Advanced Shell Programming

Unix has a lot of filter commands like awk, grep, sed, spell, and wc.
A **filter** takes input from one command, does some processing, and gives output.
A filter is a Unix command that does some manipulation of the text of a file.
Two of the most powerful and popular Unix filters are the sed and awk commands.
Both of these commands are extremely powerful and complex.

Splitting Files

As the name suggests '**split**' command is used to split or break a file into the pieces in Linux and UNIX systems. (i.e.: split command in Unix is used to split a large file into smaller files.)
Whenever we split a large file with split command then split output file's default size is 1000 lines and its default prefix would be 'x'.
The splitting can be done on various criteria: on the basis of number of lines, or the number of output files or the byte count, etc.
Filter commands for splitting : **head, tail, cut and split.**

1. head :

The head command, as the name implies, print the top N number of data of the given input.
By default it prints the first 10 lines of the specified files.
If more than one file name is provided then data from each file is precedes by its file name.
If no FILE is specified, or when FILE is specified as a dash ("-"), head reads from standard input.

syntax:

head [option][filename(s)]

here option and argument is optional

- by default, head display top 10 line of a file.
- $ head f1 f2 f3

Example:

- $ head f1 f2 f3

```
= =>f1<= =        #header of file1
……
…..
…..10 lines of f1….
 = =>f2<= =        #header of file2
……
…..
…..10 lines of f3….
= =>f3<= =        #header of file3
……
…..
…..10 lines of f1….
```

- $ head –   #prints only 10 line from keyboard then $
- $ head -5**c** f1 f2 f3   # first 5character of each file
- $ head –n f1   # shows first n line of file
- $ head -1q f1 f2 f3

```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
```

```
usage : ./if.sh pattern file
nidhi    pts/1       Aug 26 02:29 (192.168.0.64)
```

- $ head -1 f1 f2 f3

```
==> f1 <==
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
==> f2 <==
usage : ./if.sh pattern file
==> f3 <==
neha    pts/1       Aug 26 02:29 (192.168.0.64)
```

Options with this command

Tag	Description
-c, --bytes=[-]num	**to print N bytes from each input file.** You can use the **-c** option to print the N number of bytes from the initial part of file. **$ head -c 5 flavours.txt** Ubuntu
-n, --lines=[-]num	**To print N lines from each input file.** To view the first N number of lines, pass the file name as an argument with -n option as shown below. **$ head -n 5 flavours.txt** Ubuntu Debian Redhat Gentoo Fedora core
-q, --quiet, --silent	**–quietPrevent printing of header information that contains file name** It is used if more than 1 file is given. Because of this command, data from each file is not precedes by its file name. **Without using -q option** ==> state.txt capital.txt <== Hyderabad Itanagar Dispur Patna Raipur Panaji Gandhinagar Chandigarh Shimla Srinagar **With using -q option** **$ head -q state.txt capital.txt** Andhra Pradesh Arunachal Pradesh Assam Bihar Chhattisgarh

	Goa Gujarat Haryana Himachal Pradesh Jammu and Kashmir Hyderabad Itanagar Dispur Patna Raipur Panaji Gandhinagar Chandigarh Shimla Srinagar
-v, --verbose	t**o print header information always.** By using this option, data from the specified file is always preceded by its file name. **$ head -v state.txt** ==> state.txt <== Andhra Pradesh Arunachal Pradesh Assam Bihar Chhattisgarh Goa Gujarat Haryana Himachal Pradesh Jammu and Kashmir

2. tail

The tail command, as the name implies, print the **last** N number of data of the given input.
By default it prints the last 10 lines of the specified files.
If more than one file name is provided then data from each file is precedes by its file name.

syntax:

tail [option][filename(s)]

- by default display last 10 lines of the file

Example

- $ tail f1 f2 f3   #display last 10 lines with filename as header
- With – as file, it reads std. input until a user press <ctrl+d> and then display last 10 lines from input.
- $ tail -5c f1   #display last 5 characters
- $ tail -3 f1   # display last 3 lines of file f1
- $ tail –q f1 f2 f3   # not show header filename
- $tail +5 f1   #display lines from 5th to last line of file f1.

Options with tail command:

Short Option	Long Option	Option Description
-c	–bytes	to print last N bytes from each input file
-f	–follow	to print appended data as and when the file grows
-n	–lines	to print last N lines from each input file
	–pid	with -f, to terminate after PID dies
-q	–silent, –quiet	to prevent printing of header information
	–retry	to keep retrying to open a file even when it is not exist or becomes inaccessible. Useful when it is used with -f
-s	–sleep-interval	to sleep for N seconds between iterations
-v	–verbose	to print header information always

3. cut

Cut command in unix (or linux) is used to select sections of text from each line of files.
You can use the cut command to select fields or columns from a line by specifying a delimiter or you can select a portion of text by specifying the range or characters.
Basically the cut command slices a line and extracts the text.
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**.
Basically the cut command slices a line and extracts the text.
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*]...**

OPTIONS:

Tag	Description
-b BYTE-LIST --bytes=BYTE-LIST	Print only the bytes in positions listed in BYTE-LIST. Tabs and backspaces are treated like any other character; they take up 1 byte.
-c CHARACTER-LIST --characters=CHARACTER-LIST	Print only characters in positions listed in CHARACTER-LIST. The same as '-b' for now, but internationalization will change that. Tabs and backspaces are treated like any other character; they take up 1 character.
-f FIELD-LIST --fields=FIELD-LIST	Print only the fields listed in FIELD-LIST. Fields are separated by a TAB character by default.
-d INPUT_DELIM_BYTE --delimiter=INPUT_DELIM_BYTE	For '-f', fields are separated in the input by the first character in INPUT_DELIM_BYTE (default is TAB).
-n	Do not split multi-byte characters (no-op for now).
-s	For '-f', do not print lines that do not contain the field separator

--only-delimited	character.
--output-delimiter=OUTPUT_DELIM_STRING	For '-f', output fields are separated by OUTPUT_DELIM_STRING The default is to use the input delimiter.

For most of the example, we'll be using the following test file.

```
$ cat test.txt
cat command for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
```

1. Select Column of Characters (-c)
To extract only a desired column from a file use -c option. The following example displays 2nd character from each line of a file test.txt

```
$ cut -c2 test.txt
a
p
s
```

As seen above, the characters a, p, s are the second character from each line of the test.txt file.

2. Select Column of Characters using Range
Range of characters can also be extracted from a file by specifying start and end position delimited with -. The following example extracts first 3 characters of each line from a file called test.txt

```
$ cut -c1-3 test.txt
cat
cp
ls
```

3. Select Column of Characters using either Start or End Position
Either start position or end position can be passed to cut command with -c option.
The following specifies only the start position before the '-'. This example extracts from 3rd character to end of each line from test.txt file.

```
$ cut -c3- test.txt
t command for file oriented operations.
 command for copy files or directories.
 command to list out files and directories with its attributes.
```

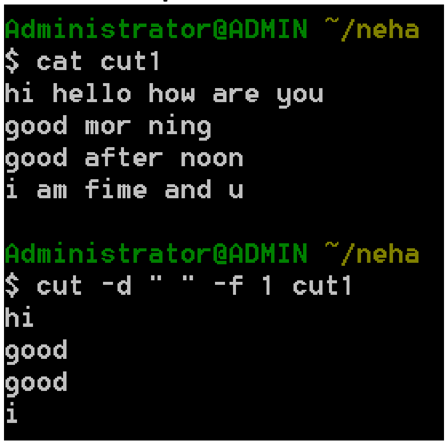
The following specifies only the end position after the '-'. This example extracts 8 characters from the beginning of each line from test.txt file.

```
$ cut -c-8 test.txt
cat comm
cp comma
ls comma
```

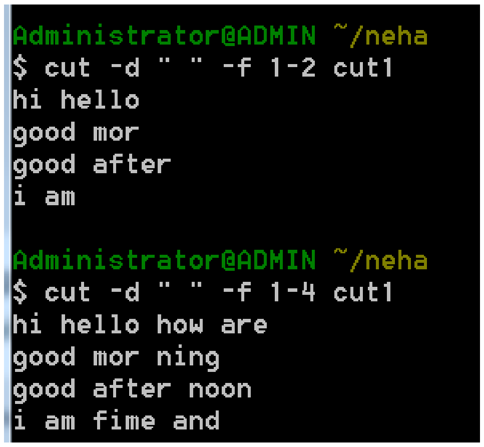
The entire line would get printed when you don't specify a number before or after the '-' as shown below.

```
$ cut -c- test.txt
cat command for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
```

4. Select a Specific Field from a File



```
Administrator@ADMIN ~/neha
$ cat cut1
hi hello how are you
good mor ning
good after noon
i am fime and u

Administrator@ADMIN ~/neha
$ cut -d " " -f 1 cut1
hi
good
good
i
```



```
Administrator@ADMIN ~/neha
$ cut -d " " -f 1-2 cut1
hi hello
good mor
good after
i am

Administrator@ADMIN ~/neha
$ cut -d " " -f 1-4 cut1
hi hello how are
good mor ning
good after noon
i am fime and
```

Instead of selecting x number of characters, if you like to extract a whole field, you can combine option -f and -d. The option -f specifies which field you want to extract, and the option -d specifies what is the field delimiter that is used in the input file.
The following example displays only first field of each lines from /etc/passwd file using the field delimiter : (colon). In this case, the 1st field is the username. The file

```
$ cut -d':' -f1 /etc/passwd
root
daemon
bin
sys
sync
games
bala
```

5. Select Multiple Fields from a File
You can also extract more than one fields from a file or stdout. Below example displays username and home directory of users who has the login shell as “/bin/bash”.

```
$ grep "/bin/bash" /etc/passwd | cut -d':' -f1,6
root:/root
```

```
bala:/home/bala
```

To display the range of fields specify start field and end field as shown below. In this example, we are selecting field 1 through 4, 6 and 7

```
$ grep "/bin/bash" /etc/passwd | cut -d':' -f1-4,6,7
root:x:0:0:/root:/bin/bash
bala:x:1000:1000:/home/bala:/bin/bash
```

6. Select Fields Only When a Line Contains the Delimiter
In our /etc/passwd example, if you pass a different delimiter other than : (colon), cut will just display the whole line.
In the following example, we've specified the delimiter as | (pipe), and cut command simply displays the whole line, even when it doesn't find any line that has | (pipe) as delimiter.

```
$ grep "/bin/bash" /etc/passwd | cut -d'|'  -f1
root:x:0:0:root:/root:/bin/bash
bala:x:1000:1000:bala,,,:/home/bala:/bin/bash
```

But, it is possible to filter and display only the lines that contains the specified delimiter using -s option.
The following example doesn't display any output, as the cut command didn't find any lines that has | (pipe) as delimiter in the /etc/passwd file.

```
$ grep "/bin/bash" /etc/passwd | cut -d'|' -s -f1
```

7. Select All Fields Except the Specified Fields
In order to complement the selection field list use option –complement.
The following example displays all the fields from /etc/passwd file except field 7

```
$ grep "/bin/bash" /etc/passwd | cut -d':' --complement -s -f7
root:x:0:0:root:/root
bala:x:1000:1000:bala,,,:/home/bala
```

8. Change Output Delimiter for Display
By default the output delimiter is same as input delimiter that we specify in the cut -d option.
To change the output delimiter use the option –output-delimiter as shown below. In this example, the input delimiter is : (colon), but the output delimiter is # (hash).

```
$ grep "/bin/bash" /etc/passwd | cut -d':'  -s -f1,6,7 --output-delimiter='#'
root#/root#/bin/bash
bala#/home/bala#/bin/bash
```

9. Change Output Delimiter to Newline
In this example, each and every field of the cut command output is displayed in a separate line.
We still used –output-delimiter, but the value is $'\n' which indicates that we should add a newline as the output delimiter.

```
$ grep bala /etc/passwd | cut -d':' -f1,6,7 --output-delimiter=$'\n'
bala
/home/bala
/bin/bash
```

4. split

split large files into a number of smaller files (i.e. Split a file into pieces.)

To split large files into smaller files in UNIX, use the split command.
At the Unix prompt, enter:

```
Syntax : split [options] filename prefix
```

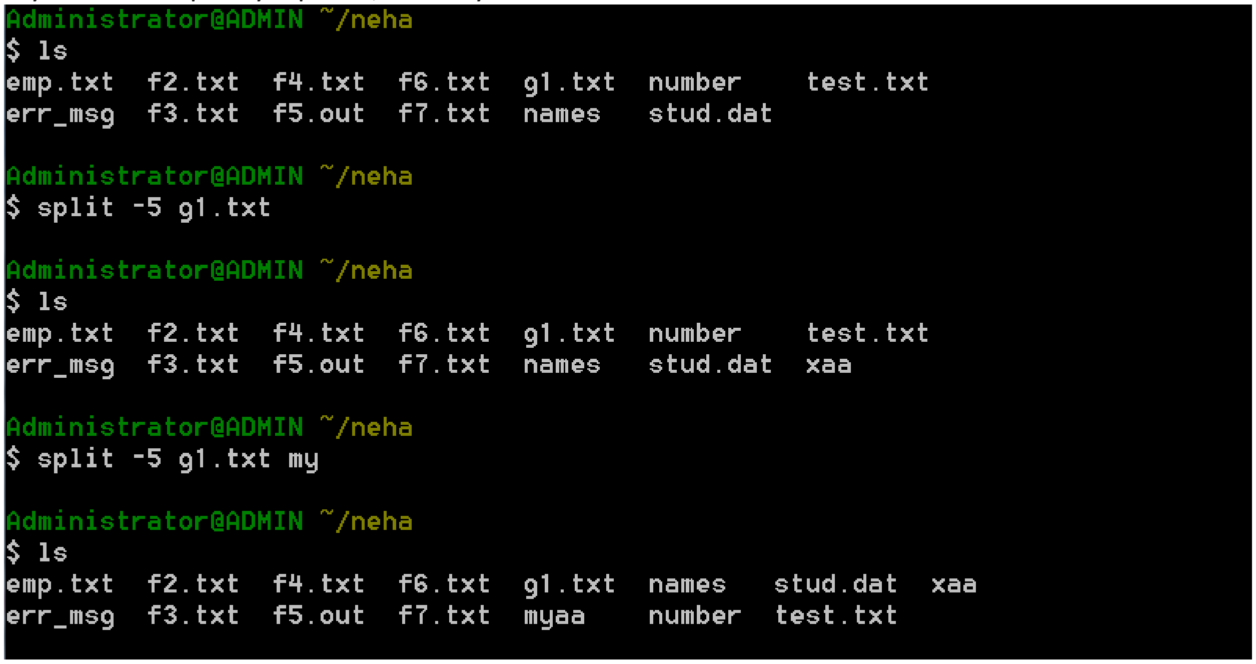
Replace filename with the name of the large file you wish to split.
Replace prefix with the name you wish to give the small output files.

You can exclude [options], or replace it with either of the following:

```
-l linenumber
-b bytes
```

If you use the -l (a lowercase L) option, replace linen umber with the number of lines you'd like in each of the smaller files (the default is 1,000).

If you use the -b option, replace bytes with the number of bytes you'd
like in each of the smaller files.
The split command will give each output file it creates the name prefix with an extension tacked to the end that indicates its order.
By default, the split command adds aa to the first output file, proceeding through the alphabet to zz for subsequent files.
If you do not specify a prefix, most systems use x.



```
Administrator@ADMIN ~/neha
$ ls
emp.txt  f2.txt  f4.txt  f6.txt  g1.txt  number    test.txt
err_msg  f3.txt  f5.out  f7.txt  names   stud.dat

Administrator@ADMIN ~/neha
$ split -5 g1.txt

Administrator@ADMIN ~/neha
$ ls
emp.txt  f2.txt  f4.txt  f6.txt  g1.txt  number    test.txt
err_msg  f3.txt  f5.out  f7.txt  names   stud.dat  xaa

Administrator@ADMIN ~/neha
$ split -5 g1.txt my

Administrator@ADMIN ~/neha
$ ls
emp.txt  f2.txt  f4.txt  f6.txt  g1.txt  names   stud.dat  xaa
err_msg  f3.txt  f5.out  f7.txt  myaa    number  test.txt
```

Options

Tag	Description
-a, **--suffix-length**=*N*	
	use suffixes of length N (default 2)
-b, **--bytes**=*SIZE*	
	put SIZE bytes per output file
-C, **--line-bytes**=*SIZE*	

	put at most SIZE bytes of lines per output file
-d, --numeric-suffixes	
	use numeric suffixes instead of alphabetic
-l, --lines=*NUMBER*	
	put NUMBER lines per output file
--verbose	
	print a diagnostic to standard error just before each output file is opened
--help	display this help and exit
--version	
	output version information and exit

Examples

- In this simple example, assume myfile is 3,000 lines long:

```
split myfile
```

  This will output three 1000-line files: xaa, xab, and xac.
- Working on the same file, this next example is more complex:

```
split -l 500 myfile segment
```

  This will output six 500-line files: segmentaa, segmentab, segmentac, segmentad, segmentae, and segmentaf.
- Finally, assume myfile is a 160KB file:

```
split -b 40k myfile segment
```

  This will output four 40KB files: segmentaa, segmentab, segmentac, and segmentad.

Sorting and merging files

1. Sort

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.

- SORT command sorts the contents of a text file, line by line.
- sort is a standard command line program that prints the lines of its input or concatenation of all files listed in its argument list in sorted order.
- The sort command is a command line utility for sorting lines of text files.
- It supports sorting alphabetically, in reverse order, by number, by month and can also remove duplicates.
- The sort command can also sort by items not at the beginning of the line, ignore case sensitivity and return whether a file is sorted or not.
- Sorting is done based on one or more sort keys extracted from each line of input.
- By default, the entire input is taken as sort key.
- Blank space is the default field separator.

The sort command follows these features as stated below:

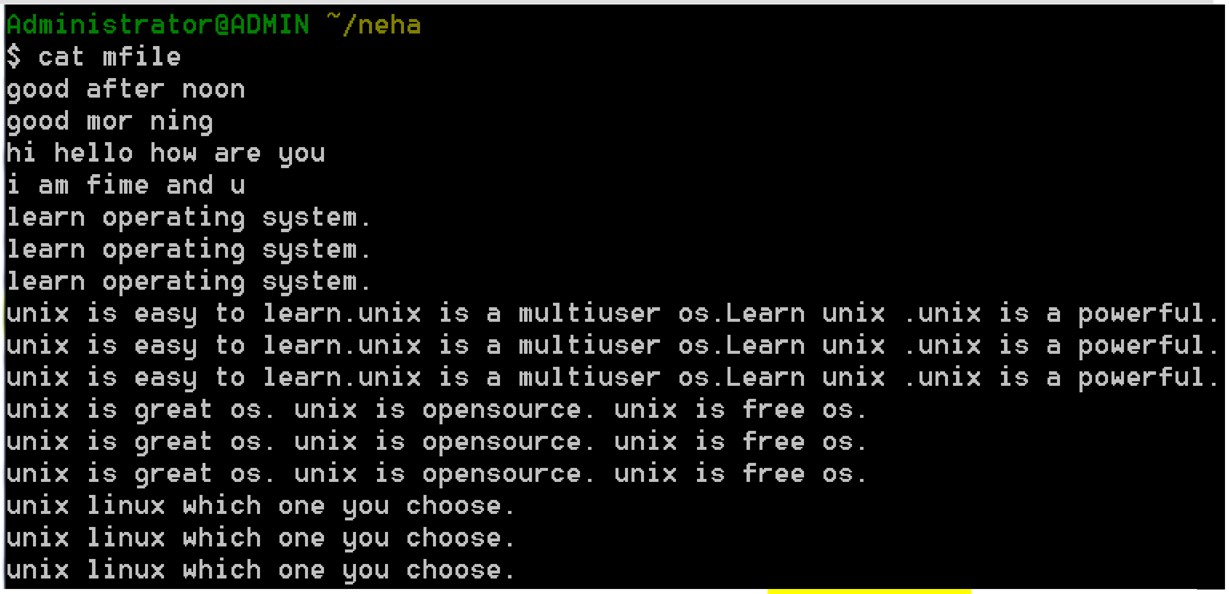
1. Lines starting with a number will appear before lines starting with a letter.
2. Lines starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet.
3. Lines starting with a lowercase letter will appear before lines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

Syntax : sort [*OPTION*]... [*FILE*]...

Options with sort function

1. **-o Option :** Unix also provides us with special facilities like if you want to write the **output to a new file**, output.txt, redirects the output like this or you can also use the built-in sort option -o, which allows you to specify an output file. Using the -o option is functionally the same as redirecting the output to a file. Note: Neither one has an advantage over the other. Example: The input file is the same as mentioned above.
Syntax :

```
$ sort inputfile.txt > filename.txt                    OR
$ sort -o filename.txt inputfile.txt
Command:
$ sort file.txt > output.txt
$ sort -o output.txt file.txt
$ cat output.txt
Output :
abhishek
chitransh
divyam
harsh
naveen
rajan
satish
```



```
Administrator@ADMIN ~/neha
$ cat mfile
good after noon
good mor ning
hi hello how are you
i am fime and u
learn operating system.
learn operating system.
learn operating system.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
unix is great os. unix is opensource. unix is free os.
unix is great os. unix is opensource. unix is free os.
unix is great os. unix is opensource. unix is free os.
unix linux which one you choose.
unix linux which one you choose.
unix linux which one you choose.
```

2. **-r Option: Sorting In Reverse Order** : You can perform a reverse-order sort using the -r flag. the -r flag is an option of the sort command which sorts the input file in reverse order i.e. descending order by default. Example: The input file is the same as mentioned above.
Syntax : **$ sort -r inputfile.txt**

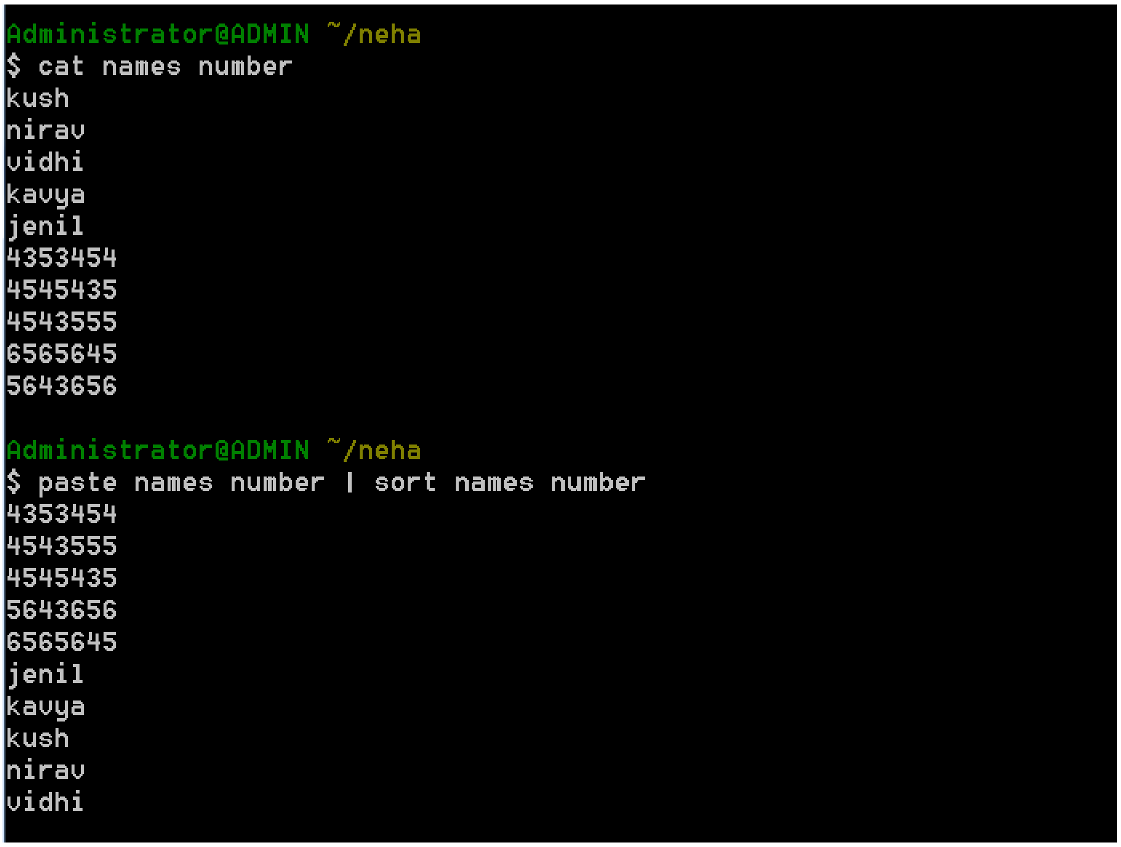
```
Command :
$ sort -r file.txt
Output :
satish
rajan
naveen
harsh
divyam
chitransh
abhishek
```

3. **-n Option** : To sort a file **numerically** used –n option. -n option is also predefined in unix as the above options are. This option is used to sort the file with numeric data present inside.
   Example: :
   Let us consider a file with numbers:

```
Command :
$ cat > file1.txt
50
39
15
89
200
```

Syntax : $ sort -n filename.txt

```
Command :
$ sort -n file1.txt
Output :
15
39
50
89
200
```



```
Administrator@ADMIN ~/neha
$ cat names number
kush
nirav
vidhi
kavya
jenil
4353454
4545435
4543555
6565645
5643656

Administrator@ADMIN ~/neha
$ paste names number | sort names number
4353454
4543555
4545435
5643656
6565645
jenil
kavya
kush
nirav
vidhi
```

4. **-nr option** : To sort a file with **numeric data in reverse order** we can use the combination of two options as stated below. Example :The numeric file is the same as above.
   Syntax : $ sort -nr filename.txt

```
Command :
$ sort -nr file1.txt
Output :
200
89
50
39
15
```

5. **-k Option** : Unix provides the feature of sorting a table on the **basis of any column number by using -k option.**
   Use the -k option to sort on a certain column. For example, use "-k 2" to sort on the second column.
   Example :
   Let us create a table with 2 columns

```
$ cat > employee.txt
manager  5000
clerk    4000
employee  6000
peon     4500
director 9000
guard    3000
```

Syntax : $ sort -k filename.txt

```
Command :
```

```
$ 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. This will write to standard output if there are lines that are out of order. The sort tool can be used to understand if this file is sorted and which lines are out of order
Example: :
Suppose a file exists with a list of cars called cars.txt.

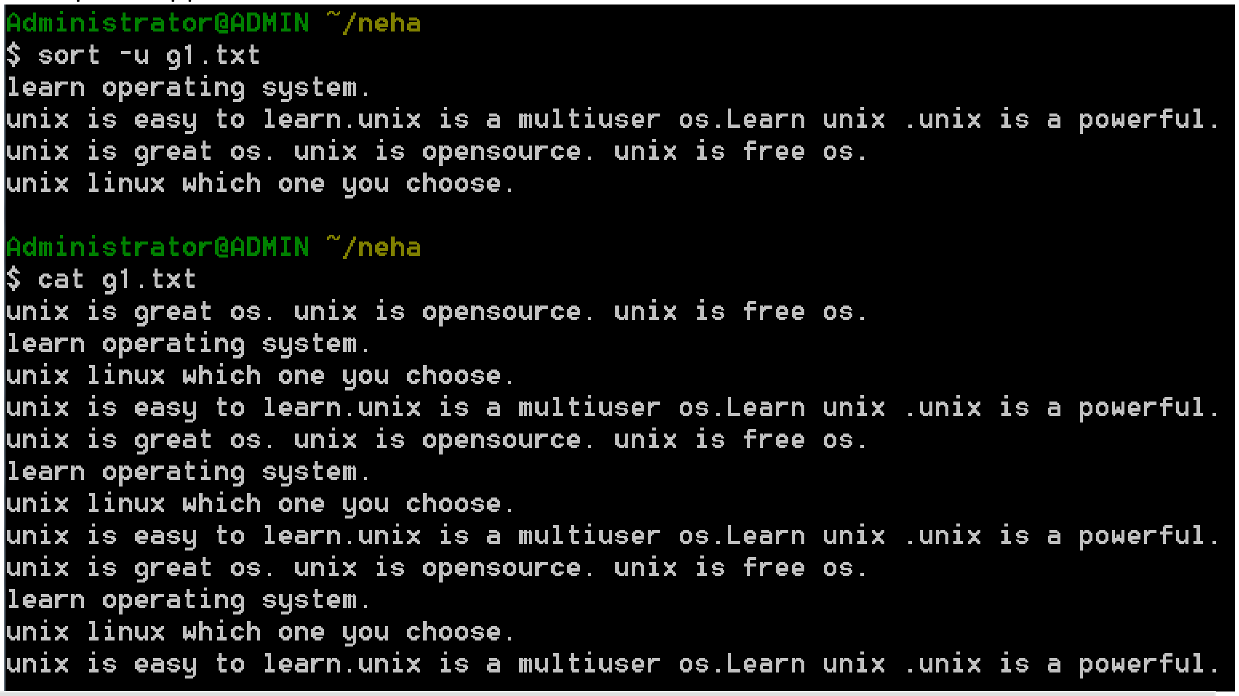
```
Audi
Cadillac
BMW
Dodge
```

Syntax : **$ sort -c filename.txt**

```
Command :
$ sort -c cars.txt
Output :
sort: cars.txt:3: disorder: BMW
```

Note : If there is no output then the file is considered to be already sorted

7. **-u option :** To **sort and remove duplicates** pass the -u option to sort. This will write a sorted list to standard output and remove duplicates.
This option is helpful as the duplicates being removed gives us an redundant file.
Example : Suppose a file exists with a list of cars called cars.txt.



```
Administrator@ADMIN ~/neha
$ sort -u g1.txt
learn operating system.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
unix is great os. unix is opensource. unix is free os.
unix linux which one you choose.

Administrator@ADMIN ~/neha
$ cat g1.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.
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.
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.
```

```
Audi
BMW
Cadillac
```

```
BMW
Dodge
```

Syntax : $ sort -u filename.txt

```
Command :
$ sort -u cars.txt
$ cat cars.txt
Output :
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.
Example:
Suppose the following file exists and is saved as months.txt

```
$ cat > months.txt
```

```
February
January
March
August
September
```

Using The -M option with sort allows us to order this file.

```
Command :
$ sort -M months.txt
$ cat months.txt
Output :
January
February
March
August
September
```

2. Paste

Paste command is one of the useful commands in unix or linux operating system.
The paste command **merges the lines from multiple files**.
The paste command sequentially writes the corresponding lines from each file separated by a **TAB** delimiter on the unix terminal.

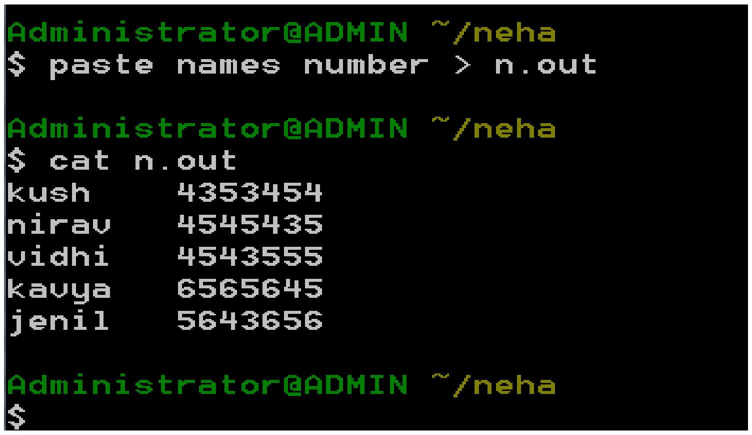
The syntax of the paste command is

```
paste [options] files-list
```

The options of paste command are:

```
-d : Specify of a list of delimiters.
-s : Paste one file at a time instead of in parallel.
```

```
--version : version information
--help : Help about the paste command.
```



```
Administrator@ADMIN ~/neha
$ paste names number > n.out

Administrator@ADMIN ~/neha
$ cat n.out
kush    4353454
nirav   4545435
vidhi   4543555
kavya   6565645
jenil   5643656

Administrator@ADMIN ~/neha
$
```

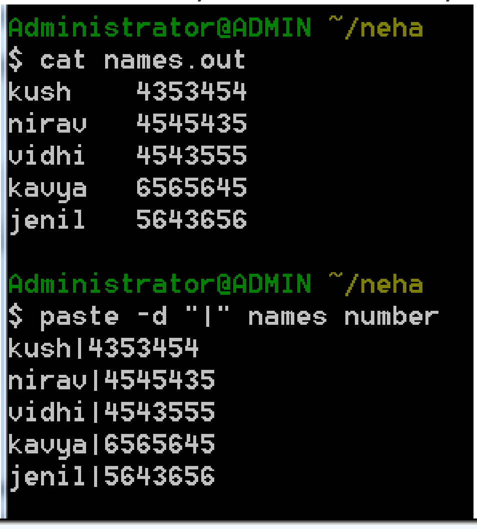
Paste Command Examples:
Create the following three files in your unix or linux servers to practice to practice the examples:

```
> cat file1          > cat file2               > cat file3
Unix                 Dedicated server          Hosting
Linux                Virtual server            Machine
Windows                                        Operating system
```

1. Merging files in parallel
By default, the paste command merges the files in parallel. The paste command writes corresponding lines from the files as a tab delimited on the terminal.

```
> paste file1 file2               |     > paste file2 file1
Unix        Dedicated server      |     Dedicated server   Unix
Linux       Virtual server        |     Virtual server     Linux
Windows                                                    Windows
```

2. Specifying the delimiter
Paste command uses the tab delimiter by default for merging the files. You can change the delimiter to any other character by using the -d option.



```
Administrator@ADMIN ~/neha
$ cat names.out
kush    4353454
nirav   4545435
vidhi   4543555
kavya   6565645
jenil   5643656

Administrator@ADMIN ~/neha
$ paste -d "|" names number
kush|4353454
nirav|4545435
vidhi|4543555
kavya|6565645
jenil|5643656
```

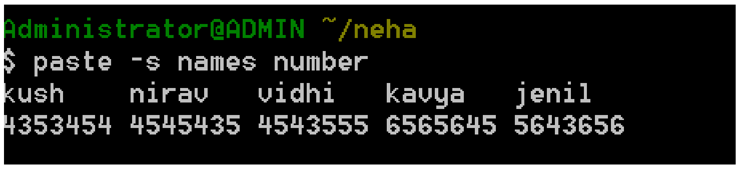
```
> paste -d"|" file1 file2
```

```
Unix|Dedicated server
Linux|Virtual server
Windows|
```

In the above example, pipe delimiter is specified

3. Merging files in sequentially.
You can merge the files in sequentially using the -s option. The paste command reads each file in sequentially. It reads all the lines from a single file and merges all these lines into a single line.

```
> paste -s file1 file2
Unix    Linux   Windows
Dedicated server        Virtual server
```



```
Administrator@ADMIN ~/neha
$ paste -s names number
kush    nirav   vidhi   kavya   jenil
4353454 4545435 4543555 6565645 5643656
```

The following example shows how to specify a delimiter for sequential merging of files:

```
> paste -s -d"," file1 file2
Unix,Linux,Windows
Dedicated server,Virtual server
```

4. **Specifying multiple delimiters.**
Multiple delimiters come in handy when you want to merge more than two files with different delimiters.
For example I want to merge file1, file2 with pipe delimiter and file2, file3 with comma delimiter. In this case multiple delimiters will be helpful.

```
> paste -d"|," file1 file2 file3
Unix|Dedicated server,Hosting
Linux|Virtual server,Machine
Windows|,Operating system
```

5. Combining N consecutive lines
The paste command can also be used to merge N consecutive lines from a file into a single line.
The following example merges 2 consecutive lines into a single line
> cat file1 | paste - -

```
Unix    Linux
Windows
```

- $ cat file1|**paste - - -**
```
unix    linux   windows
```
- $ cat file1|**paste -**
```
unix
linux
windows
```

Comparing Files

Sometimes user wants to know that 2 files are identical or not.
That means the content of the files are same or different.
commands for comparing files : cmp, diff, comm

1. cmp (compare) :

cmp command in Linux/UNIX is used to compare the two files byte by byte and helps you to find out whether the two files are identical or not.

- When cmp is used for comparison between two files, it reports the location of the first mismatch to the screen if difference is found and if no difference is found i.e the files compared are identical.
- cmp displays no message and simply returns the prompt if the the files compared are identical.

Syntax:
cmp [OPTION]... FILE1 [FILE2 [SKIP1 [SKIP2]]]

SKIP1 ,SKIP2 & OPTION are optional and FILE1 & FILE2 refer to the filenames .

The syntax of cmp command is quite simple to understand.
If we are comparing two files then obviously we will need their names as arguments (i.e as FILE1 & FILE2 in syntax).
In addition to this, the optional SKIP1 and SKIP2 specify the number of bytes to skip at the beginning of each file which is zero by default and OPTION refers to the options compatible with this command about which we will discuss later on.

cmp Example : As explained that the cmp command reports the byte and line number if a difference is found. Now let's find out the same with the help of an example. Suppose there are two files which you want to compare one is file1.txt and other is file2.txt :

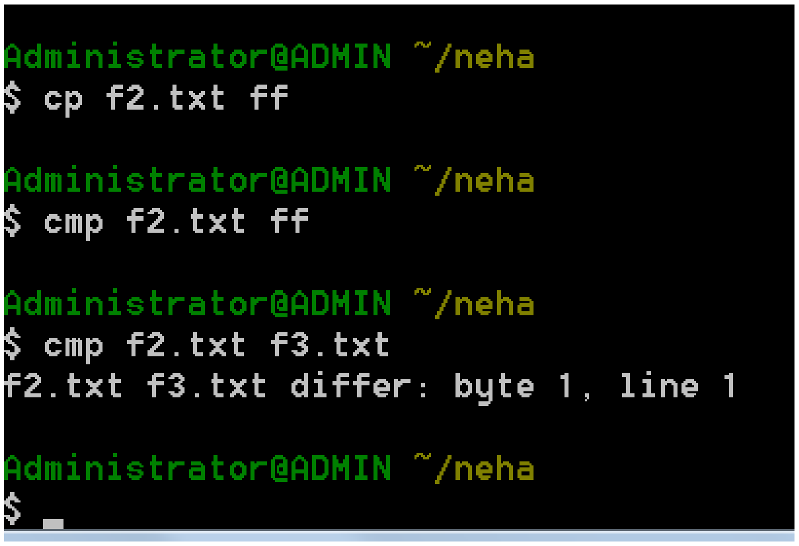
```
$cmp file1.txt file2.txt
```

1. If the files are not identical : the output of the above command will be :

```
$cmp file1.txt file2.txt
file1.txt file2.txt differ: byte 9, line 2

 /*indicating that the first mismatch found in two files at byte 20 in second line*/
```

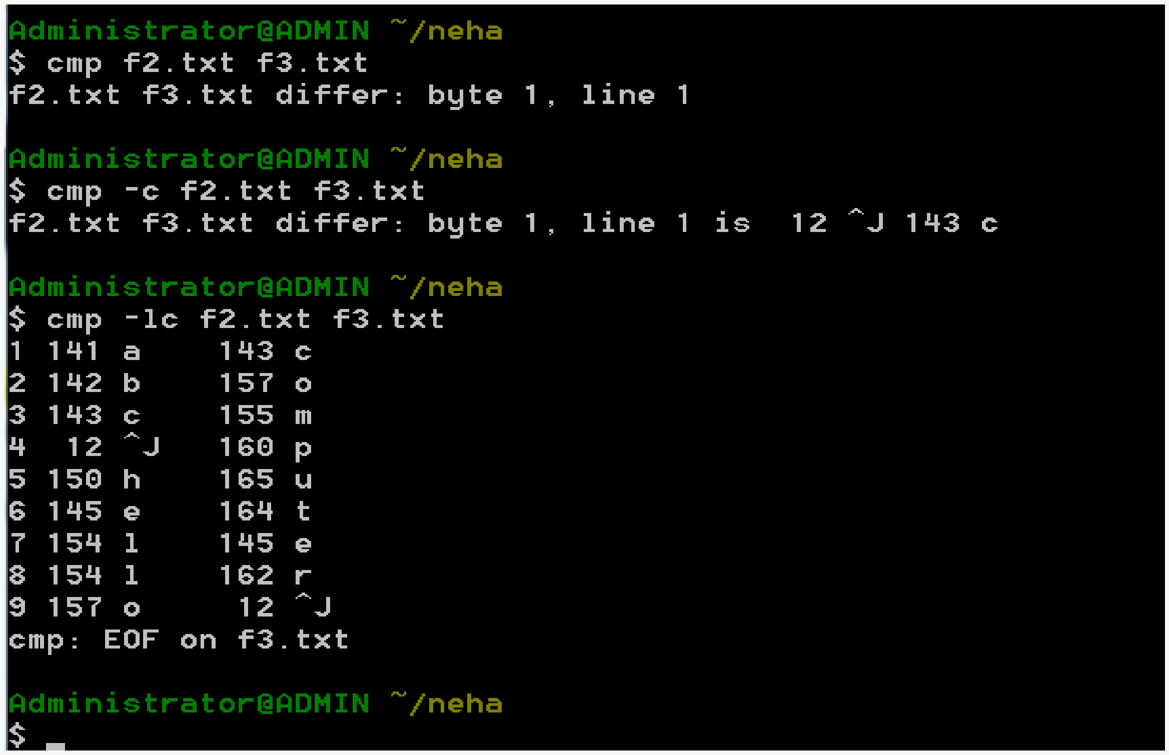
2. If the files are identical : you will see something like this on your screen:

```
$cmp file1.txt file2.txt
$ _
/*indicating that the files are identical*/
```



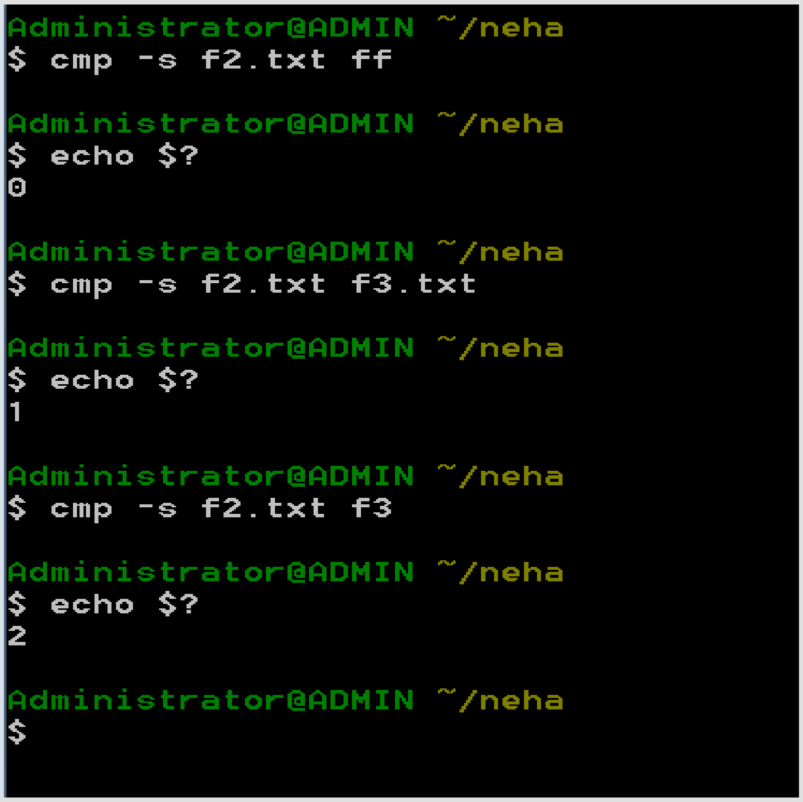
```
Administrator@ADMIN ~/neha
$ cp f2.txt ff

Administrator@ADMIN ~/neha
$ cmp f2.txt ff

Administrator@ADMIN ~/neha
$ cmp f2.txt f3.txt
f2.txt f3.txt differ: byte 1, line 1

Administrator@ADMIN ~/neha
$ _
```

OPTIONS

-c	Print the differing characters.
	Display control characters as a '^' followed by a letter of the alphabet and precede characters that have the high bit set with 'M-' (which stands for "meta").
	e.g.: $cmp -c f1 f2
	f1 f2 differ: byte 2, line1 is 160p 53+
	
	$cmp -lc f1 f2
	ans: 2 160 53+
	3 160 53+
	
	`Administrator@ADMIN ~/neha`
	`$ cmp f2.txt f3.txt`
	`f2.txt f3.txt differ: byte 1, line 1`
	
	`Administrator@ADMIN ~/neha`
	`$ cmp -c f2.txt f3.txt`
	`f2.txt f3.txt differ: byte 1, line 1 is  12 ^J 143 c`
	
	`Administrator@ADMIN ~/neha`
	`$ cmp -lc f2.txt f3.txt`
	`1 141 a    143 c`
	`2 142 b    157 o`
	`3 143 c    155 m`
	`4  12 ^J   160 p`
	`5 150 h    165 u`
	`6 145 e    164 t`
	`7 154 l    145 e`
	`8 154 l    162 r`
	`9 157 o     12 ^J`
	`cmp: EOF on f3.txt`
	
	`Administrator@ADMIN ~/neha`
	`$ _`

--ignore-initial=BYTES	Ignore any differences in the the first BYTES bytes of the input files. Treat files with fewer than BYTES bytes as if they are empty. e.g.: $cmp i3 f1 f2 $
-l (L)	Print the byte/character number in decimal and the differing character value in octal for each character is differ in both files. e.g.: $cmp -l f1 f2 ans: 2 160 53

	3 160 53 i.e. : it display detailed list in 3 column. the 1st shows position of different characters in files, 2nd shows the octal value of different characters in file f1 and third shows the octal value of differ character in file f2.

--quiet
-s
--silent
	Do not print anything; only return an exit status indicating whether the files differ.
	Return Values
	The **cmp** utility exits with one of the following values:
	0—The files are identical.
	1—The files are different; this value includes the case where one file is identical to the first part of the other. In the latter case, if the **-s** option has not been specified, **cmp** writes to standard output that EOF was reached in the shorter file (before any differences were found).
	>1—An error occurred.
	//...cmp command used with -s option...//
	$cmp -s file1.txt file.txt
	1
	/*indicating files are different without displaying the differing byte and line*/
	
	Administrator@ADMIN ~/neha
	$ cmp -s f2.txt ff
	
	Administrator@ADMIN ~/neha
	$ echo $?
	0
	
	Administrator@ADMIN ~/neha
	$ cmp -s f2.txt f3.txt
	
	Administrator@ADMIN ~/neha
	$ echo $?
	1
	
	Administrator@ADMIN ~/neha
	$ cmp -s f2.txt f3
	
	Administrator@ADMIN ~/neha
	$ echo $?
	2
	
	Administrator@ADMIN ~/neha
	$

2. **comm:**
   Compare two sorted files line-by-line.
   Compare sorted files FILE1 and FILE2 line-by-line.

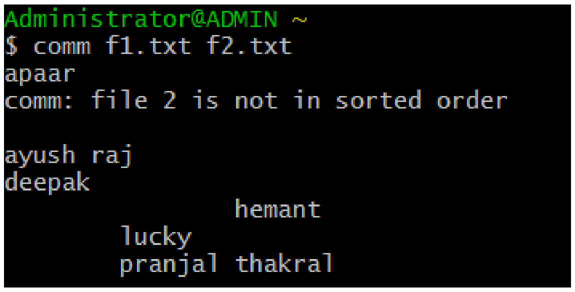
   comm syntax : **comm [OPTION]... FILE1 FILE2**

   With no options, comm produces three-column output.
   Column 1 contains lines unique to FILE1,
   column 2 contains lines unique to FILE2, and
   column 3 contains lines common to both files.

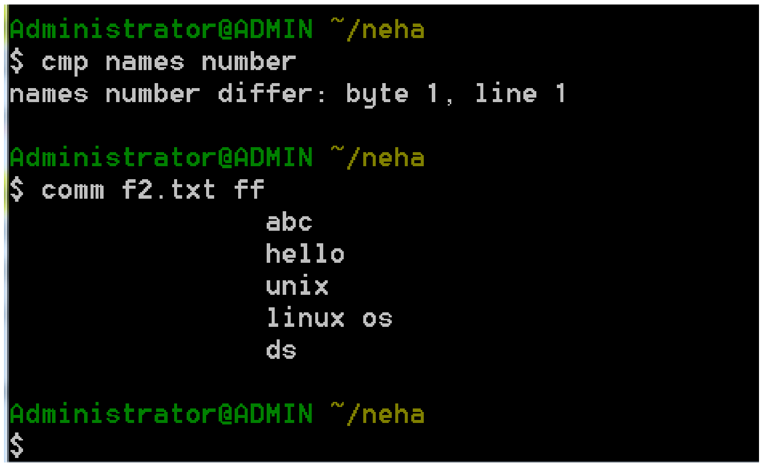
```
// displaying contents of file1 //
$cat file1.txt
Apaar
Ayush Rajput
Deepak
Hemant

// displaying contents of file2 //
$cat file2.txt
Apaar
Hemant
Lucky
Pranjal Thakral
```

```
$comm file1.txt file2.txt
```

result :



```
Administrator@ADMIN ~
$ comm f1.txt f2.txt
apaar
comm: file 2 is not in sorted order

ayush raj
deepak
                hemant
        lucky
        pranjal thakral
```



```
Administrator@ADMIN ~/neha
$ cmp names number
names number differ: byte 1, line 1

Administrator@ADMIN ~/neha
$ comm f2.txt ff
                abc
                hello
                unix
                linux os
                ds

Administrator@ADMIN ~/neha
$
```

Each of these columns can be suppressed individually with options.

Options:

-1	suppress column 1 (lines unique to **FILE1**)
-2	suppress column 2 (lines unique to **FILE2**)
-3	suppress column 3 (lines that appear in both files)
--check-order	check that the input is correctly sorted, even if all input lines are pairable
--nocheck-order	do not check that the input is correctly sorted
--output-delimiter=*STR*	separate columns with string *STR*
--help	display a help message, and exit.
--version	output version information, and exit.

Examples

Let's say you have two text files, **recipe.txt** and **shopping-list.txt**.

recipe.txt contains these lines:

```
All-Purpose Flour
Baking Soda
Bread
Brown Sugar
Chocolate Chips
Eggs
Milk
Salt
Vanilla Extract
White Sugar
```

And

shopping-list.txt contains these lines:

```
All-Purpose Flour
Bread
Brown Sugar
Chicken Salad
Chocolate Chips
Eggs
Milk
Onions
Pickles
Potato Chips
Soda Pop
Tomatoes
White Sugar
```

If we run the **comm** command on the two files, it will read both files and give us three columns of output:

```
comm recipe.txt shopping-list.txt
```

```
        All-Purpose Flour
Baking Soda
        Bread
        Brown Sugar
    Chicken Salad
        Chocolate Chips
        Eggs
        Milk
    Onions
    Pickles
    Potato Chips
Salt
    Soda Pop
    Tomatoes
Vanilla Extract
        White Sugar
```

Here, each line of output has either zero, one, or two tabs at the beginning, separating the output into three columns:

1. The first column (zero tabs) is lines that only appear in the first file.
2. The second column (one tab) is lines that only appear in the second file.
3. The third column (two tabs) is lines that appear in both files.

3. **diff:**
   diff stands for **difference**.
   This command is used to display the differences in the files by comparing the files line by line.
   Unlike its fellow members, cmp and comm, it tells us which lines in one file have is to be changed to make the two files identical.
   The important thing to remember is that **diff** uses certain **special symbols** and **instructions** that are required to make two files identical.
   It tells you the instructions on how to change the first file to make it match the second file.

Special symbols are:

```
a : add
c : change
d : delete
```

Syntax :

```
diff [options] File1 File2
```

Lets say we have two files with names **a.txt** and **b.txt** containing 5 Indian states.

```
$ ls a.txt  b.txt
```

```
$ cat a.txt
Gujarat
Uttar Pradesh
Kolkata
Bihar
Jammu and Kashmir
$ cat b.txt
```

```
Tamil Nadu
Gujarat
Andhra Pradesh
Bihar
Uttar pradesh
```

Now, applying **diff** command without any option we get the following output:

```
$ diff a.txt b.txt
0a1
> Tamil Nadu
2,3c3
< Uttar Pradesh
 Andhra Pradesh
5c5
 Uttar pradesh
```

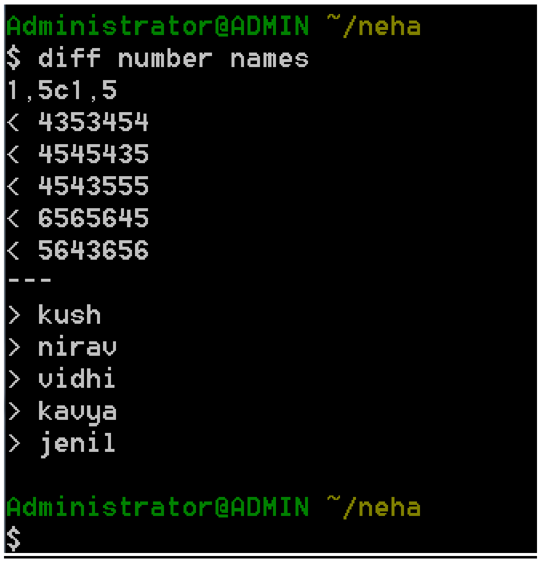
Let’s take a look at what this output means.
The first line of the **diff** output will contain:

- Line numbers corresponding to the first file,
- A special symbol and
- Line numbers corresponding to the second file.

Like in our case, **0a1** which means **after** lines 0 (at the very beginning of file) you have to add **Tamil Nadu** to match the second file line number 1.
It then tells us what those lines are in each file preceded by the symbol:

- **Lines preceded by a < are lines from the first file.**
- **Lines preceded by > are lines from the second file.**
- Next line contains **2,3c3** which means from line 2 to line 3 in the first file needs to be changed to match line number 3 in the second file. It then tells us those lines with the above symbols.
- The three dashes **(“—“)** merely separate the lines of file 1 and file 2.

As a summary to make both the files identical, first add *Tamil Nadu* in the first file at very beginning to match line 1 of second file after that change line 2 and 3 of first file i.e. *Uttar Pradesh* and *Kolkata* with line 3 of second file i.e. *Andhra Pradesh*.
After that change line 5 of first file i.e. *Jammu and Kashmir* with line 5 of second file i.e. *Uttar pradesh*.


```
Administrator@ADMIN ~/neha
$ diff number names
1,5c1,5
< 4353454
< 4545435
< 4543555
< 6565645
< 5643656
---
> kush
> nirav
> vidhi
> kavya
> jenil

Administrator@ADMIN ~/neha
$
```

Now let's see what it looks like when **diff** tells us that we need to delete a line.

```
$ cat a.txt
Gujarat
Andhra Pradesh
Telangana
Bihar
Uttar pradesh
```

```
$ cat b.txt
Gujarat
Andhra Pradesh
Bihar
Uttar pradesh
```

```
$ diff a.txt b.txt
3d2
< Telangana
```

Here above output **3d2** means delete line 3rd of first file i.e. *Telangana* so that both the files **sync up** at line 2.

Options:

-i : By default this command is *case sensitive*.
To make this command *case in -sensitive*use **-i** option with **diff**.

```
$ cat file1.txt
dog
mv
CP
comm
$ cat file2.txt
```

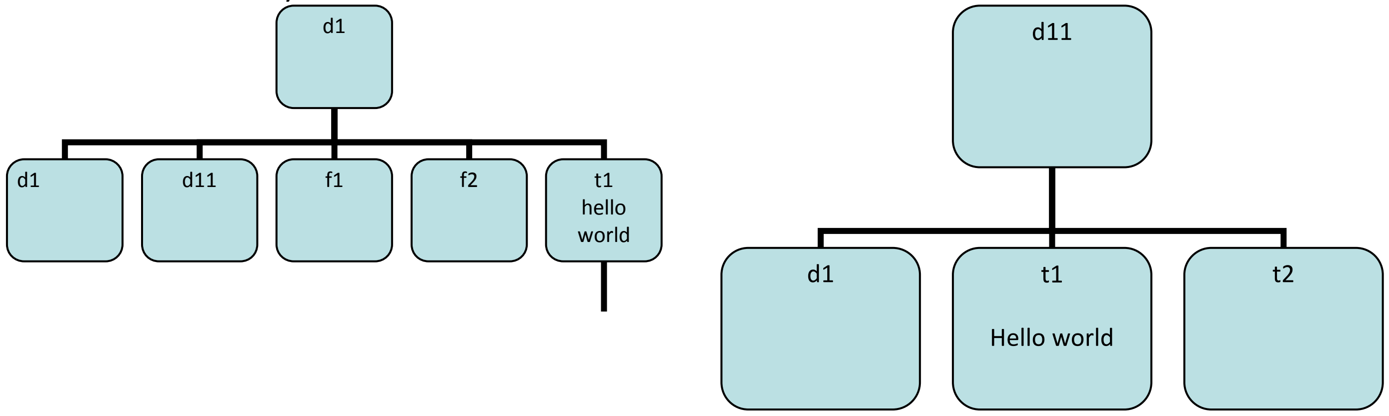
```
DOG
cp
diff
comm
```

Without using this option:

```
$ diff file1.txt file2.txt
1,3c1,3
< dog
< mv
 DOG
> cp
> diff
```

```
Using this option:
$ diff -i file1.txt file2.txt
2d1
 diff
```

-r (recursive) : it recursively compare files of subdirectories with same name and display nothing if identical otherwise display differences to make both files identical.
Consider the directory structure as below:



-s : It reports when 2 files are identical otherwise display differences between them.

e.g.: $diff -s f1 f1   // display message if files are identical
ans: Files f1 and F1 are identical

What is the difference between cmp and diff commands? Provide an example for each.

cmp

-Byte by byte comparision performed for two files comparision and displays the first mismatch byte.
-cmp returns the 1st byte and the line no of the fileone to make the changes to make the fileone identical to filetwo.
-Directory names cannot be used.

diff

-Indicates the changes that are to be done to make the files identical.
-returns the text of filetwo that is different from filetwo.
-Directory names can be used
///////////////////////////////////////

Translating characters:

tr: tr stands for translate.

The tr command in UNIX is a command line utility for translating or deleting characters.
It supports a range of transformations including uppercase to lowercase, squeezing repeating characters, deleting specific characters and basic find and replace.
It can be used with UNIX pipes to support more complex translation.

Syntax :

```
$ tr [OPTION] SET1 [SET2]
```

Options

-c: complements the set of characters in string. i.e., operations apply to characters not in the given set.

-d: delete characters in the first set from the output.

-s: replaces repeated characters listed in the set1 with single occurrence

-t: truncates set1

Tr command Examples:

1**. Convert lower case letters to upper case**

The following tr command translates the lower case letters to capital letters in the give string:

```
> tr "[:lower:]" "[:upper:]"
linux dedicated server
LINUX DEDICATED SERVER
> echo "linux dedicated server" | tr "[a-z]" "[A-Z]"
LINUX DEDICATED SERVER

$ tr '[a-z]' '[A-Z]'
hiii
HIII
hello
HELLO
```

```
tr '[a-z]' '[A-Z]' > translate.txt
```

```
 $ cat f1
unix   or linux os
is unix good os
is linux good os
$ tr "[a-z]" "[A-Z]" <f1
UNIX   OR LINUX OS
IS UNIX GOOD OS
IS LINUX GOOD OS
```

Note: tr does' not take a filename as its argument, but it takes input through redirection or a pipe or std. input

2. Transform upper case letters to lower case.
Similar to the above example, you can translate the uppercase letters to small letters.

```
> echo "UNIX DEDICATED SERVER" | tr "[:upper:]" "[:lower:]"
unix dedicated server
> echo "UNIX DEDICATED SERVER" | tr "[A-Z]" "[a-z]"
unix dedicated server
```

```
$cat>f2
a/b
c/d
e-f
$ tr '/' '-' <f2
a-b
```

```
c-d
e-f
$ tr '/' '-' f2
tr: too many arguments
Try `tr --help' for more information.
```

3. Replace non-matching characters.
The -c option is used to replace the non-matching characters with another set of characters.

```
> echo "unix" | tr -c "u" "a"
uaaa
```

In the above example, **except the character "c"** other **characters are replaced with "a"**

- **$cat f2**

```
a/b8
c/d4
e-fA
$tr -c 'a-z0-9' '*' <f2

a*b8*c*d4*e*f**$
```

- **to replace newline character(newline having octal value 012) visible with dollar symbol**

```
$ tr '\012' '$'<f2
a/b8$c/d4$e-fA$
$
```

- it replaces 'a' with 'x' and characters 'b,c,d,e' with y and rest characters will be unchanged.

```
$cat f2
a/b8
c/d4
e-fA
$ tr 'abcde' 'xy'<f2
x/y8
y/y4
y-fA
```

- **here a,b,c,d,e all will be replaced by x**

```
$ tr 'abcde' 'x'<f2
x/x8
x/x4
x-fA
```

4. Delete non-printable characters
The -d option can be used to delete characters. The following example deletes all the non-printable characters from a file.

```
> tr -cd "[:print:]" < filename
```

5. **Squeezing characters**
You can squeeze more than one occurrence of continuous characters with single occurrence.

The following example squeezes two or more successive blank spaces into a single space.

```
> echo "linux    server" | tr -s " "
linux server
```

Here you can replace the space character with any other character by specifying in set2.

```
> "linux    server" | tr -s " " ","
linux,server
```

6. Delete characters
The following example removes the word linux from the string.

```
> echo "linuxserver" | tr -d "linux"
server
```

```
$ cat f2
a/b8
c/d4
e-fA
$tr -d '0-9a-z'<f2
/
/
-A
#deletes number and all small letters from file.
$tr -d '0-9a-c'<f2
/
/d
e-fA
```

Formatting text files:

1. pr

pr command is used to paginating the files
pr command prepares a file for printing by adding suitable headers, footers, and formatted text to an input file.

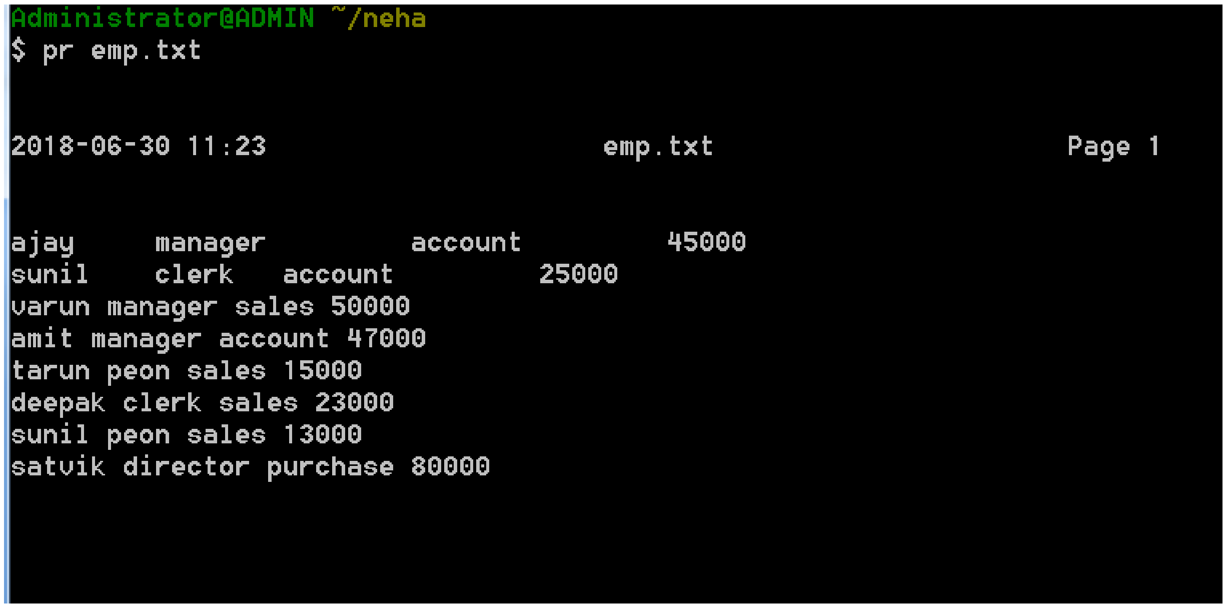
Syntax,

$ pr options filename

By default, pr command inserts 5- lines of header at the top and 5- lines of footer at the bottom of each page of the input file.

Example,

$pr file1



```
Administrator@ADMIN ~/neha
$ pr emp.txt


2018-06-30 11:23                      emp.txt                      Page 1


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
```

PR COMMAND OPTIONS:

a. **pr –l (length)**
   i. By default, the page size used by pr command is 66 (header as well as footer included) lines, which can be changed with the –l (length) option along with argument.
   ii. Example,

$ pr –l 72 file1

```
2015-07-15    8:25                  file1                        Page1
This is mkics
$ _
```

This command sets the page length of 72 lines instead of 66 lines.

b. **pr +k**
   i. When printing large file that spans to several pages, we can instruct pr command to start printing from a specific page.
   ii. This is done with +k option, where k → any integer which indicate that formatting start from kth page of the input file.
   iii. Example,

$ pr +10 file1 *Starts formatting from page 10*

c. **pr –k**
   i. Here, k is any integer.
   ii. This option produces output in more than one column and print down the page (top to bottom)
   iii. Example,

To print first 3 column

$ pr -3 file1

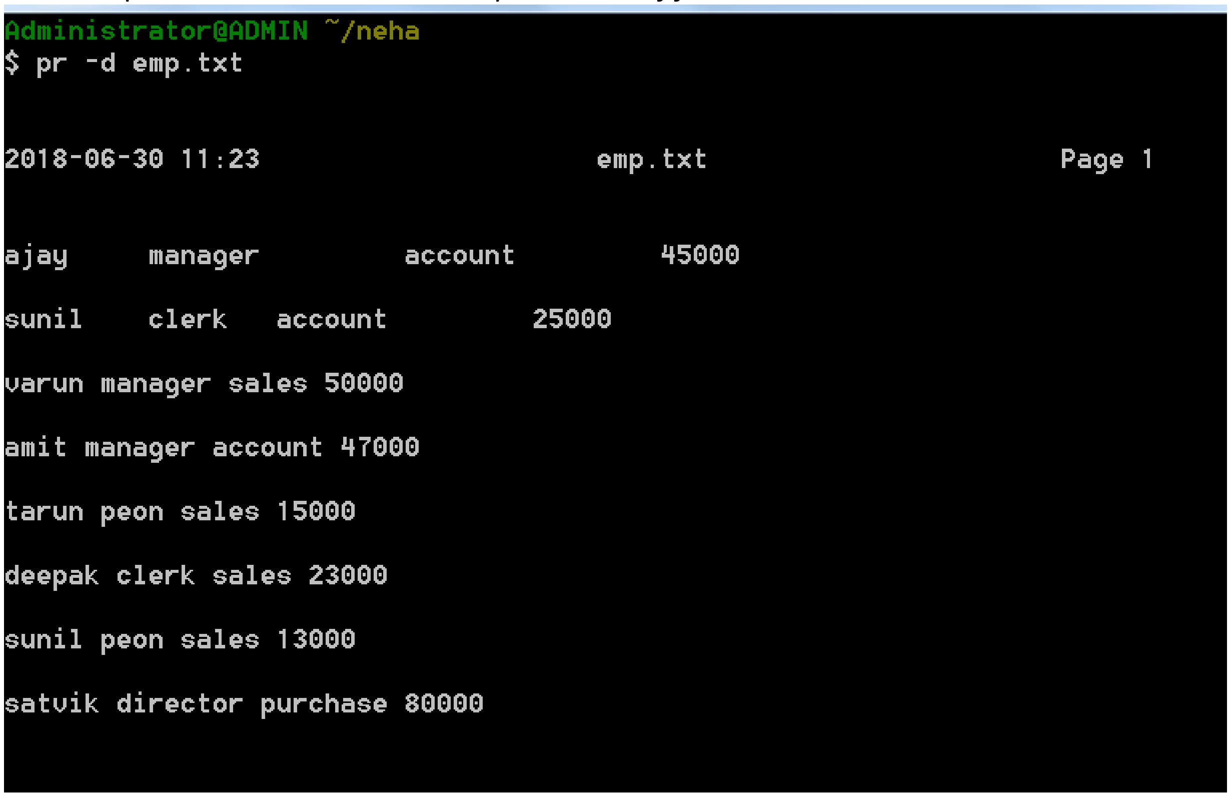
d. **pr –a**
   i. It prints column across (left to write) the page rather than down the page.
   ii. It is used together with –k option.
   iii. Example,

$ pr –a -3 file1

e. **pr –d**

i. It double-spaces the text of input file.
ii. Example,

$ pr –d file1 *double spaces lines of file1*

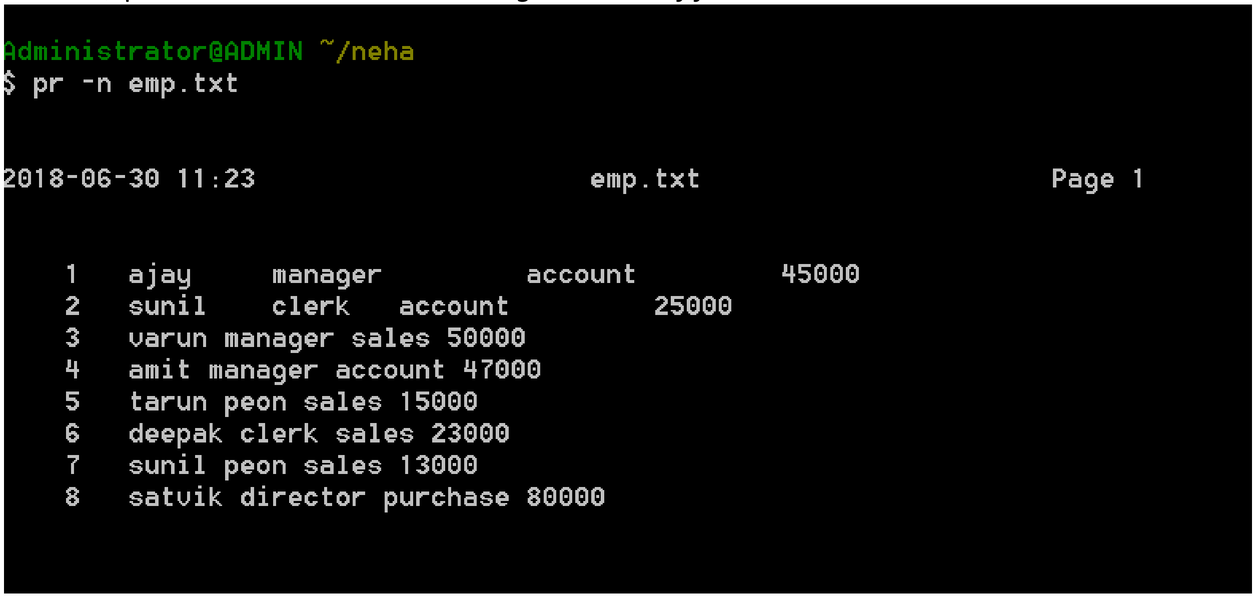


```
Administrator@ADMIN ~/neha
$ pr -d emp.txt


2018-06-30 11:23                  emp.txt                  Page 1


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
```

f. pr –n

i. It prints a line number before each line.
ii. It gives the line number to empty as well as non-empty lines.
iii. Example,

$ pr –n file1 *Numbering each line of file1*



```
Administrator@ADMIN ~/neha
$ pr -n emp.txt


2018-06-30 11:23                  emp.txt                  Page 1


    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
```

g. pr –oN

i. It sets a left margin N-characters wide where N → any positive number
ii. Example,

$ pr -o10 file1 *set left margin 10-characters wide*

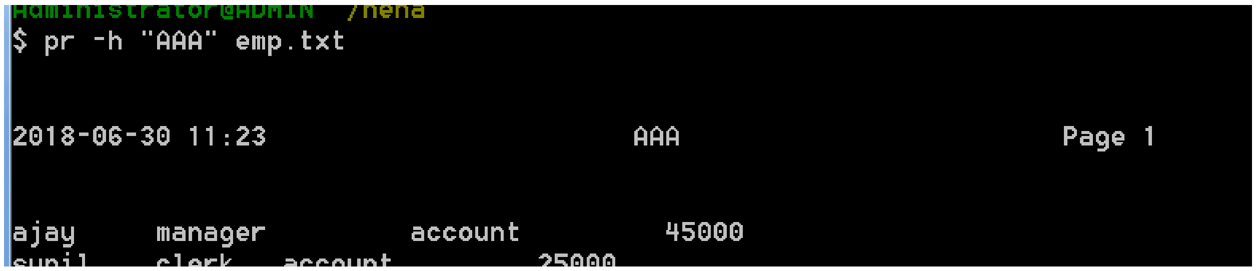
h. pr –t

i. This command does not print header and footer of input file.
ii. Example,

$ pr –t file1 *do not print header and footer*

i. **pr –h**

  i. It uses a suitable centered header instead of filename in page header.
  ii. Example,
  $ pr –h "MKICS.doc" file1   *display "MKICS.doc" instead of file1*



```
Administrator@ADMIN ~/neha
$ pr -h "AAA" emp.txt


2018-06-30 11:23                      AAA                        Page 1


ajay     manager          account         45000
sunil    clerk    account             25000
```

j. **pr –wN**

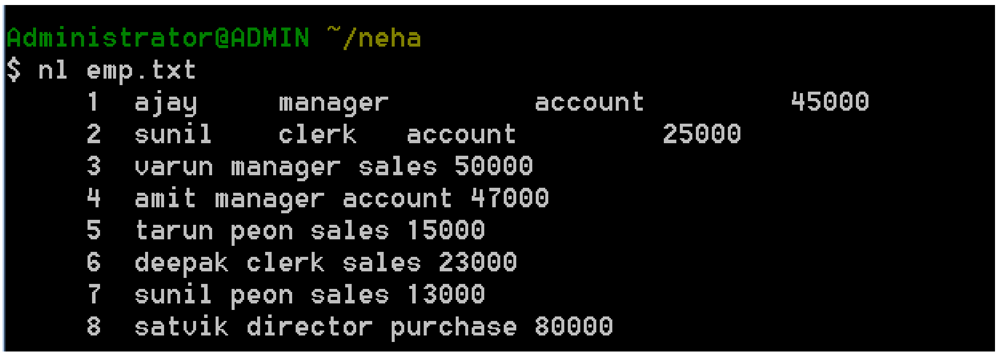
  i. It sets page width to N-characters for multiple text columns.
  ii. If line length is greater than N then remaining characters are truncated from right.
  iii. It is used with –k option.
  iv. Example,
  $ pr –w90 -3 file1

It displays 3- column output. Here, size 90 is equally distributed to each column. So, size of each column is 30 characters.

2. nl

- It provides line number to each logical line of files.
- Logical line means non-empty lines which consists something apart from the new line character.
- **Syntax,**

  $ nl [option] [filenames]



```
Administrator@ADMIN ~/neha
$ nl emp.txt
     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
```

- To give number to each logical lines

  $ nl file1

  1 C++ Lang
  2 C Lang
  3 Asp.net

➢ **NL COMMAND OPTIONS:**

- 7 options:

1. **nl –n** format_characters
   - It inserts line number according to format characters.
   - Format characters:

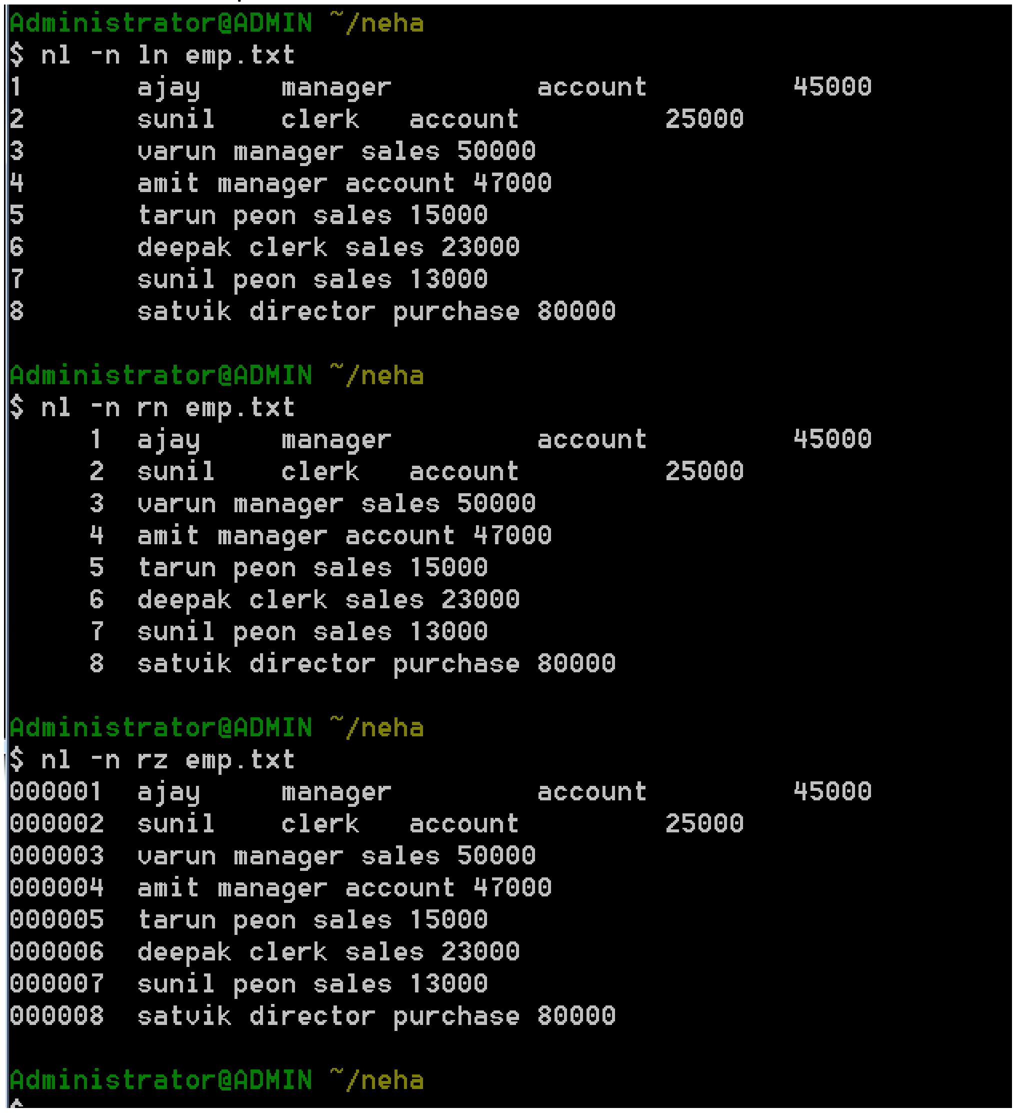
Format Characters	**Meaning**
Ln	Left justified, no leading zeros
Rn	Right justified, no leading zeros

Rz	Right justified, leading zeros

- By default, **width of line number is 6-characters**.
- To display leading zeros with line numbers

  $ nl –n rz file1

      000001 C++ Lang
      000002 C Lang
      000003 Asp.net



```
Administrator@ADMIN ~/neha
$ nl -n ln emp.txt
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

Administrator@ADMIN ~/neha
$ nl -n rn emp.txt
     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

Administrator@ADMIN ~/neha
$ nl -n rz emp.txt
000001  ajay    manager         account         45000
000002  sunil   clerk   account         25000
000003  varun manager sales 50000
000004  amit manager account 47000
000005  tarun peon sales 15000
000006  deepak clerk sales 23000
000007  sunil peon sales 13000
000008  satvik director purchase 80000

Administrator@ADMIN ~/neha
```

2. nl –wN

- It sets width of line number column (i.e. 1st column) to N, where N → any positive number.
- Example,

  To set width of line number column

      $ cat file2
      $ nl –nrz –w3 file1
          001 C++ Lang
          002 C Lang
          003 Asp.net

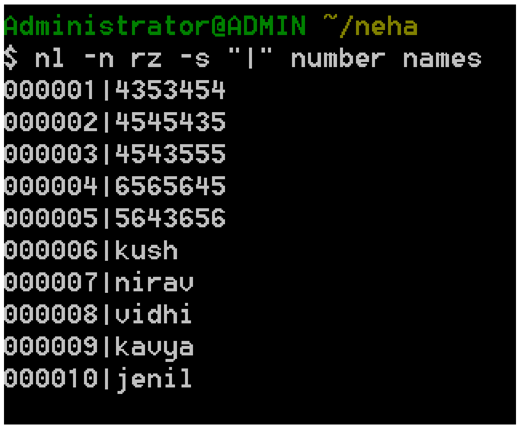
3. nl –s sep

- It adds separator sep after line number instead of default separator TAB
- Example,

  To add separator "|" between number column and file content

$ nl –n rz –w3 –s'|' file1

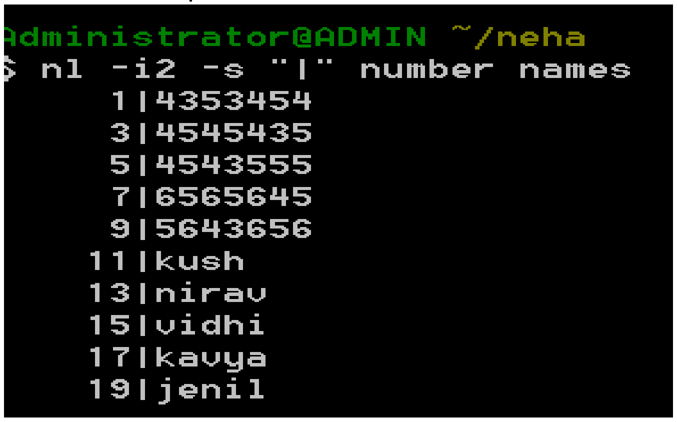
```
001|C++ Lang
002|C Lang
003|Asp.net
```



```
Administrator@ADMIN ~/neha
$ nl -n rz -s "|" number names
000001|4353454
000002|4545435
000003|4543555
000004|6565645
000005|5643656
000006|kush
000007|nirav
000008|vidhi
000009|kavya
000010|jenil
```

4. **nl –iN**
   - It sets line number increment N at each line where N→ any positive number.
   - Example,
     To set odd number to each line
     $ nl –i2 file1

```
001 C++ Lang
003 C Lang
005 Asp.net
```



```
Administrator@ADMIN ~/neha
$ nl -i2 -s "|" number names
     1|4353454
     3|4545435
     5|4543555
     7|6565645
     9|5643656
    11|kush
    13|nirav
    15|vidhi
    17|kavya
    19|jenil
```

5. **nl –b body-style**
   - It uses body-style for numbering body lines.
   - Body-style used with –b option

Body-Style	**Meaning**
A	Number all lines
T	Number only non empty lines
N	Number no lines
pREGEXP	Number only lines that contain a match for REGEXP

   - Example,
     To assign line numbers to line that contain 'hello' pattern
     $ nl –bp hello file1

```
1 hello world
  unix
2 hello asp.net
3 hello
```

cn

6. nl –lN
   - It joins a group of N empty lines and is counted as one where N → any positive number.
   - It is used with –ba option.
   - Example,
     To make groups of 2 consecutive blank line
     $ nl –l2 –ba file1

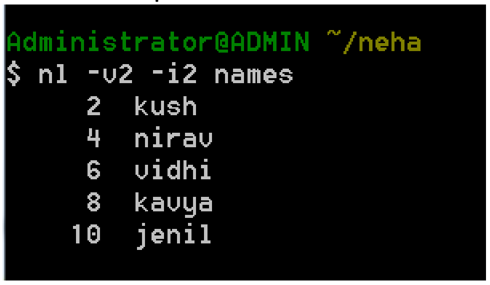
```
1   C++
2   C
3   Unix
4
5   Asp.net
6
7   CN
```

7. nl –vN
   - It sets initial value for line number on each logical page.
   - Example,
     To assign even numbers to each line
     $ nl –v2 –i2 file1

```
2 C++
4 Unix
6 Asp.net
```

     $ nl –v2 file1

```
2 C++
3 Unix
4 Asp.net
```



```
Administrator@ADMIN ~/neha
$ nl -v2 -i2 names
     2  kush
     4  nirav
     6  vidhi
     8  kavya
    10  jenil
```

Other filtering utilities:

1. uniq

- It gets one copy of each line and writes it to standard output.
- Means it reads unique lines from successive repeated lines and writes it to standard output.
- Syntax,

  $ uniq [option] … [input file [output file]]

- It discard all but one of the successive identical line from input file or standard input and writes it to output file or standard output.
  Example,
  If we apply both input and output file with uniq command then it writes unique lines into output files

  $ cat uniq1

```
Cpp Language
C++ Language
Hello surat
Hello surat
Red hat linux
Unix os
```

$ uniq uniq1 uniz1.out

```
Cpp Language
C++ Language
Hello surat
Red hat linux
Unix os
```

- **UNIQ COMMAND OPTIONS:**

  - 8 options:

1. **uniq –c (Count)**
   - This option prefixes each line by the number that indicates occurrences of line.
   - Example,
     To print the frequency or occurrence of all lines

```
$ uniq –c uniq1
1 Cpp Language
1 C++ Language
2 Hello surat
1 Red hat linux
1 Unix os
```

2. uniq –d (Duplicate)
   - It prints only duplicate lines.
   - Example,

```
$ uniq –d uniq1
Hello surat
```

3. uniq –D (All repeated)
   - It prints all duplicate lines.
   - Example,

```
$ uniq –D uniq1
Hello surat
Hello surat
```

4. uniq –fN (Skip-fields → N)
   - It avoids first N fields of each line during comparison.
   - Example,
     To display unique lines after avoiding 1st field of each lines

$ uniq –f1 uniq1

```
C++ language
Hello surat
Red hat linux
Unix os
```

Here, 1st field of lines 1 and 2 are c++ and cpp respectively.

If we ignored 1st field of each line then first two lines of input file is considered as same.
So, **2nd line is not display on screen**.

5. uniq –i (Ignore Case)
   - It ignores differences in case when comparing.
   - Example,

     $ uniq u2
     HELLO
     Hello
     Unix OS
     $ uniq –i u2
     HELLO
     Unix Os

6. uniq –u (Unique)
   - Sometimes, a user is interested only in unique lines of the file then –u is used.
   - This option prints non-repeated lines of input file.
   - Example,

     To display only unique lines

     $ uniq –u uniq1
     Cpp Language
     C++ Language
     Red hat linux
     Unix os

7. uniq –sN (Skip-chars → N)
   - It avoids comparing first N-characters.
   - Example,

     $ cat u3
     Cpp programming
     C++ programming
     J++ programming
     Java programming
     If you ignore 1st character of each line
     $ uniq –s1 u3
     Cpp programming
     C++ programming
     Java programming
     Here, we ignore 1st character of each line then during comparison 2nd and 3rd lines become unique/ identical. So, 3rd line do not display on screen.

8. uniq –wN (Check-chars → N)
   - It compares no more than N characters in lines.
   - Example,

     If we compare 1st characters of each line then 1st 2 lines and last 2 lines of file becomes identical.
     $ uniq –w1 u3
     Cpp programming
     Java programming

2. wc

wc, or "word count," prints a count of newlines, words, and bytes for each input file.

As the name implies, it is mainly used for counting purpose.

- It is 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]... [FILE]...
```

Let us consider two files having name **state.txt** and **capital.txt** containing 5 names of the Indian states and capitals respectively.

```
$ cat state.txt
Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh

$ cat capital.txt
Hyderabad
Itanagar
Dispur
Patna
Raipur
```

Passing only one file name in the argument.

```
$ wc state.txt
 5  7 63 state.txt
      OR
$ wc capital.txt
 5  5 45 capital.txt
```

Options:

1. -l: This option prints the **number of lines** present in a file. With this option wc command displays two-columnar output, 1st column shows number of lines present in a file and 2nd itself represent the file name.

```
With one file name
$ wc -l state.txt
5 state.txt

With more than one file name
$ wc -l state.txt capital.txt
  5 state.txt
  5 capital.txt
 10 total
```

2. -w: This option prints the **number of words** present in a file. With this option wc command displays two-columnar output, 1st column shows number of words present in a file and 2nd is the file name.

```
With one file name
$ wc -w state.txt
```

```
7 state.txt

With more than one file name
$ wc -w state.txt capital.txt
  7 state.txt
  5 capital.txt
 12 total
```

3. -c: This option displays **count of bytes** present in a file. With this option it display two-columnar output, 1st column shows number of bytes present in a file and 2nd is the file name.

```
With one file name
$ wc -c state.txt
63 state.txt

With more than one file name
$ wc -c state.txt capital.txt
 63 state.txt
 45 capital.txt
108 total
```

4. -m: Using **-m** option 'wc' command displays **count of characters** from a file.

```
With one file name
$ wc -m state.txt
63 state.txt

With more than one file name
$ wc -m state.txt capital.txt
 63 state.txt
 45 capital.txt
108 total
```

5. -L: The 'wc' command allow an argument **-L**, it can be used to print out the length of longest (number of characters) line in a file. So, we have the longest character line *Arunachal Pradesh*in a file **state.txt** and *Hyderabad* in the file **capital.txt**. But with this option if more than one file name is specified then the last row i.e. the extra row, doesn't display total but it display the maximum of all values displaying in the first column of individual files. **Note:** A **character** is the smallest unit of information that includes space, tab and newline.

```
With one file name
$ wc -L state.txt
17 state.txt

With more than one file name
$ wc -L state.txt capital.txt
 17 state.txt
 10 capital.txt
 17 total
```

3. more

It display file content page-wise.

more syntax: **more [-option] [-num *lines*] [+/*pattern*] [+*linenum*] [*file* ...]**

Options

-num *lines*	Sets the number of lines that makes up a screenful. The *lines* must be an integer.
-d	With this option, **more** will prompt the user with the message **"[Press space to continue, 'q' to quit.]"** and display **"[Press 'h' for instructions.]"** when an illegal key is pressed, instead of ringing a bell.
-l	**more** usually treats **^L** (**CONTROL-L**, the form feed) as a special character, and will pause after any line that contains it. The **-l** option will prevent this behavior.
-f	Causes **more** to count logical, rather than screen lines (i.e., long lines are not wrapped).
-p	Do not scroll. Instead, clear the whole screen and then display the text. This option is switched on automatically if the **more** executable is named **page**.
-c	Do not scroll. Instead, paint each screen from the top, clearing the remainder of each line as it is displayed.
-s	Squeeze multiple blank lines into one blank line.
-u	Do not display underlines.
+/*string*	Search for the string *string*, and advance to the first line containing *string* when the file is displayed.
+*num*	Start displaying text at line number *num*.

Commands

When displaying a file, **more** can be controlled with a set of commands loosely based on the text editor vi. Some commands can be preceded by a decimal number referred to as *k* in the following descriptions.

h, **?**	Show help (display a brief command summary). If you forget all the other commands, remember this one!
[*k*]**SPACE**	Pressing the spacebar displays the next *k* lines of text. If *k* is not specified, **more** displays a full screen of new text.
[*k*]**z**	Like pressing **SPACE**, but *k* becomes the new default number of lines to display.
[*k*]**RETURN**	Pressing the return key displays next *k* lines of text. The default is 1 line. If specified, *k*becomes the new default.
[*k*]**d**, [*k*]**^D**	Pressing **d** or **CONTROL-D** scrolls *k* lines. The default is the current scroll size, which is initially 11 lines. If specified, *k* becomes the new default.
q, **Q**, **^C**	Pressing **q**, **Q**, or **CONTROL-C** (the interrupt key) exits the program.
[*k*]**s**	Skip forward *k* lines of text. Defaults to 1.
[*k*]**f**	Skip forward *k* screenfuls of text. Defaults to 1.
b, **^B**	Pressing **b** or **CONTROL-B** skips backward *k* lines of text. Defaults to 1. (This only works when viewing files, not piped input).
'	Go to the place where the previous search started.
=	Display the current line number.
[*k*]**/***pattern*	Search for the *k*th occurrence of the regular expression *pattern*. Defaults to 1.
[*k*]**n**	Search for the *k*th occurrence of the last regular expression searched for, which

	defaults to 1.
!*command*, **:!***command*	Execute *command* in a subshell.
v	Start up an editor at current line. The editor is taken from the environment variable**VISUAL** if it is defined, or **EDITOR** if **VISUAL** is undefined; if neither is defined, defaults to "vi".
^L	Pressing **CONTROL-L** redraws the screen.
[*k*]**:n**	Go to the *k*th next file. Defaults to 1.
[*k*]**:p**	Go to the *k*th previous file. Defaults to 1.
:f	Display the current file name and line number.
.	Repeat previous command.

4. tee

Tee command reads from standard input and writes to standard output as well as a file.
Means it has one input and two outputs.

Syntax,

tee [option] … [filename]…

We know that all intermediate output in a pipe is discarded by UNIX i.e. it is not saved on the disk.
Sometimes a user may want to pipe the standard output of a command to another command and also save it on disk for later use i.e. sends one copy of the output as standard input to next command and one copy is redirect to a disk file.

TEE COMMAND OPTIONS:

a. tee –a (Append):
    i. It does not overwrite. The output is appended to a given file.
    ii. The user also wishes to preserve the user's list in a file called *alluser* and display the list of logged-in user on a screen

```
$ who | tee alluser
root     :0          Aug 13 02:07
root     pts/1       Aug 13 02:07
nirzari  pts/2       Aug 13 03:06 (192.168.0.64)
```

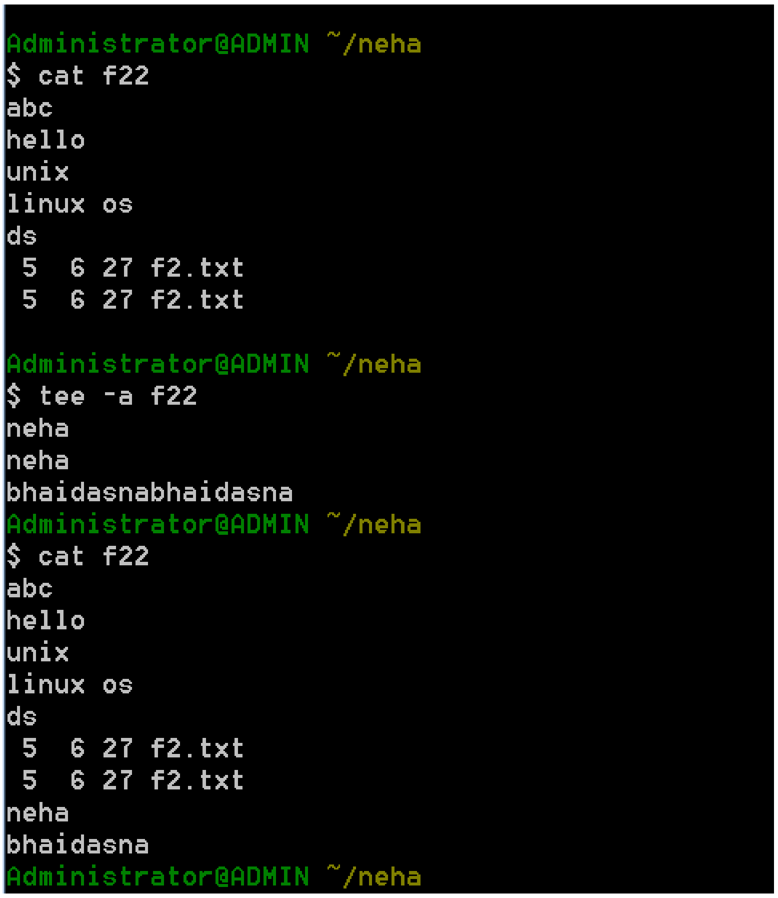
    iii. To display both, list of logged in users as well as their counts on a screen

```
$ who | tee /dev/tty | wc –l
root     :0          Aug 13 02:07
root     pts/1       Aug 13 02:07
nirzari  pts/2       Aug 13 03:06 (192.168.0.64)
    3
```

    iv. It is also useful to create a new file.

```
$ tee t2
    Unix
    Unix
    Asp.net
```

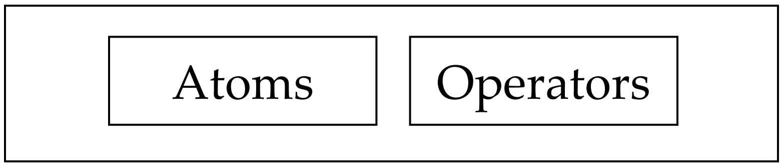
```
     Asp.net
     cn
     cn
$ cat t2
     Unix
     Asp.net
     cn
```



```
Administrator@ADMIN ~/neha
$ cat f22
abc
hello
unix
linux os
ds
 5  6 27 f2.txt
 5  6 27 f2.txt

Administrator@ADMIN ~/neha
$ tee -a f22
neha
neha
bhaidasnabhaidasna
Administrator@ADMIN ~/neha
$ cat f22
abc
hello
unix
linux os
ds
 5  6 27 f2.txt
 5  6 27 f2.txt
neha
bhaidasna
Administrator@ADMIN ~/neha
```

Advance filtering utilities:

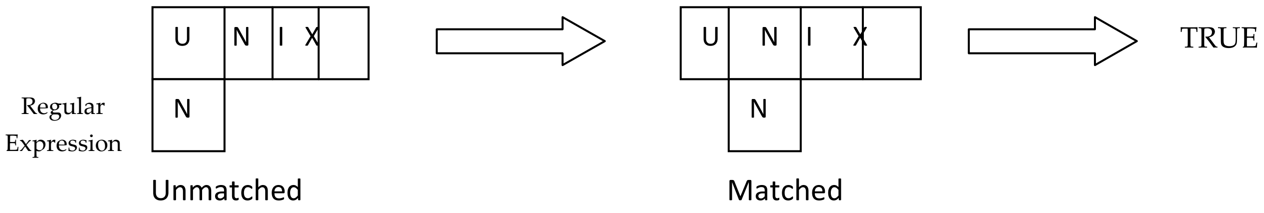
- **<u>REGULAR EXPRESSIONS:</u>**
  - It is consist of a sequence of characters that is used to match against text.
  - Regular expression consists of atoms and operators.
  - Atom specifies **what we are looking for**.
  - Operators are used to **combine atoms into complex expression**.
  - <u>Structure of Regular Expression:</u>



- **<u>ATOMS:</u>**
  - Atoms available in regular expression are:
    - ✓ Single character
    - ✓ Dot character
    - ✓ Class character
    - ✓ Anchor
    - ✓ Back reference

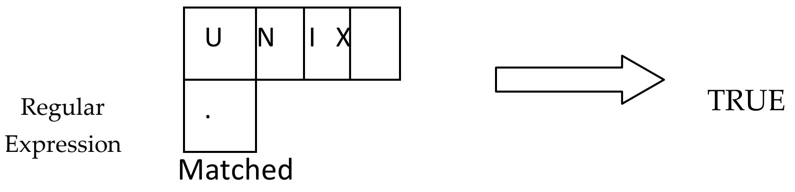
1) Single character:

- A single character matches itself.
- Length of single character atom is one.
- Comparison is done **character by character** and if **match found** then it returns **TRUE, else FALSE**.
- Example,
  The successful and unsuccessful matches of regular expression with a text.

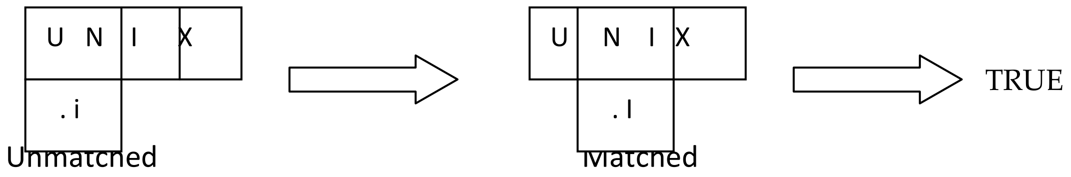


(Successful Search)

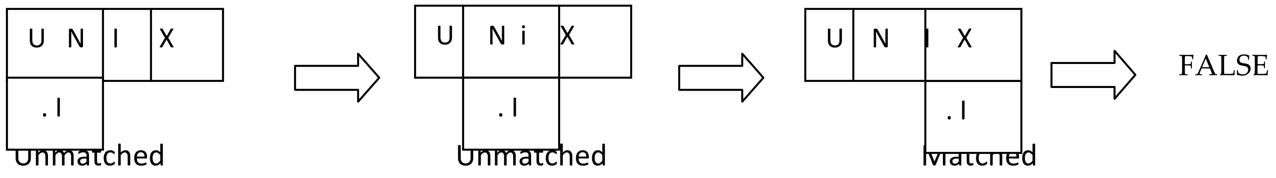
2) Dot Character:
- This atom is denoted by dot (.)
- It is also single character atom that matches any single character except new line character.
- If any **single character occurs anywhere** in a text then the **pattern match succeed otherwise failed**.
- Example,
  Successful match of regular expression:



(Successful Search)

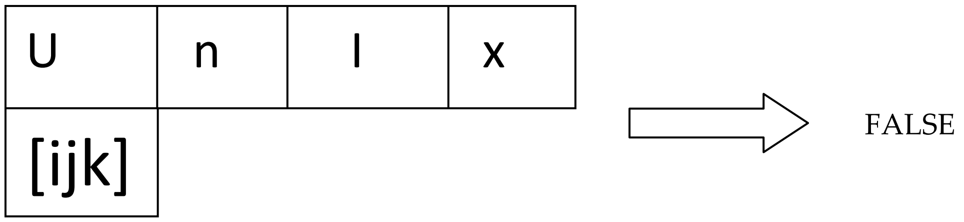


(Successful Search)

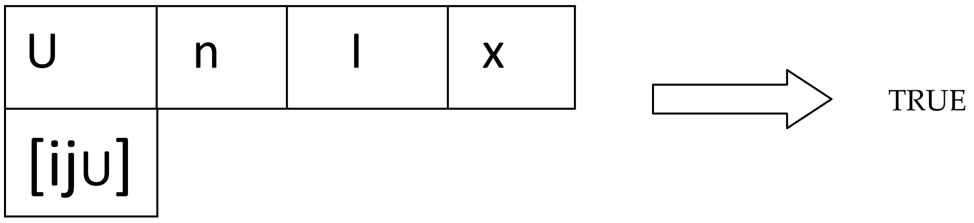


(Unsuccessful Search)

3) Class Character:
- It defines set of ASCII characters within a pair of square bracket.
- Length of this character atom is one.
- If any of the character within the class matches in a text then the pattern match is succeeded otherwise not.
- Example



(Unsuccessful Search)



(Successful Search)

- Regular expression that matches only vowels:
  [aeiouAEIOU]
- Regular expression that matches any text which contains numeric digit 0-9:
  [0-9]
- Regular expression that matches only alphabets:
  [A-Za-z]
- Regular expression that matches any character other than digit is:
  [^0-9] **//no digits only characters**
- Regular expression that matches character other than vowel is:
  [^AEIOUaeiou]
- Regular expression that matches all digits and dash:
  [0-9\-]
- Regular expression that matches text that contains alphabets digit and ^
  [\^0-9A-Za-z]
- Regular expression that matches any character other than alphabet is:
  [^A-Za-z]
- Regular expression that matches any special character
  [^A-Za-z0-9]
- Regular expression that matches only capital vowel:
  [AEIOU]

4) **Anchor**:
- Anchors are the atoms that are not matched to text but it defines where the character in the regular expression must be located in a text.
- There are 4 types of anchor:

Anchor	Meaning
^	It matches pattern at the beginning of line
$	It matches pattern at the end of line
\<	It matches pattern at the beginning of word
\>	It matches pattern at the end of word

Example:
- ✓ ^a → It matches a line that starts with character 'a'.
- ✓ a$ → It matches a line that ends with character 'a'.
- ✓ \<a → It matches a line in which any word starts with character 'a'.
- ✓ a\> → It matches a line in which any word ends with character 'a'.

5) **Back Reference**:
- It is used **to match one or more characters to text**, previously saved in a buffer. We can use **up to 9 buffers**.
- So we can use **9 back reference (\1,\2,…,\9)**.

- Back references are used with **save operator.**

- **OPERATORS:**
  - It plays powerful role in creation of regular expression.
  - To combine atoms with operator, we can create more complex regular expression.
  - 5 types of operators:
    - ✓ Sequence Operator
    - ✓ Alternation Operator
    - ✓ Repetition Operator
    - ✓ Group Operator
    - ✓ Save Operator

1) Sequence Operator:

- This operator concatenates a series of atoms in a regular expression. Then this operator is implicitly included between them.
- Whenever we use one or more atoms one after another in a regular expression then there is one sequence operator between each of them.
- Examples of sequence operator:

Expression	**Meaning**
Software	Matches pattern 'Software'.
^The	Matches pattern 'The' at the **beginning of the line.**
\<[a-z]..[0-9]\>	Matches **4-characters** pattern that starts with **small alphabet and end with digit.**

2) Alternation Operator:

- This operator is **denoted by pipe (|)**
- It is used to define **one or more alternatives of patterns**.
- Examples,

Expression	**Meaning**
Hardware | Software	Matches pattern 'Hardware' or 'Software'.
Unix | UNIX	Matches pattern 'UNIX' or 'unix'.

3) Repetition Operator:

- It is represented by a pair of escaped curly braces i.e. \{…\}
- It specifies that the atom or expression written before may be repeated.
- It should be written as: **atom/expression \{m, n\}**
- It shows that **previous** atom or expression will be **repeated from m to n times**.
- The expression written in escaped curly braces is known as **repetition operator**.
- It is also known as **braced regular expression** (BRE).
- Example,

REGULAR EXPRESSION	**MEANING**
a\{5,10\}	It repeats character **'a' 5 to 10** times
[a-zA-Z]\{15,20\}	It repeats **alphabets 15 to 20 times**
.\{5,10\}	It repeats **any character 5 to 10 times**
[a-zA-Z]\{10\}	It matches a line that contains 10 alphabets
[a-zA-Z]\{5,\}	It matches a line that contains at least 5 alphabets and max can be any. OR Matches lines that contains 5 or more alphabets
[a-zA-Z]\{,5\}	It matches a line that contains at most 5 alphabets. . OR Matches lines that contains less than or equal to 5 alphabets
Ch* is equivalent to ch\{0,\}	Ch can occur 0 or more times

Ch? is equivalent to ch\{0,1\}	Ch can occur 0 or one times
Ch+ is equivalent to ch\{1,\}	Ch can occur 1 or more times

4) **Group Operator:**
   - It is a **pair of opening and closing parenthesis** that allows the **next operator** to be applied to the **whole group.**
   - It can be written as: **(exp1|exp2|exp3|…|exp N) exp**
   - It **concatenates** any of the expression **between parentheses with exp**.
   - Example,

REGULAR EXPRESSION	MEANING
(unix|linux)OS	Match a line which contains pattern unix OS or linux OS
(hard|soft|firm)ware	Match a line which contains pattern hardware or Software or firmware.

5) **Save Operator:**
   - It is denoted by a pair of **escaped parenthesis** i.e. **\(…\)**
   - The save operator saves **one or more characters** enclose within escaped parenthesis in a buffer to be matched later with.
   - General form of save operator is: **\(exp 1\)\(exp 2\)…\(exp 9\)exp**
     Here exp 1 saved in buffer 1, exp 2 saved in buffer 2 and so on up to 9th buffer which saves exp9.
   - These buffers can be referred by **using back reference**.
   - Buffer 1 can be referred by \1
   - Buffer 2 can be referred by \2 and so on.
   - To match a line **that start and end with same character** then regular expression will be**: ^\(.\).*\1$**
   - To match a pattern
     like **hello, programming** etc.: \(.\)\1.*
   - To match a pattern
     like **12321kj, madam, nayana etc**.: \(.\)\(.\).\2\1.*
   - To match pattern
     like **11, 1abcfd1, 4hd4 etc**. : ^\([0-9]\).*\1$

grep

- grep stands for **Globally search regular Expression and Print it**.
- It is also known as pattern matching utility.
- grep scans its **input for a pattern**, and **display the selected pattern**, the **line numbers** or the **filenames** where the pattern occurs.
- The pattern that is searched in the file is referred to as the **regular expression**.
- Syntax:

  $ grep [options] pattern filename(s)
- grep is a filter.
- It can search its standard input for the pattern and stores the output in a file:

  $ who|grep kumar >file2
- Here search will be performed in **output of who**, and pattern to be searched will be **kumar,** then it will be saved in file **file2**
- Another example: $ **grep "sales" file1**
- Pattern can be given with quotes as well as without quote.
- grep gives **prompt if pattern not found.**
- Example with two filenames:

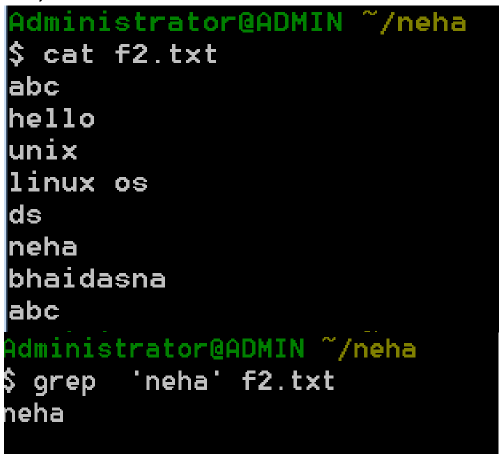
  $grep "director" f1 f2

- When a pattern contains multiple words then quoting is essential as:
  $grep 'jai sharma' f1
- If there is command substitution or variable evaluation in a pattern then double quotes should be used.

GREP COMMAND OPTIONS:

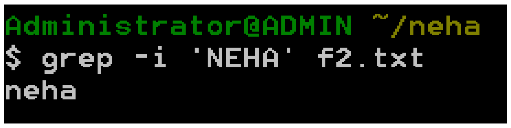
OPTIONS	SIGNIFICANCE
-i	**Ignores** case for matching
-v	**Doesn't** display lines **matching expression**
-n	Display **line number** along with line
-c	Displays count of **occurrences**
-l	Display list of **filenames only**
-e exp	Specifies expression exp with this option. It can use multiple times.
-x	Matches pattern with **entire line**
-f filename	Takes pattern from file, **one per line**
-E	Treats pattern as an extended regular expression(ERE)
-F	Matches multiple **fixed strings**

Examples,



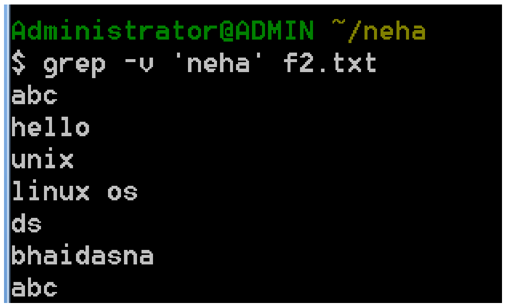
```
Administrator@ADMIN ~/neha
$ cat f2.txt
abc
hello
unix
linux os
ds
neha
bhaidasna
abc
Administrator@ADMIN ~/neha
$ grep  'neha' f2.txt
neha
```

1. $grep –i 'NEHA file1



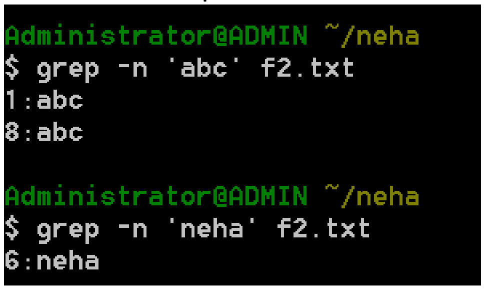
```
Administrator@ADMIN ~/neha
$ grep -i 'NEHA' f2.txt
neha
```

2. $grep –v 'director' file1>f1   #selects unmatched lines



```
Administrator@ADMIN ~/neha
$ grep -v 'neha' f2.txt
abc
hello
unix
linux os
ds
bhaidasna
abc
```

3. $grep –n ‘marketing’ file1 #displays line number with selected lines at which line that line is present

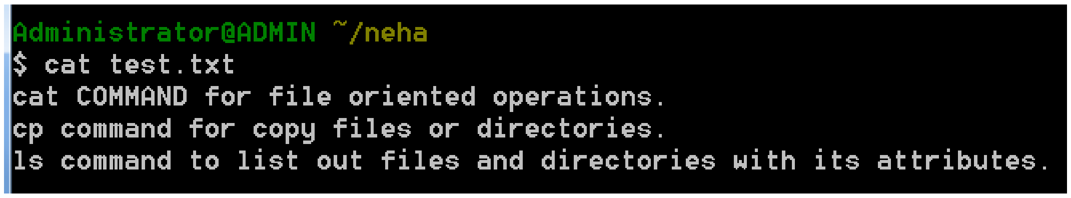


```
Administrator@ADMIN ~/neha
$ grep -n 'abc' f2.txt
1:abc
8:abc

Administrator@ADMIN ~/neha
$ grep -n 'neha' f2.txt
6:neha
```

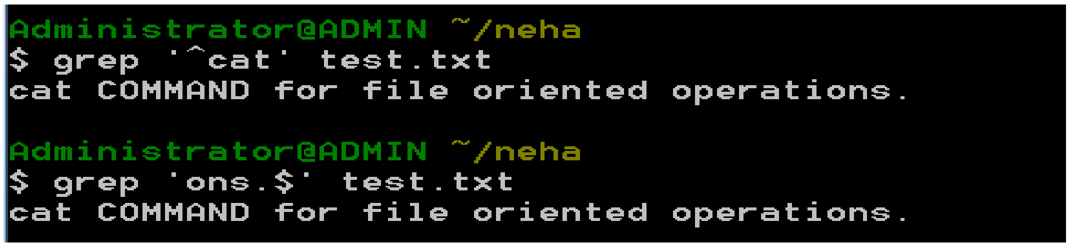
4. $grep –c ‘director’ file1 #output:3
   $ grep –c director emp*.lst # counts lines containing pattern
   Output: emp.lst:4
   emp1.lst:2
   empold.lst:6
   emp2.lst:6
5. $grep –l ‘manager’ *.lst #diisplays name of file containing pattern
6. $grep –e “Agarwal” -e “aggarwal” -e “agarwal” file1
   # -e is used to select multiple patterns at a time
7. $grep –f pattern.lst file1

➢ **BASIC REGULAR EXPRESSION (BRE):**

SYMBOLS OR EXPRESSION	**MATCHES**
*****	Zero or more occurrences of the previous character
g*	Nothing or g,gg,ggg etc
.	A **single characte**r OR Matches any **one character**
.*	Nothing or any number of characters
[pqr]	A single **character p,q or r** Matches any **one of a set** characters
[c1-c2]	A single character within the **ASCII range** represented by c1 and c2 Matches any one of a range characters
[1-3]	A digit between **1 and 3**
[^pqr]	A single character which is **not** a p,q or r
[^a-zA-Z]	A non-alphabetic character
^pat	Pattern pat at **beginning of line**
pat$	Pattern pat at **end of line**
bash$	bash at **end of line**
^bash$	bash as the only word in line
^$	Lines containing nothing



```
Administrator@ADMIN ~/neha
$ cat test.txt
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
```



```
Administrator@ADMIN ~/neha
$ grep '^cat' test.txt
cat COMMAND for file oriented operations.

Administrator@ADMIN ~/neha
$ grep 'ons.$' test.txt
cat COMMAND for file oriented operations.
```

- **THE CHARACTER CLASS:**
  - $ **grep "[aA]g[ar][ar]wal" f1**

  - $ **grep "[aA]gg*[ar][ar]wal" f1**

  - **THE DOT (.)**
    $ **grep "j.*saxena" f1**
    $ **grep a.*Agarwal f1**
    ```
    2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
    ```
    $ grep a.Agarwal f1

  - **SPECIFYING PATTERN LOCATION(^ and $)**
    - **$ - for matching at the end of line**
    - Example,
      $ grep "6...$" f1
      ```
      2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
      1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
      ```

      $ grep "2...$" f1
      ```
      2567 | anju agarwal | accountant | purchase | 12/7/76 |2000
      ```

    - **^ - for matching at the beginning of a line**
    - Example,
      $ grep "^[^2]" f1 # Displays the files which are not beginning with 2
      $ ls –l | grep "^d" #shows only directory
      $ ls -l | grep ^d
      ```
      drwxrwxr-x    3 nirzari   nirzari     4096 Jul 28 02:26 d1
      dr-xr-xr-x    2 nirzari   nirzari   4096 Jul 29 02:13 d3
      drwxrwxr-x    2 nirzari   nirzari   4096 Jul 29 05:40 d5
      drwxrwxr-x     3 nirzari   nirzari    4096 Jul 14  2014 d6
      ```

- **EXTENDED REGULAR EXPRESSIONS(ERE) AND egrep:**
  - **Solaris** user uses grep for extended regular expression with –E option.
  - If your system not support this then use egrep without –E
  - The ERE set includes 2 special characters:

  1. **+** → It is used to matches **one or more occurrence** of the previous character
  2. **?** → It is used to matches **zero or one occurrence** of the previous character

  Example:
  $ grep –E "[aA]gg?arwal" f1
  $ **grep -E [aA]gg?arwal f1**
  ```
  2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
  2567 | anju agarwal | accountant | purchase | 12/7/76 |2000
  ```

  $ grep [aA]gg?arwal f1
  $ **egrep [aA]gg?arwal f1**

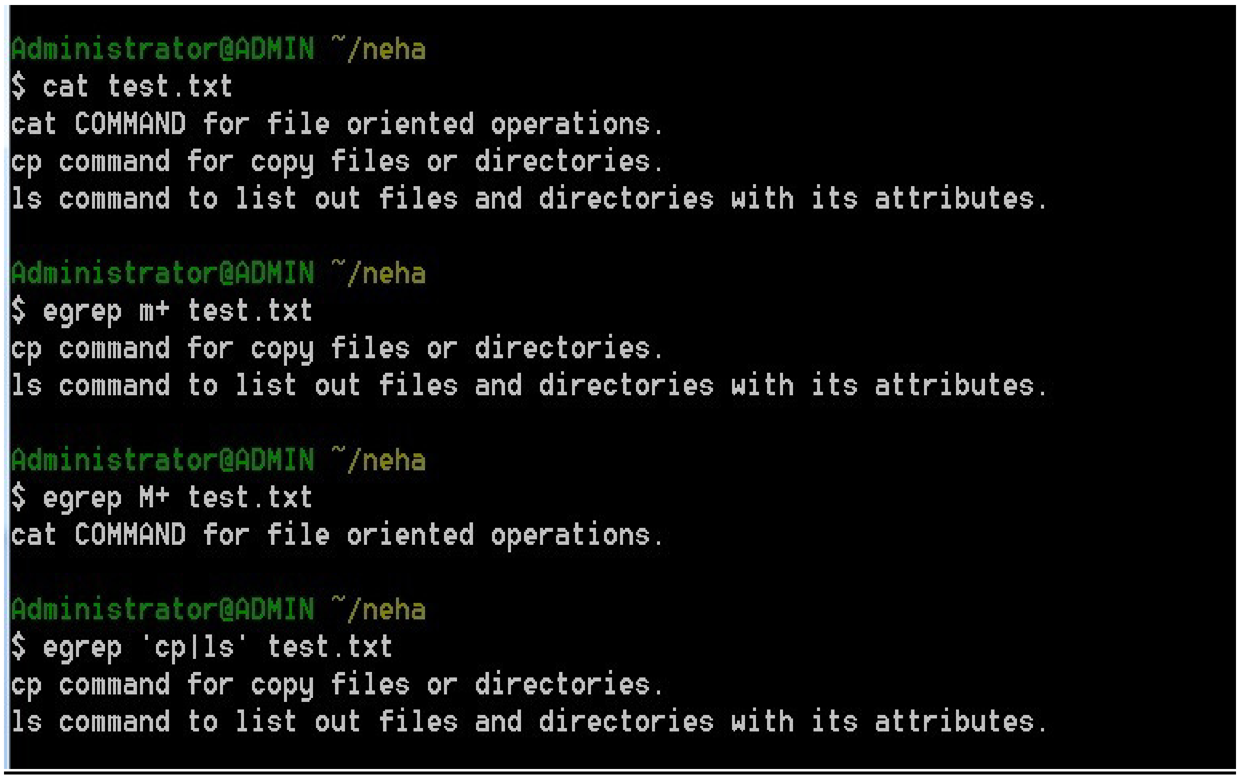
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76 |2000

$ grep –E "#?include+<stdio.h>"

- **MATCHING MULTIPLE PATTERNS (|,(AND))**
  - Pipe is the delimiter for multiple patterns.
  - **Note**: here single quote is must
  - Example,
    $ **grep –E 'sengupta|dasgupta' f1**
    The characters '(' and ')' group pattern, and do same as above
    $ **grep –E '(sen|das)gupta' f1**
  - ERE's when combines with BRE's forms very powerful regular expression.
  - Example: **$ grep –E 'agg?[ar]+wal' file1**

  - **The Extended Regular expression(ERE) used by grep, egrep and awk are as follows:**

EXPRESSION	SIGNIFICANCE
Ch +	Matches one or more occurrence of character ch
Ch?	Matches zero or one occurrence of character ch
Exp1|exp2	Matches exp1 or exp2
(x1|x2)x3	Matches x1x3 or x2x3
(lock|ver)wood	Matches lockwood or verwood



```
Administrator@ADMIN ~/neha
$ cat test.txt
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.

Administrator@ADMIN ~/neha
$ egrep m+ test.txt
cp command for copy files or directories.
ls command to list out files and directories with its attributes.

Administrator@ADMIN ~/neha
$ egrep M+ test.txt
cat COMMAND for file oriented operations.

Administrator@ADMIN ~/neha
$ egrep 'cp|ls' test.txt
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
```

- **FGREP : SEARCH A FILE FOR A FIXED -** CHARACTER STRING**:**
  - fgrep command is used to extract **fixed patterns**.
  - Patterns cannot extract from character class or special meta-character. Here f in fgrep stands for fixed pattern.
  - If pattern to be searched is a simple string, or a group of string then fgrep command is recommended.

- fgrep command is faster than grep and egrep.
- Example,
  - $ fgrep -x 'manager' f1
  - $ fgrep 'manager' f1
  - $ fgrep manager f1

```
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
```

  - $ fgrep '[a-z]*' f1
  - $ grep '[a-z]*' f1

```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76 |2000
```

- **#more than one pattern can be given by a newline character**
  - $ fgrep 'manager
    sales' f1

```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
```

- # to take pattern from a file
  $ fgrep -f f2 f1

```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
```

# points to be noted:

1. In egrep, if a pattern contains some special characters then it must be quoted.
2. –f option to extract pattern from a file also works in egrep.

# Some more grep commands:

1. grep '.' F1 #displays all lines except blank line.
2. grep '\.' F1 #hides special meaning of . ,dot can occur anywhere in a line
3. quotes compulsory in variable substitution as:
   $a=1
   $ grep "$a" f1 #display lines containing 1.
4. quotes compulsory in command substitution:
   $ grep "`echo hello`" f1
   $ grep " `echo sales` " f1

```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
```

5. $ grep 'mm*' f1 # * means 0 or more occurrence of previous character i.e. 0 or
   more time 'm' is occurred
6. $ grep '^.\{8\}$' f1 # Display the lines from file having length 8

7. $ grep '^.\{5,15\}$' f1 # Display the lines from file having Minimum length 5 and Maximum length 15

8. $ grep '^\(.\).*\1' f1

SED : THE STREAM EDITOR:

- A special editor for **modifying files automatically**.
- To write a program to make changes in a file, sed tool is use.
- Sed command is the ultimate stream editor.
- Sed command performs non-interactive operations on a data stream.
- Sed command uses **instructions** to act on text.
- An instruction combines an address for selecting lines, with an action to be taken on them.
- SED command in UNIX is stands for stream editor and it can perform lot's of function on file like, searching, find and replace, insertion or deletion.
- Though most common use of SED command in UNIX is for substitution or for find and replace.
- By using SED you can edit files even without opening it, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.
- 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.

- **Syntax:** **$ sed options 'address action' file(s)**
- The address and action are enclosed within single quotes

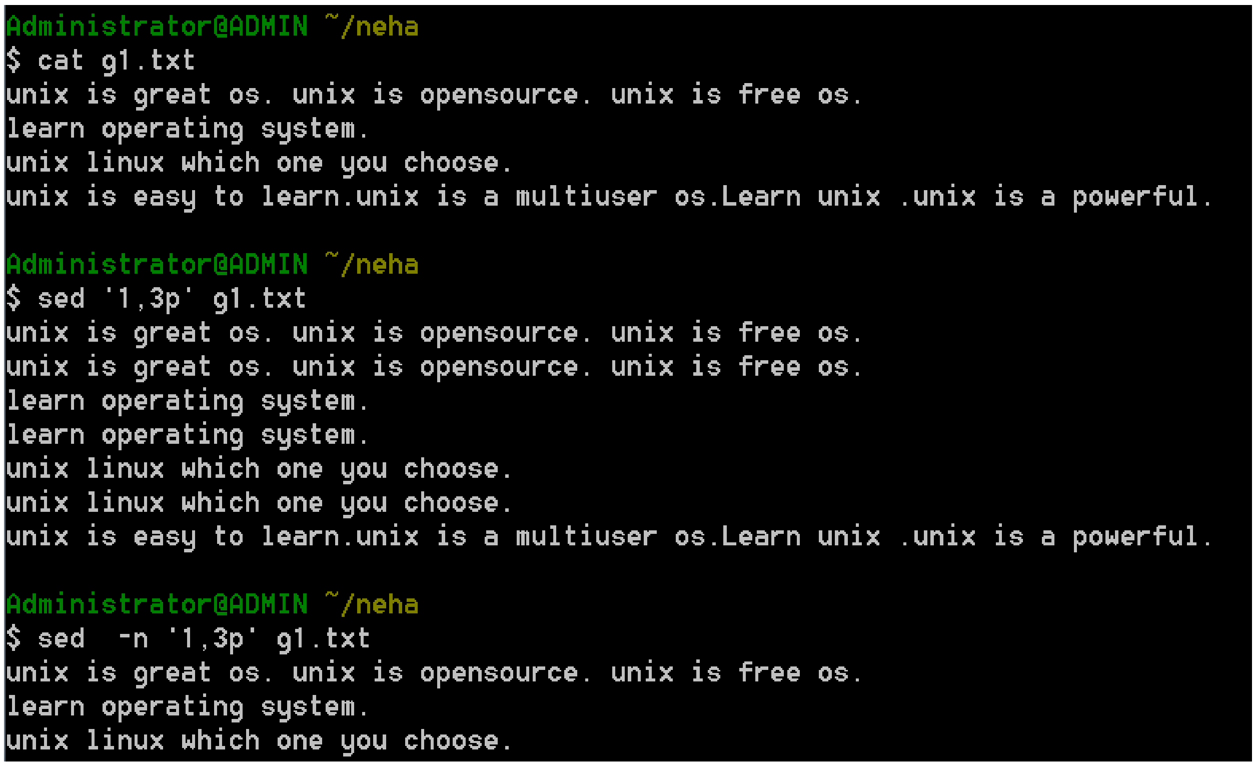
sed commands:

- **The sed s**upports several commands. Commands are used to apply on specified lines. They are

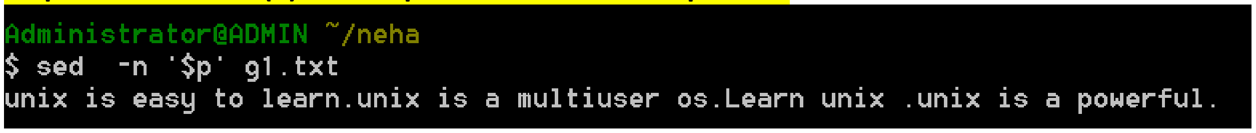
1.Print
2.Quit
3.Line number
4.Modify
5.Files
6.Substitute

1.Print :

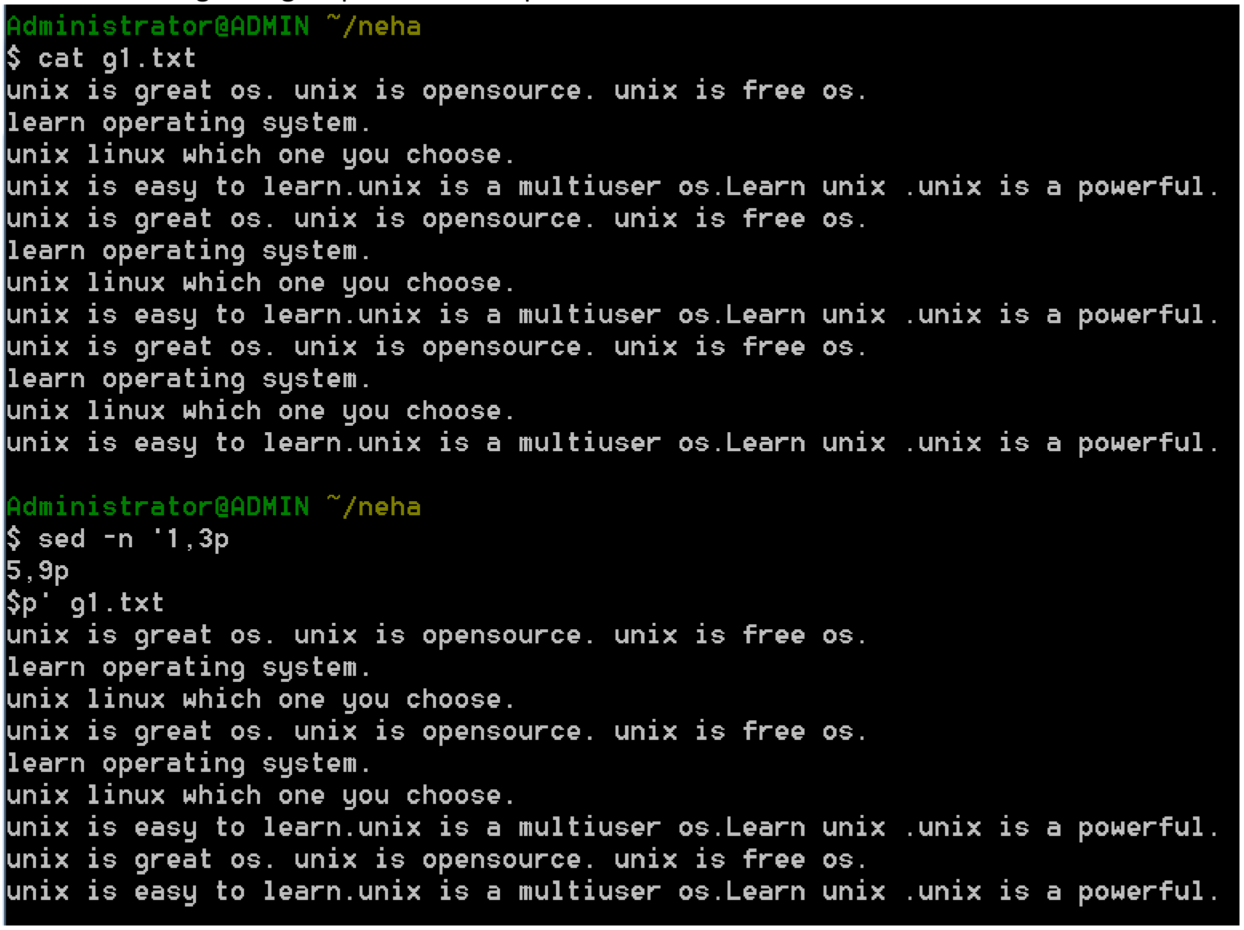
It is denoted by character **p.** It prints selected lines on standard output.
p (print) action shows all lines as well as selected lines , so selected lines will comes twice to remove delicacy –n option is used with p(print action).
Example:


```
Administrator@ADMIN ~/neha
$ cat g1.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.

Administrator@ADMIN ~/neha
$ sed '1,3p' g1.txt
unix is great os. unix is opensource. unix is free os.
unix is great os. unix is opensource. unix is free os.
learn operating system.
learn operating system.
unix linux which one you choose.
unix linux which one you choose.
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.

Administrator@ADMIN ~/neha
$ sed  -n '1,3p' g1.txt
unix is great os. unix is opensource. unix is free os.
learn operating system.
unix linux which one you choose.
```

A special character ($) is used print last line of an input file.


```
Administrator@ADMIN ~/neha
$ sed  -n '$p' g1.txt
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

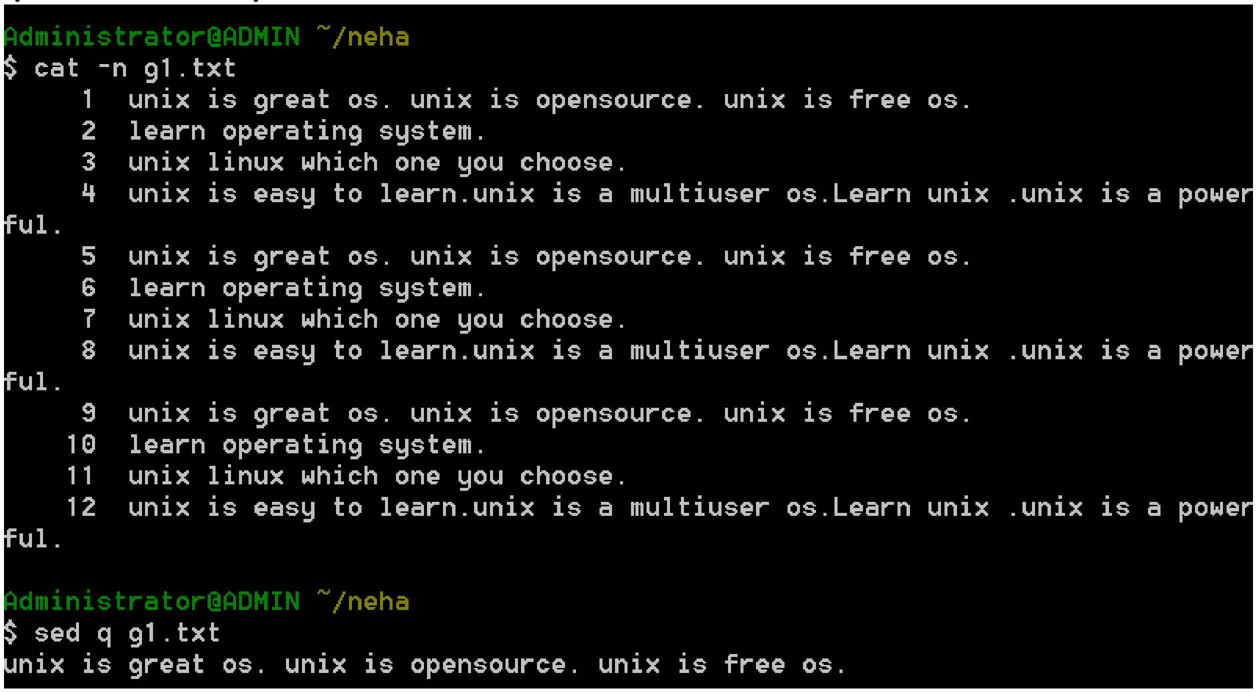
- command p without any option displays all lines from particular file. $ sed -n p g1.txt
- select non contiguous group of lines of input file then command is as follow:


```
Administrator@ADMIN ~/neha
$ cat g1.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.
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.
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.

Administrator@ADMIN ~/neha
$ sed -n '1,3p
5,9p
$p' g1.txt
unix is great os. unix is opensource. unix is free os.
learn operating system.
unix linux which one you choose.
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.
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.
```
It will display 1 to 3 lines, 7 to 9 and last line from file g1.txt

- **Selecting lines from anywhere:** **$ sed -n '9,11p' f1**
- **Negating the action(!):** $Sed –n '3,$!p' f1 don't print line 3 to the end

2.Quit:

This command denoted by character q.
It uses a single address i.e. it does not allow range of address.
It quits after reading up to address lines.
example:
q without address prints first line.

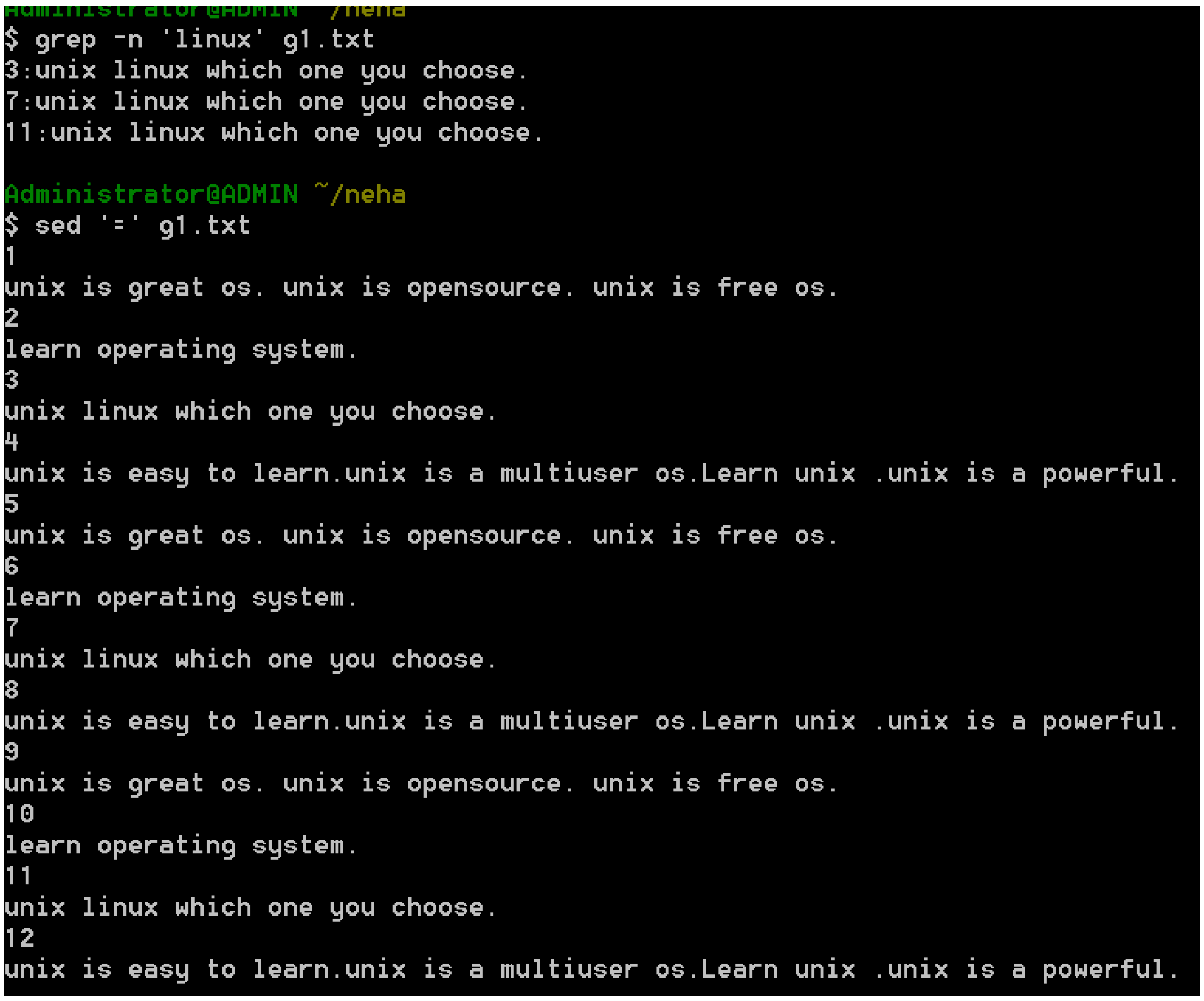


```
Administrator@ADMIN ~/neha
$ cat -n g1.txt
     1  unix is great os. unix is opensource. unix is free os.
     2  learn operating system.
     3  unix linux which one you choose.
     4  unix is easy to learn.unix is a multiuser os.Learn unix .unix is a power
ful.
     5  unix is great os. unix is opensource. unix is free os.
     6  learn operating system.
     7  unix linux which one you choose.
     8  unix is easy to learn.unix is a multiuser os.Learn unix .unix is a power
ful.
     9  unix is great os. unix is opensource. unix is free os.
    10  learn operating system.
    11  unix linux which one you choose.
    12  unix is easy to learn.unix is a multiuser os.Learn unix .unix is a power
ful.

Administrator@ADMIN ~/neha
$ sed q g1.txt
unix is great os. unix is opensource. unix is free os.
```

LINE ADDRESSING **:$ sed '3q' f1** #quits after line number 3,here 3 is address and q(quit) is action

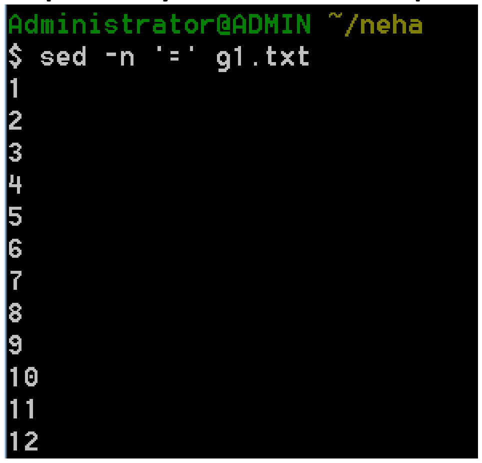
3.Line number:

It is denoted by equal (=).
It write line number of addressed line at the beginning of line.
It similar to -n option of grep. but here line numbers are written in separate line.
Example;



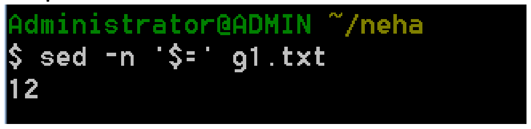
```
Administrator@ADMIN ~/neha
$ grep -n 'linux' g1.txt
3:unix linux which one you choose.
7:unix linux which one you choose.
11:unix linux which one you choose.

Administrator@ADMIN ~/neha
$ sed '=' g1.txt
1
unix is great os. unix is opensource. unix is free os.
2
learn operating system.
3
unix linux which one you choose.
4
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
5
unix is great os. unix is opensource. unix is free os.
6
learn operating system.
7
unix linux which one you choose.
8
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
9
unix is great os. unix is opensource. unix is free os.
10
learn operating system.
11
unix linux which one you choose.
12
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

To print only line numbers -n option is used.



```
Administrator@ADMIN ~/neha
$ sed -n '=' g1.txt
1
2
3
4
5
6
7
8
9
10
11
12
```

To print last line number $ is used with -n.

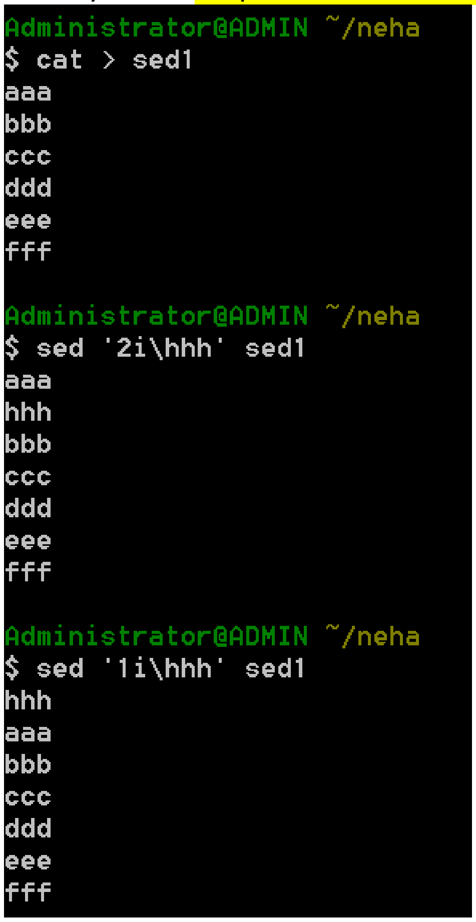


```
Administrator@ADMIN ~/neha
$ sed -n '$=' g1.txt
12
```

4.Modify:

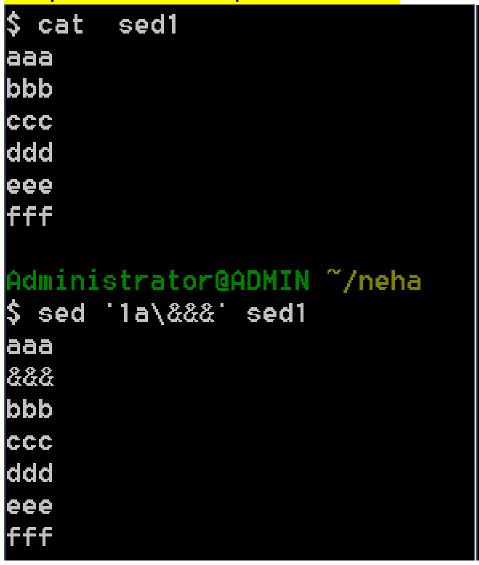
There are different purpose of this command.
it allows you to insert, append, change or delete lines.
They do not modify just a part of a line that means they work on entire line.
This command has different options.

- **Insert command (i) :** it is denoted by character i. It is insert one or more lines directly to the output before the address lines.

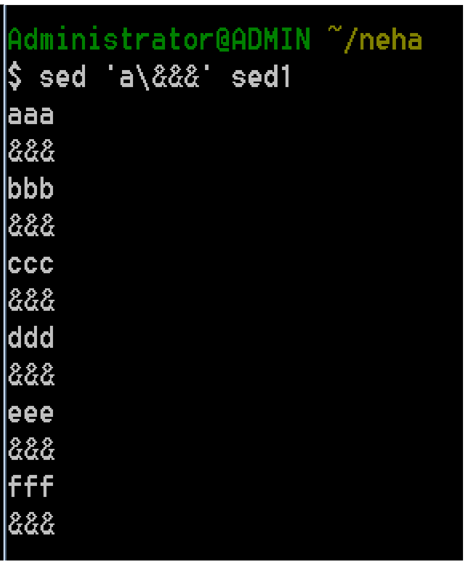


```
Administrator@ADMIN ~/neha
$ cat > sed1
aaa
bbb
ccc
ddd
eee
fff

Administrator@ADMIN ~/neha
$ sed '2i\hhh' sed1
aaa
hhh
bbb
ccc
ddd
eee
fff

Administrator@ADMIN ~/neha
$ sed '1i\hhh' sed1
hhh
aaa
bbb
ccc
ddd
eee
fff
```

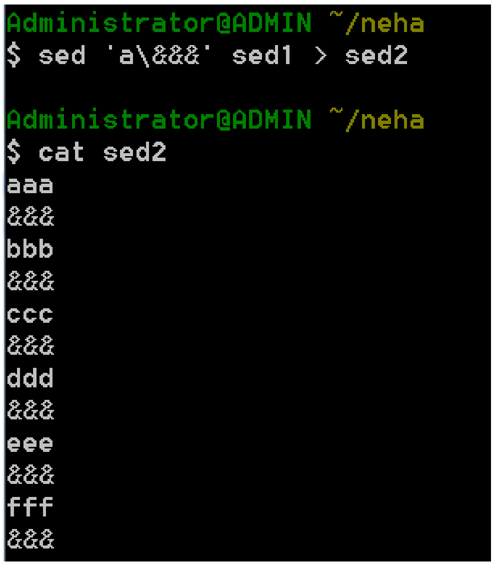
- **append command (a) :** it is denoted by character **a**.
  It is similar to insert command except that it writes the text directly to the output after the specified line.



```
$ cat  sed1
aaa
bbb
ccc
ddd
eee
fff

Administrator@ADMIN ~/neha
$ sed '1a\&&&' sed1
aaa
&&&
bbb
ccc
ddd
eee
fff
```

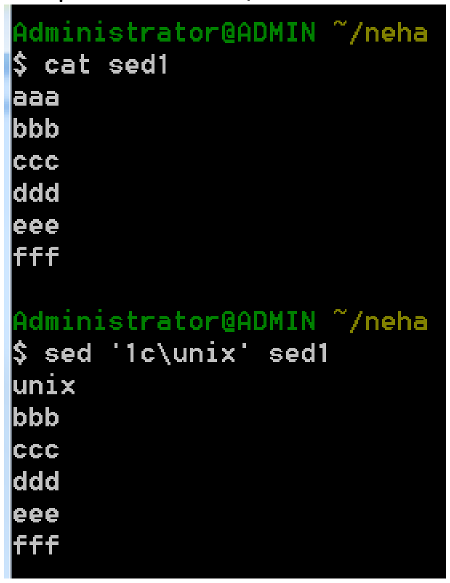


```
Administrator@ADMIN ~/neha
$ sed 'a\&&&' sed1
aaa
&&&
bbb
&&&
ccc
&&&
ddd
&&&
eee
&&&
fff
&&&
```

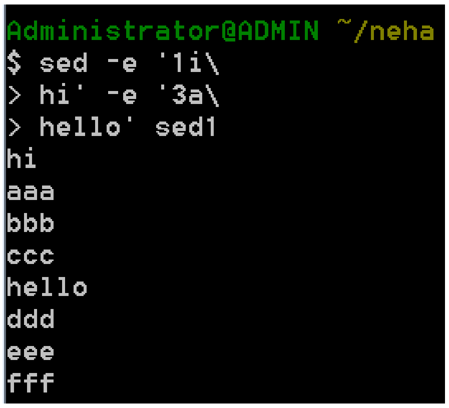


```
Administrator@ADMIN ~/neha
$ sed 'a\&&&' sed1 > sed2

Administrator@ADMIN ~/neha
$ cat sed2
aaa
&&&
bbb
&&&
ccc
&&&
ddd
&&&
eee
&&&
fff
&&&
```

- **change command (c) :** it is denoted by character c. It replaces address/matched line with new text.

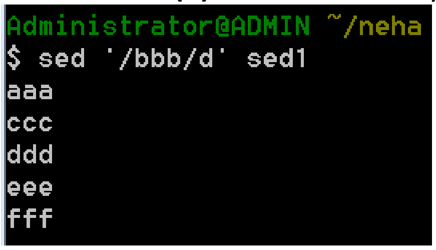


```
Administrator@ADMIN ~/neha
$ cat sed1
aaa
bbb
ccc
ddd
eee
fff

Administrator@ADMIN ~/neha
$ sed '1c\unix' sed1
unix
bbb
ccc
ddd
eee
fff
```



```
Administrator@ADMIN ~/neha
$ sed -e '1i\
> hi' -e '3a\
> hello' sed1
hi
aaa
bbb
ccc
hello
ddd
eee
fff
```

- **delete command (d) :** it is denoted by character d.



```
Administrator@ADMIN ~/neha
$ sed '/bbb/d' sed1
aaa
ccc
ddd
eee
fff
```

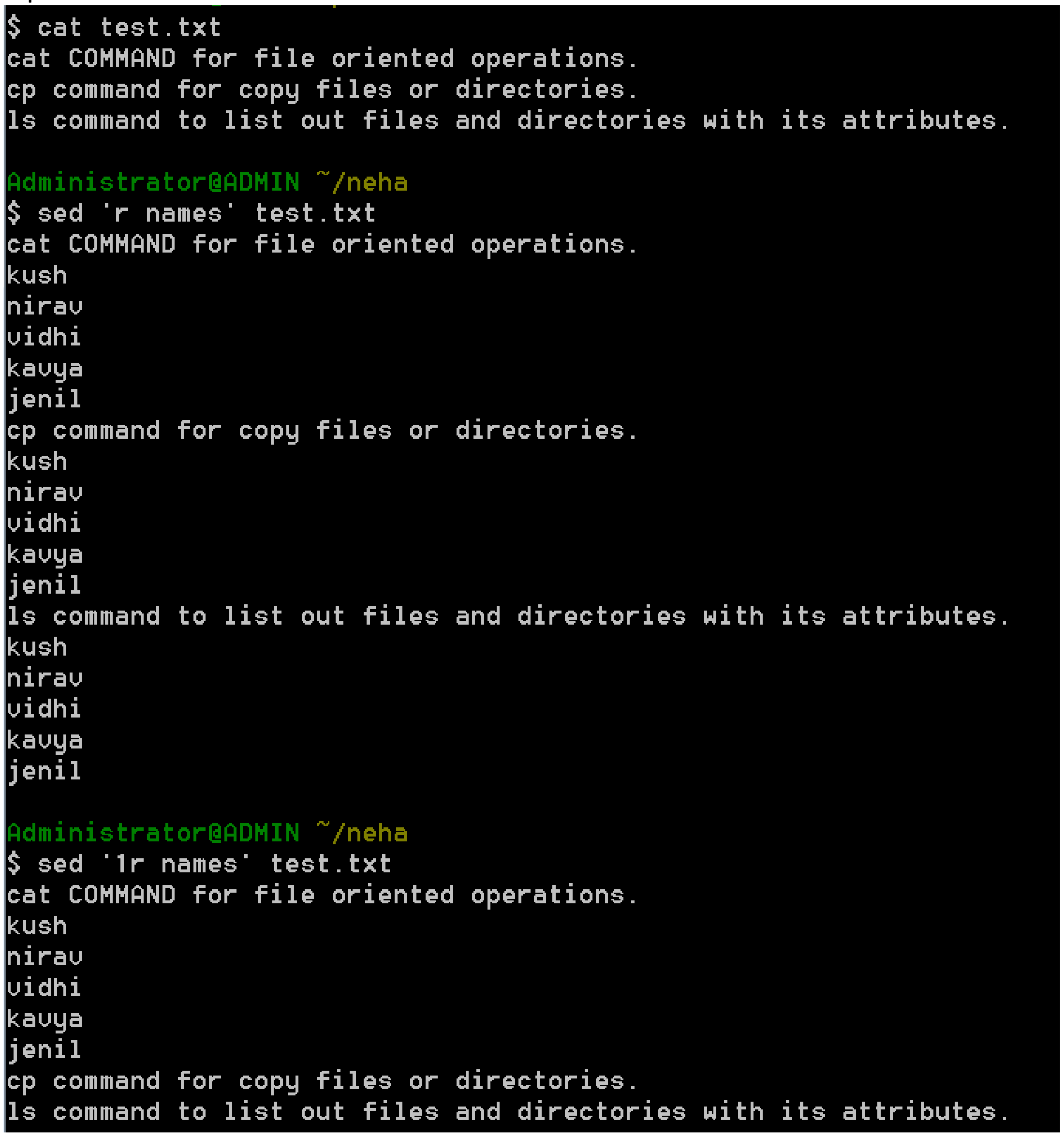
5.Files:

File command is used to read or write data from other file respectively.

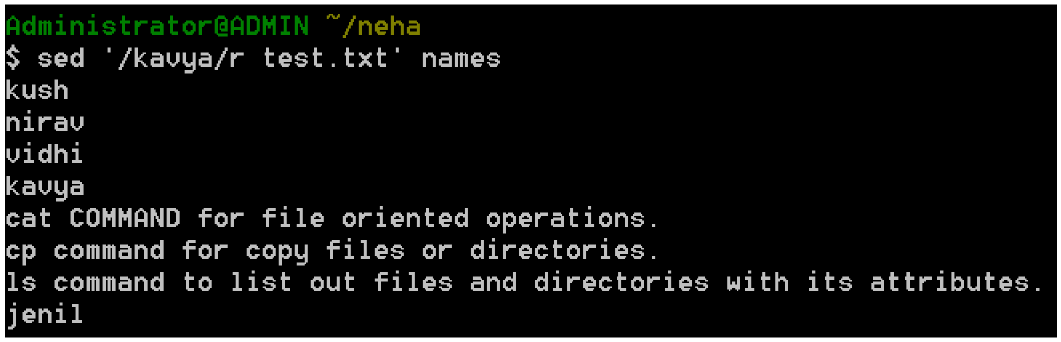
There are two types of commands :

- **read file :** it is denoted by **r fname.** When a user wants to insert common content of a file after specified line of an input file then this command is useful.

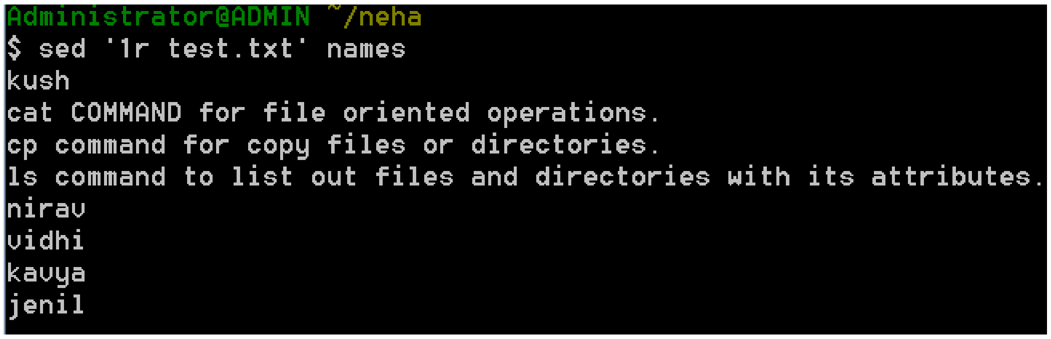
  It reads text from file **fname** and place its content after a specified line of input file.



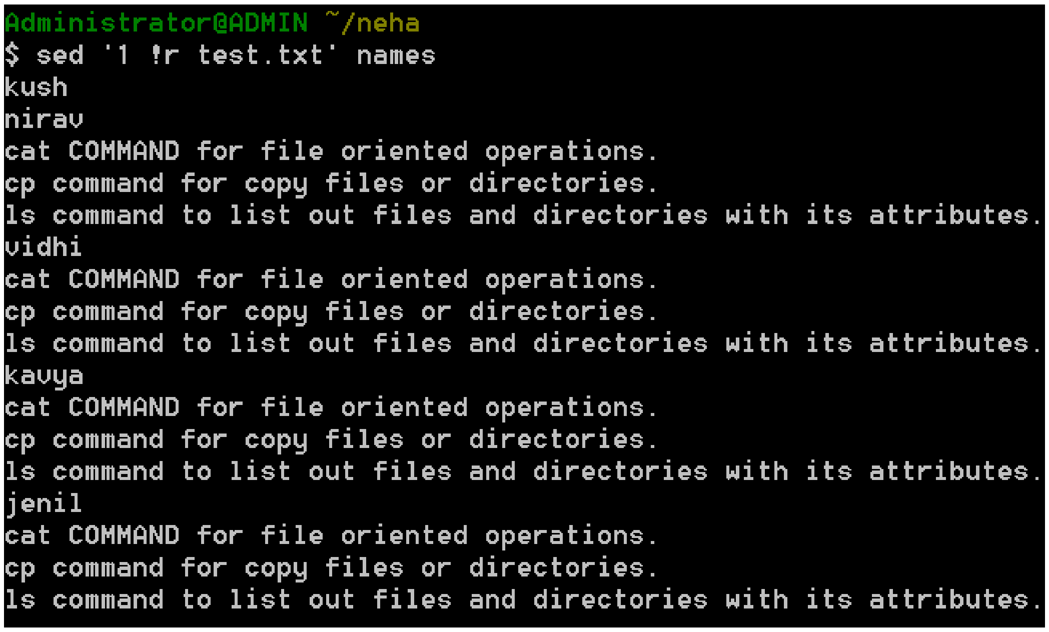
```
$ cat test.txt
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.

Administrator@ADMIN ~/neha
$ sed 'r names' test.txt
cat COMMAND for file oriented operations.
kush
nirav
vidhi
kavya
jenil
cp command for copy files or directories.
kush
nirav
vidhi
kavya
jenil
ls command to list out files and directories with its attributes.
kush
nirav
vidhi
kavya
jenil

Administrator@ADMIN ~/neha
$ sed '1r names' test.txt
cat COMMAND for file oriented operations.
kush
nirav
vidhi
kavya
jenil
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
```



```
Administrator@ADMIN ~/neha
$ sed '/kavya/r test.txt' names
kush
nirav
vidhi
kavya
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
jenil
```

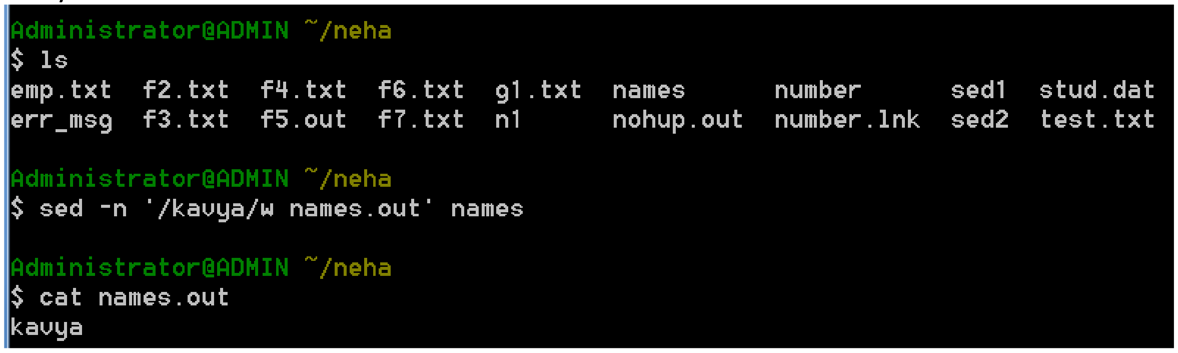


```
Administrator@ADMIN ~/neha
$ sed '1r test.txt' names
kush
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
nirav
vidhi
kavya
jenil
```



```
Administrator@ADMIN ~/neha
$ sed '1 !r test.txt' names
kush
nirav
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
vidhi
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
kavya
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
jenil
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.
```

- **Write file :** it is denoted by **w fname.**
  The write file command makes possible to write the selected lines in a separate file.
  To write selected lines of input file then command id as below:
  Here in output it writes lines from names file to names.out file having pattern 'kavya'.



```
Administrator@ADMIN ~/neha
$ ls
emp.txt  f2.txt  f4.txt  f6.txt  g1.txt  names      number      sed1  stud.dat
err_msg  f3.txt  f5.out  f7.txt  n1      nohup.out  number.lnk  sed2  test.txt

Administrator@ADMIN ~/neha
$ sed -n '/kavya/w names.out' names

Administrator@ADMIN ~/neha
$ cat names.out
kavya
```

similarly if you want to write top 5 lines from input file to the output file command is as below:

$ sed '1,5w f4.txt' names.out

A user can create multiple output files that contain selected lines of input file.

$ sed -n '/linux/w file   <enter>
> /unix/w ufile' f1

OR

$ sed -ne '/linux/w file' -e '/unix/w ufile' f1

It will writes lines that contain pattern linux to a file and lines that contain unix pattern from ufile to a f1.

6.Substitute commands: (s)

It is denoted by character S.

It scan lines for search pattern and substitution it with replacement string.

This command is similar to the search and replacement feature of text editor.

This feature provides us to add, delete or change text in one or more lines.

The format of the substitution command is as follow:

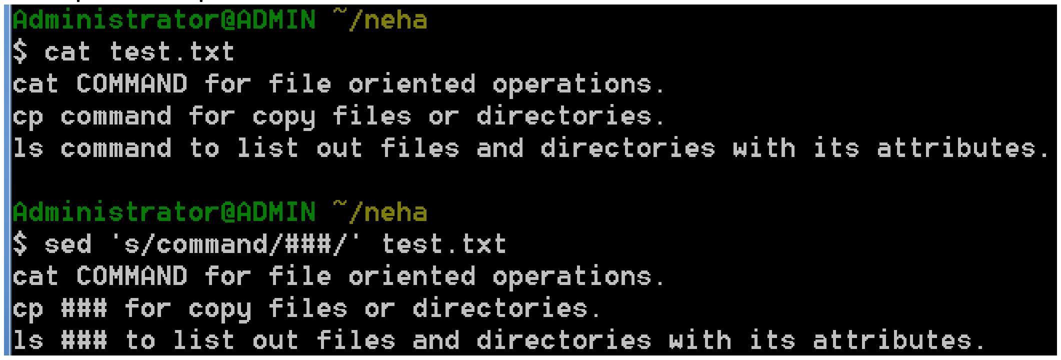
[address or scanned_pattern] s/search_pattern/replace_string/[flags(s)]

Here if address is not specified, the substitution will be performs for all lines containing first occurrence of search_pattern may be regular expression or literal string.

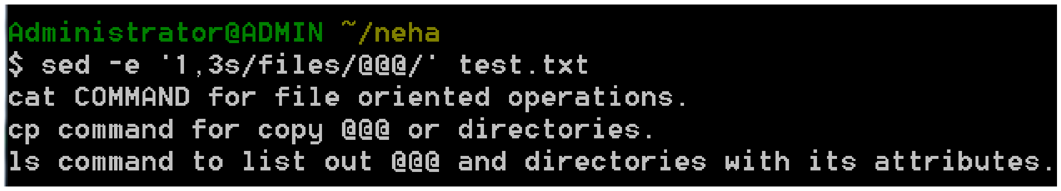
Both search_pattern and replace_string are delimited by slash(/).

The replace_string is a string that consist of either ordinary character or an atom or meta-characters or combination of them.

Example: To replace 1st occurrences of "command" with '###' command is as below:



```
Administrator@ADMIN ~/neha
$ cat test.txt
cat COMMAND for file oriented operations.
cp command for copy files or directories.
ls command to list out files and directories with its attributes.

Administrator@ADMIN ~/neha
$ sed 's/command/###/' test.txt
cat COMMAND for file oriented operations.
cp ### for copy files or directories.
ls ### to list out files and directories with its attributes.
```



```
Administrator@ADMIN ~/neha
$ sed -e '1,3s/files/@@@/' test.txt
cat COMMAND for file oriented operations.
cp command for copy @@@ or directories.
ls command to list out @@@ and directories with its attributes.
```

Flag(g) : To replace all occurrence user need to use global (g) flag at the end of the instruction. This referred as global substitution.

Example : **$sed 's/for/###/g' test.txt**

Remembered Pattern :

- **USING MULTIPLE INSTRUCTIONS(-e and –f)**
  - **Sed -n –e '1,2p' -e '7,9p' -e '$p' f1** # Giving address action from a file to sed.
  - **$ cat instr.fil**
    1,2p
    7,9p
    $p
  - $ sed -n -f instr.fil f1
  - $ sed -n -f instr.fil1 -f instr.fil2 f1
  - $ sed -n -e '/saxena/p' -f instr.fill -f instr.fil2 emp?.lst

- **CONTEXT ADDRESSING**
  - $ sed –n '/director/p' f1

- $ sed -n '/dasgupta/,/saksena/p' f1
- $ sed   -n '1,/dasgupta/p' f1
- $ sed -n '/[aA]gg*[ar][ar]wal/p' f1
- $ sed -n '/dasgupta/p > /sales/p' f1

  ```
  2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
  1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
  1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
  1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
  2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
  ```
- $ sed –n '/50…..$/p' f1

➢ **WRITING SELECTED LINES TO A FILE:**

- W (write) command is used to write the selected lines to a separate file.
- Without –n selected lines will be written to the respective file; -n is used to suppress printing of all lines on the terminal.
- $ sed   -n '/director/w dlist' f1

-
  ```
  $ sed –n '/director/w dlist
  /manager/w mlist
  /executive/w  elist'   f1
  ```

-
  ```
  $ sed  -n  '1,55w  f1
  501,$w  F2'   f.main
  ```

➢ **TEXT EDITING**

- Here, we have some editing commands available in sed's action component.
- **Inserting and changing lines (i , a, c)**

  -
    ```
    $ sed '1i\
          > #include  <stdio.h>\
          > #include<unistd.h>
          > ' foo.c> $$
    ```

  -
    ```
    $ mv $$ foo.c; head –2 foo.c
          #include <stdio.h>
          #include<unistd.h>
    ```

  -
    ```
    $ sed  'a\
    ' emp.lst
    ```

<u>TELNET VERSION:</u>

-
  ```
  $ cat f3
  nirzari    pts/1       Aug 26 02:29 (192.168.0.64)
  ```

-
  ```
  $ sed '1i\
  > hello user\
  > hiii
  > ' f3
      hello user
      hiii
      nnn    pts/1       Aug 26 02:29 (192.168.0.64)
  ```

-
  ```
  $ cat f3
  nnn    pts/1       Aug 26 02:29 (192.168.0.64)
  ```

-
  ```
  $ sed '1i\
  hello user\
  hiii
  ' f3>$$
  ```

-
  ```
  $ cat $$
  ```

```
hello user
hiii
nnn    pts/1       Aug 26 02:29 (192.168.0.64)
```

-
```
$ sed '1i\
> hiiii\
> be ok
> ' f2 >$$
```
-
```
$ cat $$
hiiii
be ok
sales
```

- **DELETING LINES(D)**
  - `$ sed  '/director/d'  f1 > olist`   -n option not to be used with d

<u>TELNET VERSION:</u>

-
```
$ sed '/director/d' f1
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76|2000
```
-
```
$ cat f1
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76|2000
```
- `$ sed  -n  '/director/!p'  f1 > olist`
-
```
$ sed -n '/director/!p' f1
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76|2000
```

- **DELETING BLANK LINES**
  - `$ sed  '/^[]*$/d'   f1`   # a space and a tab inside []

- **<u>SUBSTITUTION (S)</u>**
  - [ADDRESS]s / EXPRESSION1 / EXPRESSION2 / FLAGS
  -
```
$ cat > j3
a b c d
a b c
a b
a
```
  -
```
$ sed 's/a/A/' j3
A b c d
A b c
A b
A
```
  -
```
$ cat j3
a b c d
a b c
```

```
a b
a
```

- $ sed 's/a/A/g' j3
  ```
  A b c d
  A b c
  A b
  A
  ```
- $ sed ‘s/|/:/’ f1 | head -2
- $ sed ‘s/|/:/g’ f1 | head -2

TELNET VERSION:

- **$ sed 's/|/:/' f1 | head -2**
  ```
  2233 : a.k. shukla | g.m | sales | 12/12/52 | 6000
  1006 : chanchal singhvi | director | sales | 13/9/87 | 6700
  ```
- **$ sed 's/|/:/g' f1 | head -2**
  ```
  2233 : a.k. shukla : g.m : sales : 12/12/52 : 6000
  1006 : chanchal singhvi : director : sales : 13/9/87 : 6700
  ```
- **$ sed ‘1,3s/|/:/g’ f1   first 3 lines only**
- **$ sed ‘1,5s/director/member /’   f1**
- **$ sed ‘s/[Aa]gg*[ar][ar]wal/Agarwal/g’ f1**
- **$ sed ‘s/^/2/’ f1| head –n 1**
- **$ sed ‘s/$/.00/’ f1 | head –n 1**

TELNET VERSION:

- **$ sed 's/^/2/' f1**
  ```
  22233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
  21006 | chanchal singhvi | director | sales | 13/9/87 | 6700
  21265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
  22476 | anil Agarwal | manager | sales | 12/7/56 | 5000
  22567 | anju agarwal | accountant | purchase | 12/7/76 |2000
  ```
- $ cat f1
  ```
  2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
  1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
  1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
  2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
  2567 | anju agarwal | accountant | purchase | 12/7/76 |2000
  ```
- **$ sed 's/$/.00/' f1**
  ```
  2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000.00
  1006 | chanchal singhvi | director | sales | 13/9/87 | 6700.00
  1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600.00
  2476 | anil Agarwal | manager | sales | 12/7/56 | 5000.00
  2567 | anju agarwal | accountant | purchase | 12/7/76 |2000.00
  ```

➢ **PERFORMING MULTIPLE SUBSTITUTION**

- **$ sed ‘s/<I>/<EM>/g**
  ```
  > s/<B>/<strong>/g
  > s/<U>/<EM>/g’  form.html
  ```
- **$ sed ‘s/<I>/<EM>g**
  ```
  > s/<EM>/<STRONG>g’  form.html
  ```

➢ **COMPRESSING MULTIPLE SPACES:**

- **$ Sed ‘S/ *|/|/g’ emp.lst | tee empn.lst | head –n 3**

- **THE REMEMBERED PATTERN(//)**
  1) $ sed ‘s/director/member/’ f1
  2) $ sed ‘ /director/s//member/’ f1
  - **$ sed '/director/s//member/' f1**

```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | member | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76|2000
```

  - **$ sed ‘ /director/s/director/member/’ f1**
  - **$ sed ‘s/|//g’ f1**   **removes every | from file**

TELNET VERSION:

  - **$ sed 's/|//g' f1**

```
2233  a.k. shukla  g.m  sales  12/12/52  6000
1006  chanchal singhvi  director  sales  13/9/87  6700
1265  s.n. dasgupta  manager  sales  12/8/67  5600
2476  anil Agarwal  manager  sales  12/7/56  5000
2567  anju agarwal  accountant  purchase  12/7/76 2000
```

  - **$ sed –n ‘ /marketing/s/director/member   /p’   f1**

NOTE: The significance of // depends on its position in the instruction. If it is in the source string, it implies that the scanned pattern is stored there. If the target string is //, it means that the source pattern is to be removed.

- **BASIC REGULAR EXPRESSION REVISITED:**
  - 3 types of expressions:
    1) The Repeated Pattern → This uses a single, &, to make the entire source pattern appear at the destination also.
    2) The Interval Regular Expression (IRE) → This expression uses the characters { and } with a single pair of numbers between them.
    3) The Tagged Regular Expression (TRE) → This expression groups pattern with (and) and represents them at the destination with numbered tags.

THE REPEATED PATTERN (&)

  - **$ sed ‘s/director/executive director/’ f1**

  - **$ sed ‘s/director/executive &/’ f1**

  - **$sed ‘/director/s//executive &/’ f1**

TELNET VERSION:

  - **$ sed 's/director/executive director/' f1**

```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | executive director | sales |13/9/87 | 6700
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76|2000
```

  - **$ sed 's/director/executive &/' f1**

```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | executive director | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76 |2000
```

- **$ sed '/director/s//executive &/' f1**

```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | executive director | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | manager | sales | 12/8/67 | 5600
2476 | anil Agarwal | manager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76|2000
```

<u>INTERVAL REGULAR EXPRESSION (IRE)</u>

1) **ch\{m\}** → The meta-character ch can occur m times
2) **ch\{m,n\}** → Here, ch can occur between m and n times
3) **ch\{m,\}** → Here, ch can occur at least m times

- These are used to mention no of character/character set reputation info. Note that interval regular expression and extended reg require -E option with grep
- **Note:** In order to use this set of regular expressions you have to us -E with grep command and -r option with sed commands.
- **{n} → n occurrence of previous character**
- **{n,m} → n to m times occurrence of previous character**
- **{m, } → m or more occurrence of previous character.**

<u>TELNET VERSION:</u>

- **$ ls -l | grep -E 't{3}'**

```
-rw-rw-r--    1 nirzari    nirzari    5 Sep  3 06:51 ttt
```

- **$ ls -l | grep 't\{3\}'**

```
-rw-rw-r--    1 nirzari    nirzari    5 Sep  3 06:51 ttt
```

- **$ ls -l | grep -E 't\{3\}'**
- **$ ls -l | grep 't{3}'**

Example 1:

Find all the file names which contain “t” and t repeats for 3 times consecutively.
$ ls -l | grep -E ‘t{3}’
-E option is used to extend regexp understanding for grep.

Example 2:

Find all the file names which contain l letter in filename with 1 occurrence to 3 occurrences consecutively.

- $ ls -l | grep -E ‘l{1,3}’
- $ ls | grep -E 'l{1,3}'

```
file2
l1
ll
olist
```

- $ ls | grep -E l{1,3}

```
grep: l3: No such file or directory
```

Example 3:

Find all the file names which contains k letter 5 and more in a file name.
$ ls -l | grep -E 'k{5,}'

This is bit tricky, let me explain this. Actually we had given a range i.e 5 to infinity (Just given only comma after 5).

1) # to select lines that contains mobile numbers, from file teledir.txt
```
$ grep '[0-9]\{10\}' teledir.txt
```
2) To have list of file that have the write bit set for either for group or others:
```
$ ls –l | sed –n '/^.\{5,8\}w/p'
```
3) `$ sed –n '/.\{101,\}/p'  f1`   # line length at least 101
4) `$ grep '^.\{101,150\}$'  f2`   # line length between 101 and 150

THE TAGGED REGULAR EXPRESSION (TRE)

- # the name like amit Sharma will be substituted as Sharma amit
- `$ sed  's/\([a-z]*\) *\([a-z]*\)/\2,  \1/'  teledir.txt | sort`
- `$ sed 's/\(m\)\(a\)/\2\1/' f1`
```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | amnager | sales | 12/8/67 | 5600
2476 | anil Agarwal | amnager | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76 |2000
```
- `$ sed  's/\(m\)\([a-z]*\)/\2\1/'  f1`
```
2233 | a.k. shukla | g.m | sales | 12/12/52 | 6000
1006 | chanchal singhvi | director | sales | 13/9/87 | 6700
1265 | s.n. dasgupta | anagerm | sales | 12/8/67 | 5600
2476 | anil Agarwal | anagerm | sales | 12/7/56 | 5000
2567 | anju agarwal | accountant | purchase | 12/7/76 |2000
```

awk utility:

- Awk is a scripting language used for manipulating data and generating reports.
- The awk command programming language requires no compiling, and allows the user to use **variables, numeric functions, string functions, and logical operators**.
- Awk is a utility that enables a programmer to write tiny but effective programs 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.
- Awk is mostly used for **pattern scanning and processing**. It searches one or more files to see if they contain lines that matches with the specified patterns and then performs the associated actions.
- Awk is abbreviated from the names of the developers – **Aho, Weinberger, and Kernighan.**

1. AWK Operations:
(a) Scans a file line by line
(b) Splits each input line into fields
(c) Compares input line/fields to pattern
(d) Performs action(s) on matched lines

2. Useful For:
(a) Transform data files
(b) Produce formatted reports

3. Programming Constructs:
(a) Format output lines
(b) Arithmetic and string operations
(c) Conditionals and loops

Typical Uses of AWK

- Text processing,
- Producing formatted text reports,
- Performing arithmetic operations,
- Performing string operations, and many more.
- Awk views a text file as records and fields.
- Like common programming language, Awk has variables, conditionals and loop.
- Awk has arithmetic and string operators.
- Awk can generate formatted reports

Example :

```
$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. Default behavior of Awk : By default Awk prints every line of data from the specified file.

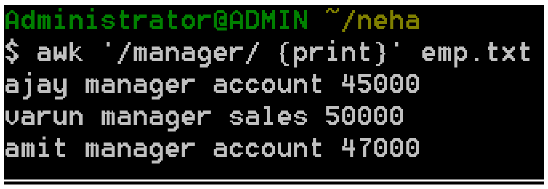
```
$ awk '{print}' employee.txt
```

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.

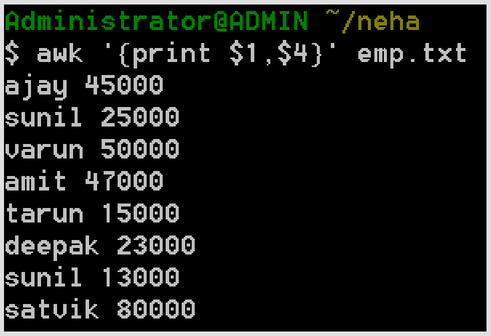
```
$ awk '/manager/ {print}' employee.txt
```



```
Administrator@ADMIN ~/neha
$ awk '/manager/ {print}' emp.txt
ajay manager account 45000
varun manager sales 50000
amit manager account 47000
```

3. Spliting a Line Into Fields : For each record i.e line, the awk command splits the record delimited by whitespace character by default and stores it in the $n variables.

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



```
Administrator@ADMIN ~/neha
$ awk '{print $1,$4}' emp.txt
ajay 45000
sunil 25000
varun 50000
amit 47000
tarun 15000
deepak 23000
sunil 13000
satvik 80000
```

Built In Variables In Awk

There are some system variables defined by awk.
All system variables are in **capital letters** and a user can change the value of system variables if desired.

The following are the system variables:
1.FS variable- Field Separator variable
2.OFS-The Output Field Separators.
3.RS-Record Separator variable.
4.ORS- A Output Record variable
5.NR -The Number of Records.

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

6.NF -Number of Field in a Record
Use of NF built-in variables (Display Last Field)

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

Output:

```
ajay 45000

sunil 25000

varun 50000

amit 47000

tarun 15000

deepak 23000

sunil 13000

satvik 80000
```

In the above example $1 represents Name and $NF represents Salary. We can get the Salary using $NF , where $NF represents last field.

7.FILENAME-Name of the Current Input File
8.FNR -Number of Records Relative to the Current Input File

Another use of NR built-in variables (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
```

More Examples

For the given text file:

```
$cat > geeksforgeeks.txt


A        B     C

Tarun   A12    1

Man     B6     2

Praveen M42    3
```

1) To print the first item along with the row number(NR) separated with ” – “ from each line in geeksforgeeks.txt:

```
$ awk '{print NR "- " $1 }' geeksforgeeks.txt

1 - Tarun

2 – Manav

3 - Praveen
```

2) To return the second row/item from geeksforgeeks.txt:

```
$ awk '{print $2}' geeksforgeeks.txt

A12

B6

M42
```

3) To print any non empty line if present

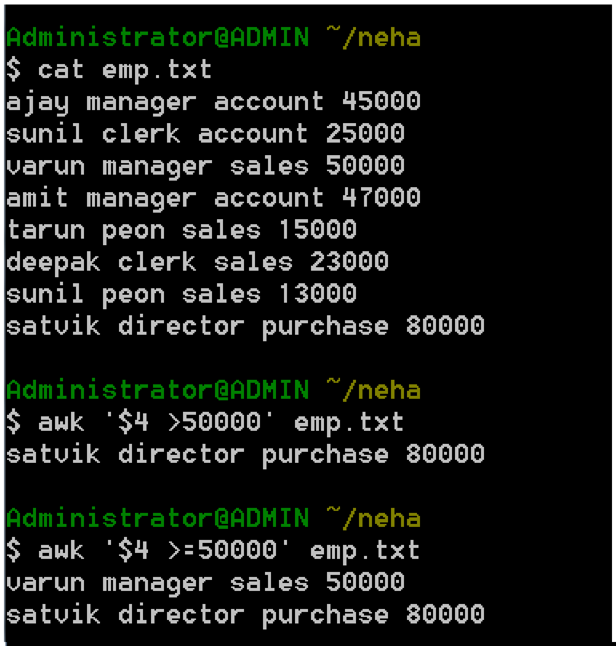
```
$ awk 'NF > 0' geeksforgeeks.txt

0
```

5) To count the lines in a file:

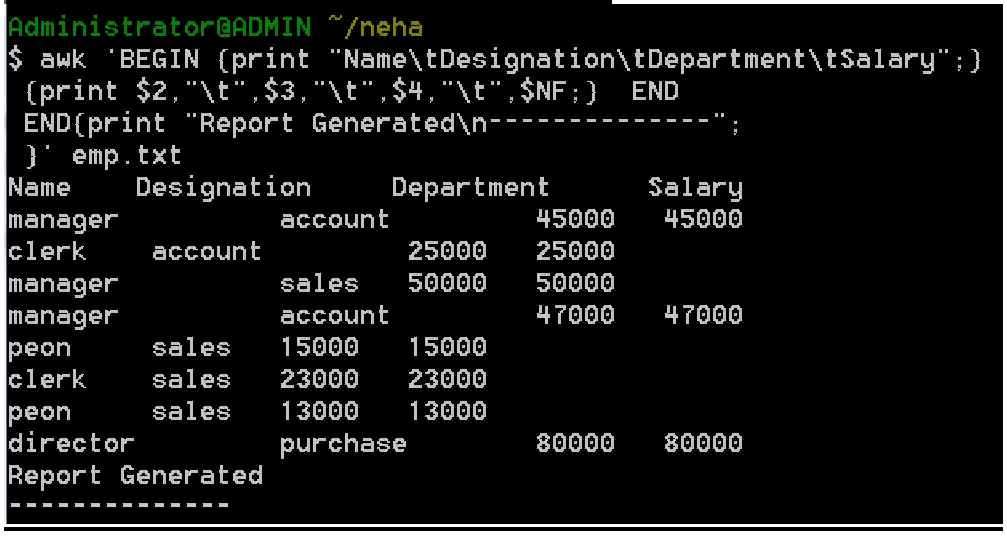
```
$ awk 'END { print NR }' geeksforgeeks.txt

3
```

6) Printing lines with more than 10 characters:

```
$ awk 'length($0) > 10' geeksforgeeks.txt

Tarun    A12    1

Praveen    M42    3
```



```
Administrator@ADMIN ~/neha
$ cat emp.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

Administrator@ADMIN ~/neha
$ awk '$4 >50000' emp.txt
satvik director purchase 80000

Administrator@ADMIN ~/neha
$ awk '$4 >=50000' emp.txt
varun manager sales 50000
satvik director purchase 80000
```



```
Administrator@ADMIN ~/neha
$ awk 'BEGIN {print "Name\tDesignation\tDepartment\tSalary";}
 {print $2,"\t",$3,"\t",$4,"\t",$NF;}  END
 END{print "Report Generated\n--------------";
 }' emp.txt
Name    Designation     Department      Salary
manager          account         45000   45000
clerk    account         25000   25000
manager          sales   50000   50000
manager          account         47000   47000
peon     sales   15000   15000
clerk    sales   23000   23000
peon     sales   13000   13000
director         purchase        80000   80000
Report Generated
--------------
```

Awk Example 6. Print the list of employees in Technology department

Now department name is available as a fourth field, so need to check if $4 matches with the string “Technology”, if yes print the line.

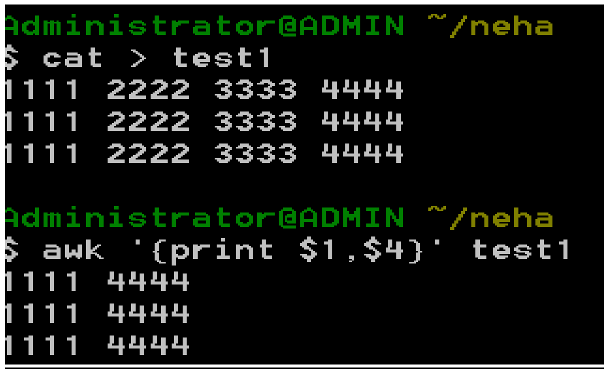
```
$ awk '$4 ~/Technology/' employee.txt
200  Jason   Developer  Technology  $5,500
300  Sanjay  Sysadmin   Technology  $7,000
500  Randy   DBA        Technology  $6,000
```

Operator ~ is for comparing with the regular expressions. If it matches the default action i.e print whole line will be performed.

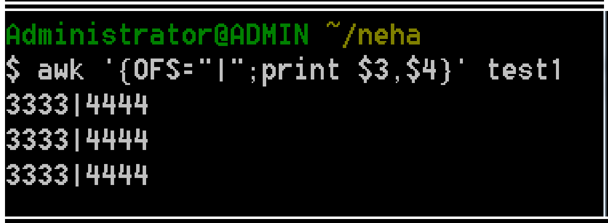
Awk Example 7. Print number of employees in Technology department

The below example, checks if the department is Technology, if it is yes, in the Action, just increment the count variable, which was initialized with zero in the BEGIN section.

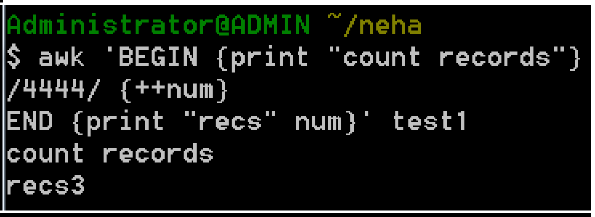
```
$ awk 'BEGIN { count=0;}
$4 ~ /Technology/ { count++; }
END { print "Number of employees in Technology Dept =",count;}' employee.txt
Number of employees in Tehcnology Dept = 3
```



```
Administrator@ADMIN ~/neha
$ cat > test1
1111 2222 3333 4444
1111 2222 3333 4444
1111 2222 3333 4444

Administrator@ADMIN ~/neha
$ awk '{print $1,$4}' test1
1111 4444
1111 4444
1111 4444
```



```
Administrator@ADMIN ~/neha
$ awk '{OFS="|";print $3,$4}' test1
3333|4444
3333|4444
3333|4444
```



```
Administrator@ADMIN ~/neha
$ awk 'BEGIN {print "count records"}
/4444/ {++num}
END {print "recs" num}' test1
count records
recs3
```

AWK workflow

To become an expert AWK programmer, you need to know its internals.

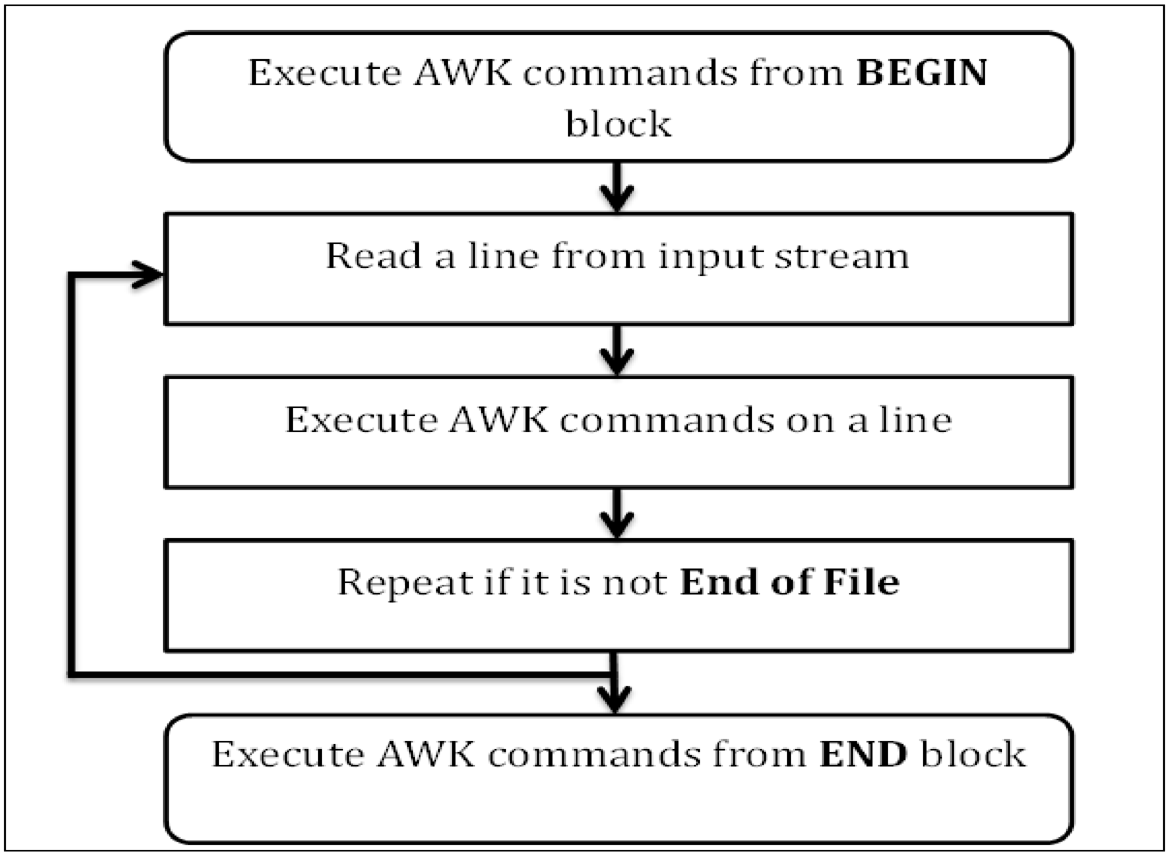
AWK follows a simple workflow – **Read, Execute, and Repeat**.

The following diagram depicts the workflow of AWK – figure

Read : AWK reads a line from the input stream (file, pipe, or stdin) and stores it in memory.

Execute: All AWK commands are applied sequentially on the input. By default AWK execute commands on every line. We can restrict this by providing patterns.

Repeat: This process repeats until the file reaches its end.


Execute AWK commands from **BEGIN** block
Read a line from input stream
Execute AWK commands on a line
Repeat if it is not **End of File**
Execute AWK commands from **END** block

Structure of awk:
Syntax: ***awk option 'instruction' filename(s)***

Instruction part of awk program has 3 sections.
[1] BEGIN [2] Processing section [3] END section

```
BEGIN { action }
 selection criteria {action}
END { action}'[filename(S)]
```

Output statement in awk

Output statement are used for display purpose.
The print and printf are generate output.

1. **print:**

The print statement does output with simple, standardized formatting.
You specify only the strings or numbers to be printed, in a list separated by commas.
They are output, separated by single spaces, followed by a newline. The statement looks like this: **print *item1*, *item2*, ...**

The entire list of items may optionally be enclosed in parentheses.
The parentheses are necessary if any of the item expressions uses the `>' relational operator; otherwise it could be confused with a redirection.
The items to be printed can be constant strings or numbers, fields of the current record (such as $1), variables, or any awk expressions.
Numeric values are converted to strings, and then printed.

Example:
Consider the following text file as the input file for all cases below.

```
$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. Default behavior of Awk : 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
```

In the above example, no pattern is given. So the actions are applicable to all the lines. Action print without any argument prints the whole line by default, so it prints all the lines of the file without failure.

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
```

In the above example, the awk command prints all the line which matches with the 'manager'.

3. Spliting a Line Into Fields : For each record i.e line, the awk command splits the record delimited by whitespace character by default and stores it in the $n variables. If the line has 4 words, it will be stored in $1, $2, $3 and $4 respectively. Also, $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
```

2. **printf:**

printf is similar to AWK print statement but the advantage is that it can print with formatting the output in a desired manner.
So before learning printf command I suggest you to learn about print command and then come to this printf statement.

Syntax:

```
awk '{printf "format", Arguments}' filename
```

For example you want to print decimal values of column 3 then the example will be.

```
awk '{printf "%d", $3}' example.txt
```

Printf can do two things which AWK print command can’t

```
1)Defining type of Data.
2)Padding between columns.
```

AWK PRINTF SUPPORTED DATA TYPES

The printf can be useful when specifying data type such as integer, decimal, octal etc. Below are the list of some data types which are available in AWK.

```
%i or d --Decimal%o --Octal
%x --hex
%c --ASCII number character
%s --String
%f --floating number
```

Note: Make sure that you pass exact data types when using corresponding formats as shown below. If you pass a string to a decimal formatting, it will print just zero instead of that string.

Lets start with some examples. for this post our test file contents are

```
Jones 21 78 84 77
Gondrol 23 56 58 45
RinRao 25 21 38 37
Edwin 25 87 97 95
Dayan 24 55 30 47
```

Example 1: Print first column values from db.txt file.

```
awk '{printf "%sn", $1}' db.txt
```

Output:

```
Jones
Gondrol
RinRao
Edwin
Dayan
```

Note: printf will not have default new line char, so you have to include tat when ever you execute printf command as shown above.

Example 2: Try printing a string with decimal format and see the difference.

```
awk '{printf "%dn", $1}' db.txt
```

Output:

```
0
0
0
0
0
```

-F option :

The default field separator can be changed by option -F.
In such cases, after -F option we can write a single character constant to indicate the field separator.
This character should be enclosed within marks if it is having special meaning like meta-character.
for rest data photos from mobile

AWK operators:

Like other programming languages, AWK also provides a large set of operators
There are two types of operators in Awk.

1. Unary Operator – Operator which accepts single operand is called unary operator.
2. Binary Operator – Operator which accepts more than one operand is called binary operator.

Few Operators are:

1. Arithmetic operators
2. Assignment Operator
3. Relational Operator
4. Logical Operator
5. Regular expression matching operator

1. **Arithmetic operators**
   The following operators are used for performing arithmetic calculations.

Operator	Description

+	**Addition**
	Ex:
	`[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a + b) = ", (a + b) }'`
	On executing this code, you get the following result –**Output**
	`(a + b) = 70`

–	**Subtraction**
	Example
	`[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a - b) = ", (a - b) }'`
	On executing this code, you get the following result –
	Output
	`(a - b) = 30`

*	Multiplication Example

	`[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a * b) = ", (a * b) }'`
	On executing this code, you get the following result –
	Output
	`(a * b) = 1000`

/	Division
	Example
	`[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a / b) = ", (a / b) }'`
	On executing this code, you get the following result –
	Output
	`(a / b) = 2.5`

%	Modulo Division
	Example
	`[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a % b) = ", (a % b) }'`
	On executing this code, you get the following result –
	Output
	`(a % b) = 10`

^ or **	exponentiation

2. **Assignment Operator**
   Awk has Assignment operator and Shortcut assignment operator as listed below.

Operator	**Description**

=	**Assignment**
	Example
	`[jerry]$ awk 'BEGIN { name = "Jerry"; print "My name is", name }'`
	On executing this code, you get the following result –
	Output
	`My name is Jerry`

+=	**Shortcut addition assignment**
	Example
	`[jerry]$ awk 'BEGIN { cnt = 10; cnt += 10; print "Counter =", cnt }'`
	On executing this code, you get the following result –
	Output
	`Counter = 20`
	In the above example, the first statement assigns value 10 to the variable **cnt**. In the next statement, the shorthand operator increments its value by 10.

-=	**Shortcut subtraction assignment**
	Example
	`[jerry]$ awk 'BEGIN { cnt = 100; cnt -= 10; print "Counter =", cnt }'`
	On executing this code, you get the following result –
	Output
	`Counter = 90`
	In the above example, the first statement assigns value 100 to the variable **cnt**. In the next statement, the shorthand operator decrements its value by 10.

*=	**Shortcut multiplication assignment** Example

	`[jerry]$ awk 'BEGIN { cnt = 10; cnt *= 10; print "Counter =", cnt }'`
	On executing this code, you get the following result –
	Output
	`Counter = 100`
	In the above example, the first statement assigns value 10 to the variable **cnt**. In the next statement, the shorthand operator multiplies its value by 10.

/=	**Shortcut division assignment**
	Example
	`[jerry]$ awk 'BEGIN { cnt = 100; cnt /= 5; print "Counter =", cnt }'`
	On executing this code, you get the following result –
	Output
	`Counter = 20`
	In the above example, the first statement assigns value 100 to the variable **cnt**. In the next statement, the shorthand operator divides it by 5.

%=	**Shortcut modulo division assignment**
	Example
	`[jerry]$ awk 'BEGIN { cnt = 100; cnt %= 8; print "Counter =", cnt }'`
	On executing this code, you get the following result –
	Output
	`Counter = 4`

^=	Shorthand Exponential
	Example
	`[jerry]$ awk 'BEGIN { cnt = 2; cnt ^= 4; print "Counter =", cnt }'`
	On executing this code, you get the following result –
	Output
	`Counter = 16`
	The above example raises the value of **cnt** by 4.

3. **Relational Operator**
   awk has the following list of conditional operators which can be used with control structures and looping statement which will be covered in the coming article.

Operator	Description

>	**Is greater than**
	It is represented by **>**. It returns true if the left-side operand is greater than the right-side operand, otherwise it returns false.
	Example
	`[jerry]$ awk 'BEGIN { a = 10; b = 20; if (b > a ) print "b > a" }'`
	On executing the above code, you get the following result –
	Output
	`b > a`

>=	**Is greater than or equal to** It is represented by **>=**. It returns true if the left-side operand is greater than or equal to the right-side operand; otherwise it returns false. b >= a
<	**Is less than** It is represented by **<**. It returns true if the left-side operand is less than the right-side

	operand; otherwise it returns false.
	Example
	`[jerry]$ awk 'BEGIN { a = 10; b = 20; if (a < b) print "a  < b" }'`
	On executing this code, you get the following result –
	Output
	`a < b`

<=	**Is less than or equal to**
	It is represented by **<=**. It returns true if the left-side operand is less than or equal to the right-side operand; otherwise it returns false.
	Example
	`[jerry]$ awk 'BEGIN { a = 10; b = 10; if (a <= b) print "a <= b" }'`
	On executing this code, you get the following result –
	Output
	`a <= b`

<=	**Is less than or equal to**

==	**Is equal to**
	It is represented by ==. It returns true if both operands are equal, otherwise it returns false. The following example demonstrates this –
	Example
	`awk 'BEGIN { a = 10; b = 10; if (a == b) print "a == b" }'`
	On executing this code, you get the following result –
	Output
	`a == b`

!=	**Is not equal to**
	It is represented by **!=**. It returns true if both operands are unequal, otherwise it returns false.
	Example
	`[jerry]$ awk 'BEGIN { a = 10; b = 20; if (a != b) print "a != b" }'`
	On executing this code, you get the following result –
	Output
	`a != b`

4. Logical Operator

Operator	Description

&&	**Both the conditional expression should be true**
	It is represented by **&&**. Its syntax is as follows –
	Syntax
	`expr1 && expr2`
	It evaluates to true if both expr1 and expr2 evaluate to true; otherwise it returns false. expr2 is evaluated if and only if expr1 evaluates to true. For instance, the following example checks whether the given single digit number is in octal format or not.
	Example
	`[jerry]$ awk 'BEGIN {`
	`   num = 5; if (num >= 0 && num <= 7) printf "%d is in octal format\n", num`
	`}'`
	On executing this code, you get the following result –
	Output

	5 is in octal format

||	**Any one of the conditional expression should be true**
	It is represented by **||**. The syntax of Logical OR is −
	Syntax
	`expr1 || expr2`
	It evaluates to true if either expr1 or expr2 evaluates to true; otherwise it returns false. expr2 is evaluated if and only if expr1 evaluates to false. The following example demonstrates this −
	Example
	`[jerry]$ awk 'BEGIN {`
	`   ch = "\n"; if (ch == " " || ch == "\t" || ch == "\n")`
	`   print "Current character is whitespace."`
	`}'`
	On executing this code, you get the following result −
	Output
	`Current character is whitespace`

!	Logical NOT
	It is represented by **exclamation mark (!)**. The following example demonstrates this −
	Example
	`! expr1`
	It returns the logical compliment of expr1. If expr1 evaluates to true, it returns 0; otherwise it returns 1. For instance, the following example checks whether a string is empty or not.
	Example
	`[jerry]$ awk 'BEGIN { name = ""; if (! length(name)) print "name is empty string." }'`
	On executing this code, you get the following result −
	Output
	`name is empty string.`

5. **Regular expression matching operator**
   Awk Regular Expression Operator

Operator	Description
~	Match operator It is represented as ~. It looks for a field that contains the match string. For instance, the following example prints the lines that contain the pattern 9. Example [jerry]$ awk '$0 ~ 9' marks.txt On executing this code, you get the following result − Output 2) Rahul Maths 90 5) Hari History 89
!~	No Match operator It is represented as !~. It looks for a field that does not contain the match string. For instance, the following example prints the lines that do not contain the pattern 9. Example [jerry]$ awk '$0 !~ 9' marks.txt On executing this code, you get the following result −

	Output 1) Amit Physics 80 3) Shyam Biology 87 4) Kedar English 85

AWK: expression, variables and constants

AWK provides several built-in variables. They play an important role while writing AWK scripts.

Standard AWK variables

ARGC :It implies the number of arguments provided at the command line.

Example

```
[jerry]$ awk 'BEGIN {print "Arguments =", ARGC}' One Two Three Four
```

On executing this code, you get the following result –**Output**

```
Arguments = 5
```

But why AWK shows 5 when you passed only 4 arguments? Just check the following example to clear your doubt.

ARGV: It is an array that stores the command-line arguments. The array's valid index ranges from 0 to ARGC-1.

Example

```
[jerry]$ awk 'BEGIN {
  for (i = 0; i < ARGC - 1; ++i) {
    printf "ARGV[%d] = %s\n", i, ARGV[i]
  }
}' one two three four
```

On executing this code, you get the following result –

Output

```
ARGV[0] = awk
ARGV[1] = one
ARGV[2] = two
ARGV[3] = three
```

FILENAME: It represents the current file name.

Example

```
[jerry]$ awk 'END {print FILENAME}' marks.txt
```

On executing this code, you get the following result –

Output

```
marks.txt
```

Please note that FILENAME is undefined in the BEGIN block.

FS :It represents the (input) field separator and its default value is space. You can also change this by using -F **command line option.**

Example

```
[jerry]$ awk 'BEGIN {print "FS = " FS}' | cat -vte
```

On executing this code, you get the following result –

Output

```
FS =  $
```

NF: It represents the number of fields in the current record. For instance, the following example prints only those lines that contain more than two fields.

Example

```
[jerry]$ echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NF > 2'
```

On executing this code, you get the following result –

Output

```
One Two Three
One Two Three Four
```

NR:It represents the number of the current record. For instance, the following example prints the record if the current record number is less than three.

Example

```
[jerry]$ echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NR < 3'
```

On executing this code, you get the following result –

Output

```
One Two
One Two Three
```

FNR: It is similar to NR, but relative to the current file. It is useful when AWK is operating on multiple files. Value of FNR resets with new file.

OFS: It represents the output field separator and its default value is space.

Example

```
[jerry]$ awk 'BEGIN {print "OFS = " OFS}' | cat -vte
```

On executing this code, you get the following result –

Output

```
OFS =  $
```

ORS: It represents the output record separator and its default value is newline.

Example

```
[jerry]$ awk 'BEGIN {print "ORS = " ORS}' | cat -vte
```

On executing the above code, you get the following result –

Output

```
ORS = $
$
```

RS: It represents (input) record separator and its default value is newline.

Example

```
[jerry]$ awk 'BEGIN {print "RS = " RS}' | cat -vte
```

On executing this code, you get the following result –

Output

```
RS = $
$
```