RollNo||Name||Grade

1||harsh||A

2||sam||C

3||Kim||D

4||Tom||B

5||kelly||A

awk 'BEGIN {print "start"} {print $0} END{print "stop"}' dat\_one\_awk

start

RollNo||Name||Grade

1||harsh||A

2||sam||C

3||Kim||D

4||Tom||B

5||kelly||A

stop

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

RollNo Name Grade

1 harsh A

2 sam C

3 Kim D

4 Tom B

5 kelly A

awk 'begin {print "Name\tGrade"} {print$2,$3} end {print "--Done--"}' dat\_two\_awk

Name Grade

harsh A

sam C

Kim D

Tom B

kelly A

awk 'begin {print "Name\tGrade"} {print$2 $3} end {print "--Done--"}' dat\_two\_awk

NameGrade

harshA

samC

KimD

TomB

kellyA

awk 'begin {print "Name\tGrade"} {print$2 $3} END {print "--Done--"}' dat\_two\_awk

NameGrade

harshA

samC

KimD

TomB

kellyA

--Done—

awk 'BEGIN {print "Start"} {print $2,"\t", $3} END {print "Done"}' dat\_two\_awk

Start

Name Grade

harsh A

sam C

Kim D

Tom B

kelly A

Done

awk -f awk\_ex1.awk dat2

Start

Name Grade

harsh A

sam C

Kim D

Tom B

kelly A

---Done---

cat awk\_ex1.awk

#!/bin/awk -f

BEGIN {print "Start"}

{print $2 , "\t", $3 }

END {print "---Done---"}

cat dat2

RollNo Name Grade

1 harsh A

2 sam C

3 Kim D

4 Tom B

5 kelly A

Uniq :

-c, --count prefix lines by the number of occurrences

-d, --repeated only print duplicate lines, one for each group

-D print all duplicate lines

-i, --ignore-case ignore differences in case when comparing

-s, --skip-chars=N avoid comparing the first N characters

-u, --unique only print unique lines

Sort:

-b, --ignore-leading-blanks ignore leading blanks

-d, --dictionary-order consider only blanks and alphanumeric characters

-f, --ignore-case fold lower case to upper case characters

-i, --ignore-nonprinting consider only printable characters

-M, --month-sort compare (unknown) < 'JAN' < ... < 'DEC'

-n, --numeric-sort compare according to string numerical value

-r, --reverse reverse the result of comparisons

--sort=WORD sort according to WORD:

general-numeric -g, human-numeric -h, month -M,

numeric -n, random -R, version -V

-V, --version-sort natural sort of (version) numbers within text

Other options:

-c, --check, --check=diagnose-first check for sorted input; do not sort

-k, --key=KEYDEF sort via a key; KEYDEF gives location and type

-m, --merge merge already sorted files; do not sort

-o, --output=FILE write result to FILE instead of standard output

-s, --stable stabilize sort by disabling last-resort comparison

-t, --field-separator=SEP use SEP instead of non-blank to blank transition

-u, --unique with -c, check for strict ordering;

without -c, output only the first of an equal run

WC:

The options below may be used to select which counts are printed, always in

the following order: newline, word, character, byte, maximum line length.

-c, --bytes print the byte counts

-m, --chars print the character counts

-l, --lines print the newline counts

-L, --max-line-length print the maximum display width

-w, --words print the word counts

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To access bash variable in awk.

#!/bin/sh  
column="$1"  
awk '{print $'"$column"'}'

${*variable*:-*defaultvalue*}

#!/bin/sh  
column="${1:-1}"  
awk '{print $'"$column"'}'

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

AWK Table 1

Binary Operators

Operator Type Meaning

+ Arithmetic Addition

- Arithmetic Subtraction

\* Arithmetic Multiplication

/ Arithmetic Division

% Arithmetic Modulo

<space> String Concatenation

Expression Result

7+3 10

7-3 4

7\*3 21

7/3 2.33333

7%3 1

7 3 73

|  |  |
| --- | --- |
| **AWK Table 2 Assignment Operators** | |
| **Operator** | **Meaning** |
| += | Add result to variable |
| -= | Subtract result from variable |
| \*= | Multiply variable by result |
| /= | Divide variable by result |
| %= | Apply modulo to variable |

|  |  |
| --- | --- |
| **AWK Table 3 Relational Operators** | |
| **Operator** | **Meaning** |
| == | Is equal |
| != | Is not equal to |
| > | Is greater than |
| >= | Is greater than or equal to |
| < | Is less than |
| <= | Is less than or equal to |

|  |  |
| --- | --- |
| **AWK Table 4 Regular Expression Operators** | |
| **Operator** | **Meaning** |
| ~ | Matches |
| !~ | Doesn't match |

And 🡪 && ; Or 🡪 ||; Not 🡪 !

There are only a few commands in AWK. The list and syntax follows:

if ( *conditional* ) *statement* [ else *statement* ]  
while ( *conditional* ) *statement*  
for ( *expression* ; *conditional* ; *expression* ) *statement*  
for ( *variable* in *array* ) *statement*  
break  
continue  
{ [ *statement* ] ...}  
*variable*=*expression*  
print [ *expression-list* ] [ > *expression* ]  
printf *format* [ , *expression-list* ] [ > *expression* ]  
next  
exit

* Use BEGIN + EXIT if you do not want to read any data.

#!/bin/awk -f

BEGIN {

# Print the squares from 1 to 10 the first way

i=1;

while (i <= 10) {

printf "The square of ", i, " is ", i\*i;

i = i+1;

}

# do it again, using more concise code

for (i=1; i <= 10; i++) {

printf "The square of ", i, " is ", i\*i;

}

# now end

exit;

}

The following asks for a number, and then squares it:

#!/bin/awk -f

BEGIN {

print "type a number";

}

{

print "The square of ", $1, " is ", $1\*$1;

print "type another number";

}

END {

print "Done"

}

Here is a filter that you should find useful. It counts lines, totals up the numbers in the first column, and calculates the average. Pipe "wc -c \*" into it, and it will count files, and tell you the average number of words per file, as well as the total words and the number of files.

#!/bin/awk -f

BEGIN {

# How many lines

lines=0;

total=0;

}

{

# this code is executed once for each line

# increase the number of files

lines++;

# increase the total size, which is field #1

total+=$1;

}

END {

# end, now output the total

print lines " lines read";

print "total is ", total;

if (lines > 0 ) {

print "average is ", total/lines;

} else {

print "average is 0";

}

}

$2="";  
print;

deletes the second field.

FS – The input field separator

#!/bin/awk -f

BEGIN {

FS=":";

}

{

if ( $2 == "" ) {

print $1 ": no password!";

}

}

#!/bin/awk -f

{

if ( $0 ~ /:/ ) {

FS=":";

$0=$0

} else {

FS=" ";

$0=$0

}

#print the third field, whatever format

print $3

}

**OFS – The output field separator variable**

#!/bin/awk -f

BEGIN {

FS=":";

OFS=":";

}

{

$2="";

print

}

**NF – The number of fields variable**

#!/bin/awk -f

# parse the output of "ls -l"

# print owner and filename

# remember - Berkeley ls -l has 8 fields, System V has 9

{

if (NF == 8) {

print $3, $8;

} else if (NF == 9) {

print $3, $9;

}

}

**NR – The number of records variable**

#!/bin/awk -f

{ if (NR > 100) {

print NR, $0;

}

**RS – The record separator variable**

#!/bin/awk -f

BEGIN {

# change the record separator from newline to nothing

RS=""

# change the field separator from whitespace to newline

FS="\n"

}

{

# print the second and third line of the file

print $2, $3;

}

You can use it to break words up, one word per line, using this:

#!/bin/awk -f

BEGIN {

RS=" ";

}

{

print ;

}

**ORS – The output record separator variable**

#!/bin/awk -f

# this filter adds a carriage return to all lines

# before the newline character

BEGIN {

ORS="\r\n"

}

{ print }

**FILENAME – The current filename variable**

#!/bin/awk -f

# reports which file is being read

BEGIN {

f="";

}

{ if (f != FILENAME) {

print "reading", FILENAME;

f=FILENAME;

}

print; # == cat filename

}

The *printf* is very similar to the C function with the same name. C programmers should have no problem using *printf* function.

*Printf* has one of these syntactical forms:

printf ( format);  
printf ( format, arguments...);  
printf ( format) >expression;  
printf ( format, arguments...) > expression;

|  |  |
| --- | --- |
| **AWK Table 6 Format Specifiers** | |
| **Specifier** | **Meaning** |
| %c | ASCII Character |
| %d | Decimal integer |
| %e | Floating Point number (engineering format) |
| %f | Floating Point number (fixed point format) |
| %g | The shorter of e or f, with trailing zeros removed |
| %o | Octal |
| %s | String |
| %x | Hexadecimal |
| %% | Literal % |

Again, I'll cover the differences quickly. Table 3 illustrates the differences. The first line states "printf(%c\n",100.0)"" prints a "d".

|  |  |  |
| --- | --- | --- |
| **AWK Table 7 Example of format conversions** | | |
| **Format** | **Value** | **Results** |
| %c | 100.0 | d |
| %c | "100.0" | 1 (NAWK?) |
| %c | 42 | " |
| %d | 100.0 | 100 |
| %e | 100.0 | 1.000000e+02 |
| %f | 100.0 | 100.000000 |
| %g | 100.0 | 100 |
| %o | 100.0 | 144 |
| %s | 100.0 | 100.0 |
| %s | "13f" | 13f |
| %d | "13f" | 0 (AWK) |
| %d | "13f" | 13 (NAWK) |
| %x | 100.0 | 64 |

|  |  |
| --- | --- |
| **AWK Table 10 String Functions** | |
| **Name** | **Variant** |
| index(string,search) | AWK, NAWK, GAWK |
| length(string) | AWK, NAWK, GAWK |
| split(string,array,separator) | AWK, NAWK, GAWK |
| substr(string,position) | AWK, NAWK, GAWK |
| substr(string,position,max) | AWK, NAWK, GAWK |
| sub(regex,replacement) | NAWK, GAWK |
| sub(regex,replacement,string) | NAWK, GAWK |
| gsub(regex,replacement) | NAWK, GAWK |
| gsub(regex,replacement,string) | NAWK, GAWK |
| match(string,regex) | NAWK, GAWK |
| tolower(string) | GAWK |
| toupper(string) | GAWK |
| asort(string,[d]) | GAWK |
| asorti(string,[d]) | GAWK |
| gensub(r,s,h [,t]) | GAWK |
| strtonum(string) | GAWK |

if (length($0) > 1) {  
. . .  
}

Unix handson 4

sort -k3,3 -rn -t" " | head -n1 | awk '{print $2}'

export HISTCONTROL=erasedups

export HISTCONTROL=ignoredups

export HISTCONTROL=”erasedups:ignorespace”

export HISTIGNORE=”history\*”

export HISTTIMEFORMAT=”%h %d %H:%M:%S> ”

export HISTSIZE=1000 (=0 🡪 turnoff, = -ve num 🡪 infinite)

echo

passwd 🡪 to change user password

uname 🡪 to get system information

who 🡪 information about all users who are currently logged in

find 🡪 command finds the file named Abc in current(.) directory and all its sub-directories

find . -name “Abc” or “Abc\*”

cmd current directory by name filename wild card

find Dir1/Dir2 -name “Abc” or “Abc\*”

find . -size +10M (more than10MB)

find . -atime -3 ( last three days)

find . -perm 700 (u rwx, g ---, o --- )

**Perform Action on the search result :** Using exec option any action (command)  
can be executed on the search result.  
>> To remove all the files having extension .swp from the current directory the  
command will be :

|  |
| --- |
|  |
|  |
|  |
|  |

$ find . -name “\*.swp” -exec rm {} \;

**bc**  🡪 is the command used for basic mathematical calculations.

$ echo "2+5" | bc

Tty 🡪 Prints the file name of the terminal connected in standard input.

VI editor:

**Commands to insert or add text:**

Following are the commands to be used to insert or add text.

|  |  |
| --- | --- |
| key | Purpose of the command |
| u | Undo the last command’s action |
| i | insert text before the cursor,until is hit |
| I | insert text at beginning of current line, until  hit |
| a | append text after cursor, until  hit |
| A | append text to end of current line, until  hit |
| o | open and put text in a new line below current line, until  hit |
| O | open and put text in a new line above current line, until  hit |

**Commands to modify texts:**

|  |  |
| --- | --- |
| **key** | **Purpose of the command** |
| **r** | replace single character under cursor (no  needed) |
| **R** | replace characters, starting with current cursor position, until  hit |
| **cw** | change the current word with new text,  starting with the character under cursor, until  hit |
| **cNw** | change N words beginning with character under cursor, until  hit;    e.g., c5w changes 5 words |
| **C** | change (replace) the characters in the current line, until  hit |
| **cc** | change (replace) the entire current line, stopping when  is hit |
| **Ncc or cNc** | change*(*replace) the next N lines, starting with the current line, stopping when  is hit |

**Commands to Delete texts:**

|  |  |
| --- | --- |
| **key** | **Purpose of the command** |
| **x** | delete single character under cursor |
| **Nx** | delete N characters, starting with character under cursor |
| **cw** | delete the single word beginning with character under cursor |
| **dw** | change N words beginning with character under cursor, until  hit;    e.g., c5w changes 5 words |
| **dNw** | delete N words beginning with character under cursor;    e.g., d5w deletes 5 words |
| **D** | delete the remainder of the line, starting with current cursor position |
| **dd** | delete entire current line |
| **Ndd or dNd** | delete N lines, beginning with the current line;    e.g., 5dd deletes 5 lines |

**Commands to copy and paste text:**

|  |  |
| --- | --- |
| **yy** | copy (yank, cut) the current line into the buffer |
| **Nyy** | copy (yank, cut) the next N lines, including the current line, into the buffer |
| **p** | put (paste) the line(s) in the buffer into the text after the current line |

**Commands for searching text**

|  |  |
| --- | --- |
| **/string** | search forward for occurrence of string in text |
| **?string** | search backward for occurrence of string in text |

* touch
* cat
* cat > (single redirection)
* cat >>(double redirection)
* cp
* mv
* cmp 🡪  This command is used to compare two files, and if they differ, tells the first byte and line number where they differ. If input files are same , returns nothing.
* Comm 🡪 This command is used to compare two sorted files

Syntax:

comm [options] file1 file2

* One set of options allows selection of ‘columns’ to suppress.
  + -1: suppress lines unique to file1 (column 1)
  + -2: suppress lines unique to file2 (column 2)
  + -3: suppress lines common to file1 and file2 (column3)

**Example:** To show only lines common between file1 and file2, command is

$ comm -12 file1 file2

**# cat >test2**

Creates test2 file

**. Diff**-  This command is used to compare two files line by line. The output indicates how the lines in each file are different, and the steps that will be involved in changing file1 to file2.

The change commands are in the format [range][acd][range].  The range on the left may be a line number or a comma-separated range of line numbers referring to file1, and the range on the right similarly refers to file2.  The character in the middle indicates the action i.e. add, change or delete.

* ‘LaR’ – Add lines in range ‘R’ from file2 after line ‘L’ in file1.
* ‘FcT’ – Change lines in range ‘F’ of file1 to lines in range ‘T’ of file2.
* ‘RdL’ – Delete lines in range ‘R’ from file1 that would have appeared at line ‘L’ in file2

**Syntax**: diff [options] file1 file2

**Some common filters in UNIX are**:

* uniq – Removes identical adjacent lines
* head – displays first n lines of a file . -n -4 everything except last four lines
* 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

tr

tr command automatically translates or substitute characters.

**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.

Uniq

**Options:**

-u lists only the lines that are unique

lists only the lines that are duplicates

-c counts the frequency of occurrences

Cut:

The following example displays 2nd character from each line of a file test.txt.

$cut –c2 test.txt

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.

**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

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

**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

grep “string” filename

-i ignore case

-w full word

-r recursive search

-v inverted search

-c matched lines count

-o output only word

-n num in the matched file

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.

To prints third line of input file

$sed -n '3p' demo\_file

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

**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

To prints third line of input file

$sed -n '3p' demo\_file

**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

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

**sed 3d filename**

**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 /

sed -n ‘s/Linux/Linux-Unix/gpw output’ demo\_file

sed – stream editor

-n – silence the output

‘ – beginning

s – substitution option

/ - delimiter

Linux – present string

/ - delimiter

Linux-Unix – to be replaced by this

/ - delimiter

g – change all instances

p – print the modified

w – write to a file

output – name of the file data written to

‘ – ending

Demo\_file – file name

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

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

Example:

To delete the 4th and 2nd line from the input

$ sed -e ‘4d’ -e ‘2d’ demo\_file

To delete the first, last and all the blank lines from input

$ sed -e ‘1d’ -e ‘$d’ -e ‘/^$/d’ demo\_file

**FILTERS USING 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.

**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 expression "^[^aeiou]" is to search for a line which does not start with the vowel letter.

First ^ - to denote at the beginning

Second ^ - denotes other than aeiou.

**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 \}**

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}

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**

To Match a character "v" two times  
**egrep "v{2}" filename**  
To match both "col" and "cool"  
**egrep 'co{1,2}l' filename**

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'

Here is simple example to add two numbers:

**Example:**

#!/bin/sh

val=`expr 2 + 2`

echo "Total value : $val"

**Output:**

$ Total value : 4

There are following points to note down:

* There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
* Complete expression should be enclosed between ``, called inverted commas to execute expr command correctly.

**Arithmetic Operators**

Assume variable a holds 10 and variable b holds 20 then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition | `expr $a + $b ` will give 30 |
| - | Substraction | `expr $a - $b ` will give -10 |
| \* | Multiplication | `expr $a \\* $b` will give 200 |
| / | Division | `expr $b / $a` will give 2 |
| % | Modulus | `expr $a % $b` will give 0 |
| != | Not equal | [ $a != $b ] will give true |
| = | assignment | a=$b will assign value of b to a. |
| == | Equality | [ $a == $b ] will return false. |

It is very important to note here that all the conditional expressions would be put inside square braces **with one spaces**around them, for example [ $a == $b ] is correct where as [$a==$b] is incorrect.

**Relational Operators**

Below are relational operators which are specific to numeric values. These operators would not work for string values unless their value is numeric.

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

Assume variable a holds 10 and variable b holds 20 then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-eq** | Check if the values of 2 operands are equal or not, if yes then condition becomes true. | [ $a -eq $b ] is false |
| **-ne** | Check if the values of 2 operands are equal or not, if values are not equal then condition becomes true. | [ $a -eq $b ] is true |
| **-gt** | Check if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | [ $a -gt $b ] is false |
| **-lt** | Check if the value of left operand is less than the value of right operand, if yes then condition becomes true. | [ $a -lt $b ] is true |
| **-ge** | Check if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | [ $a -ge $b ] is false |
| **-le** | Check if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | [ $a -le $b ] is true |

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [ $a <= $b ] is correct where as [$a <= $b] is incorrect.

**Boolean Operators**

Assume variable a holds 10 and variable b holds 20 then

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

**String Operators**

These are string operators. Assume variable a holds "abc" and variable b holds "efg" then:

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

**File Test Operators:**

Assume a variable file holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-b file** | Returns true, if file is a block special file | [ -b $file ] is false. |
| **-c file** | Returns true, if file is a character special file | [ -b $file ] is false. |
| **-d file** | Returns true, Check if file is a directory | [ -d $file ] is not true. |
| **-f file** | Returns true, Check if file is an ordinary file or special file | [ -f $file ] is true. |
| **-r file** | Returns true, Checks if file is readable | [ -r $file ] is true. |
| **-w file** | Returns true, Check if file is writable | [ -w $file ] is true. |
| **-x file** | Returns true, Check if file is execute | [ -x $file ] is true. |
| **-s file** | Returns true, Check if file has size greater than 0 | [ -s $file ] is true. |
| -**e file** | Returns true, Check if file exists | [ -e $file ] is true. |

**Wild Card Characters**

Symbol used to replace or represent one or more characters. Wildcards or wild characters are either asterisk (\*), which represent one or more characters or question mark (?), which represent a single character.

|  |  |  |  |
| --- | --- | --- | --- |
| **Wild card /Shorthand** | **Meaning** |  | **Examples** |
| **\*** | Matches any string or group of characters. | **$ ls \*** | will show all files |
| **$ ls a\*** | will show all files whose first name is starting with letter 'a' |
| **$ ls \*.c** | will show all files having extension .c |
| **$ ls ut\*.c** | Will show all files having extension .c but file name must begin with 'ut'. |
| **?** | Matches any single character. | **$ ls ?** | will show all files whose names are 1 character long |
| **$ ls fo?** | will show all files whose names are 3 character long and file name begin with fo |
| **[...]** | Matches any one of the enclosed characters | **$ ls [abc]\*** | Will show all files beginning with letters a,b,c |

**Note: [..-..]** A pair of characters separated by a minus sign denotes a range.

***Example*:**

**$ ls /bin/[a-c]\***

Will show all File name beginning with letter a,b or c.

***Output:***

/bin/arch /bin/awk /bin/bsh /bin/chmod /bin/cp /bin/as /bin/basename /bin/cat

**Shell Quoting Mechanism**

**The Metacharacters**

Unix Shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

**Example:**

**?** Matches with a single character while listing files in a directory and an **\*** would match more than one characters.

Here is a list of most of the shell special characters (also called metacharacters):

\* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab

A character may be quoted (i.e., made to stand for itself) by preceding it with a \.

**Example:**

#!/bin/sh echo Hello; Word

This would produce following result.

Hello ./test.sh: line 2: Word: command not found shell returned 127

Now let us try using a quoted character:

#!/bin/sh echo Hello\; Word

This would produce following result:

Hello; Word

The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell:

#!/bin/sh echo "I have \$1200"

This would produce following result:

I have $1200

|  |  |
| --- | --- |
| **Quote** | **Description** |
| Single quote | All special characters between these quotes lose their special meaning. |
| Double quote | Most special characters between these quotes lose their special meaning with these exceptions:  $  `  \$  \'  \"  \\ |
| Backslash | Any character immediately following the backslash loses its special meaning. |
| Back Quote | Anything in between back quotes would be treated as a command and would be executed. |

**The Single Quotes**

Consider an echo command that contains many special shell characters:

echo <-$1500.\*\*>; (update?) [y|n]

Putting a backslash in front of each special character is tedious and makes the line difficult to read:

echo \<-\$1500.\\*\\*\>\; \(update\?\) \[y\|n\]

There is an easy way to quote a large group of characters. Put a single quote ( ') at the beginning and at the end of the string:

echo '<-$1500.\*\*>; (update?) [y|n]'

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you would precede that using a backslash (\) as follows:

echo 'It\'s Shell Programming

**The Double Quotes:**

Try to execute the following shell script. This shell script makes use of single quote:

VAR=ZARA

echo '$VAR owes <-$1500.\*\*>; [ as of (`date +%m/%d`) ]'

This would produce following result:

$VAR owes <-$1500.\*\*>; [ as of (`date +%m/%d`) ]

So this is not what you wanted to display. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make invert commas work as expected then you would need to put your commands in double quotes as follows:

VAR=ZARA

echo "$VAR owes <-\$1500.\*\*>; [ as of (`date +%m/%d`) ]"

Now this would produce following result:

ZARA owes <-$1500.\*\*>; [ as of (07/02) ]

Double quotes take away the special meaning of all characters except the following:

* $ for parameter substitution.
* Backquotes for command substitution.
* \$ to enable literal dollar signs.
* \` to enable literal backquotes.
* \" to enable embedded double quotes.
* \\ to enable embedded backslashes.
* All other \ characters are literal (not special).

**test command in if condition**

We can use test command as condition of if condition as used in the below script.

$cat if\_test.sh

#!/bin/sh

echo “Do you want to quit (Y/y)? ”

read ans

if  test $ans ==’y’ –o $ans==’Y’

then

      exit

else

      echo “ to exit enter N or n”

fi

There is no maximum number of patterns, but the minimum is one.

**Example:**

#!/bin/sh

COURSE=”DB”

case “$COURSE” in

 “Java”) echo “Java is a programming language”

   ;;

 “Perl”)echo “Perl is scripting language”

  ;;

 “DB”)echo “Oracle is a DB”

  ;;

esac

**Output:**

Oracle is a DB

**String Handling**

* String handling with test command:

|  |  |  |  |
| --- | --- | --- | --- |
| test str | Returns true |  | if str is not null |
| test –n str | Returns true |  | if length of str is greater than zero |
| test –z str | Returns true |  | if length of str is equal to zero |

* String handling with expr command

The expr is quite handy for finding the length of a string and extracting a sub-string:

**Length of the string:**

|  |  |
| --- | --- |
|  | $ str=”abcdefghijk” ;  $ n=`expr "$str" : ‘.\*’` ;  $ echo $n  11 |

expr gave how many times any character (.\*) occurs. This feature is very useful in validating data entry.

**Extracting a sub-string:**

|  |  |
| --- | --- |
|  | $ str=”abcdefghijk” ;  $ expr “$str” : ‘……\(..\)’  gh |

Note that there are 6 dots preceding the sequence \(..\). This advanced regular expression signifies that the first six characters of the string are to be ignored and extraction should start from the 7th character. Two dots inside \(..\) suggests that this extraction is limited to two characters only (backslashes override the usual interpretation of ‘()’).

**Extracting string from 3rd character to end of the string:**

|  |  |
| --- | --- |
|  | $ str="abcdefghijk"  $ expr "$str" : '..\(.\*\)'  cdefghijk |

**Location of first occurrence of a character “d” inside string:**

|  |  |
| --- | --- |
|  | $ str=”abcdefghijk” ;  $ expr "$str" : '[^d]\*d‘  4 |

**Location of last occurrence if a character inside string:**

Below will give the last occurrence of character 'a' from string str.

|  |  |
| --- | --- |
|  | $str=”abc def abc”  $expr "$str" : '[^u]\*a' |

#!/bin/sh

echo “Program name is $0”

echo “First argument is $1”

echo “Number of arguments passed =$#”

echo “The arguments are $@”

**The PS1 Variable**

The system prompt may be changed by setting the value of this variable to the desired prompt:

$ PS1=“Hello>”

Hello>   #can be changed only at the UNIX command line, not within a shell script.

The PS2 Variable: prompt string for continued command line (default ‘> ‘).

**The LOGNAME Variable**

contains user’s login name. Its contents cannot be changed by the user, but can be displayed:

echo “${LOGNAME}”

**The TERM Variable**

Names the kind of terminal you are using; setting it helps to manage your screen more effectively, say,

$ TERM=vt100

**The PWD Variable**

The current working directory can be displayed:

echo “${PWD}”

In fact the whole environment table can be displayed.

$IFS: String of characters which are used as word separators in command line ( space, tab, newline chars).

**The MAIL Variable**

Names the standard file where your mail is kept

**The .profile File**

Some of above variables like HOME and LOGNAME are set automatically each time the user logs in. The others, however, have to be set. The .profile is used for this purpose as it is executed as soon the user logs in. A sample .profile  file would look like:

PATH=.:/bin:/usr/bin

export HOME PATH PS1 MAIL

**Process state:**- The state may be new, ready, running, waiting, halted, and so on.  
**Program counter:**- The counter indicates the address of the next instruction to be  
executed for this process.  
**CPU registers:**- The registers vary in number and type, depending on the computer  
architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.  
**CPU-scheduling information**:- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.  
**Memory-management information**:- This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the OS.  
**Accounting information:**- This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.  
**I/O status information**:- This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

**ps command:**  
The ps (i.e., process status) command is used to provide information about the currently running processes in the system.

When ps is used without any option four columns of information labeled PID,TTY, TIME and CMD for at least two processes, the shell and ps will be visible.

PID :- The process are identified by a 5 digit number known as PID (Process Identification Number).  
TTY :- Is the name of the console or terminal that the user logged into (Stands for terminal type now  
but originally stood for teletype).  
TIME :- Is the amount of CPU time in minutes and seconds that the process has been running.  
CMD :- is the name of the command that launched the process.

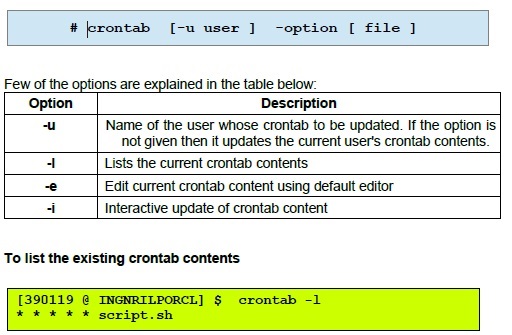
The information that ps -aux command provides about each process are :

* The user of the process,
* PID of the process,
* Percentage of CPU used by the process,
* Percentage of memory used by the process,
* VSZ (virtual size in kilobytes),
* RSS (real memory size or resident set size in 1024 byte units),
* STAT (the process state code, explained later),
* Starting time of the process,
* Length of time the process has been active
* The command that initiated the process.

**Types of process:**  
**Parent and Child Process**  
A process can initiate a sub process, which is a called a child process, the initiating process is referred to as its parent.The child processes, in turn create other child processes forming a tree of processes  
( which can be displayed using ps command with –forest option)  
**Orphan Process**  
When a child process is killed, parent process gets the notification via a signal. Parent then, can continue other task. However if the parent process is killed before, its child, is called an orphan process.  
**Zombie Process**  
When a process finished its execution and exit status not received by the parent ( or parent did not read the exit status till now), the process state becomes zombie.  
The process is dead (not to be scheduled for further execution) but cannot becompletely removed from process table, until it has been determined that exit status is no longer needed.

**Daemon Process**  
Some programs are not designed to be run with continuous user input and disconnect from the terminal when task completed. For example, a web server responds to web requests, rather than user input. Mail servers are another example of this type of application. These types of programs are known as daemons.

**Crontab**  
The word 'crontab' in UNIX stand for chron(chronograph) table, or time table for Unix system .  
The crontab is a special table where it is possible to specify commands and time or interval to execute the command once or repeatedly. Commands can be any executable programs, for example, a script can be written to take regular backup of all the required tables in database.  A crontab instruction can be created to execute the script on the specified time. The cron daemon reads the table and executes the commands at the times specified. **Crontab Command Syntax**



Grep, sed, vi, .