/linux commands. (Understand - L2)

CO 2: Learn importance of shell scripting. (Understand - L2)

CO 3: Apply shell programming to various files. (Apply - L3)

CO 4: Improve individual / teamwork skills, communication & report writing skills with ethical values.

9. Course Articulation Matrix:

Course Code	COs	Programme Outcomes												PSOs		
		1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
L184	CO1	3	2	2	1	-	-	-	-	-	-	-	2	3	-	-
	CO2	3	2	3	1	-	-	-	-	-	-	-	2	3	-	-
	CO3	3	2	3	1	-	-	-	-	-	-	-	2	3	-	-
	CO4	-	-	-	-	-	-	-	2	2	2	-	-	-	-	-
1 = Slight (Low) 2 = Moderate (Medium) 3-Substantial(High)																

10. List of Programs:

Module – I: Basic Linux Commands Study of Unix/Linux general purpose utility command list obtained from (man, who, cat, cd, cp, ps, ls, mv, rm, mkdir, rmdir, echo, more, date, time, kill, history, chmod, chown, finger, pwd, cal, logout, shutdown) commands, study of vi editor, study of Unix/Linux file system

Module – II: Introduction to Shell Introduction to Shell, Shell responsibilities, running a shell script. Variables, passing arguments, Basic Operators, Basic String Operations, Decision Making, Loops, Arrays, Arrays – Comparison, Shell functions.

Module – III: Advanced Shell Special Variables, Bash trap command, File Testing, Input Parameter Parsing, Pipelines, Process Substitution, Regular Expressions, Special Commands: sed, awk, grep, sort.

Example Programs:

1. Use of Basic UNIX Shell Commands: ls, mkdir, rmdir, cd, cat, touch, file, wc, sort, cut, grep, dd, dfspace, du, ulimit

2. Commands related to inode, I/O redirection and piping, process control commands, mails.

3. Shell Programming: Shell script exercises based on following:

(i) Interactive shell scripts

(ii) Positional parameters

(iii) Arithmetic

(iv) if-then-fi, if-then- else-fi, nested if-else

(v) Logical operators

(vi) else + if equals elif, case structure

(vii) while, until, for loops, use of break

4. Write a shell script to create a file. Follow the instructions

(i) Input a page profile to yourself, copy it into other existing file

(ii) Start printing file at certain line

(iii) Print all the difference between two file, copy the two files.

(iv) Print lines matching certain word pattern.

5. Write shell script for-

(i) Showing the count of users logged in,

ii) Printing Column list of files in your home directory

(iii) Listing your job with below normal priority

(iv) Continue running your job after logging out.

6. Write a shell script to change data format. Show the time taken in execution of this script.

7. Write a shell script to print files names in a directory showing date of creation & serial number of the file.

8. Write a shell script to count lines, words, and characters in its input (do not use wc).

Reference books:

1. Learning the bash Shell, 3rd Edition by Cameron Newham, Publisher(s): O'Reilly Media, Inc., ISBN: 9780596009656

2. UNIX and Shell Programming by Behrouz A. Forouzan, Richard F. GilbergPublisher: Thomson Press (India) Ltd, ISBN: 9788131503256, 9788131503256

3. Shell Scripting: Expert Recipes for Linux, Bash, and More by Steve Parker

GUIDELINES TO STUDENTS

How to Run Shell Scripts:

There are two ways you can execute your shell scripts. Once you have created a script file:

Method 1:

Pass the file as an argument to the shell that you want to interpret your script.

Step 1: Create the script using vi, ex or ed

For example, the script file show has the following lines

```
echo Here is the date and time
date
```

Step 2: To run the script, pass the filename as an argument to the sh (shell)

```
$ sh show
Here is the date and time
Sat jun 03 13:40:15 PST 2006
```

Method 2:

Make your script executable using the chmod command. When we create a file, by default it is created with read and write permission turned on and execute permission turned off. A file can be made executable using chmod.

Step 1: create the script using vi, ex or ed

For example, the script file show has the following lines

```
echo Here is the date and time
date
```

Step 2: Make the file executable

```
$ chmod u+x script_file
$ chmod u+x show
```

Step 3: To run the script, just type the filename

```
$ show
Here is the date and time
Sat jun 03 13:40:15 PST 2006
```

Man:

- **man command** in Linux is used to display the user manual of any command that we can run on the terminal.
- It provides a detailed view of the command which includes NAME, SYNOPSIS, DESCRIPTION, OPTIONS, EXIT STATUS, RETURN VALUES, ERRORS, FILES, VERSIONS, EXAMPLES, AUTHORS and SEE ALSO.

Every manual is divided into the following sections:

- Executable programs or shell commands
- System calls (functions provided by the kernel)
- Library calls (functions within program libraries
- Special files (usually found in /dev)
- File formats and conventions eg /etc/passwd
- Miscellaneous (including macro packages and conventions)
- System administration commands (usually only for root)
- Kernel routines [Non standard]

Syntax : $man COMMAND NAME

Ex: $man ls

Echo command

- **echo command** in linux is used to display line of text/string that are passed as an argument .
- This is a built in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

Syntax: $echo [option] [string]

Examples: $echo "this is Linux operating system"

this is Linux operating system

Options of echo command

Note: -e enables the interpretation of backslash escapes

1. \b: it removes all the spaces in between the text

Example: $echo -e "**Linux** \b**operating** \b**system**"

Linuxoperatingsystem

2. \c: suppress trailing new line with backspace interpreter '-e'to continue without emitting new line.

Example: $echo -e "Linux \coperating system"
Linux

3. \n : this option creates new line from where it is used.
Example: $ echo -e "Linux \noperating \nsystem"
Linux
operating
system

4. \t : this option is used to create horizontal tab spaces.
Example: $echo -e "Linux \toperating \tsystem"
Linux operating system

5. \r : carriage return with backspace interpretor '-e' to have specified carriage return in output.
Example: $echo -e " Linux \roperating system"
operating system

Note: In the above example, text before \r is not printed

6. \v : this option is used to create vertical tab spaces.
Example: $echo -e " Linux \voperating \vsystem"
Linux
operating
System
7. echo * : this command will print all files/folders, similar to ls command .
Example: $echo *

8. -n : this option is used to omit echoing trailing newline .
Example: $echo -n "Linux operating system"
Linux operating system$

Script command:

- script command in Linux is used to make typescript or record all the terminal activities. After executing the script command it starts recording everything printed on the screen including the inputs and outputs until exit.
- script is mostly used when we want to capture the output of a command or a set of command while installing a program or the logs generated on the terminal while compiling an opensource codes, etc.
- script command uses two files i.e. one for the terminal output and other for the timing information.

Syntax: $script [options] [filename]

Examples:

1) $script filename.txt
In order to stop the typescript, we just need to execute exit command and script will stop the capturing process.

2) $ exit

-a, –append: when we want to append the output, retaining the prior content of the file.

3) $script –a filename.txt

ls command:

- By using ls command we can list the File attributes.
- ls command is also used to obtain the list of all filenames in the current directory.

Syntax: $ls

The output of the ls command will display the filenames in ASCII Collating sequence i.e. Numbers first, uppercase and lowercase is the sequence.

Options:

$ls –x

Displays output in Multiple columns (vertical by default it is horizontal)

$ls –a

Shows all filenames beginning with a dot includes . and .. files.

$ls –l

Long listing in ASCII collating sequence showing seven attributes of a file.

$ls –F

Marks Executable files with ***** and directories with / and symbolic links with **@**

$ls –r

Sorts filenames in reverse order

$ls –t

Sorts filenames by last modification time

$ls –u

Sorts filenames by last access time

$ls –lu

Long listing with last access time

$ls –i

Display inode number for all files

Pwd command:

- pwd stands for Print Working Directory.
- It prints the path of the working directory, starting from the root.
- pwd is shell built-in command(pwd)

Syntax: $pwd

Passwd command

- passwd command in Linux is used to change the user account passwords.
- The root user reserves the privilege to change the password for any user on the system, while a normal user can only change the account password for his or her own account.

Syntax: $passwd [options] [username]

Example:
linux@labvm:~$ passwd
Changing password for linux.
Current password:
New password:
Retype new password:
passwd: password updated successfully

who command:

- who command is used to get information about currently logged in user on to system.
- who command is used to find out the following information :
  - Time of last system boot
  - Current run level of the system
  - List of logged in users and more.

Syntax: $who [options] [filename]

Examples:

1. The who command displays the following information for each user currently logged in to the system if no option is provided :

- Login name of the users
- Terminal line numbers
- Login time of the users in to system
- Remote host name of the user

linux@labvm:~$ who

linux tty7 2021-04-26 06:58 (:0)

2. To display host name and user associated with standard input such as keyboard

linux@labvm:~$ who -m -H

NAME LINE TIME COMMENT

3. To show all active processes which are spawned by INIT process

linux@labvm:~$ who -p -H

NAME LINE TIME PID COMMENT

4. To show list of users logged in to system

linux@labvm:~$ who -u

linux tty7 2021-04-26 06:58 old 1254 (:0)

5. To show time of the system when it booted last time

linux **@labvm:~$ who -b -H**

NAME LINE TIME PID COMMENT

system boot 2021-04-26 12:25

6. To count number of users logged on to system

linux@labvm:~$ who -q -H
devasc
users=1

7. To display all details of current logged in user

linux@labvm:~$ who -a

system boot 2021-04-26 12:25
run-level 5 2021-04-26 06:57
LOGIN tty1 2021-04-26 06:57 1153 id=tty1
devasc + tty7 2021-04-26 06:58 old 1254 (:0)

8. To display system's username

linux@labvm:~$ whoami

linux

9. To display list of users and their activities

devasc@labvm:~$ w

07:27:17 up 31 min, 1 user, load average: 0.05, 0.10, 0.43
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
devasc tty7 :0 06:58 17:49 53.69s 0.93s mate-session

uname command

- uname' displays the information about the system.

Syntax: $uname [OPTION]

Example: **devasc@labvm:~$ uname**
Linux

Options with examples:

-a option: It prints all the system information in the following order:

Kernel name, network node hostname, kernel release date, kernel version, machine hardware name, hardware platform, operating system

linux@labvm:~$ uname -a

Linux labvm 5.4.0-37-generic #41-Ubuntu SMP Wed Jun 3 18:57:02 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux

-s option: It prints the kernel name.

devasc@labvm:~$ uname -s

Linux

-n option: It prints the hostname of the network node(current computer).

devasc@labvm:~$ uname -n

labvm

-r option: It prints the kernel release date.

devasc@labvm:~$ uname -r

5.4.0-37-generic

-v option: It prints the version of the current kernel.

devasc@labvm:~$ uname -v

#41-Ubuntu SMP Wed Jun 3 18:57:02 UTC 2020

-p option: It prints the type of the processor.

devasc@labvm:~$ uname -p

x86_64

-i option: It prints the platform of the hardware.

devasc@labvm:~$ uname -i

x86_64

-o option: It prints the name of the operating system.

devasc@labvm:~$ uname –o GNU/Linux

Chmod: Changing file permissions

Chmod used to change a files permissions

Syntax: chmod [-R] filename

The chmod can be represented in two ways:

1. Relative Assignment – by specifying the changes to the current permission

2. Absolute Assignment – by specifying the final permission.

Relative Manner:

Here chmod only changes the permissions specified in mode and leaves the other permissions unchanged

Structure of a chmod command

Chmod u + x file

chmod indicates command name

u is the category name

+ is operator

x indicates permission

File indicated permission

The mode in the syntax contains 3 components:

1. User category (user, group, others)

2. Operation to be performed (assign or remove)

3. Type of permission (read, write, execute)

Abbreviations used by chmod command:

Category	Operation	Permission
u -user	+ Assigns permission	r –read
g -group	- removes permission	w –write
o -others	= absolute permission	x -execute
a –all(ugo)		

Examples:

1) To assign a execute permission for the user

$chmod u+x file.txt.

2) To remove execute permission for the user

$chmod u-x file.txt

3) To assign a execute permission to all

$chmod ugo+x file.txt

or

$chmod a+x file.txt

4) To assign multiple permissions to multiple category

$chmod go-rx file.txt

5) To remove execute permission from user and assign read permission to other two categories

$chmod u-x,go+r file.txt

2. Absolute Assignment

The '=' operator can perform a limited form of absolute assignment

It assigns only by the specified permissions and removes other permissions.

Example: If a file is to be made read only to all,

$chmod ugo=r file.txt or

$chmod a=r file.txt or

$chmod =r file.txt

Note: you can't set all nine permissions to all categories.

- To set all nine permissions we have to use octal numbers
- Octal numbers for read, write and execute permissions

Read permission – **4**

Write permission – **2**

Execute permission **-1**

For each category, we add up the numbers

Ex: 6 represents r & w, 7 represents all permissions.

Ex: $chmod 644 file.txt

$ls –l

_ r w r_ _ r_ _

Octal Number	Permissions	significance
0	– – –	No permissions
1	– – x	Executable only
2	– w –	Writable only
3	– w x	Writable & executable
4	r – –	Readable only
5	r – x	Readable & executable
	r w –	
7	r w x	All permissions

Recursive Operation (-R)

– Chmod –R descends a directory hierarchy and applies the expression to every file and subdirectory.

Ex: $chmod –R a+x dir

File System & inodes

- All files & directories are held together in one big superstructure or hardisk is split into distinct partitions with a separate filesystem in each partition.
- Every file system has a directory structure headed by root called as **root file system**.
- Each file is associated with a table called **inode**. (**index node**)
- inode number of a file is unique in single file system

inode contains the following attributes of a file:

- file type (regular, directory or device)
- file permissions (contains 9 permissions and UID,GID and Sticky bit)
- number of links
- UID of the owner
- GID of the group owner
- File size in bytes
- Date & time of the last modification
- Date & time of last access
- Date & time of last change of the inode

To display inode number of a file, we have to use following command.

$ls –i file.txt

Output: 255414 file.txt

Umask (Default file & Directory Permissions)

- Umask stands for user mask
- When you create files & directories, the permissions assigned to them depend on the system default setting

The unix file system has the following default permissions for all files and directories.

Regular files → 666 (octal) rw_ rw_ rw_

Directories → 777 (octal) rwx rwx rwx

Default is transformed by subtracting the user mask from it to remove one or more permissions.

To display current value of umask: $umask

Umask Value	File Permissions	Directory Permissions
000	rw_ rw_ rw_	rwx rwx rwx
002	rw_ rw_ r_ _	rwx rwx r_x
026	rw_ r_ _ _ _ _	rwx r_x _ _ x
600	_ _ _ rw_ rw_	_ _ x rwx rwx
777	_ _ _ _ _ _ _ _ _	_ _ _ _ _ _ _ _ _

Example:

Files: 666-022 = 644

Directories: 777-022 = 755

- Umask is shell built in command.
- A user can also use this command to set a new default**.**

  Example: $umask 000

File Ownership

- Chmod and ln commands will fail if uoy don't have authority to use them

Ex: rw _ r_ _ r _ _ 1 romeo Juliet 29 Apr----- file.txt

Username groupname

- Here, Romeo can change all the attributes of file1.txt but Juliet can't even she belongs to same group
- But if Juliet copies the file.txt to her directory, then she will be the owner of the copy and can change all attributes of a file.
- The privileges of the group are set by the owner of the file and not by the group members.
- When the system administrator cretes a user account he/she assigns the parameters to the user:

  1 .user-id (uid)- both its name & numeric

  2. group-id (gid) - both its name & numeric
- To know UID,GID use command **$id**

  Uid=1003(romeo) Gid=101 (julet)

1) chown: (changing file ownership)

- Allows a super user to change the ownership of a file

  $su (login to super user)

  Pwd: enter password

  $chown –R new username filename

Example:

$ ls –l file.txt

-rw_ r _ _ r _ _ 1 **romeo** juliet date file.txt

$chown john file.txt

$ ls –l file.txt

-rw_ r _ _ r _ _ 1 **john** Juliet date file.txt

- **Once the ownership was done, romeo can no longer edit file.txt**

2) chgrp (changing group owner)

- By default the group owner of a file is the group to which the owner belongs
- A user can change the group owner of a file, but only to a group to which user belongs.
- A user can belong to more than one group.

$ ls –l file.txt

- r w_ r _ _ r _ _ 1 romeo **juliet** date file.txt

$chgrp ram file.txt ; $ls –l file.txt

- r w_ r _ _ r _ _ 1 romeo **ram** date file.txt

cp command: (copying files)

- cp stands for copy.
- This command is used to copy files or group of files or directory.
- It creates an exact image of a file on a disk with different file name.
- cp command require at least two filenames in its arguments.

Syntax:

cp [OPTION] Source Destination

cp [OPTION] Source Directory

cp [OPTION] Source-1 Source-2 Source-3 Source-n Directory

cp command works on three principal modes of operation and these operations depend upon number and type of arguments passed in cp command :

1) Two file names:

- If the command contains two file names, then it copy the contents of 1st file to the 2nd file.
- If the 2nd file doesn't exist, then first it creates one and content is copied to it.
- But if it existed then it is simply overwritten without any warning.

Syntax: $cp Src_file Dest_file

Example: $ cp a.txt b.txt

2) One or more arguments: If the command has one or more arguments, specifying file names and following those arguments, an argument specifying directory name then this command copies each source

file to the destination directory with the same name, created if not existed but if already existed then it will be overwritten.

Syntax: $cp Src_file1 Src_file2 Src_file3 Dest_directory

Example: $ cp a.txt b.txt newdir

3) Two directory names:

- If the command contains two directory names, cp copies all files of the source directory to the destination directory, creating any files or directories needed.
- This mode of operation requires an additional option, typically R, to indicate the recursive copying of directories.

Syntax: $cp -R Src_directory Dest_directory

Options:

–i (interactive):

- i stands for Interactive copying.
- With this option system first warns the user before overwriting the destination file.
- cp prompts for a response, if you press y then it overwrites the file and with any other option leave it uncopied.

Example: $ cp -i a.txt b.txt

cp: overwrite 'b.txt'? y

-f (force): If the system is unable to open destination file for writing operation because the user doesn't have writing permission for this file then by using -f option with cp command, destination file is deleted first and then copying of content is done from source to destination file.

Example: $ ls -l b.txt

-r-xr-xr-x+ 1 User User 3 Nov 24 08:45 b.txt

User, group and others doesn't have writing permission.

Without -f option, command not executed

$ cp a.txt b.txt

cp: cannot create regular file 'b.txt': Permission denied

With -f option, command executed successfully

$ cp -f a.txt b.txt

-r or -R: Copying directory structure.

- With this option cp command shows its recursive behavior by copying the entire directory structure recursively.
- Suppose we want to copy linux directory containing many files, directories into linuxlab directory(not exist).

Example:
$ ls linux /
a.txt b.txt b.txt~ Folder1 Folder2 **//Without -r option, error**
$ cp linux linuxlab
cp: -r not specified; omitting directory ' linux'
With -r, execute successfully
$ cp -r linux linuxlab
$ ls linuxlab /
a.txt b.txt b.txt~ Folder1 Folder2

Copying using * wildcard: The star wildcard represents anything i.e. all files and directories. Suppose we have many text document in a directory and wants to copy it another directory, it takes lots of time if we copy files 1 by 1 or command becomes too long if specify all these file names as the argument, but by using * wildcard it becomes simple.

Example:

Initially Folder1 is empty
$ ls
a.txt b.txt c.txt d.txt e.txt Folder1
$ cp *.txt Folder1
$ ls Folder1
a.txt b.txt c.txt d.txt e.txt
mv command: (Renaming files)

mv stands for move. mv is used to move one or more files or directories from one place to another in a file system like UNIX. It has two distinct functions:
(i) It renames a file or folder.
(ii) It moves a group of files to a different directory.
No additional space is consumed on a disk during renaming. This command normally works silently means no prompt for confirmation.

Syntax: $mv [Option] source destination

Example:

$ ls
a.txt b.txt c.txt d.txt

$ mv a.txt linux.txt
$ ls
b.txt c.txt d.txt linux.txt

options:
-i (Interactive):
- -i option makes the command ask the user for confirmation before moving a file that would overwrite an existing file, you have to press y for confirm moving, any other key leaves the file as it is.
- This option doesn't work if the file doesn't exist, it simply rename it or move it to new location.

$ mv -i linux.txt b.txt
mv: overwrite 'b.txt'? y

rm command: (Deleting Files)

- rm stands for remove.
- Used to remove objects such as files, directories, and symbolic links and so on from the file system like UNIX.
- To be more precise, rm removes references to objects from the filesystem, where those objects might have had multiple references (for example, a file with two different names).
- By default, it does not remove directories.

Syntax: rm [OPTION] FILENAME

Examples:

1) Removing one file at a time

$ rm a.txt

2) Removing more than one file at a time

$ rm b.txt c.txt

Options:

1) -i (Interactive Deletion): -i option makes the command ask the user for confirmation before removing each file

$ rm -i d.txt /

rm: remove regular empty file 'd.txt'? y

2) -f (Force Deletion): rm prompts for confirmation removal if a file is write protected.

$ ls –l file.txt

-r--r--r--+ 1 User User 0 Jun 2 12:46 file.txt

$ rm e.txt

rm: remove write-protected regular empty file 'e.txt'? n

$ rm -f e.txt

$ ls

3) -r (Recursive Deletion): With -r(or -R) option rm command performs a tree-walk and will delete all the files and sub-directories recursively of the parent directory. At each stage it deletes everything it finds.

$ rm -r *

cat command:

- Cat(concatenate) command is very frequently used in Linux.
- It reads data from the file and gives their content as output.
- It helps us to create, view, and concatenate files.

1) To view a single file
Command: $cat filename

2) To view multiple files
Command: $cat file1 file2

3) To view contents of a file preceding with line numbers.
Command: $cat -n filename

4) Create a file
Command: $ cat > newfile

5) Copy the contents of one file to another file.
Command: $cat [source filename] > [destination-filename]

6) To suppress repeated empty lines in output
Command: $cat -s file.txt

7) To append the contents of one file to the end of another file.
Command: $cat file1 >> file2

8) To display content in reverse order using tac command.
Command: $tac filename

9) To highlight the end of line.
Command: $cat -E "filename"

10) To open dashed files.
Command: $cat -- "-dashfile"

11) To merge the contents of multiple files.
Command: $cat "filename1" "filename2" "filename3" > "merged_filename"

wc (counting lines, words and characters)

- wc stands for word count, mainly used for counting purpose.
- Used to find out number of lines, word count, byte and characters count in the files specified in the file arguments.
- By default it displays four-columnar output.
- **First column** shows number of lines present in a file specified, **second column** shows number of words present in the file, **third column** shows number of characters present in file and **fourth column** itself is the file name which are given as argument.

Syntax: wc Option Filename

Example:

$ cat state.txt	**$ cat capital.txt**
Andhra Pradesh	Hyderabad
Arunachal Pradesh	Itanagar
Assam	Dispur
Bihar	Patna
Chhattisgarh	Raipur

1) Passing only one file name in the argument.

$ wc state.txt

5 7 63 state.txt

$ wc capital.txt

5 5 45 capital.txt

2) Passing more than one file name in the argument.

$ wc state.txt capital.txt

5 7 63 state.txt

5 5 45 capital.txt

10 12 108 total

Options:

-l: prints the number of lines present in a file.
-w: prints the number of words present in a file.
-c: displays count of bytes present in a file.
-m: displays count of characters from a file.
-L: used to print out the length of longest (number of characters) line in a file.

Examples:

$ wc -l state.txt	**$ wc -c state.txt**	**$ wc -L state.txt**
5 state.txt	17 state.txt	63 state.txt

$ wc -w state.txt	**$ wc -m state.txt**
7 state.txt	63 state.txt

df(disk free)

- Used to display information related to file systems about total space and available space.

Syntax: $ df options filename

Examples:

1) $ df //displays the space available on all currently mounted file systems

2) $ df –h // display size in power of 1024 (human readable form)

3) $df –i // display the information of number of used inodes and their percentage for the file system.

4) $df –T // display file system type along with other information.

du(disk usage)

- used to estimate file space usage.
- The du command can be used to track the files and directories which are consuming excessive amount of space on hard disk drive.
- The du command also displays the files and directory sizes in a recursively manner.

Syntax: $ du options filename

Examples:

1) $ du //displays the disk usage summary

2) $ du –h // display in human readable form

3) $ du –a // displays the disk usage of all the files and directories

4) $ du –ch // provides a grand total usage disk space at the last line

ps command (Process Status)

- ps command is used to list the currently running processes and their PIDs along with some other information depends on different options.
- It reads the process information from the virtual files in /proc file-system.
- /proc contains virtual files.
- ps provides numerous options for manipulating the output according to our need.

Syntax: ps [options]

Examples:

1) Simple process selection: Shows the processes for the current shell

[root@rhel7 ~] $ ps

```
  PID TTY          TIME CMD
12330 pts/0    00:00:00 bash
21621 pts/0    00:00:00 ps
```

PID – the unique process ID
TTY – terminal type that the user is logged into
TIME – amount of CPU in minutes and seconds that the process has been running
CMD – name of the command that launched the process.

Options:
-f: lists the pid of parent process also // $ps -f
-u: list the processes of a given user
-a: lists the processes of all the users
-e: lists the processes including system processes.

More command:

- more command is used to view the text files in the command prompt, displaying one screen at a time in case the file is large.
- The more command also allows the user do scroll up and down through the page.
- Apart from the first page, you can also see the filename and percentage of the file that has been viewed.

Syntax: more [-options] [-num] [+/pattern] [+linenum] [filename]

[-options]: any option that you want to use in order to change the way the file is displayed. Choose any one from the followings: (-d, -l, -f, -p, -c, -s, -u)

[-num]: type the number of lines that you want to display per screen.

[+/pattern]: replace the pattern with any string that you want to find in the text file.

[+linenum]: use the line number from where you want to start displaying the text content.

[file_name]: name of the file containing the text that you want to display on the screen.

Options:

-d : help the user to navigate. It displays "[Press space to continue, 'q' to quit.]" and displays "[Press 'h' for instructions.]" when wrong key is pressed.

-f : This option does not wrap the long lines and displays them as such.

-p : This option clears the screen and then displays the text.

-c : used to display the pages on the same area by overlapping the previously displayed text.

-s : squeezes multiple blank lines into one single blank line.

Examples:

1) $more filename

2) $more –d filename

3) $more –f filename

Navigation Keys

f or space bar –one page forward

b – one page back

4) $more –p filename

5) $more -2 filename

6) $more +2 filename

7) $more +/string filename

8) $ls | more

5f – 5 pages forward

5b – 5 pages back

. – used to repeat last command

Logout Command

logout command allows you to programmatically logout from your session. causes the session manager to take the requested action immediately.

EXAMPLES:

To logout from current user session:

$ logout

output:

no output on screen, current user session will be logged out.

Shutdown Command

- The shutdown command in Linux is used to shutdown the system in a safe way.
- You can shutdown the machine immediately, or schedule a shutdown using 24 hour format.
- It brings the system down in a secure way.
- When the shutdown is initiated, all logged-in users and processes are notified that the system is going down, and no further logins are allowed.
- Only root user can execute shutdown command.

Syntax: $ shutdown [OPTIONS] [TIME] [MESSAGE]

options – Shutdown options such as halt, power-off (the default option) or reboot the system.
time – The time argument specifies when to perform the shutdown process.
message – The message argument specifies a message which will be broadcast to all users.

Options:

-r : Requests that the system be rebooted after it has been brought down.

-h : Requests that the system be either halted or powered off after it has been brought down, with the choice as to which left up to the system.

-H : Requests that the system be halted after it has been brought down.

-P : Requests that the system be powered off after it has been brought down.

-c : Cancels a running shutdown. TIME is not specified with this option, the first argument is MESSAGE.

-k : Only send out the warning messages and disable logins, do not actually bring the system down.

How to use shutdown : $ sudo shutdown

How to shutdown the system at a specified time

The time argument can have two different formats. It can be an absolute time in the format hh:mm and relative time in the format +m where m is the number of minutes from now.

The following example will schedule a system shutdown at 05 A.M:

$ sudo shutdown 05:00

The following example will schedule a system shutdown in 20 minutes from now:

$ sudo shutdown +20

To shutdown your system immediately you can use +0 or its alias now:

$ sudo shutdown now

The following command will shut down the system in 10 minutes from now and notify the users with message "System upgrade":

$ sudo shutdown +10 "System upgrade"

How to halt your system : This can be achieved using the -H option.

$ shutdown -H

Halting means stopping all CPUs and powering off also makes sure the main power is disconnected.

How to make shutdown power-off machine

Although this is by default, you can still use the -P option to explicitly specify that you want shutdown to power off the system.

$ shutdown -P

For reboot, the option is -r.

$ shutdown -r

You can also specify a time argument and a custom message:

$ shutdown -r +5 "Updating Your System"

The command above will reboot the system after 5 minutes and broadcast Updating Your System"

If you have scheduled a shutdown and you want to cancel it you can use the -c argument:

$ sudo shutdown -c

When canceling a scheduled shutdown, you cannot specify a time argument, but you can still broadcast a message that will be sent to all users.

$ sudo shutdown -c "Canceling the reboot"

Pwd command:

- pwd stands for Print Working Directory.
- It prints the path of the working directory, starting from the root.
- pwd is shell built-in command(pwd)

Syntax: $pwd

Vi Editor

- This editor enables you to edit lines in context with other lines in the file.
- An improved version of the vi editor which is called the VIM has also been made vailable now. Here, VIM stands for Vi IMproved.
- It's usually available on all the flavors of Unix system.
- Its implementations are very similar across the board.
- It requires very few resources.
- It is more user-friendly than other editors such as the ed or the ex.
- You can use the vi editor to edit an existing file or to create a new file from scratch. You can also use this editor to just read a text file.

Command & Description:

$vi filename // Creates a new file if it already does not exist, otherwise opens an existing file.

$vi -R filename //Opens an existing file in the read-only mode.

$view filename //Opens an existing file in the read-only mode.

Operation Modes:

Command mode − This mode enables you to perform administrative tasks such as saving the files, executing the commands, moving the cursor, cutting and pasting the lines or words, as well as finding and replacing.

Insert mode − this mode enables you to insert text into the file.

Last Line Mode(Escape Mode)- Line Mode is invoked by typing a colon [:], while vi is in Command Mode. The cursor will jump to the last line of the screen and vi will wait for a command. This mode enables you to perform tasks such as saving files, executing commands.

NOTE:
vi always starts in the command mode. To enter text, you must be in the insert mode for which simply type i. To come out of the insert mode, press the Esc key, which will take you back to the command mode.

VI Editing commands (You should be in the "command mode" to execute these commands.)

i - Insert at cursor (goes into insert mode)
a - Write after cursor (goes into insert mode)
A - Write at the end of line (goes into insert mode)
ESC - Terminate insert mode
u - Undo last change
U - Undo all changes to the entire line
o - Open a new line (goes into insert mode)
dd - Delete line
3dd - Delete 3 lines.
D - Delete contents of line after the cursor
C - Delete contents of a line after the cursor and insert new text. Press ESC key to end insertion.
dw - Delete word
4dw - Delete 4 words
cw - Change word
x - Delete character at the cursor
r - Replace character
R - Overwrite characters from cursor onward
s - Substitute one character under cursor continue to insert
S - Substitute entire line and begin to insert at the beginning of the line
~ - Change case of individual character

Moving within a file (You need to be in the command mode to move within a file)

k - Move cursor up
j - Move cursor down
h - Move cursor left
l - Move cursor right

Copy and Paste Commands:

Yy : Copies the current line.
9yy : Yank current line and 9 lines below.
p : Puts the copied text after the cursor.
P : Puts the yanked text before the cursor.

Saving and Closing the file

Shift+zz - Save the file and quit
:w - Save the file but keep it open
:q - Quit without saving
:wq - Save the file and quit

Shell Programming:

- Usually shells are interactive that mean, they accept command as input from users and execute them.
- However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.
- As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work.
- These files are called Shell Scripts or Shell Programs. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with .sh file extension

Ex. myscript.sh

Shell Types:

In Unix, there are two major types of shells –

Bourne shell – If you are using a Bourne-type shell, the $ character is the default prompt.

C shell – If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories –

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow –

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

Example Script:

```
#!/bin/bash

pwd

ls
```

NOTE:

Shebang construct- to alert the system that a shell script is being started.

It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.

A shell script comprises following elements –

1. Shell Keywords – if, else, break etc.
2. Shell commands – cd, ls, echo, pwd, touch etc.
3. Functions
4. Control flow – if..then..else, case and shell loops etc.

Why do we need shell scripts:

There are many reasons to write shell scripts –

1. To avoid repetitive work and automation
2. System admins use shell scripting for routine backups
3. System monitoring
4. Adding new functionality to the shell etc.

Advantages of shell scripts

1. The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax
2. Writing shell scripts are much quicker
3. Quick start
4. Interactive debugging etc.

Disadvantages of shell scripts

1. Prone to costly errors, a single mistake can change the command which might be harmful
2. Slow execution speed
3. Design flaws within the language syntax or implementation
4. Not well suited for large and complex task
5. Provide minimal data structure unlike other scripting languages. etc

SHELL VARIABLES

- A variable is a character string to which we assign a value.
- The value assigned could be a number, text, filename, device, or any other type of data.
- The shell enables you to create, assign, and delete variables.

Variable Names:

- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
- Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names:

_ALI

TOKEN_A

VAR_1

VAR_2

Following are the examples of invalid variable names:

2_VAR

-VARIABLE

VAR1-VAR2

VAR_A!

NOTE:

Dont use other characters such as !, *, or - is that these characters have a special meaning for the shell.

Defining Variables:

Variables are defined as follows –

syntax: variable_name=variable_value

For example, NAME="APPLE"

Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign ($)

EXAMPLE:

NAME="apple"

echo $NAME

output: apple

Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME –

NAME="apple"

readonly NAME

NAME="banana"

output: /bin/sh: NAME: This variable is read only.

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables.

Once you unset a variable, you cannot access the stored value in the variable.

Syntax: unset variable_name

Example:

NAME="apple"

unset NAME

echo $NAME

output: does not print anything

Special Variables:

$echo $$

$ character represents the process ID number

$0 -The filename of the current script.

$# -The number of arguments supplied to a script.

$* -All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.

$@ -All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.

$? -The exit status of the last command executed.

$! -The process number of the last background command.

Examples:

echo "File Name: $0"

echo "First Parameter : $1"

echo "Second Parameter : $2"

echo "Quoted Values: $@"

echo "Quoted Values: $*"

echo "Total Number of Parameters : $#"

Array in Shell Scripting

An array is a systematic arrangement of the same type of data. But in Shell script Array is a variable which contains multiple values may be of same type or different type since by default in shell script everything is treated as a string. An array is zero-based ie indexing start with 0.

Indirect Declaration

ARRAYNAME[INDEXNR]=value

Compound Assignment

ARRAYNAME=(value1 value2 .....valueN)

or

[indexnumber=]string

ARRAYNAME=([1]=10 [2]=20 [3]=30)

To Print All elements

[@] & [*] means All elements of Array.

Example:

```
arr=(apple banana 1 linux windows carrot)
 # To print all elements of array
echo  ${arr[@]}
echo  ${arr[*]}
echo  ${arr[@]:1}
echo  ${arr[*]:3}

#To print first element
echo ${arr[0]}
echo  ${arr}

#To Print Selected index element
echo ${ARRAYNAME[INDEXNR]}
echo  ${arr[3]}
echo  ${arr[1]}

#To print elements from a particular index
echo  ${ARRAYNAME[WHICH_ELEMENT]:STARTING_INDEX}
echo  ${arr[@]:0}
echo  ${arr[@]:1}
echo  ${arr[@]:2}
echo  ${arr[0]:1}
```

#To print elements in range

echo ${ARRAYNAME[WHICH_ELEMENT]:STARTING_INDEX:COUNT_ELEMENT}

echo ${arr[@]:1:4}

echo ${arr[@]:2:3}

echo ${arr[0]:1:3}

#To count Length of in Array

Use #(hash) to print length of particular element

echo ${#arr[0]}

echo ${#arr}

Example:

arr=(apple banana 1 linux windows carrot)

echo ${#arr[1]}

echo ${#arr}

echo ${#arr[@]}

echo ${#arr[*]}

#To Search in Array

Search Returns 1 if it found the pattern else it return zero. It does not alter the original array elements.

echo ${arr[@]/*[aA]*/}

#To Search & Replace in Array

//Search_using_Regular_Expression/Replace : Search & Replace

echo ${arr[@]//a/A}

echo ${arr[0]//r/R}

To delete Array Variable in Shell Script?

To delete index-1 element

unset ARRAYNAME[1]

To delete the whole Array

unset ARRAYNAME

Example:

```
arr=(apple banana 1 linux windows carrot)
  # To print all elements of array
echo  ${arr[@]}
echo  ${arr[*]}
echo  ${arr[@]:1}
echo  ${arr[*]:3}

# To print first element
echo ${arr[0]}
echo  ${arr}

# To print particular element
echo ${arr[3]}
echo  ${arr[1]}

# To print elements from a particular index
echo ${arr[@]:0}
echo  ${arr[@]:1}
echo  ${arr[@]:2}
echo  ${arr[0]:1}
```

```
# To print elements in range
echo ${arr[@]:1:4}
echo  ${arr[@]:2:3}
echo  ${arr[0]:1:3}

# Length of Particular element
echo ${#arr[0]}
echo  ${#arr}

# Size of an Array
echo ${#arr[@]}
echo ${#arr[*]}

# Search in Array
echo ${arr[@]/*[aA]*/}

# Replacing Substring Temporary
echo ${arr[@]//a/A}
echo  ${arr[0]//r/R}
```

SHELL OPERATORS

There are various operators supported by each shell.

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

✓ Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.

Example for how to add two numbers:

```
val=`expr 2 + 2`
echo "Total value : $val"
```

Output: Total value : 4

Arithmetic Operators:

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then –

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

Example shell script:

Shell script	Output

a=10
b=20
 	
val=`expr $a + $b`
echo "a + b : $val"
 	
val=`expr $a - $b`
echo "a - b : $val"
 	
val=`expr $a \* $b`
echo "a * b : $val"
 	
val=`expr $b / $a`
echo "b / a : $val"
 	
val=`expr $b % $a`
echo "b % a : $val"
 	
if [$a == $b]
then
  echo "a is equal to b"
fi
 	
if [$a != $b]
then
  echo "a is not equal to b"
fi
	a + b : 30
	a - b : -10
	a * b : 200
	b / a : 2
	b % a : 0
	a is not equal to b

Relational Operators

These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.

-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

Example:

Shell script	Output

a=10
b=20
 	
if [$a -eq $b]
then
  echo "$a -eq $b : a is equal to b"
else
  echo "$a -eq $b: a is not equal to b"
fi
 	
if [$a -ne $b]
then
  echo "$a -ne $b: a is not equal to b"
else
  echo "$a -ne $b : a is equal to b"
fi
 	
if [$a -gt $b]
then
  echo "$a -gt $b: a is greater than b"
else
  echo "$a -gt $b: a is not greater than b"
fi
 	
if [$a -lt $b]
then
  echo "$a -lt $b: a is less than b"
else
  echo "$a -lt $b: a is not less than b"
fi
	10 -eq 20: a is not equal to b
	10 -ne 20: a is not equal to b
	10 -gt 20: a is not greater than b
	10 -lt 20: a is less than b
	10 -ge 20: a is not greater or equal to b
	10 -le 20: a is less or equal to b

if [$a -ge $b]
then
  echo "$a -ge $b: a is greater or equal to b"
else
  echo "$a -ge $b: a is not greater or equal to b"
fi
 	
if [$a -le $b]
then
  echo "$a -le $b: a is less or equal to b"
else
  echo "$a -le $b: a is not less or equal to b"
fi	

Boolean Operators

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

Examples:

Shell script	Output

a=10
b=20
 	
if [$a != $b]
then
  echo "$a != $b : a is not equal to b"
else
  echo "$a != $b: a is equal to b"
fi
 	
if [$a -lt 100 -a $b -gt 15]
then
	10 != 20 : a is not equal to b
	10 -lt 100 -a 20 -gt 15 : returns true
	10 -lt 100 -o 20 -gt 100 : returns true
	10 -lt 5 -o 20 -gt 100 : returns false

```
   echo "$a -lt 100 -a $b -gt 15 : returns
true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns
false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns
true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns
false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns
true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns
false"
fi
```

String Operators:

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

Examples:

Shell script	Output

a="abc"
b="efg"
 	
if [$a = $b]
then
  echo "$a = $b : a is equal to b"
else
  echo "$a = $b: a is not equal to b"
fi
 	
if [$a != $b]
then
  echo "$a != $b : a is not equal to b"
else
  echo "$a != $b: a is equal to b"
fi
 	
if [-z $a]
then
  echo "-z $a : string length is zero"
else
  echo "-z $a : string length is not zero"
fi
 	
if [-n $a]
then
  echo "-n $a : string length is not zero"
else
  echo "-n $a : string length is zero"
fi
 	
if [$a]
then
  echo "$a : string is not empty"
else
  echo "$a : string is empty"
fi
	abc = efg: a is not equal to b
	abc != efg : a is not equal to b
	-z abc : string length is not zero
	-n abc : string length is not zero
	abc : string is not empty

File Test Operators:

We have a few operators that can be used to test various properties associated with a Unix file.

Assume a variable file **holds an existing file name "sample" the size of which is 100 bytes and has** read, write **and** execute **permission on**

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Control structures

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here –

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

The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if...else** statement –

1. **if...fi statement**
2. **if...else...fi statement**
3. **if...elif...else...fi statement**

if...fi statement

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax:

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

If the resulting value is true, given statement(s) are executed.

If the expression is false then no statement would be executed.

Example:

```
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
```

```
fi

if [ $a != $b ]

then

   echo "a is not equal to b"

fi
```

output: a is not equal to b

if...else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

Syntax

```
if [ expression ]

then

   Statement(s) to be executed if expression is true

else

   Statement(s) to be executed if expression is not true

fi
```

If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

Example:

```
a=10

b=20

if [ $a == $b ]

then

   echo "a is equal to b"

else

   echo "a is not equal to b"

fi
```

Output: a is not equal to b

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

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax:

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

Example:

```
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
```

```
else
   echo "None of the condition met"
fi
```

Output:

a is less than b

The case...esac Statement

The case...esac statement in the Unix shell is very similar to the switch...case statement we have in other programming languages like C or C++ and PERL, etc.

Syntax:

The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
case word in
  pattern1)
     Statement(s) to be executed if pattern1 matches
     ;;
  pattern2)
     Statement(s) to be executed if pattern2 matches
     ;;
  pattern3)
     Statement(s) to be executed if pattern3 matches
     ;;
  *)
    Default condition to be executed
     ;;
esac
```

Example 1:

```
FRUIT="kiwi"
case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."       ;;
   "banana") echo "I like banana nut bread."  ;;
   "kiwi") echo "New Zealand is famous for kiwi."  ;;
esac
```

output:

New Zealand is famous for kiwi.

Example 2:

```
vehicle=$1
case $vehicle in
         "car"  )
      echo " Rent of vehicle is 10" ;;
        "bus" )
      echo " Rent of vehicle is 20" ;;
        * )
      echo " unkown vehicle" ;;
esac
```

Output:

$./sample.sh car

Rent of vehicle is 10

Example 3:

```
echo -e "enter some character :\c"
read value
case $value in
        [a-z] )
     echo " you entered $value a to z" ;;
       [A-Z] )
     echo " you entered $value A to Z" ;;
        [0-9] )
     echo " you entered $value 0 to 9" ;;
       ? )
     echo " you entered $value special character" ;;
        * )
     echo " unkown input" ;;
esac
```

Example 4:

```
echo “Enter a number”
read num
case $num in
[0-9])
echo “you have entered a single digit number”  ;;
[1-9][1-9])
echo “you have entered a two-digit number”        ;;
[1-9][1-9][1-9])
echo “you have entered a three-digit number”    ;;
*) echo “your entry does not match any of the conditions” ;;
esac
```

Example 5:

```
echo "Enter your lucky number"
read n
case $n in
101)
echo echo "You got 1st prize" ;;
510)
echo "You got 2nd prize" ;;
999)
echo "You got 3rd prize" ;;
*)
echo "Sorry, try for the next time" ;;
esac
```

LOOPS IN SHELL SCRIPT

* list of commands executed repeatedly

while loop

syntax:

```
while [ condition ]
do
        command1
        command2
        -----
        -----
done
```

EXAMPLE: TO PRINT 1 TO n NUMBERS

```
n=1
while [ $n -le 10 ]  or while (( $n <= 10 ))
do
   echo "$n"
   n = $(( n + 1 ))  or      (( n++ ))
done
```

NOTE:

1. To delay the value while printing use sleep 1

2. for infinite loop use ctrl+c to come out from loop

How to read a file using while loop:

```
while read f      # f is a varaible name
do
  echo $f
done < sample.sh
                (or)
cat sample.sh | while read f
do
   echo $f
done
```

UNTIL LOOPS

* If the condition is false then only it will execute commands

* similar to while loop

syntax:

```
until [ condition ]
do
   command1
   command2
   --------
  --------
done
```

EXAMPLE: TO PRINT 1 TO n NUMBERS

```
n=1
until [ $n -le 10 ]
do
  echo "$n"
  n = $(( n + 1 ))
done
```

FOR LOOP:

SYNTAX:

1) for variable in 1 2 3 4 5 .. N

```
  do
        command1
        command2
        --------
       --------
  done
```

2) for variable in file1 file2

```
do
    command1 on $varaible
    command2
    --------
   --------
done
```

3) for variable in $linux_command

```
do
    command1 on $varaible
    command2
    --------
   --------
done
```

4) for ((exp1; exp2; exp3))

```
do
    command1
    command2
    --------
   --------
done
```

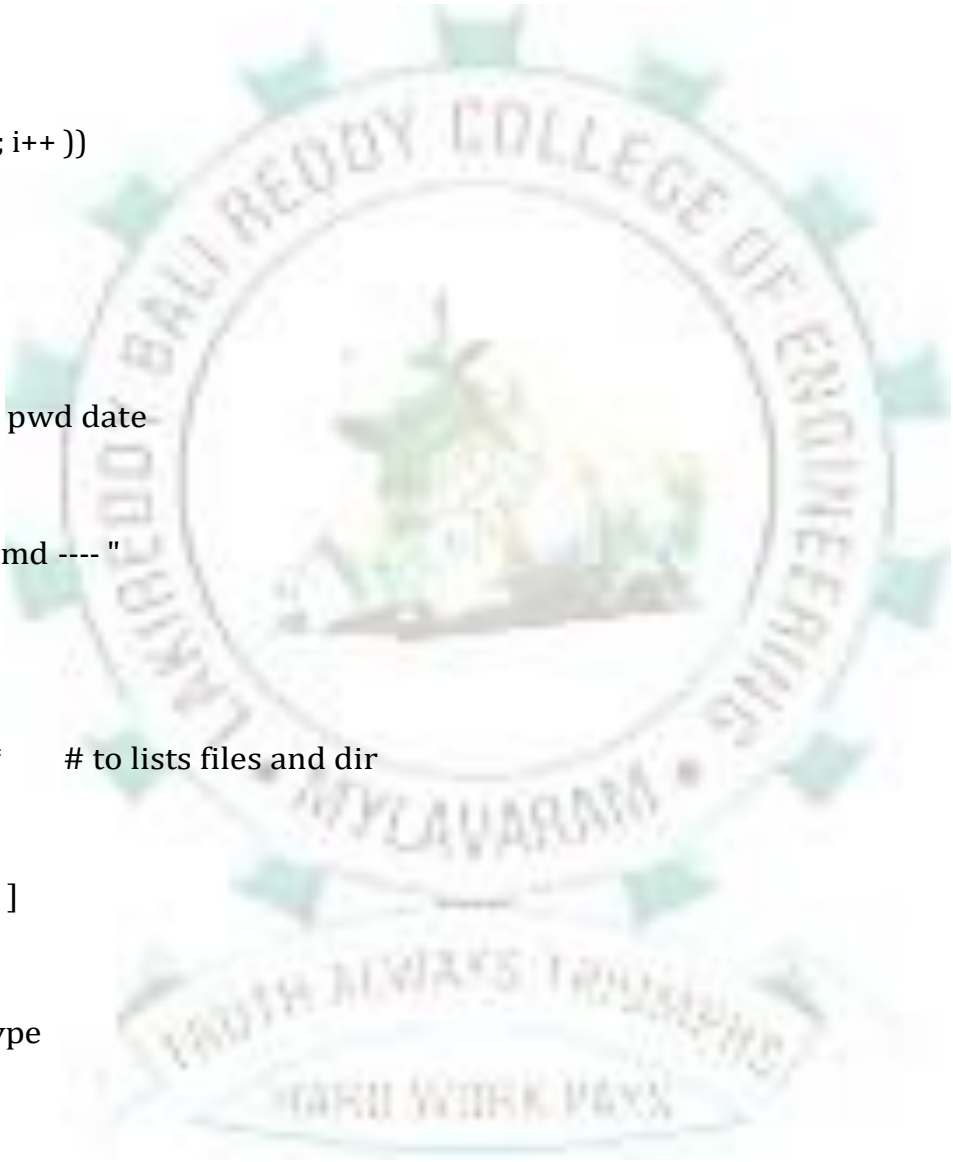
Examples:

1) for i in 1 2 3 4 5

```
do
    echo $i
done
```

2) for i in {1..10}

```
do
        echo $i
done
3) for i in  {1..10..2}  # { start..end..increment }
do
        echo $i
done
4) for (( i=0; i<5; i++ ))
do
        echo $i
done
5) for cmd in ls pwd date
do
    echo  "---$cmd ---- "
    $cmd
done
6) for type  in *       # to lists files and dir
do
    if [ -d $type ]
    then
        echo  $type
    fi
```



SELECT LOOP:

- Similar to for loop and case statements

- to generate easy menus

SYNTAX:

```
select variable_name in list
    do
        command1
        command2
        --------
       --------
    done
```

Example 1:

```
select name in ant bat cat
do
   echo "$name selected"
done
```

output:

```
1)ant
2)bat
3)cat
#? 2
bat selected
#?
```

Example 2:

```
select ch in cp rm mv ln
do
        case $ch in
                cp) echo "enter source file name"
                        read file1
                        echo " enter destination file name "
                        read file2
                        cp $file1 $file2
                        echo " file copy is successful"
                        ;;
                rm) echo "enter file name to remove "
                        read file
                        rm $file
                        echo " file has removed"
                        ;;
                  mv) echo "enter the source filename"
                        read file1
                echo " enter destination file name to rename source file"
                read file2
                mv $file1 $file2
                echo " file renaming is successful"
                ;;
                ln) echo " enter source file"
                read file1
                echo " enter destination file"
                read file2
```

```
            ln $file1 $file2
            echo " linking completed"
              ;;
             *) exit
esac
done
```

BREAK & CONTINUE

Example for Break:

```
for (( i=1; i<=10; i++ ))
  do
    if [ $i -gt 5 ]
      then
      break
    fi
       echo "$i"
  done
```

Example for Continue:

```
for (( i=1; i<=10; i++ ))
  do
    if [ $i -eq 3  -o $i -eq 6 ]
      then
      continue
    fi
       echo "$i"
  done
```

Positional parameters

```
echo
echo "Positional parameters before set \`uname -a\` :"
echo "Command-line argument #1 = $1"
echo "Command-line argument #2 = $2"
echo "Command-line argument #3 = $3"
set `uname -a` # Sets the positional parameters to the output
echo
echo "Positional parameters after set \`uname -a\` :"
# $1, $2, $3, etc. reinitialized to result of `uname -a`
echo "Field #1 of 'uname -a' = $1"
echo "Field #2 of 'uname -a' = $2"
echo "Field #3 of 'uname -a' = $3"
exit 0
```

Output:

```
devasc@labvm:~$ ./ex.sh LBRCE CSE SHELLSCRIPTING
Positional parameters before set `uname -a` :
Command-line argument #1 = LBRCE
Command-line argument #2 = CSE
Command-line argument #3 = SHELLSCRIPTING
Positional parameters after set `uname -a` :
Field #1 of 'uname -a' = Linux
Field #2 of 'uname -a' = labvm
Field #3 of 'uname -a' = 5.4.0-37-generic
```

Example Programs: (as per Syllabus)

1. Use of Basic UNIX Shell Commands: ls, mkdir, rmdir, cd, cat, touch, file, wc, sort, cut, grep,dd, dfspace, du, ulimit

cut command

- The cut command in UNIX is a command for cutting out the sections from each line of files and writing the result to standard output.
- It can be used to cut parts of a line by byte position, character and field.
- It is necessary to specify option with command otherwise it gives error.
- If more than one file name is provided then data from each file is not precedes by its file name.

Syntax: cut OPTION... [FILE]...

$ cat state.txt

Andhra Pradesh

Arunachal Pradesh

Assam

Bihar

Chhattisgarh

1. -b(byte):

- To extract the specific bytes, you need to follow -b option with the list of byte numbers separated by comma.
- Range of bytes can also be specified using the hyphen(-).
- It is necessary to specify list of byte numbers otherwise it gives error.
- Tabs and backspaces are treated like as a character of 1 byte.

List without ranges

$ cut -b 1,2,3 state.txt

And

Aru

Ass

Bih

Chh

List with ranges

$ cut -b 1-3,5-7 state.txt

Andra

Aruach

Assm

Bihr

Chhtti

Here, 1- indicate from 1st byte to end byte of a line

$ cut -b 1- state.txt

Andhra Pradesh

Arunachal Pradesh

Assam

Bihar

Chhattisgarh

Here, -3 indicate from 1st byte to 3rd byte of a line

$ cut -b -3 state.txt

And

Aru

Ass

Bih

Chh

2. -c (column):

- To cut by character use the -c option..
- This can be a list of numbers separated comma or a range of numbers separated by hyphen (-).
- Tabs and backspaces are treated as a character.
- It is necessary to specify list of character numbers otherwise it gives error with this option.

$ cut -c 2,5,7 state.txt

nr

rah

sm

ir

hti

$ cut -c 1-7 state.txt

Andhra

Arunach

Assam

Bihar

Chhatti

$ cut -c 1- state.txt

Andhra Pradesh

Arunachal Pradesh

Assam

Bihar

Chhattisgarh

$ cut -c -5 state.txt

Andhr

Aruna

Assam

Bihar

Chhat

3. -f (field):

- used to cut by fields rather than columns.
- List of the fields number specified must be separated by comma.
- Ranges are not described with -f option.
- cut uses tab as a default field delimiter but can also work with other delimiter by using -d option.

Syntax: $cut -d "delimiter" -f (field number) file.txt

If -d option is used then it considered space as a field separator or delimiter:

$ cut -d " " -f 1 state.txt

Andhra

Arunachal

Assam

Bihar

Chhattisgarh

$ cut -d " " -f 1-4 state.txt

Andhra Pradesh

Arunachal Pradesh

Assam

Bihar

Chhattisgarh

4. –complement: As the name suggests it complement the output. This option can be used in the combination with other options either with -f or with -c.

$ cut --complement -d " " -f 1 state.txt

Pradesh

Pradesh

Assam

Bihar

Chhattisgarh

$ cut --complement -c 5 state.txt

Andha Pradesh

Arunchal Pradesh

Assa

Biha

Chhatisgarh

5. –output-delimiter: By default the output delimiter is same as input delimiter that we specify in the cut with -d option.

To change the output delimiter use the option –output-delimiter=”delimiter”.

$ cut -d " " -f 1,2 state.txt --output-delimiter='%'

Andhra%Pradesh

Arunachal%Pradesh

Assam

Bihar

Chhattisgarh

Sort command:

- SORT command is used to sort a file, arranging the records in a particular order.
- By default, the sort command sorts file assuming the contents are ASCII.
- Using options in sort command, it can also be used to sort numerically.

The sort command follows these features as stated below:

- Lines starting with a number will appear before lines starting with a letter.
- Lines starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet.
- Lines starting with a lowercase letter will appear before lines starting with the same letter in uppercase.

$ cat > sample.txt
abc
apple
BALL
Abc
bat

$ sort sample.txt
abc
Abc
apple
bat
BALL

Options:

1) -o Option: if you want to write the output to a new file, output.txt, redirects the output

$ sort -o output.txt newfilename.txt

2) -r Option: Sorting In Reverse Order, which sorts the input file in reverse order i.e. descending order by default.

Syntax: $ sort -r filename

$ cat file.txt

apple

cat

egg

ball

ant

$ sort -r file.txt

egg

cat

ball

apple

ant

3) -n Option: This option is used to sort the file with numeric data present inside.

Syntax: $ sort -n filename.txt

$ cat > file1.txt

50

39

15

89

200

$sort –n file1.txt

15

39

50

89

200

4) -nr option: To sort a file with numeric data in reverse order we can use the combination of two options as stated below.

$ sort -nr filename.txt

$ sort -nr file1.txt

200

89

50

39

15

5) -k Option: Use the -k option to sort on a certain column. For example, use "-k 2" to sort on the second column. Syntax : $ sort -k filename.txt

$ cat > employee.txt

manager 5000

clerk 4000

employee 6000

peon 4500

director 9000

guard 3000

$ sort -k 2n employee.txt

guard 3000

clerk 4000

peon 4500

manager 5000

employee 6000

director 9000

6) -c option: This option is used to check if the file given is already sorted or not & checks if a file is already sorted pass the -c option to sort.

Syntax: $ sort -c filename.txt

$ cat cars.txt.

Audi

Cadillac

BMW

Dodge

$ sort -c cars.txt

sort: cars.txt:3: disorder: BMW

7) -u option: This option is helpful as the duplicates being removed gives us an redundant file.

Syntax : $ sort -u filename.txt

$ cat cars.txt

Audi

BMW

Cadillac

BMW

Dodge

$ sort -u cars.txt

Audi

BMW

Cadillac

Dodge

8) -M Option: To sort by month pass the -M option to sort. This will write a sorted list to standard output ordered by month name.

Syntax : $ sort -M filename.txt

$ cat > months.txt

February

January

March

August

September

$ sort -M months.txt

January

February

March

August

September

Grep command:

The grep filter searches a file for a particular pattern of characters, and displays all lines that contain that pattern. The pattern that is searched in the file is referred to as the regular expression (grep stands for globally search for regular expression and print out).

Syntax: grep [options] pattern [files]

Options Description

-c : This prints only a count of the lines that match a pattern

-h : Display the matched lines, but do not display the filenames.

-i : Ignores, case for matching

-l : Displays list of a filenames only.

-n : Display the matched lines and their line numbers.

-v : This prints out all the lines that do not matches the pattern

-e exp : Specifies expression with this option. Can use multiple times.

-f file : Takes patterns from file, one per line.

-E : Treats pattern as an extended regular expression (ERE)

-w : Match whole word

-o : Print only the matched parts of a matching line, with each such part on a separate output line.

cat > grep.txt

unix is great os. unix is opensource. unix is free os.

learn operating system.

Unix linux which one you choose.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

1. Case insensitive search: The -i option enables to search for a string case insensitively in the give file. It matches the words like "UNIX", "Unix", "unix".

$grep -i "UNix" grep.txt
Output:

unix is great os. unix is opensource. unix is free os.

Unix linux which one you choose.

uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

2. Displaying the count of number of matches: We can find the number of lines that matches the given string/pattern

$grep -c "unix" grep.txt
Output:

2

3. Display the file names that matches the pattern: We can just display the files that contains the given string/pattern.

$grep -l "unix" *

or

$grep -l "unix" f1.txt f2.txt f3.xt f4.txt
Output:

grep.txt

4. Checking for the whole words in a file: By default, grep matches the given string/pattern even if it found as a substring in a file. The -w option to grep makes it match only the whole words.
$ grep -w "unix" grep.txt
Output:
unix is great os. unix is opensource. unix is free os.
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

5. Displaying only the matched pattern: By default, grep displays the entire line which has the matched string. We can make the grep to display only the matched string by using the -o option.

$ grep -o "unix" grep.txt

Output:
unix

unix

unix

unix

unix

unix

6. Show line number while displaying the output using grep -n: To show the line number of file with the line matched.

$ grep -n "unix" grep.txt
Output:

1: unix is great os. unix is opensource. unix is free os.
4: uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

7. Inverting the pattern match: You can display the lines that are not matched with the specified search sting pattern using the -v option.

$ grep -v "unix" grep.txt

Output:
learn operating system.
Unix linux which one you choose.

8. Matching the lines that start with a string: The ^ regular expression pattern specifies the start of a line. This can be used in grep to match the lines which start with the given string or pattern.

$ grep "^unix" grep.txt
Output:

unix is great os. unix is opensource. unix is free os.

9. Matching the lines that end with a string: The $ regular expression pattern specifies the end of a line. This can be used in grep to match the lines which end with the given string or pattern.

$ grep "os$" grep.txt

10. Specifies expression with -e option. Can use multiple times:

$grep –e "Agarwal" –e "Aggarwal" –e "Agrawal" grep.txt

11. -f file option Takes patterns from file, one per line.

$cat pattern.txt

Agarwal
Aggarwal
Agrawal

$grep –f pattern.txt grep.txt

Grep with REGULAR EXPRESSIONS

woodhouse

wodehouse

$grep "wo[od][de]house" filename

Symbols:

g*	nothing or g,gg,ggg etc
gg*	g,gg,ggg etc
.*	nothing or any no of characters
[1-3]	a digit b/w 1 & 3
[^a-zA-Z]	a non-alphabetic character
bash$	bash at end of file
^bash$	bash as the only word in line
^$	line containing nothing
[ ]	Matches any one of a set characters
[ ] with hyphen	Matches any one of a range characters
^	The pattern following it must occur at the beginning of each line
^ with [ ]	The pattern must not contain any character in the set specified
$	The pattern preceding it must occur at the end of each line

. (dot) Matches any one character

\ (backslash) Ignores the special meaning of the character following it

* Zero or more occurrences of the previous character

(dot).* Nothing or any numbers of characters.

Example 1:

trueman

truman

$ grep "true*man" filename

Example 2:

sprintf

ssprintf

sssprintf

printf

$ grep "s*printf" filename

Example 3:

wilcocks

wilcox

$ grep "wilco[cx]k*s*" filename

Example 4:

$ grep 2... // matches a four character pattern beginning with 2

Example 5:

p.j.woodhouse

$ grep "p.*woodhouse" filename

Example 6:

$ grep "^2" filename //displays line starts with 2

$ grep "^[^2]" filename // displays line doesnot begin with 2

Example 7:

$ ls-l | grep "^d" // listing only directories

$ ls-l | grep '^.....w' // files have write permissions in group

$ls –l |grep " ^-" // Display list of regular files only

Example 8:

$grep "New[abc]" filename

It specifies the search pattern as : Newa , Newb or Newc

Example 9:

$grep "[aA]g[ar][ar]wal" filename

It specifies the search pattern as: Agarwal , Agaawal , Agrawal , Agrrwal, agarwal , agaawal , agrawal , agrrwal

Example 10:

$grep "New[a-e]" filename

It specifies the search pattern as: Newa , Newb or Newc , Newd, Newe

$grep "New[0-9][a-z]" filename

It specifies the search pattern as: New followed by a number and then an alphabet. New0d, New4f etc

Example 11:

$grep "^san" filename

Search lines beginning with san. It specifies the search pattern as: sanjeev ,sanjay, sanrit , sanchit , sandeep etc.

$grep "New[^a-c]" filename

It specifies the pattern containing the word "New" followed by any character other than an 'a','b', or 'c'

$grep "^[^a-z A-Z]" filename

Search lines beginning with an non-alphabetic character

$ grep "vedik$" file.txt // pattern preceding it must occur at the end of each line

Example 12:

. (dot): Matches any one character

$ grep "..vik" file.txt

$ grep "7..9$" file.txt

Example 13:

\ (backslash): Ignores the special meaning of the character following it

$ grep "New\.\[abc\]" file.txt

It specifies the search pattern as New.[abc]

$ grep "S\.K\.Kumar" file.txt

It specifies the search pattern as S.K.Kumar

Example 14:

***: zero or more occurrences of the previous character**

$ grep "[aA]gg*[ar][ar]wal" file.txt

Example 15:

(dot).*: Nothing or any numbers of characters.

$ grep "S.*Kumar" file.txt

ERE (Extended Regular Expression -E)

- possible to match dissimilar pattern with a single character

Expression	Meaning
g+	Matches atleast one g (g,gg,ggg,.....)
g?	Matches nothing or one g
gif|jpeg	Matches gif or jpeg
(lock|ver)wood	Matches lockwood or verwood

Example1:

Matching multiple Pattern

woodhouse

woodcock

$grep -E 'woodhouse|woodcock' filename

$ grep -E 'wood(house|cock)' filename

Example2:

woodhouse

woodcock

wilcocks

wilcox

$ grep -E 'wilco[cx]k*s*|wood(house|cock)' filename

fgrep (-f) (Fixed Grep)

- fgrep searches files for one or more pattern arguments.
- It does not use regular expressions; instead, it does direct string comparison to find matching lines of text in the input.
- egrep works in a similar way, but uses R.E

Example:

File1	**File2**
appl\e	apple
pur*poses	purposes
ball	mango
cat	cat

$grep -f file1 file2

Output:

apple

purposes

cat

$fgrep -f file1 file2

Output:

Cat

sed: The Stream Editor

- sed is a multipurpose tool that combines the work of several filters.
- sed is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- sed command in unix supports regular expression which allows it perform complex pattern matching.
- sed uses instructions to act on text. An instruction combines an address forselecting lines, with an action to be taken on them.

Syntax: $ sed options 'address action' file(s)

Addressing in sed is done in two ways:

• By one or two line numbers (like 3, 7).

• By specifying a /-enclosed pattern which occurs in a line (like /From:/).

- Address specifies either one line number to select a single line or a set of two (3,7) to select a group of contiguous lines.
- The action component can be internal commands or an editing function like insertion, deletion, or substitution of text.

Command	Description
i,a,c	Inserts, appends, and changes text
d	Deletes line(s)
p	Prints line(s) on standard output
q	Quits after reading up to addressed line
r	flname Places contents of file flname after line
w	flname Writes addressed lines to file flname
=	Prints line number addressed
s/s1/s2/ s2	Replaces first occurrence of expression s1 in all lines with expression
s/s1/s2/g	As above but replaces all occurrences

Examples:

Command	Description
1,4d	Deletes lines 1 to 4
10q	Quits after reading the first 10 lines
3,$p	Prints lines 3 to end (-n option required)
$!p	Prints all lines except last line (-n option required)
/begin/,/end/p	Prints line containing begin through line containing end
10,20s/-/:/	Replaces first occurrence of - in lines 10 to 20 with a :
s/echo/printf/g	Replaces all occurrences of echo in all lines with printf

Line Addressing:

Consider, the instruction 3q can be broken down into the address 3 and the action q (quit).

$sed '3q' filename #Quits after line number 3 or head -n 3 filename

$sed -n '1,2p' filename #prints the first two lines

$sed -n '$p' filename #select the last line of file

Selecting Lines from Anywhere: sed can also select a contiguous group of lines from any location.

$sed -n '9,11p' filename // To select lines 9 through 11

Selecting Multiple Groups of Lines: sed is not restricted to selecting only one group of lines.

sed -n '1,2p

7,9p

$p' filename

or

$ sed -n '1,2p;7,9p;$p' filename

Negating the Action (!)

selecting the first two lines is the same as not selecting lines 3 through the end.

$sed -n '3,$!p' filename #Don't print lines 3 to end

sed Options: sed supports only three options (-n, -e, and -f).

Multiple Instructions in the Command Line (-e)

The -e option allows you to enter as many instructions as you wish, each preceded by the option.

$ sed -n -e '1,2p' -e '7,9p' -e '$p' filename

Instructions in a File (-f)

When you have too many instructions to use or when you have a set of common instructions that you execute often, they are better stored in a file.

For ex: the above three instructions can be stored in a file, with each instruction on a separate line.

$ cat instructions

1,2p

7,9p

$p

$sed -n -f instructions filename

Note: we can use the -f option with multiple files and can also combine the -e and -f options as many times as you want.

$sed -n -f instr1 -f instr2 filename

$sed -n -e '/wilcox/p' -f instr1 -f instr2 filename

Context Addressing:

1) We can specify a pattern (or two) rather than line numbers, where the pattern has a / on either side.

$sed -n '/wilco[cx]k*s*/p' filename

output:

wilcox or wilcocks

$sed -n "/o'br[iy][ae]n/p;/lennon/p" filename

output:

Either the o'brien or lennon

2) We can also specify a comma-separated pair of context addresses to select a group of contiguous lines.

$sed -n '/johnson/,/lightfoot/p' filename

$sed -n '1,/woodcock/p' filename

$ls -l | sed -n '/^.....w/p' // list files which have write permission for the group

Examples

$cat file.txt

unix is great os. unix is opensource. unix is free os.

learn operating system.

unix linux which one you choose.

unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

Example1:

$sed 's/unix/linux/' file.txt //replaces the word "unix" with "linux" in the file.

Output:

linux is great os. unix is opensource. unix is free os.

learn operating system.

linux linux which one you choose.

linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

Example2: Replacing the nth occurrence of pattern in a line

$sed 's/unix/linux/2' file.txt

// replaces the second occurrence of the word "unix" with "linux" in a line.

Output:

unix is great os. linux is opensource. unix is free os.

learn operating system.

unix linux which one you choose.

unix is easy to learn.linux is a multiuser os.Learn unix .unix is a powerful.

Example3: Replacing all the occurrence of the pattern in line

$sed 's/unix/linux/g' file.txt # g (global replacement)

Output:

linux is great os.linux is opensource.linux is free os.

learn operating system.

linux linux which one you choose.

linux is easy to learn.linux is a multiuser os.Learn linux .linux is a powerful.

Example4: Replacing from nth occurrence to all occurrences in a line

$sed 's/unix/linux/3g' file.txt

//replaces the third, fourth, fifth…. “unix” word with “linux” word in a line.

Output:

unix is great os. unix is opensource. linux is free os.

learn operating system.

unix linux which one you choose.

unix is easy to learn.unix is a multiuser os.Learn linux .linux is a poweful.

Example5: Parenthesize first character of each word

$echo "Welcome To The Linux Stuff" | sed 's/\(\b[A-Z]\)/\(\1\)/g'

// prints the first character of every word in parenthesis.

Output:

(W)elcome (T)o (T)he (L)inux (S)tuff

Example6: Replacing string on a specific line number

$sed '3 s/unix/linux/' file.txt

Output:

unix is great os. unix is opensource. unix is free os.

learn operating system.

linux linux which one you choose.

unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

Example7: Replacing string on a range of lines

$sed '1,3 s/unix/linux/' file.txt

Output:

linux is great os. unix is opensource. unix is free os.

learn operating system.

linux linux which one you choose.

unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

$sed '2,$ s/unix/linux/' file.txt

#replaces the text from second line to last line in the file.

Example8: Deleting lines from a particular file

SED command is used for performing deletion operation without even opening the file

1. To Delete a particular line say n in this example

$ sed '5d' filename.txt

2. To Delete a last line

$ sed '$d' filename.txt

3. To Delete line from range x to y

$ sed '3,6d' filename.txt

4. To Delete from nth to last line

$ sed '12,$d' filename.txt

5. To Delete pattern matching line

$ sed '/abc/d' filename.txt

Text Editing

Append Lines Using Sed Command

- Sed provides the command "a" which appends a line after every line with the address or pattern.

Syntax:

#sed 'ADDRESS a\

Line which you want to append' filename

#sed '/PATTERN/ a\

Line which you want to append' filename

Examples:

$cat data.txt

Linux Sysadmin

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Windows- Sysadmin, reboot etc.

Ex 1) Add a line after the 3rd line of the file

Add the line "Cool gadgets and websites" after the 3rd line.

$ sed '3 a\

> Cool gadgets and websites' data.txt

Output:

Linux Sysadmin

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Cool gadgets and websites

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Windows- Sysadmin, reboot etc.

Ex 2) Append a line after every line matching the pattern

The below sed command will add the line "Linux Scripting" after every line that matches the pattern "Sysadmin".

**$ sed '/Sysadmin/a **

> Linux Scripting' data.txt

Output:

Linux Sysadmin

Linux Scripting

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Windows- Sysadmin, reboot etc.

Linux Scripting

Ex 3) Append a line at the end of the file

The following example, appends the line "Website Design" at the end of the file.

$ sed '$ a

> Website Design' data.txt

Output:

Linux Sysadmin

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Windows- Sysadmin, reboot etc.

Website Design

Insert Lines Using Sed Command

- Sed command "i" is used to insert a line before every line with the range or pattern.

Syntax:

#sed 'ADDRESS i\

Line which you want to insert' filename

#sed '/PATTERN/ i\

Line which you want to insert' filename

Examples:

Ex 1. Add a line before the 4th line of the line.

Add a line "Cool gadgets and websites" before 4th line. "a" command inserts the line after match whereas "i" inserts before match.

$ sed '4 i

> Cool gadgets and websites' data.txt

Output:

Linux Sysadmin

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Cool gadgets and websites

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Windows- Sysadmin, reboot etc.

Ex 2. Insert a line before every line with the pattern

The below sed command will add a line "Linux Scripting" before every line that matches with the pattern called 'Sysadmin".

**$ sed '/Sysadmin/i **

> Linux Scripting' data.txt

Output:

Linux Scripting

Linux Sysadmin

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Linux Scripting

Windows- Sysadmin, reboot etc.

Ex3. Insert a line before the last line of the file.

Append a line "Website Design" before the last line of the file.

$ sed '$ i

> Website Design' data.txt

Output:

Linux Sysadmin

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Website Design

Windows- Sysadmin, reboot etc.

Replace Lines Using Sed Command

"c" command in sed is used to replace every line matches with the pattern or ranges with the new given line.

Syntax:

#sed 'ADDRESS c\

new line' filename

#sed '/PATTERN/ c\

new line' filename

Examples:

Ex 1) Replace a first line of the file

The below command replaces the first line of the file with the "The linux OS".

$ sed '1 c\

> The linux OS ' data.txt

Output:

The linux OS

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Windows- Sysadmin, reboot etc.

Ex 2) Replace a line which matches the pattern

Replace everyline which has a pattern "Linux Sysadmin" to "Linux Sysadmin – Scripting".

**$ sed '/Linux Sysadmin/c **

> Linux Sysadmin - Scripting' data.txt

Output:

Linux Sysadmin - Scripting

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Windows- Sysadmin, reboot etc.

Ex 3) Replace the last line of the file

Sed command given below replaces the last line of the file with "Last Line of the file".

$ sed '$ c

> Last line of the file' data.txt

Output:

Linux Sysadmin

Databases - Oracle, mySQL etc.

Security (Firewall, Network, Online Security etc)

Storage in Linux

Productivity (Too many technologies to explore, not much time available)

Last line of the file

Print Line Numbers Using Sed Command

"=" is a command in sed to print the current line number to the standard output.

Syntax:

#sed '=' filename

The above send command syntax prints line number in the first line and the original line from the file in the next line .

sed '=' command accepts only one address, so if you want to print line number for a range of lines, you must use the curly braces.

Syntax:

```
# sed -n '/PATTERN/,/PATTERN/ {

=

p

}' filename
```

Example 1. Find the line number which contains the pattern

The below sed command prints the line number for which matches with the pattern "Databases"

$ sed -n '/Databases/=' data.txt

Output:

2

Example 2. Printing Range of line numbers

Print the line numbers for the lines matches from the pattern "Oracle" to "Productivity".

$ sed -n '/Oracle/,/Productivity/{

> =

> p

> }' data.txt

Output:

2

Databases - Oracle, mySQL etc.

3

Security (Firewall, Network, Online Security etc)

4

Storage in Linux

5

Productivity (Too many technologies to explore, not much time available)

Example 3. Print the total number of lines in a file

Line number of the last line of the file will be the total lines in a file. Pattern $ specifies the last line of the file.

$ sed -n '$=' data.txt

Output:

6

Substitution with the sed command

1) Change the first occurrence of the pattern

$sed 's/life/leaves/' a.txt

2) Replacing the nth occurrence of a pattern in a line

$sed 's/to/two/2' a.txt

3) Replacing all the occurrence of the pattern in a line.

$Sed 's/life/learn/g' a.txt

4) Replace pattern from nth occurrence to all occurrences in a line.

Syntax: $sed 's/old_pattern/new_pattern/ng' filename

$ sed 's/to/TWO/2g' a.txt

5) Replacing pattern on a specific line number. Here, "m" is the line number.

$ sed '3 s/every/each/' a.txt

6) Replace string on a defined range of lines –

$sed '2,5 s/to/TWO/' a.txt

7) To replace multiple spaces with a single space

$sed 's/ */ /g' filename

8) Replace one pattern followed by the another pattern

$sed '/is/ s/live/love/' a.txt

9) Replace a pattern with other except in the nth line.

$sed -i '5!s/life/love/' a.txt

PRINT OR VIEW THE FILES

$ cat a.txt

life isn't meant to be easy, life is meant to be lived.

Try to learn & understand something new everyday in life.

Respect everyone & most important love everyone.

Don't hesitate to ask for love & don't hesitate to show love too.

Life is too short to be shy.

In life experience will help you differentiating right from wrong

1) Viewing a file from x to y range

$sed -n '2,5p' a.txt

2) View the entire file except the given range

$sed '2,4d' a.txt

3) Print nth line of the file

$sed -n '4'p a.txt

4) Print lines from xth line to yth line.

$sed -n '4,6'p a.txt

5) Print only the last line

$sed -n '$'p filename

6) Print from nth line to end of file

$sed -n '3,$'p a.txt

Pattern Printing

7) Print the line only which matches the pattern

Syntax: $sed -n /pattern/p filename

$sed -n /every/p a.txt

8) Print lines which matches the pattern i.e from input to xth line.

$sed -n '/everyone/,5p' a.txt

9) Prints lines from the xth line of the input, up-to the line which matches the pattern.

If the pattern doesn't found then it prints up-to end of the file.

$sed -n '1,/everyone/p' a.txt

10) Print the lines which matches the pattern up-to the next xth lines

$ sed -n '/learn/,+2p' a.txt

awk

awk abbreviated for– Aho, Weinberger, and Kernighan

1) Awk is a scripting language used for manipulating data and generating reports.

2) The awk command programming language requires no compiling, and allows the user to use variables, numeric functions, string functions, and logical operators.

3) The programs are written in the form of statements that define text patterns that are to be searched for in each line of a document and the action that is to be taken when a match is found within a line.

Syntax:

awk options 'selection_criteria {action }' input-file

Example:

Consider the following text files as the input file

$ cat > empl

2233:charles harris :g.m. :sales :12/12/52: 90000

9876:bill johnson :director :production :03/12/50: 130000

5678:robert dylan :d.g.m. :marketing :04/19/43: 85000

2365:john woodcock :director :personnel :05/11/47: 120000

5423:barry wood :chairman :admin :08/30/56: 160000

1006:gordon lightfoot :director :sales :09/03/38: 140000

6213:michael lennon :g.m. :accounts :06/05/62: 105000

1265:p.j. woodhouse :manager :sales :09/12/63: 90000

4290:neil o'bryan :executive :production :09/07/50: 65000

2476:jackie wodehouse :manager :sales :05/01/59: 110000

6521:derryk o'brien :director :marketing :09/26/45: 125000

3212:bill wilcocks :d.g.m. :accounts :12/12/55: 85000

3564:ronie trueman :executive :personnel :07/06/47: 75000

2345:james wilcox :g.m. :marketing :03/12/45: 110000

0110:julie truman :g.m. :marketing :12/31/40: 95000

$cat > employee.txt

ajay manager account 45000

sunil clerk account 25000

varun manager sales 50000

amit manager account 47000

tarun peon sales 15000

deepak clerk sales 23000

sunil peon sales 13000

satvik director purchase 80000

1) By default Awk prints every line of data from the specified file.

$ awk '{print}' employee.txt

Output:

ajay manager account 45000

sunil clerk account 25000

varun manager sales 50000

amit manager account 47000

tarun peon sales 15000

deepak clerk sales 23000

sunil peon sales 13000

satvik director purchase 80000

2) Print the lines which matches with the given pattern.

$ awk '/manager/ {print}' employee.txt

output:

ajay manager account 45000

varun manager sales 50000

amit manager account 47000

3) Splitting a Line Into Fields:

The awk command splits the record delimited by whitespace character by default and stores it in the $n variables.

$0 represents the whole line.

$ awk '{print $1,$4}' employee.txt

Output:

ajay 45000

sunil 25000

varun 50000

amit 47000

tarun 15000

deepak 23000

sunil 13000

satvik 80000

4) Tests for exact match on second field

awk '$2 ~ /^manager$/ { print }' employee.txt

5) fourth field greater than 20000

awk '$4 > 20000 { print }' employee.txt

Built In Variables in awk

NR: keeps a current count of the number of input records.

$ awk '{print NR,$0}' employee.txt

Output:

1 ajay manager account 45000

2 sunil clerk account 25000

3 varun manager sales 50000

4 amit manager account 47000

5 tarun peon sales 15000

6 deepak clerk sales 23000

7 sunil peon sales 13000

8 satvik director purchase 80000

NF: keeps a count of the number of fields within the current input record.

$ awk '{print $1,$NF}' employee.txt

$1 represents Name and $NF represents Salary i.e last field.

Output:

ajay 45000

sunil 25000

varun 50000

amit 47000

tarun 15000

deepak 23000

sunil 13000

satvik 80000

(Display Line From 3 to 6)

$ awk 'NR==3, NR==6 {print NR,$0}' employee.txt

Output:

3 varun manager sales 50000

4 amit manager account 47000

5 tarun peon sales 15000

6 deepak clerk sales 23000

FS: Field Separator, contains the field separator character which is used to divide fields on the input line.

Example:

awk 'BEGIN { FS = ":" } { print $1, $2 }' empl

awk -F ':' '{ print $1, $2 }' empl

awk 'BEGIN { FS = ":" } { print $1 "\t" $2 "\t" $3 }' empl

RS: Record Separator, stores the current record separator character.

Example:

```
awk ' BEGIN { FS = "\n"; RS = "\n\n"} { print $1, $4 } ' address
```

OFS: OUTPUT Field Separator, stores the output field separator, which separates the fields when Awk prints them.

Example:

```
awk 'BEGIN { FS = ":"; OFS = "|" } { print $1, $3 }' empl
```

```
awk 'BEGIN { FS = ":"; OFS = "|"; ORS = "\n\n" } { print $1, $3 }' empl
```

ORS: Output Record Separator, stores the output record separator, which separates the output lines when Awk prints them.

Example:

```
awk ' BEGIN { FS = "\n"; RS = "\n\n"; ORS = "\n\n" } { print $1, $4 } ' address
```

ARGC:

```
awk 'BEGIN {print "Args=", ARGC}' hi hello welcome to LINUX
```

```
Args= 6
```

In the above example, we passed 5 arguments. But we got the output as 6. Here, it considers the arguments like below:

Arg[0] = awk

Arg[1] = hi

Arg[2] = hello

Arg[3] = welcome

Arg[4] = to

Arg[5] = linux

This means, by default Arg[0] will store the command nameas an argument. So total arguments count will be: **Arguments + Command Name**

ARGV:

```
awk 'BEGIN {
  for (i = 0; i < ARGC; ++i) {
    printf "ARGV[%d]=\"%s\"\n", i, ARGV[i]
```

```
  }
}' arg1 arg2 arg3
```

OUTPUT:

```
ARGV[0]="awk"
ARGV[1]="arg1"
ARGV[2]="arg2"
ARGV[3]="arg3"
```

FNR: NUMBER OF RECORDS IN A FILE

```
awk '{print FNR}' STUDENT
awk '{print FNR}' student emp
```

options:

-F option to specify the delimiter (:) whenever we select fields from this file.

$ awk -F: '/sales/ { print $2, $1 }' empn.lst

```
charles harris 2233
gordon lightfoot 1006
p.j. woodhouse 1265
jackie wodehouse 2476
```

$ awk -F: '{ $4 = "" ; print }' empl | head -n 2

```
2233 charles harris g.m. 12/12/52 90000
9876 bill johnson director 03/12/50 130000
```

Variables and Expressions

Expressions comprise strings, numbers, variables, and entities that are built by combining them with operators.

Example: (x + 5)*12 is an expression.

1) awk doesn't have char, int, long, double, and primitive data types

2) Every expression can be interpreted either as a string or a number

3) awk also allows the use of user-defined variables but without declaring them.

4) Variables are case-sensitive: x is different from X.

5) awk variables don't use the $ either in assignment or evaluation:

x = "5"

print x

6) Strings in awk are double-quoted and can contain any character

7) awk strings can also use escape sequences and octal values

x ="\t\tBELL\7" #Prints two tabs, the string BELL and sounds a beep

print x

8) awk provides no operator for concatenating strings. Strings are concatenated by simply placing them side-by-side:

```
x = "sun" ; y = "com"
print x y          # Prints suncom
print x "." y      # Prints sun.com
```

9) A numeric and string value can also be concatenated with equal ease

```
x = "5" ; y = 6 ; z = "A"
print x y          # y converted to string; prints 56
print x + y        # x converted to number; prints 11
print y + z        # z converted to numeric 0; prints 6
```

10) Expressions also have true and false values associated with them. Any nonempty string is true; so is any positive number.

Example:
if (x)
is true if x is a nonnull string or a positive number.

Operators

Arithmetic Operators

$ echo 22 7 | awk '{print $1/$2}'

3.14286

$ echo 22 7 | awk '{print $1+$2}'

$ echo 22 7 | awk '{print $1-$2}'

$ echo 22 7 | awk '{print $1*$2}'

$ echo 22 7 | awk '{print $1^$2}'

$ echo 22 7 | awk '{print $1%$2}'

Comparison and Logical Operators

awk has a single set of comparison operators for handling strings and numbers and two separate operators for matching regular expressions.

Operator	Significance
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>=	Greater than or equal to
>	Greater than
~	Matches a regular expression
!~	Doesn't match a regular expression
&&	Logical AND
||	Logical OR
!	Logical NOT

String and Numeric Comparison

Both numeric and string equality are tested with the == operator.

The operator != tests inequality.

Examples:

1) $ awk -F: 'NR == 3, NR == 6 { print NR, $2,$3,$6 }' emp

3 robert dylan d.g.m. 85000

4 john woodcock director 120000

5 barry wood chairman 160000

6 gordon lightfoot director 140000

2) $ awk -F: '$6 > 120000 { print $2, $6 }' emp

bill johnson 130000

barry wood 160000

gordon lightfoot 140000

derryk o'brien 125000

3) the operators >, < are also used to compare two strings.

x=0.0 ; y = 0
x > y Compared numerically; not true
$ echo 0.0 0 | awk '{ print ($1 < $2) ? "true" : "false" }'
x="0.0" ; y ="0"
x > y Compared as strings; true

x=0.0 ; y = "0"
x > y y converted to number; not true

~ and !~: The Regular Expression Operators

To match a string embedded in a field, you must use ~ instead of ==. Similarly, to negate a match, use !~ instead of !=.

1) $awk $2 ~ /wilco[cx]k*s*/ emp #Matches second field

2) $awk $3 !~ /director|chairman/ #Neither director nor chairman

3) $ awk -F: '$3 ~ /0/' /etc/passwd

root:x:0:0:root:/root:/bin/bash

ftp:x:40:49:FTP account:/srv/ftp:/bin/bash

uucp:x:10:14:Unix-to-Unix CoPy system:/etc/uucp:/bin/bash

sumit:x:500:100:sumitabha das:/home/sumit:/bin/bash

4) $ awk -F: '$3 ~ /^0$/' /etc/passwd

root:x:0:0:root:/root:/bin/bash

Logical Operators

awk supports three logical or boolean operators and expressions, and uses them to return true or false values. (&&, **||**, and !)

exp1 && exp2 True if both exp1 and exp2 are true.

exp1 **||** exp2 True if either exp1 or exp2 is true.

!exp True if exp is false.

Examples:

$3 == "director" **||** $3 == "chairman" Either director or chairman

$3 != "director" && $3 != "chairman" Neither director nor chairman

NR < 5 **||** NR > 10 Either lines 1 to 4 or 11 and above

$ awk -F: '$1 ~ /^root$/ || $3 == 4' /etc/passwd

root:x:0:0:root:/root:/bin/bash

lp:x:4:7:Printing daemon:/var/spool/lpd:/bin/bash

awk -F: '($3 > 1 && $3 < 4) || ($3 >=7 && $3 <=12)' /etc/passwd

The -f Option: Storing awk Programs in a File

Consider the following program, which is stored in

the file emp.awk

cat emp.awk

$6 > 120000 { print $2, $6 }

how to run:

$awk -F: -f emp.awk empl

bill johnson 130000

barry wood 160000

gordon lightfoot 140000

derryk o'brien 125000

The BEGIN and END Sections

1) The Begin section is denoted by the keyword BEGIN.

2) The instructions in BEGIN section are executed once before the awk actually executing program statements from body.

3) The instructions in BEGIN section are used for performing tasks such as initializing variables, displaying headings.

4) End Section is denoted by the keyword END.

5) The instructions in END section are executed once after program statements from the body are executed.

6) The instructions in END section are used for displaying results.

7) The BEGIN and END sections are optional.

Syntax:

```
BEGIN
{
        Instructions in body
}
END
{
        Instructions
}
```

Example:

```
BEGIN
{
        a = 10
        b = 20
}
{
        c = a+b
}
END { printf "Sum is: %d \n", c}
```

Example: emp.awk

```
BEGIN {
printf "\t\tEmployee abstract\n\n"
} $6 > 120000 {                 # Increment variables for serial number and pay
count++ ; tot+= $6              # Multiple assignments in one line
printf "%3d %-20s %-12s %d\n", count,$2,$3,$6
}
END  {
printf "\n\tThe average salary is %6d\n", tot/count
}
```

Output:

```
$ awk -F: -f emp.awk empl

        Employee abstract

1 bill johnson              director 130000

2 barry wood                chairman  160000

3 gordon lightfoot director 140000

4 derryk o'brien     director 125000

      The average salary is 138750
```

Arrays:

An Array is a collection of elements under a single variable name.

Awk supports two types of arrays. They are:

1) One Dimensional Indexed Arrays

2) Associative Arrays

One Dimensional Indexed Arrays:

The values of an array which can be accessed with the help of index number are called as Indexed Array.

Syntax: array_name[index] = "value"

Example:

Creating a File with awk source code:

```
$ cat>array.awk
BEGIN
{
a[1] = "ram";
a[2] = "sam";
a[3] = "bam";
print a[1];
print a[2];
print a[3];
}
```

Output: $awk –f array.awk

ram

sam

bam

Associative Arrays:

It is also same as Indexed Arrays. But in this array, instead of index number, we use a name to access array values.

Syntax: array_name[“name”] = “value”

Example:

Creating a File with awk source code:

$ cat>array2.awk

```
BEGIN
{
a[“name”] = "ram";
a[“roll”] = "007";
a[“dept”] = "CSE”;
print a[“name”];
print a[“roll”];
print a[“dept”];
}
```

Output: $awk –f array2.awk

ram

007

CSE

Control Structures in awk:

1) Decision Statements
2) Looping Statements

Decision Statements:

These will execute the statements by taking the decisions based on the conditions. There are three types of decision statements supported by awk.

1) if statement
2) if..else statement
3) else…if statement

if statement:

It executes the statements only if the condition is true. If the condition is false, just it comes out of the block.

Syntax:

```
if (condition)
{
code to be executed if condition is true;
                }
```

Example:

```
$cat > if.awk

BEGIN

{

print "Enter a Value:"

getline a

if(a>0)

print "Positive"

}
```

Output1: $awk –f if.awk

Enter a value: 6

Positive

Output2: $awk –f if.awk

Enter a value: -16

Nothing will be displayed

If…else Statement:

Here, it first checks the condition. If the condition is true, then executes the statements in the true block. If the condition is false, then executes the false block statements.

Syntax:

```
if (condition)
code to be executed if condition is true;
else
code to be executed if condition is false;
```

Example1:

```
$cat > ifelse.awk
BEGIN
{
print "Enter a Value:"
getline a
print "Enter b Value:"
getline b
if(a>b)
print "a is bigger than b"
else
        print “b is bigger than a”
}
```

Output: $awk –f ifelse.awk
Enter a value: 6
Enter b value: 5
a is bigger than b

Example2:

```
$cat >odd.awk
BEGIN
{
printf("Enter a Number: ")
getline x
if(x%2==0)
print "Even"
else
print "Odd"
}
```

Output: $awk –f odd.awk
Enter a Number: 5
Odd
Enter a Number: 6
Even

If…else…if Statement:

Here, it first checks the condition. If the condition is true, then executes the statements in the true block. If the condition is false, then goes for the next condition. Then verifies the condition. If the next condition is true, executes the true statements, else goes to next condition. Process will be continued. If none of the condition is satisfied, then executes the else block.

Syntax:

```
if (condition 1)
code to be executed if condition 1 is true;
else if (condition 2)
code to be executed if condition 2 is true;
else if (condition 3)
code to be executed if condition 3 is true;
else
code to be executed if all the conditions are false;
```

Example1:

```
$cat > maxofthree.awk
BEGIN
{
print "Enter a value:"
getline a
print "Enter b value:"
getline b
print "Enter c value:"
getline c
if(a>b && a>c)
print "a is bigger than b and c"
else if(b>a && b>c)
print "b is bigger than a and c"
else
print "c is bigger than a and b"
}
```

Output:

```
$awk -f maxofthree.awk
Enter a value: 11
Enter b value: 15
Enter c value: 10
b is bigger than a and c
```

Example2:

```
$cat > minofthree.awk
BEGIN
{
print "Enter a value:"
getline a
print "Enter b value:"
getline b
print "Enter c value:"
getline c
if(a<b && a<c)
```

```
print "a is smaller than b and c"
else if(b<a && b<c)
print "b is smaller than a and c"
else
print "c is smaller than a and b"
}
```

Output: $awk -f minofthree.awk
Enter a value: 10
Enter b value: 5
Enter c value: 8
b is smaller than a and c

Loop Statements:

Loop means executing the same statements number of times until the condition fails. There are different types of loops in awk. They are:

1) while loop
2) for loop

while loop:

Here it first checks the condition. If the condition is true, then it executes the statements in the loop block. Then it performs increment or decrement operation and again checks the condition. Then if the condition is true, executes the statements. Otherwise comes out of the loop. It is also called as "Entry Controlled Loop".

Syntax:

```
while (condition)
{
  Code to be executed;
}
```

Example1:

```
$cat > cubes.awk
BEGIN
{
print "Enter a:"
getline a
i = 1
cube = 1
while(i <= a)
{
cube = i*i*i
printf("Cube of %d is %d\n",i,cube)
```

```
i++
}
}
```

Output: $awk –f cubes.awk

```
Enter a: 15
Cube of 1 is 1
Cube of 2 is 8
Cube of 3 is 27
Cube of 4 is 64
Cube of 5 is 125
Cube of 6 is 216
Cube of 7 is 343
Cube of 8 is 512
Cube of 9 is 729
Cube of 10 is 1000
Cube of 11 is 1331
Cube of 12 is 1728
Cube of 13 is 2197
Cube of 14 is 2744
Cube of 15 is 3375
```

Example2:

```
$cat>fact.awk
BEGIN
{
print "Enter a:"
getline a
fact = 1
i = 1
while(i <= a)
{
fact = fact*i
i++
}
printf("Factorial of %d is %d",a,fact)
}
```

Output: $awk –f fact.awk

```
Enter a: 6
Factorial of 6 is 720
```

Example3:

```
$cat > natural.awk
BEGIN{}
{
```

```
print "Enter a: "
getline a
i = 0
while(i<=a)
{
print i
i++
}
}
```

Output: $awk –f natural.awk

```
Enter a: 10
1
2
3
4
5
6
7
8
9
10
```

For loop:

The for loop is used when you know how many times that the same statements should run.

Syntax:

```
for (initialization; test condition; increment/decrement)
{
code to be executed;
}
```

Example1:

```
$cat>sum.awk
BEGIN
{
print "Enter a:"
getline a
sum = 0
i = 1
for(i=0;i<=a;i++)
{
sum = sum + i
```

```
}
printf("Sum is : %d", sum)
}
```

Output: $awk –f sum.awk
Enter a: 10
Sum is : 55

Example2:

```
$cat>squares.awk
BEGIN{}
{
print "Enter a:"
getline a
i = 1
square = 1
for(i=1; i <= a; i++)
{
square = i*i
printf("Square of %d is %d\n", i, square)
}
}
```

Output: $awk –f squares.awk
Enter a: 10
Square of 1 is 1
Square of 2 is 4
Square of 3 is 9
Square of 4 is 16
Square of 5 is 25
Square of 6 is 36
Square of 7 is 49
Square of 8 is 64
Square of 9 is 81
Square of 10 is 100

Getline:

This is used to read the values from the user interactively. This acts like scanf in C Programming.

Syntax: getline variable_name

Example1:

```
$cat>number.awk
BEGIN
{
print "Enter a value:"
getline a
printf("You Entered: %d",a)
}
```

Output: $awk –f number.awk
Enter a value: 6
You Entered: 6

Example2:

```
$cat>sum.awk
BEGIN
{
print "Enter a value:"
getline a
print "Enter b value:"
getline b
c = a + b
printf("Sum is: %d",c)
}
```

Output: $awk –f sum.awk
Enter a value: 6
Enter b value: 5
Sum is: 11

2. Commands related to inode, I/O redirection and piping, process control commands, mails

inode:

Example: Deleting a file by its inode number

```
ARGCOUNT=1 # Filename arg must be passed to script.
WRONGARGS=70
FILE_NOT_EXIST=71
CHANGED_MIND=72
if [ $# -ne "$ARGCOUNT" ]
then
echo "Provide filename to Delete"
exit $WRONGARGS
fi
if [ ! -e "$1" ]
then
echo "Given Filename \""$1"\" does not exist."
exit $FILE_NOT_EXIST
fi
inum=`ls -i | grep "$1" | awk '{print $1}'`
echo; echo -n "Are you absolutely sure you want to delete \"$1\" (y/n)? "
read answer
case "$answer" in
[nN]) echo "Changed your mind, huh?"
exit $CHANGED_MIND
;;
*) echo "Deleting file \"$1\".";;
esac
find . -inum $inum -exec rm {} \;
echo "File "\"$1"\" deleted!"
exit 0
```

Output:

#when filename was not given
devasc@labvm:~$ ./ex.sh
Provide filename to Delete

#when the given filename was not exists.
devasc@labvm:~$ ./ex.sh file1
Given Filename "file" does not exist.

#when the given filename is exists.
devasc@labvm:~$ ./ex.sh filename

Are you absolutely sure you want to delete "filename" (y/n)? y
Deleting file "filename".
File "filename" deleted!

Explanation:
#How to find and delete a file with confirmation.

$ find ./GFG -name sample.txt -exec rm -i {} \;

When this command is entered, a prompt will come for confirmation, if you want to delete sample.txt or not. if you enter 'Y/y' it will delete the file.

I/O Redirection:

Example1: Redirecting stdin using exec

```
# Redirectingstdin using 'exec'.
exec 6<&0 # Link file descriptor #6 with stdin.
# Saves stdin.
exec< data-file # stdin replaced by file "data-file"
read a1 # Reads first line of file "data-file".
read a2 # Reads second line of file "data-file."
echo
echo "Following lines read from file."
echo "-----------------------------"
echo $a1
echo $a2
echo; echo; echo
```

```
exec 0<&6 6<&-
# Now restore stdin from fd #6, where it had been saved,
#+ and close fd #6 ( 6<&- ) to free it for other processes to use.
#
# <&6 6<&- also works.
echo -n "Enter data "
read b1 # Now "read" functions as expected, reading from normal stdin.
echo "Input read from stdin."
echo "----------------------"
echo "b1 = $b1"
echo
exit 0
```

Output:

```
devasc@labvm:~$ cat data-file
apple
mango
unix
linux

devasc@labvm:~$ ./ex.sh

Following lines read from file.
-------------------------------
apple
mango

Enter data grapes
Input read from stdin.
----------------------
b1 = grapes
```

Explanation:

In the Bash shell environment, every process has three files opened by default. These are standard input, display, and error. The file descriptors associated with them are 0, 1, and 2 respectively.

0 - stdin
1 - stdout
2 - stderr

The syntax for declaring output.txt as output is as follows:

exec fd > output.txt

#This command will declare the number fd as an output file descriptor.

The syntax for closing the file is as follows:

exec fd<&-

To close fd, which is 5, enter the following:

exec 5<&-

For Example,

The 3>&1 in your command line will create a new file descriptor and redirect it to 1 which is STDOUT. Now 1>&2 will redirect the file descriptor 1 to STDERR and 2>&3 will redirect file descriptor 2 to 3 which is STDOUT.

Example2: Redirecting stdout using exec

```
#!/bin/bash
# reassign-stdout.sh
LOGFILE=logfile.txt
exec 6>&1 #Link file descriptor #6 with stdout.
# Saves stdout.
exec> $LOGFILE #  stdout replaced with file "logfile.txt".
# ---------------------------------------------------- #
# All output from commands in this block sent to file $LOGFILE.
echo -n "Logfile: "
date
echo  "-----------------------------------"
echo
echo "Output of \"ls -al\" command"
echo
ls -al
echo; echo
echo "Output of \"df\" command"
```

```
echo
df
# ----------------------------------------------------- #
exec 1>&6 6>&- # Restore stdout and close file descriptor #6.
echo
echo "== stdout now restored to default == "
echo
ls  -al
echo
exit 0
```

Output: devasc@labvm:~$ ./ex.sh

== stdout now restored to default ==

```
total 324
drwxr-xr-x 24 devasc devasc  4096 Sep  3 06:09  .
drwxr-xr-x  3 root   root    4096 Jun 17  2020  ..
-rw-rw-r--  1 devasc devasc     0 Aug 27 05:40 '$'
-rw-rw-r--  1 devasc devasc    23 Aug 10 04:27  a
-rw-rw-r--  1 devasc devasc    26 Sep  3 05:46  a1
```

devasc@labvm:~$ cat logfile.txt

```
Logfile: Fri 03 Sep 2021 06:09:38 AM UTC
-----------------------------------
```

Output of "ls -al" command

```
total 320
drwxr-xr-x 24 devasc devasc  4096 Sep  3 06:09 .
drwxr-xr-x  3 root   root    4096 Jun 17  2020 ..
-rw-rw-r--  1 devasc devasc     0 Aug 27 05:40 $
-rw-rw-r--  1 devasc devasc    23 Aug 10 04:27 a
-rw-rw-r--  1 devasc devasc    26 Sep  3 05:46 a1
-rw-rw-r--  1 devasc devasc    17 Sep  3 05:46 a2
```

Output of "df" command

```
Filesystem     1K-blocks      Used Available Use% Mounted on
udev             1974736         0   1974736   0% /dev
tmpfs             403068      1196    401872   1% /run
/dev/sda5       31603552  12285264  17689872  41% /
tmpfs            2015328         0   2015328   0% /dev/shm
tmpfs               5120         0      5120   0% /run/lock
```

Piping:

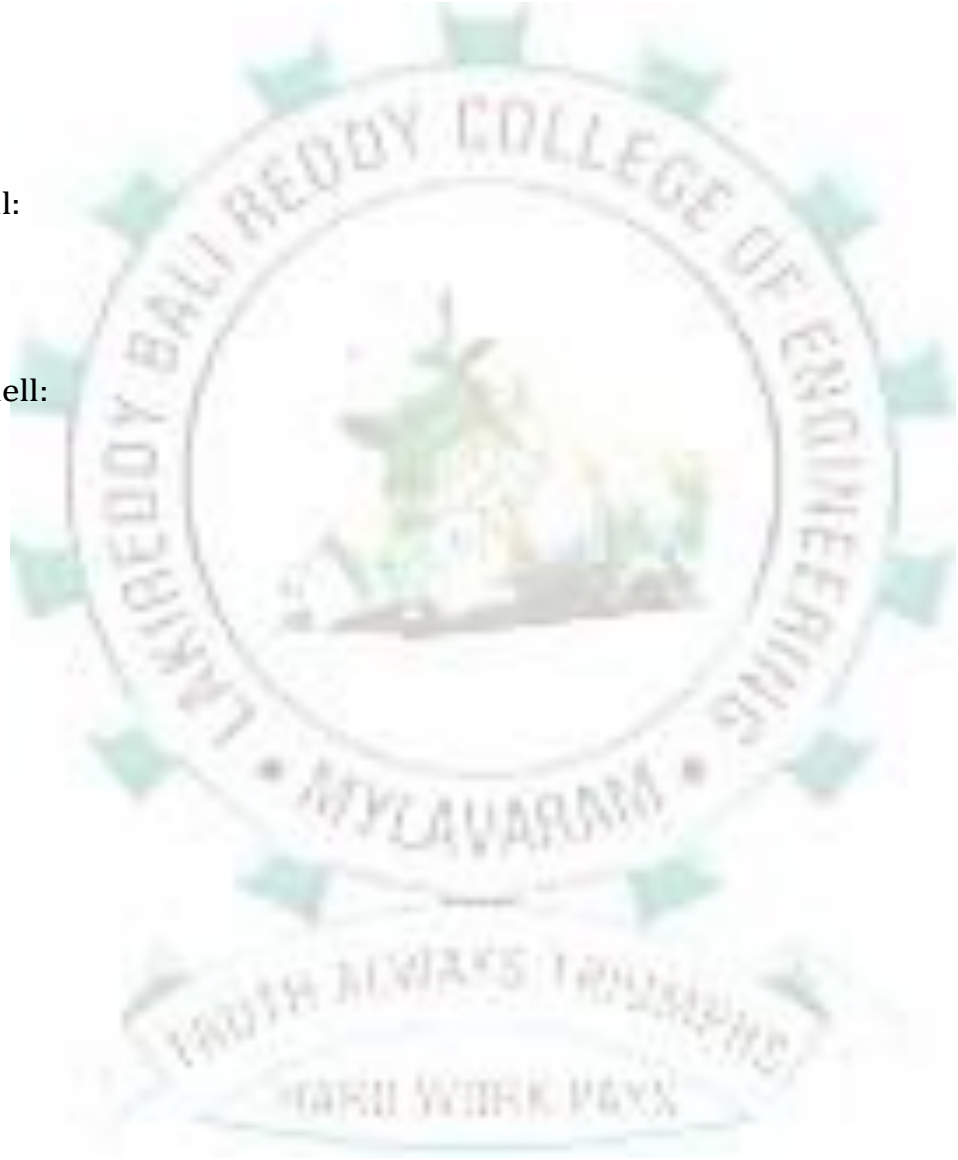
Example: Piping the output of echo to a read

```
a=apple
b=ball
c=cat
echo "one two three" | read a b c
# Try to reassign a, b, and c.
echo "a = $a" # a = apple
echo "b = $b" # b = ball
echo "c = $c" # c = cat
# Reassignment failed.
# ----------------------------
# Try the following alternative.
var=`echo "one two three"`
set -- $var
a=$1; b=$2; c=$3
echo "-------- "
echo "a = $a" # a = one
echo "b = $b" # b = two
echo "c = $c" # c = three
a=apple # Starting all over again.
b=ball
c=cat
echo; echo
echo "one two three" | ( read a b c;
echo "Inside subshell: "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
# a = one
# b = two
# c = three
echo "-----------------"
echo "Outside subshell: "
echo "a = $a" # a = apple
echo "b = $b" # b = ball
```

```
echo "c = $c" # c = cat

echo

exit 0
```

Output: devasc@labvm:~$ ./ex.sh

```
a = apple
b = ball
c = cat
-------
a = one
b = two
c = three

Inside subshell:
a = one
b = two
c = three
----------------
Outside subshell:
a = apple
b = ball
c = cat
```



4. Write a shell script to create a file. Follow the instructions

(i) Input a page profile to yourself, copy it into other existing file

Source Code:

```
echo "create a file in /home/devasc/profile in directory"
mkdir profile
echo "Present working DIRECTORY is"
cd profile
pwd
echo "Enter a file name"
read file1
echo "Enter Data contains in $file1"
cat > $file1
echo "Enter existing file name"
read file2
echo "Display copy of contains $file1 to $file2"
cp $file1 $file2
cat $file2
```

Output:

```
devasc@labvm:~$ ./ex.sh
create a file in /home/devasc/profile in directory
Present working DIRECTORY is
/home/devasc/profile
Enter a file name
file1
Enter Data contains in file1
G BALU NARASIMHARAO
SR.ASST.PROFESSOR
CSE DEPARTMENT
LBRCE, MYLAVARAM
Enter existing file name
FILE2
Display copy of contains file1 to FILE2
G BALU NARASIMHARAO
SR.ASST.PROFESSOR
```

CSE DEPARTMENT

LBRCE, MYLAVARAM

(ii) Start printing file at certain line

Source Code:

```
echo  "Enter the file name: "
read    file
echo    "Enter the starting line number: "
read    start
echo    "Enter the ending line number: "
read  end
tot=`expr    $end    -    $start    +    1`
echo
echo "Data    in    between    lines    are:"
head    -$end    $file|tail    -$tot
```

Output: devasc@labvm:~$ cat month

```
april
may
august
july
june
june
```

devasc@labvm:~$ ./ex.sh

```
Enter the file name:
month
Enter the starting line number:
2
Enter the ending line number:
5
Data in between lines are:
may
august
july
june
```

(iii) Print all the difference between two file, copy the two files.

Source Code:

```
echo "enter first file name"
read file1
echo "enter second file name"
read file2
echo "enter third file name"
read file3
echo "Enter Data contains in $file1"
cat > $file1
echo "Enter Data contains in $file2"
cat > $file2
echo "Display difference between $file1 and $file2 copy to $file3"
diff -a $file1 $file2 > $file3
cat $file3
```

Output: devasc@labvm:~$ ./ex.sh

```
enter first file name
file1
enter second file name
file2
enter third file name
file3
Enter Data contains in file1
apple
mango
grapes
orange
sapota
Enter Data contains in file2
apple
grapes
boy
girl
```

Display difference between file1 and file2 copy to file3

2d1

< mango

4,5c3,4

< orange

< sapota

> boy

> girl

(iv) Print lines matching certain word pattern.

Source Code:

```
echo "create a file "
read file1
echo "inputs data contains in file $file1"
cat > $file1
echo "Enter word we findout "
read f
grep -n $f $file1
```

Output:

```
devasc@labvm:~$ ./ex.sh
create a file
f1
inputs data contains in file f1
boy
good
men
man
Enter word we findout
men
3:men
```

5. Write shell script for-

(i) Showing the count of users logged in,

Source Code:

```
echo "list of all users who are login"
who
echo "count of all logins are"
who | wc -l
```

Output:

```
devasc@labvm:~$ ./ex.sh
list of all users who are login
devasc    tty7         2021-08-26 05:49 (:0)
count of all logins are
1
```

(ii) Printing Column list of files in your home directory

Source Code:

```
echo "list of files in a Home directory are:"
ls -l| cut -c 46-
```

Output:

```
devasc@labvm:~$ ./ex.sh
list of files in a Home directory are:
$
a
ab
apple
arrayex.sh
arraysex.sh
---
---
```

(iii) Listing your job with below normal priority

Source Code:

```
echo "list of normal priority"
ps -l | cut -c 13-18,30-32
```

Output:

```
list of normal priority
  PID PRI
27474   80
27557   80
27558   80
27821   80
27827   80
27838   80
```

(IV) Continue running your job after logging out.

Source Code:

```
# nohup command -with-options &
```

Note:- nohup (No Hang Up) is a command in Linux systems that runs the process even after logging out from the shell/terminal.

To run a command in the foreground:

```
$ nohup bash file.sh
```

To run a command in the background (with '&'):

```
$ nohup bash file.sh &
```

Output: devasc@labvm:~$ nohup bash file.sh &

```
[1] 3174

devasc@labvm:~$  nohup: ignoring input and appending output to 'nohup.out'
fg  # type fg to return to foreground process
bash: fg: job has terminated
[1]+  Done                    nohup bash file.sh
```

6. Write a shell script to change data format. Show the time taken in execution of this script.

Source Code:

```
echo "Enter file name"
read fname
echo "Input contains in $fname"
cat > fname
echo "Display created file current time"
ls -l $fname
echo "Modification of file $fname"
vi $fname
echo "show Access time "
ls -ult $fname
echo "show Modification time"
ls -clt $fname
```

Output:

```
devasc@labvm:~$ ./ex.sh
Enter file name
file11
Input contains in file11
linux
unix
Display created file current time
-rw-rw-r-- 1 devasc devasc 43 Sep  3 07:15 file11
Modification Time file11
show Access time
-rw-rw-r-- 1 devasc devasc 43 Sep  3 07:18 file11
show Modification time
-rw-rw-r-- 1 devasc devasc 43 Sep  3 07:18 file11
```

7. Write a shell script to print files names in a directory showing date of creation & serial number of the file.

Source Code:

```
echo "To display filenames, date of creation and serial number"
ls -li | awk '{print $1,$7,$8,$9,$10}'
```

8. Write a shell script to count lines, words, and characters in its input (do not use wc).

SOURCE CODE: Filename: count.awk

```
BEGIN { print "record.\t characters \t words"}
#BODY section
{
len=length($0)
total_len =len
print(NR,":\t",len,":\t",NF,$0)
words =NF
}
END{
print("\n total no of characters & letters are:")
print("characters :\t" total_len)
print("lines :\t" NR)
}
```

Output:

devasc@labvm:~$ cat data

```
shell scripting lab
linux programming lab
```

devasc@labvm:~$ awk -f count.awk data

```
record.         characters    words
1 :       19 :    3 shell scripting lab
2 :       21 :    3 linux programming lab
total no of characters & letters are:
characters : 21
lines :2
```