**Filters**

Filter is a programm in unix . It takes its input from another program, performs some operation on that input, and writes the result to the standard output. Thus the common use of filters is to modify  or  restructure output.

Some common filters in UNIX are:

* uniq – Removes identical adjacent lines
* head – displays first n lines of a file .
* tail – displays last n lines of a file .
* sort – sorts files by line (lexically or numerically)
* cut – select portions of a line.
* wc – word count (line count, character count)
* tr – translate
* grep, egrep – search files using regular expressions

**head**

This command list the beginning of a file to standard output. The default is 10 lines, but a different number can be specified. The command has a number of  options.

**Syntax:**

head [OPTION] [FILE]

**Options:**

**-c** Prints the first N bytes of file; with leading -, prints all but the last N bytes of the file.

**-n** Prints first N lines; with leading - print all but the last N lines of each file.

**Example:**To display the first 10 lines of the file myfile.txt. $head myfile.txt

To display the first 100 lines of the file myfile.txt.

$head -n100 myfile.txt

To print the first 5 bytes from the file

$ head -c5 myfile.txt

**tail**

List the (tail) end of a file to stdout. The default is 10 lines, but this can be changed with the -n option. Commonly used to keep track of changes to a system log-file, using the -f option, which outputs lines appended to the file.

**Syntax:**

tail [OPTION]... [FILE]...

**Example:**

To display the last 10 lines of the file myfile.txt.

$tail myfile

To display the last 100 lines of the file myfile.txt.

$ tail -100 myfile.txt

$tail –n 100 myfile.txt

**more**

more command allows to view text files or other output in a scrollable manner. When can command is used to view a very long file, all the output scrolls off the top of your screen and only the last page can be viewed. more command solves this problem by allowing the output of cat command one screenful of data at a time.

**Syntax:**

more [option] filename

**Options:**

-num This option specifies an integer which is the screen size (in lines).

-d more will prompt the user with the message "[Press space to continue, 'q' to quit.]" and will display "[Press 'h' for instructions.]" instead of ringing the bell when an illegal key is pressed.

-l more usually treats ^L (form feed) as a special character, and will pause after any line that contains a form feed. The -l option will prevent this behavior.

-p Do not scroll. Instead, clear the whole screen and then display the text.

**tr**  
tr command automatically translates or substitute characters.   
**Syntax:**

tr [OPTION] set1 [set2]

Translate, squeeze, and/or delete characters from standard input, writing to standard output.   
**Options:**

1. : complements the set of characters in string.
2. : deletes the characters in set1
3. : replaces repeated characters listed in the set1 with single occurrence
4. : truncates set1

**Example:**To replace any occurrence of a by x, b by y and c by z in a given string

$echo “about to call “|tr [abc] [xyz]

Output : xyout to zxll

**Example:**To replace non matching characters

$ echo "Hello"|tr -c e a

Output : aeaaaa

In the above example , except the character “e” other characters are replaced by a

**Example:**Squeez , we can squeeze more than one occurrence of continuous characters with single occurrence.

$echo “about to call “|tr – s ‘ ‘

Output : about to call

Above example squeezes two or more blank spaces into one.

**sort**

sort command reorders the lines of a file in ascending or descending order.

The default order is ascending .

**Syntax:**

sort -t field\_delemeter [OPTION] file1 [file 2]

**Options:**

**-k n sort on the nth field of the line**

**-t char use char as the field delimiter**

**-n sort numerically**

**-r reverse order sort**

**-u removes repeated lines**

**-m list merge sorted files in list**

**Examples:**

Below examples will help you to understand sort used with different options:

**Example 1:**

Consider a file named “**list”**, which has below data

1, Justin Timberlake, Title 545, Price $7.30

2, Lady Gaga, Title 118, Price $7.30

3, Johnny Cash, Title 482, Price $6.50

4, Elvis Presley, Title 335, Price $7.30

5, John Lennon, Title 271, Price $7.90

To sort on the 2ndfield of file named “list” we have to use the below command:

$sort –t’,’ –k 2 list

Note: File list is comma separated file.

**Output:**

4, Elvis Presley, Title 335, Price $7.30

5, John Lennon, Title 271, Price $7.90

3, Johnny Cash, Title 482, Price $6.50

1, Justin Timberlake, Title 545, Price $7.30

2, Lady Gaga, Title 118, Price $7.30

**Example 2: Numerically sorting:**

To numerically sort data , option to be used is –n

Suppose list is the name of the file having following data:

19

20

5

49

00

If we sort it as below:

$sort list

Output is :

19

20

200

49

5

To get the expected output , the command will be

$sort –n list

Output:

5

19

20

49

200

**Sort can sort multiple files also.**

$sort file1 file2 file3 …

**Example 3:**Numerically sort in reverse order

**$sort –**nr list

**Output :**

200

49

20

19

5

**Example 4: Sort the file list removing the repeated lines.**

**Syntax:**

$sort –u filename

File list has following content:

Unix

Unix

Linux

Linux

Solaris

Axis

Axis

**$sort –u list**

**Output:**

Unix

Linux

Solaris

Axis

**uniq**

uniq command is used to suppress the duplicate lines from a file. It discards all the successive identical lines except one from the input and writes the output. 

**Syntax:**

uniq [option] filename

**Options:**

-u lists only the lines that are unique

lists only the lines that are duplicates

-c counts the frequency of occurrences

**Suppress duplicate lines:**

The default behavior of the uniq command is to suppress the duplicate line. Note that, you have to pass sorted input to the uniq, as it compares only successive lines.

If the lines in the file are not in sorted order, then use the sort command and then pipe the output to the uniq command.

**Count of lines:**

The -c option is used to find how many times each line occurs in the file. It prefixes each line with the count.

**Display only duplicate lines:**

You can print only the lines that occur more than once in a file using the -d option. The -D option prints all the duplicate lines.

**Skip first N fields in comparison:**

the -f option is used to skip the first N columns in comparison. Here the fields are delimited by the space character.

**​​​​cut**

This command is used for text processing. You can use this command to extract portion of text from a file by selecting columns.

**Syntax:**

cut –option filename

**Select Column of Characters :**

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

**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 –c 1-3 test.txt

**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.

Following example extracts from 3rd character to end of each line from test.txt file. 

$cut –c3- test.txt

To extract 8 characters from the beginning from the file test.txt,

$cut –c-8 test.txt  
**Select a Specific Field from a File :**  
Instead of selecting x number of characters you can combine option -f and –d to extract a whole field.

The option -f specifies which field you want to extract,

The option -d specifies what 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.

$ cut -d':' -f1 etc/passwd

**paste**

This is the command  for merging together different files into a single, multi-column file. In combination with cut, useful for creating system log files.   
**Syntax:**

paste file1 file2

**join**  
This utility allows merging two files in a meaningful fashion, which essentially creates a simple version of a relational database. 

The command join operates on exactly two files, but pastes together only those lines with a common tagged field (usually a numerical label), and writes the result to standard output.

The files to be joined should be sorted according to the tagged field for the matchups to work properly.

**Example:**

The content of two files file1 and file2 are as below,  
$cat file1

100 Shoes

200 Laces

300 Socks

$cat file2

100 $40.0

200 $1.00

300 $2.00

The following command will join these two files.  
 $ join 1.data 2.data

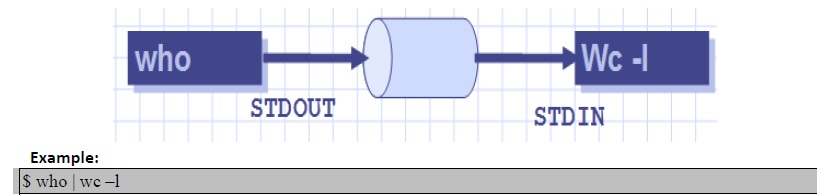
100 Shoes $40.00

200 Laces $1.00

300 Socks $2.00

**Pipe**

In unix , you can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a pipe. In shell the symbol '|’ is used to represent pipe. 



**Purpose of Pipes** :

Using pipe you can construct powerful unix command lines by combining basic unix commands. UNIX commands are powerful; however by using pipe you can combine them together, to accomplish complex tasks with ease.

Through the standard output of one command (the command to the left of the pipe) gets sent as standard input to another command (the command to the right of the pipe). Pipe functions in a similar manner like the output redirection in UNIX (using > symbol to redirect the standard output of a command to a file. However, the pipe is different because it is used to pass the output of a command to another command, not a file.

**Example:**

$ cat apple.txt | wc

3 4 21  
In this example, the contents of the file apple.txt are sent through pipe to wc (word count) command. The wc command then does its job and counts the lines, words, and characters in the file.

You can combine many commands with pipes on a single command line. Here's an example where the characters, words, and lines of the file apple.txt is sent to wc and then the output of wc mailed to nobody@december.com with the subject line "The count."

$ cat apple.txt | wc | mail -s "The count" nobody@december.com

**awk**

awk is a scripting language which is used for processing or analyzing text files.

awk is used for grouping of data based on either a column or field, or on a set of columns.

It derives its name from the first letter of the last name of its three authors namely Alfred V. Aho, Peter J.Weinberger and Brian W. Kernighan.

 awk can be used for reporting data in a useful manner. It searches one or more files to see if they contain lines that match specified patterns and then perform associated actions. awk is an advanced filter.  
***Simple awk Filtering***

**Syntax of awk:**

~$ awk 'pattern {action}' input-file  
Let’s take a input file with the following data  
~$cat awk\_file  
Name,Marks,Max\_Marks  
Peter,200,1000  
Sam,500,1000  
Greg,1000  
Abharam,800,1000  
Henry,600,1000  
Peter,400,1000

**Example: Default behavior of awk**  
Print all the lines from a file.

By default, awk prints all lines of a file, so to print every line of above  created file , use below command:  
~$ awk '{print}' awk\_file

Name,Marks,Max\_Marks  
Peter,200,1000  
Sam,500,1000  
Greg,1000  
Abharam,800,1000  
Henry,600,1000  
Peter,400,1000

**Example 2:** Print only specific field  
Print 2nd & 3rd fileds  
~$ awk -F”,” {print $2,$3;}' awk\_file

**Example: Pattern Matching**  
Print the lines which matches the pattern (lines which contains the word “Henry" or "Peter”)  
~$ awk '/Henry|Peter/' awk\_file  
Peter,200,1000  
Henry,600,1000  
Peter,400,1000

**Initialization and Final Action**

BEGIN and END blocks are helpfull in displaying information before and after executing actual awk script.

BEGIN block is evaluated before awk starts processing the actual awk script; it’s an excellent place to initialize the FS (field separator) variable, print a heading, or initialize other global variables.

**BEGIN block Usages:**

* Declaring variables.
* Initialization variables for doing increment/decrements operations in main AWK code.
* Printing Headings/info before actual AWK code output.

END block is evaluated after all the lines in the awk script have been processed.  
Typically, the END block is used to perform final calculations or print summaries that should appear at the end of the output stream.

**END block Usages:**

* Printing final results, after doing operations in main AWK block.
* Printing Completion/info after actual AWK code output.

awk tool is mainly used for reporting some data in useful manner. Without these BEGIN and END  
blocks the output will be meaningless.  
Consider db.txt which contains below data:  
Jones 2143 78 84 77  
Gondrol 2321 56 58 45  
RinRao 2122234 38 37

**awk BEGIN block**

This is a block of code which is executed before executing actual awk script.  
**BEGIN block Syntax**  
               awk ‘BEGIN{awk initializing code}{actual AWK code}’ filename.txt  
**Example:** Print a meaning full info before actual AWK output.  
~$ awk ‘BEGIN{print “########################\nThis is the output of  
filtered  
data\n########################”}{print $0}’ db.txt

**Output:**  
##########################  
This is the output of filtered data  
##########################  
Jones 2143 78 84 77  
Gondrol 2321 56 58 45  
RinRao 2122234 38 37  
Edwin 253734 87 97 95  
Dayan 24155 30 47

**awk END block**

This is the block which is executed after executing all the awk code.   
**Example:**

Print some meaning full info after processing awk code.  
~$ awk ‘{print $0} END {print “#########################\n Completed  
printing  
filtered data\n########################”}’ db.txt

**Output:**

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  
#########################  
Completed printing filtered data  
#########################

**Combining BEGIN and END block**

**Example:**

~$ awk ‘BEGIN{print “##########################\n This is the output of filtered  
data\n##########################”}{print $0}END{print  
“########################\n Completed printing filtered  
data\n########################”}’ db.txt

**Output:**

#########################  
This is the output of filtered data  
#########################  
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  
########################  
Completed printing filtered data

**awk inbuilt variables**

awk is supplied with good number of built-in variables which comes in handy when working with data files. We will see usages of awk built-in variables with one or two examples .  These variable are used to format the output of an awk command.

**List of built-in variables:**

FS field separator character (default blank & tab)  
OFS output field separator string (default blank)  
RS input record separator character (default newline)  
ORS output record separator string (default newline)  
NF number of fields in input record  
NR number of input record  
FNR output number of lines  
FILENAME name of current input file

Consider below db.txt as sample file.  
~$ cat db.txt  
John,29,MS,IBM,M,Married  
Barbi,45,MD,JHH,F,Single  
Mitch,33,BS,BofA,M,Single  
Tim,39,Phd,DELL,M,Married  
Lisa,22,BS,SmartDrive,F,Married

In order to make it simple we can divide above inbuilt variables in to groups on basis of their operations.  
**Group1**: FS(input field separator), OFS(Output Field Separator)  
**Group2**: RS(Row separator) and ORS(Output Record Separator)  
**Group3:** NR, NF and FNR  
**Group4:** FILENAME variable

**FS (Input Field Separator)**

This variable is useful in storing the input field separator. By default AWK can understand only spaces, tabs as input and output separators. But if your file contains some other character as separator other than these mention one’s, awk cannot understand them.

For example UNIX password file which contain ‘:’ as a separator. So in order to mention the input filed separator we use this inbuilt variable. We will see what issue we face if we don’t mention the field separator for our db.txt.

**Example: without using FS**

Print first column data from db.txt file.  
**~$ awk ‘{print $1}’ db.txt**

**Output:**  
John,29,MS,IBM,M,Married  
Barbi,45,MD,JHH,F,Single  
Mitch,33,BS,BofA,M,Single  
Tim,39,Phd,DELL,M,Married  
Lisa,22,BS,SmartDrive,F,Married

**OFS (Output Field Separator)**  
This variable is useful for defining the output field separator for the expected output data.

**Example:**

Display only 1st and 4th column and with $ as field  separator for the output .  
**~$ awk ‘BEGIN{FS=”,”;OFS=” $ “}{print $1,$4}’ db.txt**

**Output:**  
John $ IBM  
Barbi $ JHH  
Mitch $ BofA  
Tim $ DELL  
Lisa $ SmartDrive  
Note: Space is give before and after $ in OFS variable to show better output.

**RS (Row separator)**  
Row Separator is helpful in defining separator between rows in a file. By default awk takes row separator as new line. We can change this by using RS built-in variable.

**Example:**  
Convert a sentence to a word per line. We can use RS variable for doing it.  
~$ echo “This is how it works” | awk ‘BEGIN{RS=” ”}{print $0}’

**Output:**  
This  
is  
how  
it  
Works

**ORS (Output Record Separator)**  
This variable is useful for defining the record separator for the awk command output. By default ORS is set to new line.

**Example:**  
Print all the company names in single line which are in 4th column.  
**~$ awk -F’,’ ‘BEGIN{ORS=” “}{print $4}’ db.txt**

**Output:**  
IBM JHH BofA DELL SmartDrive

**NF**  
This variable keeps information about total fields in a given row. The final  
value of a row can be represented with $NF.

**Example:** Consider abc.txt which contains below data:  
Jones 2143 78 84 77  
Gondrol 2321 56 58 45  
RinRao 2122234 38 37  
Edwin 253734 87 97 95  
Dayan 24155 30 47  
Print number of fields in each row in abc.txt.  
**~$ awk ‘{print NF}’ abc.txt**

**Output:**  
5  
5  
4  
5  
4

**NR**

This variable keeps the value of present line number. This will come handy when you want to print line numbers in a file.

**Example:**  
Print line number for each line in a given file.  
**~$ awk ‘{print NR, $0}’ abc.txt**

**Output:**  
1 Jones 2143 78 84 77  
2 Gondrol 2321 56 58 45  
3 RinRao 2122234 38 37  
4 Edwin 253734 87 97 95  
5 Dayan 24155 30 47  
This can be treated as cat command -n option for displaying line number for a file

**FNR**  
This variable keeps count of number of lines present in a given file/data. This will come handy when  
you want to print no of line present in a given file. This command is equivalent to wc -l command.

**Example:**  
Print total number of lines in a given file.  
~$ awk ‘END{print FNR}’ abc.txt

**Output:**  
5  
**FILENAME**  
This variable contain file awk command is processing.

**Example:**  
Print filename for each line in a given file.  
**~$ awk ‘{print FILENAME, NR, $0}’ abc.txt**

**Output:**  
abc.txt 1 Jones 2143 78 84 77  
abc.txt 2 Gondrol 2321 56 58 45  
abc.txt 3 RinRao 2122234 38 37  
abc.txt 4 Edwin 253734 87 97 95  
abc.txt 5 Dayan 24155 30 47

**awk Built in Function**

A function is a self-contained computation that accepts a number of arguments as input and returns some value. awk has a number of built-in functions in two groups: arithmetic and string functions.

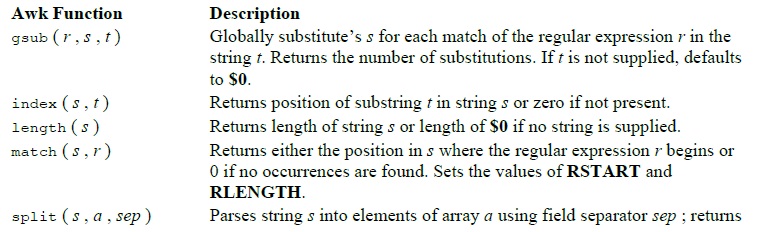
**Arithmetic Functions**  
Nine of the built-in functions can be classified as arithmetic functions. Most of them take a numeric argument and return a numeric value. Below table summarizes these arithmetic functions with some Examples.

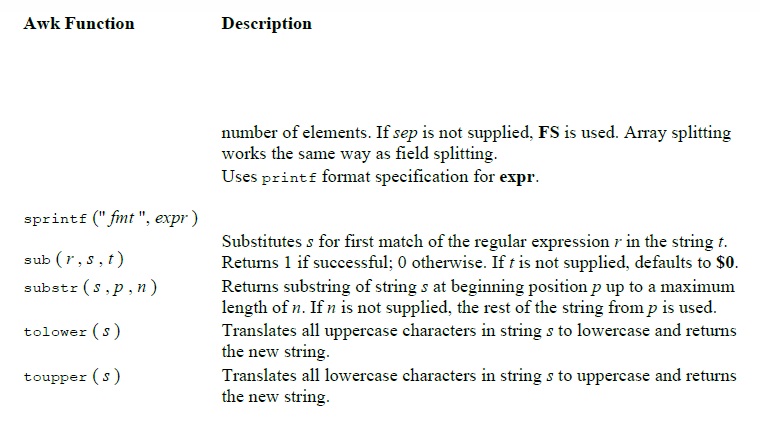
**awk Function Description**  
cos ( x ) Returns cosine of x (x is in radians).  
exp ( x ) Returns e to the power x.  
index (s1,s2) Position of string s2 in s1; returns 0 if not present  
int ( x ) Returns truncated value of x.  
log ( x ) Returns natural logarithm (base- e) of x.  
sin ( x ) Returns sine of x (x is in radians)

sqrt ( x ) Returns square root of x.  
atan2 ( y , x ) Returns arctangent of y / x in the range - to .  
rand () Returns pseudo-random number r, where 0 <= r < 1.  
sqrt(expr) Returns the square root of the expression or value given

**Examples:**  
~$ awk 'BEGIN{  
print sqrt(16);  
print sqrt(0);  
print sqrt(-12);  
}'  
**Output:**  
4  
0  
nan  
Here nan stands for not a valid number.

**String Functions**  
The built-in string functions are much more significant and interesting than the numeric functions. Because awk is essentially designed as a string-processing language, a lot of its power derives from these functions. Below table lists the string functions found in awk. awk's Built-In String Functions





**Filters and Regular Expression**

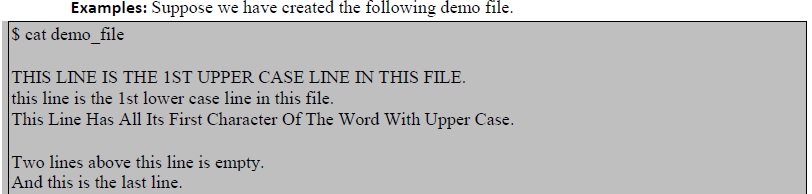
**grep**  
grep command allows you to search one file or multiple files for lines that contain a pattern.

Full form of grep is global regular expression print.

It is a powerful file pattern searcher in **Linux**

grep's exit status is 0 if matches were found, 1 if no matches were found, and 2 if errors occurred.  
grep search the target file(s) for occurrences of pattern, where pattern may be literal text or a Regular Expression.

Syntax:  
*grep pattern [file...]*

**

**Search for the given string in a single file**  
The basic usage of grep command is to search for a specific string in the specified file as shown below.

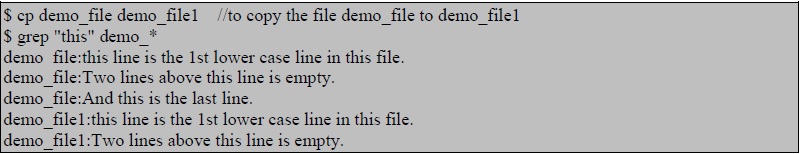


**Checking given string in multiple files**  
We can use  grep command for  searching for a given string in multiple files.

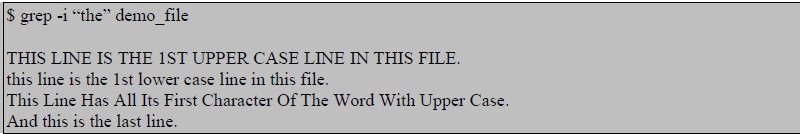
For example, let us copy the demo\_file to demo\_file1and use the grep on both the files to search the pattern this.

The output will  include the file name in front of the line that matched the specific pattern as shown below.

When the Linux shell sees the meta character, it does the expansion and gives all the files as input to grep.



**Case insensitive search**  
We can use grep to search for the given string/pattern case insensitively. So it matches all the words such as “the”, “THE” and “The” case insensitively as shown below.



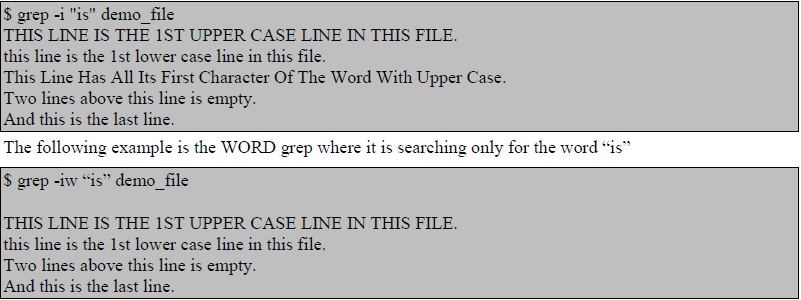
**Match regular expression in files**  
This is a very powerful feature of grep . In the following example, it searches for all the pattern that starts with “lines” and ends with “empty” with anything in-between. i.e To search “lines[anything in-between]empty” in the demo\_file.

https://g91.tcsion.com/per/g91/pub/2030/LX/ckeditor_assets/pictures/2030/39/match_reg_exp_original.jpg

A regular expression may be followed by one of several repetition operators:

1. ? The preceding item is optional and matched at most once.
2. The preceding item will be matched zero or more times.
3. + The preceding item will be matched one or more times.
4. {n} The preceding item is matched exactly n times.
5. {n,} The preceding item is matched n or more times.
6. {,m} The preceding item is matched at most m times.
7. {n,m} The preceding item is matched at least n times, but not more than m times.

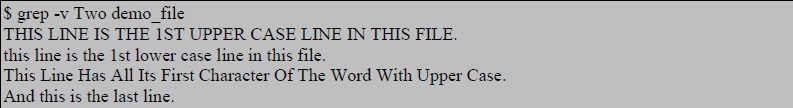
**Checking for full words**  
To search for a word, and to avoid it to match the substrings -w option is used. The following example is the regular grep where it is searching for “is”. When you search for “is”, without any option it will show out “is”, “his”, “this” and everything which has the substring “is”.



**Searching in all files recursively**  
When you want to search in all the files under the current directory and its sub directory ‘–r’ option is the one which you need to use. The following example will look for the string “ramesh” in all the files in the current directory and all its subdirectory.

$ grep -r "ramesh" \*

**Invert match**  
If you want to display the lines which does not matches the given string/pattern, use the option -v as shown below. This example will display all the lines that did not match the word “Two”.



Displaying the lines which does not matches the entire given pattern.

Syntax:  
grep -v -e pattern -e pattern

For example, the file file1 has the following content

Apple

Banana  
Cauliflower  
Grapes  
Orange



Counting the number of matches  
Count the number of lines matched in the given pattern/string, then use the option -c.  
Syntax:  
grep -c pattern filename

Displaying only the file names which matches the given pattern  
The -l option is used to display only the file names which matched the given pattern. When you give multiple files to the grep as input, it displays the names of file which contains the text that matches the pattern, will be very handy when you try to find some notes in your whole directory structure.



**Showing line number while displaying the output**  
To show the line number of file with the line matched, -n option is used.

**Syntax:**  
grep -n pattern filename

**Example:**  
grep -n "this" demo\_file   
2: this line is the 1st lower case line in this file.   
6: Two lines above this line is empty.

**sed**

sed is a stream editor used to perform basic text transformations on an input stream (a file, or input from a pipeline).

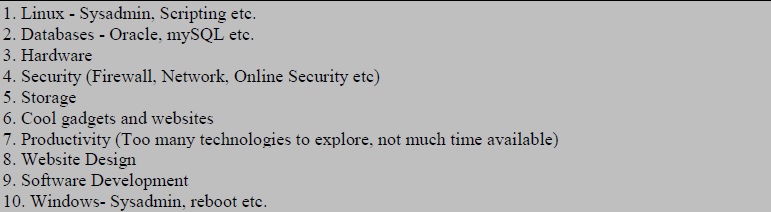
**Working methodology**  
sed works by making only one pass over the input(s) s called as one execution cycle. Cycle continues till end of file/input is reached.

* Read entire line from stdin/file.
* Removes any trailing newline.
* Places the line, in its pattern buffer.
* Modify the pattern buffer according to the supplied commands.
* Print the pattern buffer to stdout.

**Printing Operation in sed**  
sed allows you to print only specific lines based on the line number or pattern matches. “p” is the command for printing the data from the pattern buffer. To suppress automatic printing of patternspace -n option is used with sed. sed -n option will not print anything, unless an explicit request to print is found. 

Syntax:   
sed -n 'ADDRESS'p filename   
sed -n '/pattern/p' filename

Examples:  
Let us assume the demo\_file has the following content



To prints third line of input file

$sed -n '3p' demo\_file  
3. Hardware

To print every nth line starting from the line m  
$sed -n 'm~np' filename

To print only the last line

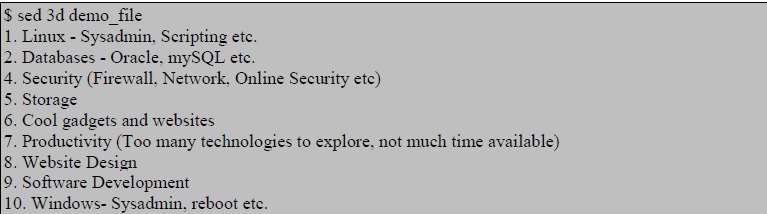
https://g91.tcsion.com/per/g91/pub/2030/LX/ckeditor_assets/pictures/2030/48/print_last_line_original.jpg

**To print the lines containing the given pattern:**  
Syntax:  
sed -n /PATTERN/p filename

**Deletion operation in sed**  
In sed the d command is used to delete the pattern space buffer and immediately starts the next cycle.  
Syntax:  
sed nd filename  
'nd’ deletes the nth line and prints the other lines.  
sed 'ADDRESS'd filename  
sed /PATTERN/d filename

The process is  
• It reads the first line and places in its pattern buffer  
• checks whether supplied command is true for this line , if true, deletes pattern space buffer and starts next cycle and reads the next line.  
• If supplied command is not true, it prints the content of the pattern space buffer.

**To delete the 3rd line and print other lines from the file demo\_file**

****

**Substitution operation in sed**  
In sed the s command is used to substitute the pattern. The `s’ command attempts to match the pat-tern space against the supplied expression/ pattern; if the match is successful, then that portion of the pattern space which was matched is replaced with the replacement given.

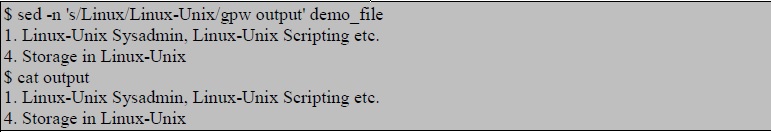
Syntax:   
$sed 'ADDRESSs/REGEXP/REPLACEMENT/FLAGS' filename   
$sed 'PATTERNs/REGEXP/REPLACEMENT/FLAGS' filename

1. s is substitute command
2. / is a delimiter
3. REGEXP is regular expression to match
4. REPLACEMENT is a value to replace

FLAGS can be any of the following:

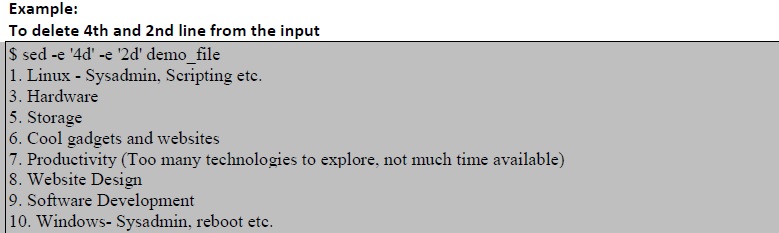
1. g Replace all the instance of REGEXP with REPLACEMENT
2. n Could be any number,replace nth instance of the REGEXP with REPLACEMENT.
3. p If substitution was made, then prints the new pattern space.
4. i match REGEXP in a case-insensitive manner.
5. w file If substitution was made, write out the result to the given file.
6. We can use different delimiters ( one of @ % ; : ) instead of /

To Write Changes to a File and Print the Changes

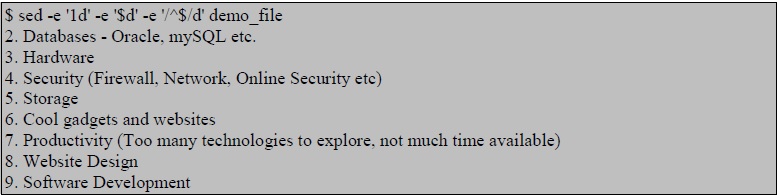


**To combine multiple sed commands we have to use option -e**

Syntax:  
$sed  -e 'command' e 'command' filename



**To Delete the first,last and all the blank lines from input**



**FILTERS USING REGULAR EXPRESSION**

A regular expression is a set of characters that specify a pattern. Regular expressions are used when you want to search for specific lines of text containing a particular pattern. Most of the UNIX utilities operate on ASCII files a line at a time. Regular expressions search for patterns on a single line, and not for patterns that start on one line and end on another.

**The Structure of a Regular Expression**

There are three important parts to a regular expression.  
• Anchors : These are used to specify the position of the pattern in relation to a line of text.  
• Character Sets : The set of characters that match one or more characters in a single position.  
• Modifiers: They specify how many times the previous character set is repeated.  
A simple example that demonstrates all three parts is the regular expression is : "^#\*"  
Here ,  
• The up arrow , “^”, is an anchor that indicates the beginning of the line.  
• The character "#" is a simple character set that matches the single  
character "#".  
• The asterisk “\*” is a modifier. In a regular expression it specifies that the  
previous character set can appear any number of times, including zero.

There are also two types of regular expressions:  
• the "Basic" regular expression,(BRE)  
• the "extended" regular expression.(ERE)

A few utilities like awk and egrep use the extended expression. Most use the "regular" regular expression. From now on, if I talk about a "regular expression," it describes a feature in both types.

The Anchor Characters: ^ and $ Anchores are used when we want to search for a pattern that is at one end or the other, of a line. The character "^" is the starting anchor, and the character "$" is the end anchor.Following list provides a summary:

**Pattern   Matches**  
^A   "A" at the beginning of a line  
A$   "A" at the end of a line  
A^   "A^" anywhere on a line  
$A   "$A" anywhere on a line  
^^   "^" at the beginning of a line  
$$   "$" at the end of a line

**The Character Set**  
The character set also called “character class” in a regular expression , is used to tell the regex engine to match only one out of several characters

* A character set matches only a single character. In case of the above example, gr[ae]y does not match graay, graey or any such thing.
* The order of the characters inside a character set does not matter. The results are identical.
* Some characters have a special meaning in regular expressions. If we want to search for such a character, we have to escape it with a backslash.

**Exception in the character class**  
If we want to search for all the characters except those in the square bracket,  
then the ^ (Caret) symbol needs to be used as the first character after open  
square bracket. The expression "^[^aeiou]" is to searc for a line which does  
not start with the vowel letter.

**Regular Expression** **Matches**  
[]                              The characters "[]"  
[0]                            The character "0"  
[0-9]                         Any number  
[^0-9]                       Any character other than a number  
[-0-9]                        Any number or a "-"  
[0-9-]                        Any number or a "-"  
[^-0-9]                      Any character except a number or a "-"  
[]0-9]                        Any number or a "]"  
[0-9]]                        Any number followed by a "]"  
[0-9-z]                      Any number, or any character between "9" and "z".  
[0-9\-a\]]                  Any number, or a "-", a "a", or a "]"

**Match any character**  
The character "." is one one of thespecial meta-characters. By itself it will match any character, except the end-of-line character. Thus the pattern that will match a line with a single characters is **^.$**

**Repeating character sets**  
The third part of a regular expression is the modifier. It is used to specify how may times you expect to see the previous character set. The repetition modifier \* find no or one, one or more, and zero or more  
repeats, respectively.

Examples:  
Expression         Matches  
Go\*gle               Gogle,Google,Gooogle, and so on.  
"[0-9]\*"              zero or more numbers.

**Matching a specific number of sets with \{ and \}**  
We cannot specify a maximum number of sets with the "\*" modifier. There is a special pattern we can use to specify the minimum and maximum number of repeats, by putting those two numbers between "\{" and "\}".

▪ A modifier can specify amounts such as none, one, or more;

For example , A user name is a string beginning with a letter followed by at least two, but not more than seven letters or numbers followed by the end of the string. Then the regular expression is

^[A-z][A-z0-9]{2,7}

▪ A repetition modifier must be combined with other patterns; the modifier has no meaning by itself.

For example , modifiers like "\*" and "\{1,5\}" only act as modifiers if they follow a character set. If they were at the beginning of a pattern, they would not be a modifier.

**grep with Regular expression**

Search for 'vivek' in /etc/passswd  
**grep vivek /etc/passwd**  
Search vivek in any case (i.e. case insensitive search)  
**grep -i -w vivek /etc/passwd**  
Search vivek or raj in any case  
**grep -E -i -w 'vivek|raj' /etc/passwd**

**Line and word anchors**

Search lines starting with the vivek only  
**grep ^vivek /etc/passwd**  
To display only lines starting with the word vivek only i.e. do not display vivekgite, vivekg  
**grep -w ^vivek /etc/passwd**  
To Find lines ending with word foo  
**grep 'foo$' filename**

**Character classes**

To match Vivek or vivek.  
**grep '[vV]ivek' filename**  
OR  
**grep '[vV][iI][Vv][Ee][kK]' filename**

To match digits (i.e match vivek1 or Vivek2 etc)  
**grep -w '[vV]ivek[0-9]' filename**

**Wildcards**  
To match all 3 character word starting with "b" and ending in "t".  
grep '\' filename  
Where,  
•\< Match the empty string at the beginning of word  
•\> Match the empty string at the end of word.  
Print all lines with exactly two characters  
**grep '^..$' filename**  
Display any lines starting with a dot and digit  
**grep '^\.[0-9]' filename**

**Escaping the dot**

To find an IP address 192.168.1.254  
**grep '192\.168\.1\.254' /etc/hosts**

**Search a Pattern Which Has a Leading – Symbol**

Searches for all lines matching '--test--' using -e option . Without -e, grep would  
attempt to parse '--test--' as a list of options  
**grep -e '--test--' filename**

**Test Sequence**

To Match a character "v" two times  
**egrep "v{2}" filename**

To match both "col" and "cool"  
**egrep 'co{1,2}l' filename**

**grep OR Operator**  
Suppose the file “employee” has the following data:

  
To find the records of those who are either from Tech or Sales dept.  
We can use the following syntaxes :

1) Syntax : grep 'word1\|word2' filename  
grep 'Tech\|Sales' employee

2) Syntax : grep -E 'pattern1|pattern2' fileName  
grep -E 'Tech|Sales' employee

**grep AND Operator**  
There is no AND operator in grep. But, we can simulate AND using

• grep -E option.  
Syntax : grep -E 'word1.\*word2 ' filename  
grep -E 'word1.\*word2|'word2.\*word1' filename

• multiple grep command separated by pipe  
Syntax : grep 'word1' filename | grep 'word2'