

# Domain Name Rs.445/- p.a.

[Get Cyberciti Domain for Just Rs.445 with 2 Free e-mail]

**Linux Shell Scripting Tutorial (LSST) v1.05r3**  
Chapter 2: Getting started with Shell Programming

[Prev](#)

[Next](#)

## How to write shell script

Following steps are required to write shell script:

- (1) Use any editor like vi or mcedit to write shell script.
- (2) After writing shell script set execute permission for your script as follows

*syntax:*

```
chmod permission your-script-name
```

*Examples:*

```
$ chmod +x your-script-name
$ chmod 755 your-script-name
```

**Note:** This will set read write execute(7) permission for owner, for group and other permission is read and execute only(5).

- (3) Execute your script as

*syntax:*

```
bash your-script-name
sh your-script-name
./your-script-name
```

*Examples:*

```
$ bash bar
$ sh bar
$ ./bar
```

**NOTE** In the last syntax ./ means current directory, But only . (dot) means execute given command file in current shell without starting the new copy of shell, The syntax for . (dot) command is as follows

*Syntax:*

```
. command-name
```

*Example:*

```
$ . foo
```

Now you are ready to write first shell script that will print "Knowledge is Power" on screen. See the [common vi command list](#), if you are new to vi.

```
$ vi first
#
# My first shell script
#
clear
echo "Knowledge is Power"
```

After saving the above script, you can run the script as follows:


```
$ ./first
```

This will not run script since we have not set execute permission for our script *first*; to do this type command

```
$ chmod 755 first
$ ./first
```

First screen will be clear, then Knowledge is Power is printed on screen.

Script Command(s)	Meaning
\$ vi first	Start vi editor
# # My first shell script #	# followed by any text is considered as comment. Comment gives more information about script, logical explanation about shell script. <i>Syntax:</i> # comment-text
clear	clear the screen
echo "Knowledge is Power"	To print message or value of variables on screen, we use echo command, general form of echo command is as follows <i>syntax:</i> echo "Message"

 How Shell Locates the file (My own bin directory to execute script)

**Tip:** For shell script file try to give file extension such as .sh, which can be easily identified by you as shell script.

**Exercise:**

1) Write following shell script, save it, execute it and note down its output.

```
$ vi ginfo
#
#
# Script to print user information who currently login , current date
& time
#
clear
echo "Hello $USER"
echo "Today is \c ";date
echo "Number of user login : \c" ; who | wc -l
echo "Calendar"
cal
exit 0
```

**Future Point:** At the end why statement exit 0 is used? See exit status for more information.

[Advertisement]



[Get Cyberciti Domain for Just Rs.445 with 2 Free e-mail]

# Variables in Shell

To process our data/information, data must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time).

In Linux (Shell), there are two types of variable:

- (1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
- (2) **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in lower letters.

You can see system variables by giving command like `$ set`, some of the important System variables are:

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/vivek	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=vivek	User name who is currently login to this PC

**NOTE** that Some of the above settings can be different in your PC/Linux environment. You can print any of the above variables contains as follows:

```
$ echo $USERNAME
```

```
$ echo $HOME
```

Exercise:

1) If you want to print your home directory location then you give command:

```
a) $ echo $HOME
```

**OR**

(b) \$ echo HOME

Which of the above command is correct & why? [Click here for answer.](#)

**Caution:** Do not modify System variable this can some time create problems.

---

[Prev](#)

How to write shell script

[Home](#)

[Up](#)

[Next](#)

How to define User defined variables  
(UDV)

# How to define User defined variables (UDV)

To define UDV use following syntax

*Syntax:*

variable name=value

'value' is assigned to given '**variable name**' and Value must be on right side = sign.

*Example:*

```
$ no=10 # this is ok
```

```
$ 10=no # Error, NOT Ok, Value must be on right side of = sign.
```

To define variable called 'vech' having value Bus

```
$ vech=Bus
```

To define variable called n having value 10

```
$ n=10
```

# Rules for Naming variable name (Both UDV and System Variable)

(1) Variable name must begin with Alphanumeric character or underscore character (`_`), followed by one or more Alphanumeric character. For e.g. Valid shell variable are as follows

```
HOME
SYSTEM_VERSION
vech
no
```

(2) Don't put spaces on either side of the equal sign when assigning value to variable. For e.g. In following variable declaration there will be no error

```
$ no=10
```

But there will be problem for any of the following variable declaration:

```
$ no =10
$ no= 10
$ no = 10
```

(3) Variables are case-sensitive, just like filename in Linux. For e.g.

```
$ no=10
$ No=11
$ NO=20
$ nO=2
```

Above all are different variable name, so to print value 20 we have to use `$ echo $NO` and not any of the following

```
$ echo $no # will print 10 but not 20
$ echo $No # will print 11 but not 20
$ echo $nO # will print 2 but not 20
```

(4) You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition) For e.g.

```
$ vech=
$ vech=""
```

Try to print it's value by issuing following command

```
$ echo $vech
```

Nothing will be shown because variable has no value i.e. NULL variable.

(5) Do not use `?,*` etc, to name your variable names.

How to define User defined variables  
(UDV)

Up

How to print or access value of UDV  
(User defined variables)

# How to print or access value of UDV (User defined variables)

To print or access UDV use following syntax

*Syntax:*

```
$variablename
```

Define variable vech and n as follows:

```
$ vech=Bus
```

```
$ n=10
```

To print contains of variable 'vech' type

```
$ echo $vech
```

It will print 'Bus', To print contains of variable 'n' type command as follows

```
$ echo $n
```

**Caution:** Do not try **\$ echo vech**, as it will print vech instead its value 'Bus' and **\$ echo n**, as it will print n instead its value '10', You must *use \$ followed by variable name.*

## Exercise

Q.1.How to Define variable x with value 10 and print it on screen.

Q.2.How to Define variable xn with value Rani and print it on screen

Q.3.How to print sum of two numbers, let's say 6 and 3?

Q.4.How to define two variable x=20, y=5 and then to print division of x and y (i.e. x/y)

Q.5.Modify above and store division of x and y to variable called z

Q.6.Point out error if any in following script

```
$ vi variscript
#
#
# Script to test MY knowledge about variables!
#
myname=Vivek
myos = TroubleOS
myno=5
echo "My name is $myname"
echo "My os is $myos"
echo "My number is myno, can you see this number"
```

[For Answers Click here](#)



Rules for Naming variable name (Both  
UDV and System Variable)

Up

echo Command

# echo Command

Use echo command to display text or value of variable.

```
echo [options] [string, variables...]
```

Displays text or variables value on screen.

Options

-n Do not output the trailing new line.

-e Enable interpretation of the following backslash escaped characters in the strings:

\a alert (bell)

\b backspace

\c suppress trailing new line


\n new line

\r carriage return

\t horizontal tab

\\ backslash

For e.g. **`$ echo -e "An apple a day keeps away \a\t\tdoctor\n"`**

 How to display colorful text on screen with bold or blink effects, how to print text on any row, column on screen, [click here for more!](#)

# Shell Arithmetic

Use to perform arithmetic operations.

*Syntax:*

```
expr op1 math-operator op2
```

*Examples:*

```
$ expr 1 + 3
$ expr 2 - 1
$ expr 10 / 2
$ expr 20 % 3
$ expr 10 \* 3
$ echo `expr 6 + 3`
```

**Note:**

expr 20 %3 - Remainder read as 20 mod 3 and remainder is 2.

expr 10 \\* 3 - Multiplication use \\* and not \* since its wild card.

For the last statement not the following points

(1) First, before expr keyword we used ` (back quote) sign not the (single quote i.e. ') sign. Back quote is generally found on the key under tilde (~) on PC keyboard OR to the above of TAB key.

(2) Second, expr is also end with ` i.e. back quote.

(3) Here expr 6 + 3 is evaluated to 9, then echo command prints 9 as sum

(4) Here if you use double quote or single quote, it will NOT work

For e.g.

```
$ echo "expr 6 + 3" # It will print expr 6 + 3
```

```
$ echo 'expr 6 + 3' # It will print expr 6 + 3
```

 See [Parameter substitution - To save your time.](#)

# More about Quotes

There are three types of quotes

Quotes	Name	Meaning
"	Double Quotes	"Double Quotes" - Anything enclosed in double quotes removed meaning of that characters (except \ and \$).
'	Single quotes	'Single quotes' - Enclosed in single quotes remains unchanged.
`	Back quote	`Back quote` - To execute command

*Example:*

```
$ echo "Today is date"
```

Can't print message with today's date.

```
$ echo "Today is `date`".
```

It will print today's date as, Today is Tue Jan ....., Can you see that the `date` statement uses back quote?

# Exit Status

By default in Linux if particular command/shell script is executed, it return two type of values which is used to see whether command or shell script executed is successful or not.

- (1) If return *value is zero* (0), command is successful.
- (2) If return *value is nonzero*, command is not successful or some sort of error executing command/shell script.

This value is know as *Exit Status*.

But how to find out exit status of command or shell script?  
Simple, to determine this exit Status you can use `$?` special variable of shell.

For e.g. (This example assumes that **unknow1file** doest not exist on your hard drive)

```
$ rm unknow1file
```

It will show error as follows

```
rm: cannot remove `unkowm1file': No such file or directory  
and after that if you give command
```

```
$ echo $?
```

it will print nonzero value to indicate error. Now give command

```
$ ls
```

```
$ echo $?
```

It will print 0 to indicate command is successful.

## Exercise

Try the following commands and not down the exit status:

```
$ expr 1 + 3
```

```
$ echo $?
```

```
$ echo Welcome
```

```
$ echo $?
```

```
$ wildwest canwork?
```


```
$ echo $?
```

```
$ date
```

```
$ echo $?
```

```
$ echon $?
```

```
$ echo $?
```

 **\$?** useful variable, want to know more such Linux variables [click here](#) to explore them!

[Prev](#)

More about Quotes

[Home](#)

[Up](#)

[Next](#)

The read Statement

# The read Statement

Use to get input (data from user) from keyboard and store (data) to variable.

*Syntax:*

```
read variable1, variable2,...variableN
```

Following script first ask user, name and then waits to enter name from the user via keyboard. Then user enters name from keyboard (after giving name you have to press ENTER key) and entered name through keyboard is stored (assigned) to variable fname.

```
$ vi sayH
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

Run it as follows:

```
$ chmod 755 sayH
$ ./sayH
Your first name please: vivek
Hello vivek, Lets be friend!
```

# Wild cards (Filename Shorthand or meta Characters)

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 files name beginning with letter a,b or c like

```
/bin/arch      /bin/awk      /bin/bsh     /bin/chmod    /bin/cp
/bin/ash      /bin/basename /bin/cat     /bin/chown    /bin/cpio
/bin/ash.static /bin/bash    /bin/chgrp   /bin/consolechars /bin/csh
```

But

```
$ ls /bin/[!a-o]
```

```
$ ls /bin/[^a-o]
```

If the first character following the [ is a ! or a ^ ,then any character not enclosed is matched i.e. do not show us file name that beginning with a,b,c,e...o, like



```
/bin/ps      /bin/rvi      /bin/sleep /bin/touch   /bin/view
/bin/pwd     /bin/rview    /bin/sort  /bin/true   /bin/wcomp
/bin/red     /bin/sayHello /bin/stty  /bin/umount /bin/xconf
/bin/remadmin /bin/sed      /bin/su    /bin/uname  /bin/ypdomainname
/bin/rm      /bin/setserial /bin/sync  /bin/userconf /bin/zcat
/bin/rmdir   /bin/sfxload   /bin/tar   /bin/usleep
/bin/rpm     /bin/sh        /bin/tcsh  /bin/vi
```

---

[Prev](#)

The read Statement

[Home](#)

[Up](#)

[Next](#)

More command on one command line

# More command on one command line

*Syntax:*

command1;command2

To run two command with one command line.

*Examples:*

**\$ date;who**

Will print today's date followed by users who are currently login. Note that You can't use

**\$ date who**

for same purpose, you must put semicolon in between date and who command.

# Command Line Processing

Try the following command (assumes that the file "grate\_stories\_of" is not exist on your system)

```
$ ls grate_stories_of
```

It will print message something like - *grate\_stories\_of: No such file or directory.*

**ls** is the name of an *actual command* and shell executed this command when you type command at shell prompt. Now it creates one more question **What are commands?** What happened when you type `$ ls grate_stories_of`?

The first word on command line is, **ls** - is name of the command to be executed.

Everything else on command line is taken *as arguments to this command*. For e.g.

```
$ tail +10 myf
```

Name of command is **tail**, and the arguments are **+10** and **myf**.

## Exercise

Try to determine command and arguments from following commands

```
$ ls foo
$ cp y y.bak
$ mv y.bak y.okay
$ tail -10 myf
$ mail raj
$ sort -r -n myf
$ date
$ clear
```

Answer:

Command	No. of argument to this command (i.e \$#)	Actual Argument
ls	1	foo
cp	2	y and y.bak
mv	2	y.bak and y.okay
tail	2	-10 and myf
mail	1	raj
sort	3	-r, -n, and myf
date	0	
clear	0	

## NOTE:

**\$#** holds number of arguments specified on command line. And **\$\*** or **\$@** refer to all arguments passed to

script.

---

[Prev](#)

More commands on one command line

[Home](#)

[Up](#)

[Next](#)

Why Command Line arguments required

# Why Command Line arguments required

1. Telling the command/utility which option to use.
2. Informing the utility/command which file or group of files to process (reading/writing of files).

Let's take rm command, which is used to remove file, but which file you want to remove and how you will tell this to rm command (even rm command don't ask you name of file that you would like to remove). So what we do is we write command as follows:

**\$ rm {file-name}**

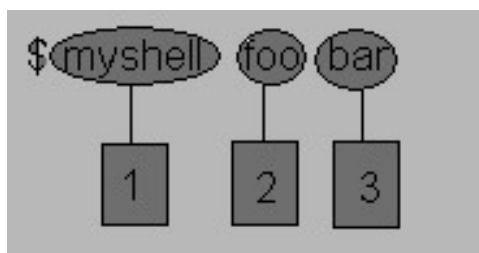
Here rm is command and filename is file which you would like to remove. This way you tell rm command which file you would like to remove. So we are doing one way communication with our command by specifying filename. Also you can pass command line arguments to your script to make it more users friendly. But how we access command line argument in our script.

Let's take ls command

**\$ ls -a /\***

This command has 2 command line arguments -a and /\* is another. For shell script,

**\$ myshell foo bar**



1 Shell Script name i.e. myshell

2 First command line argument passed to myshell i.e. foo

3 Second command line argument passed to myshell i.e. bar

In shell if we wish to refer this command line argument we refer above as follows

1 myshell it is \$0

2 foo it is \$1

2 bar it is \$2

Here \$# (built in shell variable ) will be 2 (Since foo and bar only two Arguments), Please note at a time such 9 arguments can be used from \$1..\$9, You can also refer all of them by using \$\* (which expand to ` \$1,\$2...\$9 `). Note that \$1..\$9 i.e command line arguments to shell script is know as "*positional parameters*".

### Exercise

Try to write following for commands

Shell Script Name (\$0),

No. of Arguments (i.e. \$#),

And actual argument (i.e. \$1,\$2 etc)

```
$ sum 11 20
$ math 4 - 7
$ d
$ bp -5 myf +20
$ Ls *
$ cal
$ findBS 4 8 24 BIG
```

### Answer

Shell Script Name	No. Of Arguments to script	Actual Argument (\$1,..\$9)				
		\$1	\$2	\$3	\$4	\$5
\$0	#					
sum	2	11	20			
math	3	4	-	7		
d	0					
bp	3	-5	myf	+20		
Ls	1	*				
cal	0					
findBS	4	4	8	24	BIG	

Following script is used to print command ling argument and will show you how to access them:

```
$ vi demo
#!/bin/sh
#
# Script that demos, command line args
#
echo "Total number of command line argument are $#"
```

Run it as follows

Set execute permission as follows:

```
$ chmod 755 demo
```

Run it & test it as follows:

```
$ ./demo Hello World
```

If test successful, copy script to your own bin directory (Install script for private use)

```
$ cp demo ~/bin
```

Check whether it is working or not (?)

```
$ demo
```

```
$ demo Hello World
```

**NOTE:** After this, for any script you have to used above command, in sequence, I am not going to show you all of the above command(s) for rest of Tutorial.

Also note that you *can't assigne the new value to command line arguments i.e positional parameters.*

So following all statements in shell script are invalid:

```
$1 = 5
```

```
$2 = "My Name"
```

---

[Prev](#)

[Home](#)

[Next](#)

Command Line Processing

[Up](#)

Redirection of Standard output/input  
i.e.Input - Output redirection

# Redirection of Standard output/input i.e. Input - Output redirection

Mostly all commands give output on screen or take input from keyboard, but in Linux (and in other OSs also) it's possible to send output to file or to read input from file.

For e.g.

**\$ ls** command gives output to screen; to send output to file of **ls** command give command

**\$ ls > filename**

It means put output of **ls** command to filename.

There are three main redirection symbols **>, >>, <**

(1) **>** Redirector Symbol

*Syntax:*

Linux-command **>** filename

To output Linux-commands result (output of command or shell script) to file. Note that if file already exist, it will be overwritten else new file is created. For e.g. To send output of **ls** command give

**\$ ls > myfiles**

Now if '**myfiles**' file exist in your current directory it will be overwritten without any type of warning.

(2) **>>** Redirector Symbol

*Syntax:*

Linux-command **>>** filename

To output Linux-commands result (output of command or shell script) to END of file. Note that if file exist, it will be opened and new information/data will be written to END of file, without losing previous information/data, And if file is not exist, then new file is created. For e.g. To send output of **date** command to already exist file give command

**\$ date >> myfiles**

(3) **<** Redirector Symbol

*Syntax:*

Linux-command **<** filename

To take input to Linux-command from file instead of key-board. For e.g. To take input for **cat** command give

**\$ cat < myfiles**

 [Click here to learn more about I/O Redirection](#)

You can also use above redirectors simultaneously as follows

Create text file name as follows



**\$ cat > sname**

vivek  
ashish  
zebra  
babu

*Press CTRL + D to save.*

Now issue following command.

**\$ sort < sname > sorted\_names**

**\$ cat sorted\_names**

ashish  
babu  
vivek  
zebra

In above example sort (**\$ sort < sname > sorted\_names**) command takes input from sname file and output of sort command (i.e. sorted names) is redirected to sorted\_names file.

Try one more example to clear your idea:

**\$ tr "[a-z]" "[A-Z]" < sname > cap\_names**

**\$ cat cap\_names**

VIVEK  
ASHISH  
ZEBRA  
BABU

**tr** command is used to translate all lower case characters to upper-case letters. It take input from sname file, and tr's output is redirected to cap\_names file.

**Future Point :** Try following command and find out most important point:

**\$ sort > new\_sorted\_names < sname**

**\$ cat new\_sorted\_names**

---

[Prev](#)

Why Command Line arguments required

[Home](#)

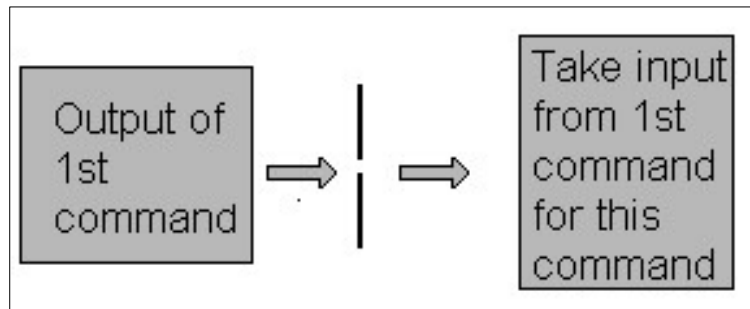
[Up](#)

[Next](#)

Pipe

# Pipes

A pipe is a way to connect the output of one program to the input of another program without any temporary file.



Pipe Defined as:

*"A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands ( Multiple commands) from same command line."*

*Syntax:*

`command1 | command2`

*Examples:*

Command using Pipes	Meaning or Use of Pipes
<code>\$ ls   more</code>	Output of ls command is given as input to more command So that output is printed one screen full page at a time.
<code>\$ who   sort</code>	Output of who command is given as input to sort command So that it will print sorted list of users
<code>\$ who   sort &gt; user_list</code>	Same as above except output of sort is send to (redirected) user_list file
<code>\$ who   wc -l</code>	Output of who command is given as input to wc command So that it will print number of user who logon to system
<code>\$ ls -l   wc -l</code>	Output of ls command is given as input to wc command So that it will print number of files in current directory.

**\$ who | grep raju**

Output of who command is given as input to grep command So that it will print if particular user name if he is logon or nothing is printed (To see particular user is logon or not)

[Prev](#)

Redirection of Standard output/input  
i.e.Input - Output redirection

[Home](#)

[Up](#)

[Next](#)

Filter

# Filter

If a Linux command accepts its input from the standard input and produces its output on standard output is know as a filter. A filter performs some kind of process on the input and gives output. For e.g.. Suppose you have file called 'hotel.txt' with 100 lines data, And from 'hotel.txt' you would like to print contains from line number 20 to line number 30 and store this result to file called 'hlist' then give command:

```
$ tail +20 < hotel.txt | head -n30 >hlist
```

Here **head** command is filter which takes its input from tail command (tail command start selecting from line number 20 of given file i.e. hotel.txt) and passes this lines as input to head, whose output is redirected to 'hlist' file.

Consider one more following example

```
$ sort < sname | uniq > u_sname
```

Here uniq is filter which takes its input from sort command and passes this lines as input to uniq; Then uniqs output is redirected to "u\_sname" file.

# What is Processes

Process is kind of program or task carried out by your PC. For e.g.

**\$ ls -lR**

**ls** command or a request to list files in a directory and all subdirectory in your current directory - It is a process.

Process defined as:

*"A process is program (command given by user) to perform specific Job. In Linux when you start process, it gives a number to process (called PID or process-id), PID starts from 0 to 65535."*

# Why Process required

As You know Linux is multi-user, multitasking Os. It means you can run more than two process simultaneously if you wish. For e.g. To find how many files do you have on your system you may give command like:

```
$ ls / -R | wc -l
```

This command will take lot of time to search all files on your system. So you can run such command in Background or simultaneously by giving command like

```
$ ls / -R | wc -l &
```

The **ampersand (&)** at the end of command tells shells start process (**ls / -R | wc -l**) and run it in background takes next command immediately.

Process & PID defined as:

*"An instance of running command is called **process** and the number printed by shell is called **process-id (PID)**, this PID can be use to refer specific running process."*

# Linux Command Related with Process

Following tables most commonly used command(s) with process:

For this purpose	Use this Command	Examples*
To see currently running process	ps	\$ ps
To stop any process by PID i.e. to kill process	kill {PID}	\$ kill 1012
To stop processes by name i.e. to kill process	killall {Process-name}	\$ killall httpd
To get information about all running process	ps -ag	\$ ps -ag
To stop all process except your shell	kill 0	\$ kill 0
For background processing (With &, use to put particular command and program in background)	linux-command &	\$ ls / -R   wc -l &
To display the owner of the processes along with the processes	ps aux	\$ ps aux
To see if a particular process is running or not. For this purpose you have to use ps command in combination with the grep command	ps ax   grep process-U-want-to see	For e.g. you want to see whether Apache web server process is running or not then give command \$ ps ax   grep httpd
To see currently running processes and other information like memory and CPU usage with real time updates.	top <small>See the output of top command.</small>	\$ top  Note that to exit from top command press q.
To display a tree of processes	pstree	\$ pstree

\* To run some of this command you need to be root or equivalent user.

**NOTE** that you can only kill process which are created by yourself. A Administrator can almost kill 95-98% process. But some process can not be killed, such as VDU Process.

## Exercise:

You are working on your Linux workstation (might be learning LSST or some other work like sending mails, typing letter), while doing this work you have started to play MP3 files on your workstation. Regarding this situation, answer the following question:

- 1) Is it example of Multitasking?
- 2) How you will you find out the both running process (MP3 Playing & Letter typing)?
- 3) "Currently only two Process are running in your Linux/PC environment", Is it True or False?, And how you will verify this?
- 4) You don't want to listen music (MP3 Files) but want to continue with other work on PC, you will take any of the following action:
  1. Turn off Speakers
  2. Turn off Computer / Shutdown Linux Os
  3. Kill the MP3 playing process
  4. None of the above

[Click here for answers.](#)

---

[Prev](#)

[Home](#)

[Next](#)

Why Process required

[Up](#)

Shells (bash) structured Language  
Constructs



# Introduction

Making decision is important part in ONCE life as well as in computers logical driven program. In fact logic is not LOGIC until you use decision making. This chapter introduces to the bash's structured language constructs such as:

- Decision making
- Loops

Is there any difference making decision in Real life and with Computers? Well real life decision are quite complicated to all of us and computers even don't have that much power to understand our real life decisions. What computer know is 0 (zero) and 1 that is Yes or No. To make this idea clear, lets play some game (WOW!) with bc - Linux calculator program.

**\$ bc**

After this command bc is started and waiting for your commands, i.e. give it some calculation as follows type  $5 + 2$  as:

**5 + 2**

7

7 is response of bc i.e. addition of  $5 + 2$  you can even try

**5 - 2**

**5 / 2**

See what happened if you type  $5 > 2$  as follows

**5 > 2**

1

1 (One?) is response of bc, How? bc compare 5 with 2 as, Is 5 is greater then 2, (If I ask same question to you, your answer will be YES), bc gives this 'YES' answer by showing 1 value. Now try

**5 < 2**

0

0 (Zero) indicates the false i.e. Is 5 is less than 2?, Your answer will be no which is indicated by bc by showing 0 (Zero). Remember in bc, relational expression always returns **true** (1) or **false** (0 - zero).

Try following in bc to clear your Idea and not down bc's response

**5 > 12**

**5 == 10**

**5 != 2**

**5 == 5**

**12 < 2**

Expression	Meaning to us	Your Answer	BC's Response
$5 > 12$	Is 5 greater than 12	NO	0
$5 == 10$	Is 5 is equal to 10	NO	0
$5 != 2$	Is 5 is NOT equal to 2	YES	1

5 == 5	Is 5 is equal to 5	YES	1
1 < 2	Is 1 is less than 2	Yes	1

It means when ever there is any type of comparison in Linux Shell It gives only two answer one is YES and NO is other.

In Linux Shell Value	Meaning	Example
Zero Value (0)	Yes/True	0
NON-ZERO Value	No/False	-1, 32, 55 anything but not zero

Remember both bc and Linux Shell uses *different ways to show True/False values*

Value	Shown in bc as	Shown in Linux Shell as
True/Yes	1	0
False/No	0	Non - zero value

[Prev](#)  
Linux Command(s) Related with Process

[Home](#)  
[Up](#)

[Next](#)  
if condition

# if condition

if condition which is used for decision making in shell script, If given condition is true then command1 is executed.

*Syntax:*

```
if condition
then
    command1 if condition is true or if exit status
    of condition is 0 (zero)
    ...
    ...
fi
```

Condition is defined as:

*"Condition is nothing but comparison between two values."*

For compression you can use test or [ expr ] statements or even exist status can be also used.

Expression is defined as:

*"An expression is nothing but combination of values, relational operator (such as >, <, <> etc) and mathematical operators (such as +, -, / etc)."*

Following are all examples of expression:

```
5 > 2
3 + 6
3 * 65
a < b
c > 5
c > 5 + 30 -1
```

Type following commands (assumes you have file called **foo**)

```
$ cat foo
```

```
$ echo $?
```

The cat command return zero(0) i.e. exit status, on successful, this can be used, in if condition as follows,

Write shell script as

```
$ cat > showfile
#!/bin/sh
#
#Script to print file
#
if cat $1
then
echo -e "\n\nFile $1, found and successfully echoed"
fi
```

Run above script as:

```
$ chmod 755 showfile
```

```
$/showfile foo
```

Shell script name is showfile (\$0) and foo is argument (which is \$1). Then shell compare it as follows: if cat \$1 which is expanded to if cat foo.

### ***Detailed explanation***

if cat command finds foo file and if its successfully shown on screen, it means our cat command is successful and its exist status is 0 (indicates success), So our if condition is also true and hence statement echo -e "\n\nFile \$1, found and successfully echoed" is proceed by shell. Now if cat command is not successful then it returns non-zero value (indicates some sort of failure) and this statement echo -e "\n\nFile \$1, found and successfully echoed" is skipped by our shell.

### **Exercise**

Write shell script as follows:

```
cat > trmif
#
# Script to test rm command and exist status
#
if rm $1
then
echo "$1 file deleted"
fi
```

Press Ctrl + d to save

```
$ chmod 755 trmif
```

Answer the following question in referance to above script:

- (A) foo file exists on your disk and you give command, \$ **./trmfi foo** what will be output?
- (B) If bar file not present on your disk and you give command, \$ **./trmfi bar** what will be output?
- (C) And if you type \$ **./trmfi** What will be output?

[For Answer click here.](#)

## Shells (bash) structured Language Constructs

Up

test command or [ expr ]

# test command or [ expr ]

test command or [ expr ] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.

*Syntax:*

test expression OR [ expression ]

*Example:*

Following script determine whether given argument number is positive.

```
$ cat > ispositive
#!/bin/sh
#
# Script to see whether argument is positive
#
if test $1 -gt 0
then
echo "$1 number is positive"
fi
```

Run it as follows

**\$ chmod 755 ispostive**

**\$ ispostive 5**

*5 number is positive*

**\$ ispostive -45**

*Nothing is printed*

**\$ ispostive**

*./ispostive: test: -gt: unary operator expected*

## ***Detailed explanation***

The line, if test \$1 -gt 0 , test to see if first command line argument(\$1) is greater than 0. If it is true(0) then test will return 0 and output will printed as 5 number is positive but for -45 argument there is no output because our condition is not true(0) (no -45 is not greater than 0) hence echo statement is skipped. And for last statement we have not supplied any argument hence error ./ispostive: test: -gt: unary operator expected, is generated by shell , to avoid such error we can test whether command line argument is supplied or not.

test or [ expr ] works with

- 1.Integer ( Number without decimal point)
- 2.File types
- 3.Character strings

**For Mathematics, use following operator in Shell Script**

Mathematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [ expr ] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if [ 5 -eq 6 ]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if [ 5 -ne 6 ]
-lt	is less than	5 < 6	if test 5 -lt 6	if [ 5 -lt 6 ]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if [ 5 -le 6 ]
-gt	is greater than	5 > 6	if test 5 -gt 6	if [ 5 -gt 6 ]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if [ 5 -ge 6 ]

**NOTE:** == is equal, != is not equal.

**For string Comparisons use**

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

**Shell also test for file and directory types**

Test	Meaning
-s file	Non empty file
-f file	Is File exist or normal file and not a directory
-d dir	Is Directory exist and not a file
-w file	Is writeable file
-r file	Is read-only file
-x file	Is file is executable

**Logical Operators**

Logical operators are used to combine two or more condition at a time

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2	Logical AND

expression1 -o expression2

Logical OR

---

[Prev](#)

Decision making in shell script ( i.e. if command)

[Home](#)

[Up](#)

[Next](#)

if...else...fi



# if...else...fi

If given condition is true then command1 is executed otherwise command2 is executed.

*Syntax:*

```
if condition
then
    condition is zero (true - 0)
    execute all commands up to else statement

else
    if condition is not true then
    execute all commands up to fi

fi
```

For e.g. Write Script as follows:

```
$ vi isnump_n
#!/bin/sh
#
# Script to see whether argument is positive or negative
#
if [ $# -eq 0 ]
then
echo "$0 : You must give/supply one integers"
exit 1
fi

if test $1 -gt 0
then
echo "$1 number is positive"
else
echo "$1 number is negative"
fi
```

Try it as follows:

```
$ chmod 755 isnump_n
```

```
$ isnump_n 5
```

```
5 number is positive
```

```
$ isnump_n -45
```

*-45 number is negative*

## **\$ isnum\_p\_n**

*./ispos\_n : You must give/supply one integers*

## **\$ isnum\_p\_n 0**

*0 number is negative*

### **Detailed explanation**

First script checks whether command line argument is given or not, if not given then it print error message as *./ispos\_n : You must give/supply one integers*. if statement checks whether number of argument (\$#) passed to script is not equal (-eq) to 0, if we passed any argument to script then this if statement is false and if no command line argument is given then this if statement is true. The echo command i.e.

```
echo "$0 : You must give/supply one integers"
```

```
  |           |
  |           |
  1           2
```

1 will print Name of script

2 will print this error message

And finally statement exit 1 causes normal program termination with exit status 1 (nonzero means script is not successfully run).

The last sample run **\$ isnum\_p\_n 0** , gives output as *"0 number is negative"*, because given argument is not > 0, hence condition is false and it's taken as negative number. To avoid this replace second if statement with **if test \$1 -ge 0**.

## **💡 Nested if-else-fi**

You can write the entire if-else construct within either the body of the if statement or the body of an else statement. This is called the nesting of ifs.

```
$ vi nestedif.sh
osch=0

echo "1. Unix (Sun Os) "
echo "2. Linux (Red Hat) "
echo -n "Select your os choice [1 or 2]? "
read osch

if [ $osch -eq 1 ] ; then

    echo "You Pick up Unix (Sun Os) "

else #### nested if i.e. if within if #####
```

```

if [ $osch -eq 2 ] ; then
    echo "You Pick up Linux (Red Hat) "
else
    echo "What you don't like Unix/Linux OS."
fi

```

```
fi
```

Run the above shell script as follows:

```
$ chmod +x nestedif.sh
```

```
$ ./nestedif.sh
```

```
1. Unix (Sun Os)
```

```
2. Linux (Red Hat)
```

```
Select you os choice [1 or 2]? 1
```

```
You Pick up Unix (Sun Os)
```

```
$ ./nestedif.sh
```

```
1. Unix (Sun Os)
```

```
2. Linux (Red Hat)
```

```
Select you os choice [1 or 2]? 2
```

```
You Pick up Linux (Red Hat)
```

```
$ ./nestedif.sh
```

```
1. Unix (Sun Os)
```

```
2. Linux (Red Hat)
```

```
Select you os choice [1 or 2]? 3
```

```
What you don't like Unix/Linux OS.
```

Note that Second *if-else* construct is nested in the first *else* statement. If the condition in the first *if* statement is false the the condition in the second *if* statement is checked. If it is false as well the final *else* statement is executed.

You can use the nested *ifs* as follows also:

*Syntax:*

```

if condition
then
    if condition
    then
        .....
        ..
        do this
    else
        ....
        ..
        do this
    fi
else

```

```
        . . .  
        . . . . .  
do this  
fi
```

---

[Prev](#)  
test command or [ expr ]

[Home](#)  
[Up](#)

[Next](#)  
Multilevel if-then-else

# Multilevel if-then-else

*Syntax:*

```

if condition
then
    condition is zero (true - 0)
    execute all commands up to elif statement
elif condition1
then
    condition1 is zero (true - 0)
    execute all commands up to elif statement
elif condition2
then
    condition2 is zero (true - 0)
    execute all commands up to elif statement
else
    None of the above condition, condition1, condition2 are true (i.e.
    all of the above nonzero or false)
    execute all commands up to fi
fi

```

For multilevel if-then-else statement try the following script:

```

$ cat > elf
#
#!/bin/sh
# Script to test if..elif...else
#
if [ $1 -gt 0 ]; then
    echo "$1 is positive"
elif [ $1 -lt 0 ]
then
    echo "$1 is negative"
elif [ $1 -eq 0 ]
then
    echo "$1 is zero"
else
    echo "Oops! $1 is not number, give number"
fi

```

Try above script as follows:

**\$ chmod 755 elf**

**\$ ./elf 1**

**\$ ./elf -2**

**\$ ./elf 0**

**\$ ./elf a**

Here o/p for last sample run:

**./elf: [: -gt: unary operator expected**

**./elf: [: -lt: unary operator expected**

**./elf: [: -eq: unary operator expected**

**Oops! a is not number, give number**

Above program gives error for last run, here integer comparison is expected therefore error like `./elf: [: -gt: unary operator expected`" occurs, but still our program notify this error to user by providing message `"Oops! a is not number, give number"`.

---

[Prev](#)

if...else...fi

[Home](#)

[Up](#)

[Next](#)

Loops in Shell Scripts

# Loops in Shell Scripts

Loop defined as:

*"Computer can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop."*

Bash supports:

- for loop
- while loop

**Note** that in each and every loop,

- (a) First, the variable used in loop condition must be initialized, then execution of the loop begins.
- (b) A test (condition) is made at the beginning of each iteration.
- (c) The body of loop ends with a statement that modifies the value of the test (condition) variable.

# for Loop

*Syntax:*

```
for { variable name } in { list }
do
    execute one for each item in the list until the list is
    not finished (And repeat all statement between do and done)
done
```

Before try to understand above syntax try the following script:

```
$ cat > testfor
for i in 1 2 3 4 5
do
echo "Welcome $i times"
done
```

Run it above script as follows:

```
$ chmod +x testfor
```

```
$ ./testfor
```

The for loop first creates i variable and assigned a number to i from the list of number from 1 to 5, The shell execute echo statement for each assignment of i. (This is usually know as iteration) This process will continue until all the items in the list were not finished, because of this it will repeat 5 echo statements. To make you idea more clear try following script:

```
$ cat > mtable
#!/bin/sh
#
#Script to test for loop
#
#
if [ $# -eq 0 ]
then
echo "Error - Number missing form command line argument"
echo "Syntax : $0 number"
echo "Use to print multiplication table for given number"
exit 1
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
do
echo "$n * $i = `expr $i \* $n`"
done
```

Save above script and run it as:

```
$ chmod 755 mtable
```

```
$ ./mtable 7
```

```
$ ./mtable
```

For first run, above script print multiplication table of given number where i = 1,2 ... 10 is multiply by given n (here



command line argument 7) in order to produce multiplication table as

```
7 * 1 = 7
7 * 2 = 14
...
..
7 * 10 = 70
```

And for second test run, it will print message -

**Error - Number missing form command line argument**

**Syntax : ./mtable number**

**Use to print multiplication table for given number**

This happened because we have not supplied given number for which we want multiplication table, Hence script is showing Error message, Syntax and usage of our script. This is good idea if our program takes some argument, let the user know what is use of the script and how to used the script.

**Note** that to terminate our script we used 'exit 1' command which takes 1 as argument (1 indicates error and therefore script is terminated)

Even you can use following syntax:

*Syntax:*

```
for ( ( expr1; expr2; expr3 ) )
do
    .....
    ...
    repeat all statements between do and
    done until expr2 is TRUE
Done
```

In above syntax BEFORE the first iteration, **expr1** is evaluated. This is usually used to initialize variables for the loop. All the statements between do and done is executed repeatedly UNTIL the value of **expr2** is TRUE. AFTER each iteration of the loop, **expr3** is evaluated. This is usually use to increment a loop counter.

```
$ cat > for2
for (( i = 0 ; i <= 5; i++ ))
do
    echo "Welcome $i times"
done
```

Run the above script as follows:

**\$ chmod +x for2**  
**\$ ./for2**

```
Welcome 0 times
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times
```

In above example, first expression ( $i = 0$ ), is used to set the value variable **i** to zero. Second expression is condition i.e. all statements between do and done executed as long as expression 2 (i.e continue as long as the value of variable **i** is less than or equal to 5) is TRUE. Last expression **i++** increments the value of **i** by 1 i.e. it's equivalent to  $i = i + 1$  statement.

# 💡 Nesting of for Loop

As you see the if statement can nested, similarly loop statement can be nested. You can nest the for loop. To understand the nesting of for loop see the following shell script.

```
$ vi nestedfor.sh
for (( i = 1; i <= 5; i++ ))      ### Outer for loop ###
do

    for (( j = 1 ; j <= 5; j++ )) ### Inner for loop ###
    do
        echo -n "$i "
    done

    echo "" ##### print the new line ###

done
```

Run the above script as follows:

```
$ chmod +x nestedfor.sh
```

```
$ ./nestedfor.sh
```

```
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

Here, for each value of **i** the inner loop is cycled through 5 times, with the variable **j** taking values from 1 to 5. The inner for loop terminates when the value of **j** exceeds 5, and the outer loop terminates when the value of **i** exceeds 5.

Following script is quite interesting, it prints the chess board on screen.

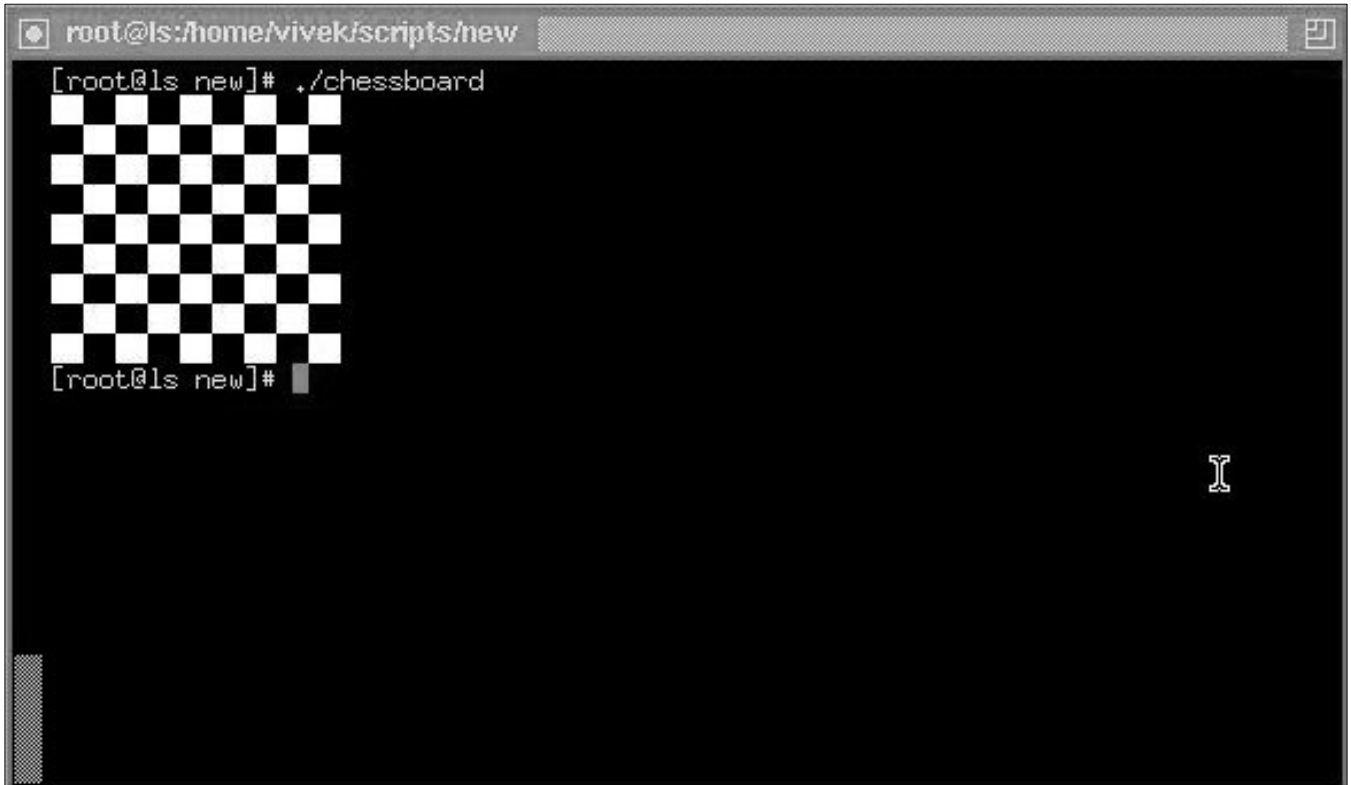
```
$ vi chessboard
for (( i = 1; i <= 9; i++ )) ### Outer for loop ###
do
    for (( j = 1 ; j <= 9; j++ )) ### Inner for loop ###
    do
        tot=`expr $i + $j`
        tmp=`expr $tot % 2`
        if [ $tmp -eq 0 ]; then
            echo -e -n "\033[47m "
        else
            echo -e -n "\033[40m "
        fi
    done
    echo -e -n "\033[40m" ##### set back background colour to black
    echo "" ##### print the new line ###
done
```

Run the above script as follows:

```
$ chmod +x chessboard
```

```
$ ./chessboard
```

On my terminal above script produce the output as follows:



Above shell script can be explained as follows:

Command(s)/Statements	Explanation
for (( i = 1; i <= 9; i++ )) do	Begin the outer loop which runs 9 times., and the outer loop terminates when the value of <b>i</b> exceeds 9
for (( j = 1 ; j <= 9; j++ )) do	Begins the inner loop, for each value of <b>i</b> the inner loop is cycled through 9 times, with the variable <b>j</b> taking values from 1 to 9. The inner for loop terminates when the value of <b>j</b> exceeds 9.
tot=`expr \$i + \$j` tmp=`expr \$tot % 2`	See for even and odd number positions using these statements.
if [ \$tmp -eq 0 ]; then echo -e -n "\033[47m " else echo -e -n "\033[40m " fi	If even number position print the white colour block (using <b>echo -e -n "\033[47m "</b> statement); otherwise for odd position print the black colour box (using <b>echo -e -n "\033[40m "</b> statement). These statements are responsible to print entire chess board on screen with alternate colours.
done	End of inner loop
echo -e -n "\033[40m"	Make sure its black background as we always have on our terminals.
echo ""	Print the blank line
done	End of outer loop and shell scripts get terminated by printing the chess board.

**Exercise**

Try to understand the shell scripts (for loops) shown in exercise chapter.

[Prev](#)

Loops in Shell Scripts

[Home](#)

[Up](#)

[Next](#)

while loop

# while loop

*Syntax:*

```
while [ condition ]
do
    command1
    command2
    command3
    ..
    ....
done
```

Loop is executed as long as given condition is true. For e.g.. [Above for loop program](#) (shown in last section of for loop) can be written using while loop as:

```
$cat > nt1
#!/bin/sh
#
#Script to test while statement
#
#
if [ $# -eq 0 ]
then
    echo "Error - Number missing form command line argument"
    echo "Syntax : $0 number"
    echo " Use to print multiplication table for given number"
exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
    echo "$n * $i = `expr $i \* $n`"
    i=`expr $i + 1`
done
```

Save it and try as

**\$ chmod 755 nt1**

**\$/nt1 7**

Above loop can be explained as follows:

n=\$1	Set the value of command line argument to variable n. (Here it's set to 7 )
i=1	Set variable i to 1
while [ \$i -le 10 ]	This is our loop condition, here if value of i is less than 10 then, shell execute all statements between do and done
do	Start loop
echo "\$n * \$i = `expr \$i \* \$n`"	Print multiplication table as $7 * 1 = 7$ $7 * 2 = 14$ .... $7 * 10 = 70$ , Here each time value of variable n is multiply be i.
i=`expr \$i + 1`	Increment i by 1 and store result to i. ( i.e. $i=i+1$ ) <b>Caution:</b> If you ignore (remove) this statement than our loop become infinite loop because value of variable i always remain less than 10 and program will only output $7 * 1 = 7$ ... ... E (infinite times)
done	Loop stops here if i is not less than 10 i.e. condition of loop is not true. Hence loop is terminated.

[Prev](#)  
for loop

[Home](#)  
[Up](#)

[Next](#)  
The case Statement

# The case Statement

The case statement is good alternative to Multilevel if-then-else-fi statement. It enable you to match several values against one variable. Its easier to read and write.

*Syntax:*

```
case $variable-name in
    pattern1)    command
                ...
                ..
                command;;
    pattern2)    command
                ...
                ..
                command;;
    patternN)    command
                ...
                ..
                command;;
    *)           command
                ...
                ..
                command;;
esac
```

The *\$variable-name* is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. The default is *\*)* and its executed if no match is found. For e.g. write script as follows:

```
$ cat > car
#
# if no vehicle name is given
# i.e. -z $1 is defined and it is NULL
#
# if no command line arg
if [ -z $1 ]
then
    rental="*** Unknown vehicle ***"
elif [ -n $1 ]
then
# otherwise make first arg as rental
```

```

rental=$1
fi

case $rental in
    "car") echo "For $rental Rs.20 per k/m";;
    "van") echo "For $rental Rs.10 per k/m";;
    "jeep") echo "For $rental Rs.5 per k/m";;
    "bicycle") echo "For $rental 20 paisa per k/m";;
    *) echo "Sorry, I can not gat a $rental for you";;
esac

```

Save it by pressing CTRL+D and run it as follows:

```
$ chmod +x car
```

```
$ car van
```

```
$ car car
```

```
$ car Maruti-800
```

First script will check, that if \$1 (first command line argument) is given or not, if NOT given set value of rental variable to "\*\*\* Unknown vehicle \*\*\*", if command line arg is supplied/given set value of rental variable to given value (command line arg). The \$rental is compared against the patterns until a match is found.

For first test run its match with van and it will show output *"For van Rs.10 per k/m."*

For second test run it print, *"For car Rs.20 per k/m"*.

And for last run, there is no match for Maruti-800, hence default i.e. \*) is executed and it prints, *"Sorry, I can not gat a Maruti-800 for you"*.

**Note** that esac is always required to indicate end of case statement.

See the one more [example of case](#) statement in chapter 4 of section shift command.

---

[Prev](#)

while loop

[Home](#)

[Up](#)

[Next](#)

How to de-bug the shell script?



# How to de-bug the shell script?

While programming shell sometimes you need to find the errors (bugs) in shell script and correct the errors (remove errors - debug). For this purpose you can use `-v` and `-x` option with `sh` or `bash` command to debug the shell script. General syntax is as follows:

*Syntax:*

```
sh option { shell-script-name }
```

**OR**

```
bash option { shell-script-name }
```

Option can be

`-v` Print shell input lines as they are read.

`-x` After expanding each simple-command, bash displays the expanded value of PS4 system variable, followed by the command and its expanded arguments.

*Example:*

```
$ cat > dsh1.sh
#
# Script to show debug of shell
#
tot=`expr $1 + $2`
echo $tot
```

Press `ctrl + d` to save, and run it as

```
$ chmod 755 dsh1.sh
```

```
$ ./dsh1.sh 4 5
```

```
9
```

```
$ sh -x dsh1.sh 4 5
```

```
#
# Script to show debug of shell
```

```
#
```

```
tot=`expr $1 + $2`
```

```
expr $1 + $2
```

```
++ expr 4 + 5
```

```
+ tot=9
```

```
echo $tot
```

```
+ echo 9
```

```
9
```

See the above output, `-x` shows the exact values of variables (or statements are shown on screen with values).

## **\$ sh -v dsh1.sh 4 5**

Use -v option to debug complex shell script.

---

[Prev](#)

The case Statement

[Home](#)

[Up](#)

[Next](#)

Advanced Shell Scripting

# Introduction

Linux contains powerful utility programs. You can use these utility to

- Locate system information
- For better file management
- To organize your data
- System administration etc

Following section introduce you to some of the essential utilities as well as expression. While programming shell you need to use these essential utilities. Some of these utilities (especially sed & awk) requires understanding of expression. After the quick introduction to utilities, you will learn the expression.

# Prepering for Quick Tour of essential utilities

For this part of tutorial create *sname* and *smark* data files as follows (Using text editor of your choice)  
**Note** Each data block is separated from the other by TAB character i.e. while creating the file if you type 11 then press "tab" key, and then write Vivek (as shown in following files):

## sname

<u>Sr.No</u>	<u>Name</u>
11	Vivek
12	Renuka
13	Prakash
14	Ashish
15	Rani

## smark

<u>Sr.No</u>	<u>Mark</u>
11	67
12	55
13	96
14	36
15	67

# Selecting portion of a file using cut utility

Suppose from *sname* file you wish to print name of student on-screen, then from shell (Your command prompt i.e. \$) issue command as follows:

```
$cut -f2 sname
```

```
Vivek
```

```
Renuka
```

```
Prakash
```

```
Ashish
```

```
Rani
```

**cut** utility cuts out selected data from *sname* file. To select Sr.no. field from *sname* give command as follows:

```
$cut -f1 sname
```

```
11
```

```
12
```

```
13
```

```
14
```

```
15
```

Command	Explanation
<b>cut</b>	Name of cut utility
<b>-f1</b>	Using (-f) option, you are specifying the extraction field number. (In this example its 1 i.e. first field)
<b>sname</b>	File which is used by cut utility and which is use as input for cut utility.

You can redirect output of cut utility as follows

```
$cut -f2 sname > /tmp/sn.tmp.$$
```

```
$cut -f2 smark > /tmp/sm.tmp.$$
```

```
$cat /tmp/sn.tmp.$$
```

```
Vivek
```

```
Renuka
```

```
Prakash
```

```
Ashish
```

```
Rani
```

```
$cat /tmp/sm.tmp.$$
```

```
67
```

```
55
```

```
96
```

```
36
```

```
67
```

## General Syntax of cut utility:

### *Syntax:*

```
cut -f{field number} {file-name}
```

### *Use of Cut utility:*

Selecting portion of a file.

---

[Prev](#)

Preparing for Quick Tour of essential utilities

[Home](#)

[Up](#)

[Next](#)

Putting lines together using paste utility

# Putting lines together using paste utility

Now enter following command at shell prompt

```
$ paste sname smark
```

```
11 Vivek 11 67
12 Renuka 12 55
13 Prakash 13 96
14 Ashish 14 36
15 Rani 15 67
```

Paste utility join *textual information together*. To clear your idea try following command at shell prompt:

```
$ paste /tmp/sn.tmp.$$ /tmp/sm.tmp.$$
```

```
Vivek 67
Renuka 55
Prakash 96
Ashish 36
Rani 67
```

Paste utility is useful to put textual information together located in various files.

General Syntax of paste utility:

*Syntax:*

```
paste {file1} {file2}
```

*Use of paste utility:*

Putting lines together.

Can you note down basic difference between cut and paste utility?

# The join utility

Now enter following command at shell prompt:

**\$join sname smark**

```
11 Vivek 67
12 Renuka 55
13 Prakash 96
14 Ashish 36
15 Rani 67
```

Here students names are matched with their appropriate marks. How ? join utility uses the Sr.No. field to join to files. Notice that Sr.No. is the first field in both sname and smark file.

General Syntax of join utility:

*Syntax:*

```
join {file1} {file2}
```

*Use of join utility:*

The join utility joins, lines from separate files.

**Note** that join will only work, if there is *common field in both file and if values are identical to each other.*



# Translateing range of characters using tr utility

Type the following command at shell prompt:

```
$ tr "h2" "3x" < sname
```

```
11 Vivek
```

```
1x Renuka
```

```
13 Prakas3
```

```
14 As3is3
```

```
15 Rani
```

You can clearly see that each occurrence of character 'h' is replace with '3' and '2' with 'x'. tr utility translate specific characters into other specific characters or range of characters into other ranges.

```
h -> 3
```

```
2 -> x
```

Consider following example: (after executing command type text in lower case)

```
$ tr "[a-z]" "[A-Z]"
```

```
hi i am Vivek
```

```
HI I AM VIVEK
```

```
what a magic
```

```
WHAT A MAGIC
```

{Press CTRL + C to terminate.}

Here tr translate range of characters (i.e. small a to z) into other (i.e. to Capital A to Z) ranges.

General Syntax & use of tr utility:

*Syntax:*

```
tr {pattern-1} {pattern-2}
```

*Use of tr utility:*

To translate range of characters into other range of characters.

After typing following paragraph, I came to know my mistake that entire paragraph must be in lowercase characters, how to correct this mistake? (Hint - Use tr utility)

```
$ cat > lcommunity.txt
```

```
THIS IS SAMPLE PARAGRAPH  
WRITTEN FOR LINUX COMMUNITY,  
BY VIVEK G GITE (WHO ELSE?)  
OKAY THAT IS OLD STORY.
```

[Prev](#)

The join utility

[Home](#)

[Up](#)

[Next](#)

Data manipulation using awk utility

# Data manipulation using awk utility

Before learning more about awk create data file using any text editor or simply vi:

## inventory

```
egg    order 4
cacke  good 10
cheese okay 4
pen    good 12
floppy good 5
```

After crating file issue command

```
$ awk '/good/ { print $3 }' inventory
```

```
10
12
5
```

awk utility, select each record from file containing the word "good" and performs the action of printing the third field (Quantity of available goods.). Now try the following and note down its output.

```
$ awk '/good/ { print $1 " " $3 }' inventory
```

General Syntax of awk utility:

*Syntax:*

```
awk 'pattern action' {file-name}
```

For `$ awk '/good/ { print $3 }' inventory` example,

<b>/good/</b>	Is the pattern used for selecting lines from file.
<b>{print \$3}</b>	This is the action; if pattern found, print on of such action. Here \$3 means third record in selected record. (What \$1 and \$2 mean?)
<b>inventory</b>	File which is used by awk utility which is use as input for awk utility.

*Use of awk utility:*

To manipulate data.

# sed utility - Editing file without using editor

For this part of tutorial create data file as follows

## teaormilk

India's milk is good.  
tea Red-Lable is good.  
tea is better than the coffee.

After creating file give command

```
$ sed '/tea/s//milk/g' teaormilk > /tmp/result.tmp.$$
```

```
$ cat /tmp/result.tmp.$$
```

```
India's milk is good.  
milk Red-Lable is good.  
milk is better than the coffee.
```

sed utility is used to find every occurrence of tea and replace it with word milk. sed - Steam line editor which uses 'ex' editors command for editing text files without starting ex. (Cool!, isn't it? no use of text editor to edit anything!!!)

<b>/tea/</b>	Find tea word or select all lines having the word tea
<b>s//milk/</b>	Replace (substitute) the word milk for the tea.
<b>g</b>	Make the changes globally.

*Syntax:*

```
sed {expression} {file}
```

*Use of sed utility:* sed is used to edit (text transformation) on given stream i.e a file or may be input from a pipeline.

# Removing duplicate lines using uniq utility

Create text file personame as follows:

## personame

```
Hello I am vivek
12333
12333
welcome
to
sai computer academy, a'bad.
what still I remeber that name.
oaky! how are u luser?
what still I remeber that name.
```

After creating file, issue following command at shell prompt

```
$ uniq personame
```

```
Hello I am vivek
12333
welcome
to
sai computer academy, a'bad.
what still I remeber that name.
oaky! how are u luser?
what still I remeber that name.
```

Above command prints those lines which are unique. For e.g. our original file contains 12333 twice, so additional copies of 12333 are deleted. But if you examine output of uniq, you will notice that 12333 is gone (Duplicate), and "what still I remeber that name" remains as its. Because the uniq utility compare only adjacent lines, duplicate lines must be next to each other in the file. To solve this problem you can use command as follows

```
$ sort personame | uniq
```

General Syntax of uniq utility:

*Syntax:*

```
uniq {file-name}
```

[Prev](#)  
sed utility - Editing file without using  
editor

[Home](#)  
[Up](#)

[Next](#)  
Finding matching pattern using grep  
utility

# Finding matching pattern using grep utility

Create text file as follows:

## demo-file

```
hello world!  
cartoons are good  
especially toon like tom (cat)  
what  
the number one song  
12221  
they love us  
I too
```

After saving file, issue following command,

```
$ grep "too" demofile
```

```
cartoons are good  
especially toon like tom (cat)  
I too
```

grep will locate all lines for the "too" pattern and print all (matched) such line on-screen. grep prints too, as well as cartoons and toon; because grep treat "too" as expression. Expression by grep is read as the letter **t** followed by **o** and so on. So if this expression is found any where on line its printed. grep don't understand words.

*Syntax:*

```
grep "word-to-find" {file-name}
```

# Introduction

In the chapter 5, "Quick Tour of essential utilities", you have seen basic utilities. If you use them with other tools, these utilities are very useful for data processing or for other works. In rest part of tutorial we will learn more about patterns, filters, expressions, and off course sed and awk in depth.

## Learning expressions with ex

What does "cat" mean to you ?

One its the word cat, (second cat is an animal! I know 'tom' cat), If same question is asked to computer (not computer but to grep utility) then grep will try to find all occurrence of "cat" word (remember grep read word "cat" as the **c** letter followed by **a** and followed by **t**) including cat, copycat, catalog etc.

Pattern defined as:

*"Set of characters (may be words or not) is called pattern."*

For e.g. "dog", "celeron", "mouse", "ship" etc are all example of pattern. Pattern can be change from one to another, for e.g. "ship" as "sheep".

Metacharacters defined as:

*"If patterns are identified using special characters then such special characters are known as metacharacters".*

expressions defined as:

*"Combination of pattern and metacharacters is known as expressions (regular expressions)."*

Regular expressions are used by different Linux utilities like

- grep
- awk
- sed

So you must know how to construct regular expression. In the next part of LSST you will learn how to construct regular expression using ex editor.

For this part of chapter/tutorial create '[demofile](#)' - text file using any text editor.



# Getting started with ex

You can start the ex editor by typing ex at shell prompt:

*Syntax:*

```
ex {file-name}
```

*Example:*

```
$ ex demofile
```

The **:** (colon) is ex prompt where you can type ex text editor command or regular expression. Its time to open our demofile, use ex as follows:

```
$ ex demofile
```

```
"demofile" [noeol] 20L, 387C
```

```
Entering Ex mode. Type "visual" to go to Normal mode.
```

```
:
```

As you can see, you will get **:** prompt, here you can type ex command, type **q** and press ENTER key to exit from ex as shown follows: (remember commands are case sensitive)

```
: q
```

```
vivek@ls vivek]$
```

After typing the **q** command you are exit to shell prompt.

# Printing text on-screen

First open the our demofile as follows:

```
$ ex demofile
```

```
"demofile" [noeol] 20L, 387C
```

```
Entering Ex mode. Type "visual" to go to Normal mode.
```

Now type 'p' in front of : as follow and press enter

```
:p
```

```
Okay! I will stop.
```

```
:
```

**NOTE** By default p command will print current line, in our case its the last line of above text file.

## Printing lines using range

Now if you want to print 1st line to next 5 line (i.e. 1 to 5 lines) then give command

```
:1,5 p
```

```
Hello World.
```

```
This is vivek from Poona.
```

```
I love linux.
```

```
It is different from all other Os
```

**NOTE** Here 1,5 is the address. if single number is used (e.g. 5 p) it indicate line number and if two numbers are separated by comma its range of line.

## Printing particular line

To print 2nd line from our file give command

```
:2 p
```

```
This is vivek from Poona.
```

## Printing entire file on-screen

Give command

```
:1,$ p
```

```
Hello World.
```

```
This is vivek from Poona.
```

```
I love linux.
```

```
It is different from all other Os
```

.....  
...  
.....

*Okay! I will stop.*

**NOTE** Here 1 is 1st line and \$ is the special character of ex which mean last-line character. So 1,\$ means print from 1st line to last-line character (i.e. end of file). Here p stands print.

## Printing line number with our text

Give command

**:set number**

**:1,3 p**

*1 Hello World.*

*2 This is vivek from Poona.*

*3*

**NOTE** This command prints number next to each line. If you don't want number you can turn off numbers by issuing following command

**:set nonumber**

**:1,3 p**

Hello World.

This is vivek from Poona.

---

[Prev](#)

Getting started with ex

[Home](#)

[Up](#)

[Next](#)

Deleting lines

# Deleting lines

Give command

```
:1, d
```

I love linux.

## NOTE

Here 1 is 1st line and d command indicates deletes (Which deletes the 1st line).

You can even delete range of line by giving command as

```
:1,5 d
```

---

# Copying lines

Give command as follows

```
:1,4 co $
```

```
:1,$ p
```

*I love linux.*

*It is different from all other Os*

....

.....

*. (DOT) is special command of linux.*

*Okay! I will stop.*

*I love linux.*

*It is different from all other Os*

*My brother Vikrant also loves linux.*

**NOTE** Here 1,4 means copy 1 to 4 lines; co command stands for copy; \$ is end of file. So it mean copy first four line to end of file. You can delete this line as follows

```
:18,21 d
```

```
:1,$ p
```

```
:1,$ p
```

*I love linux.*

*It is different from all other Os*

*My brother Vikrant also loves linux.*

*He currently lerarns linux.*

*Linux is coool.*

*Linux is now 10 years old.*

*Next year linux will be 11 year old.*

*Rani my sister never uses Linux*

*She only loves to play games and nothing else.*

*Do you know?*

*. (DOT) is special command of linux.*

*Okay! I will stop.*

---

[Prev](#)  
Deleting lines

[Home](#)  
[Up](#)

[Next](#)  
Searching the words

# Searching the words

(a) Give following command

```
:/linux/ p
```

*I love linux.*

**Note** In ex you can specify address (line) using number for various operation. This is useful if you know the line number in advance, but if you don't know line number, then you can use *contextual address* to print line on-screen. In above example */linux/* is *contextual address* which is constructed by surrounding a regular expression with two slashes. And *p* is print command of ex.

Try following and note down difference (Hint - Watch *p* is missing)

```
:/Linux/
```

(b) Give following command

```
:g/linux/ p
```

*I love linux.*

*My brother Vikrant also loves linux.*

*He currently lerarns linux.*

*Next year linux will be 11 year old.*

*. (DOT) is special command of linux.*

In previous example (*:/linux/ p*) only one line is printed. If you want to print all occurrence of the word "linux" then you have to use *g*, which mean global line address. This instruct ex to find all occurrence of pattern. Try following

```
:1,$ /Linux/ p
```

Which give the same result. It means *g* stands for *1,\$*.

## Saving the file in ex

Give command

```
:w
```

*"demofile" 20L, 386C written*

*w* command will save the file.

## Quitting the ex

Give command

```
:q
```

**q** command quits from ex and you are return to shell prompt.

**Note** use wq command to do save and exit from ex.

---

[Prev](#)

Coping lines

[Home](#)

[Up](#)

[Next](#)

Find and Replace (Substituting regular expression)



# Find and Replace (Substituting regular expression)

Give command as follows

```
:8 p
```

```
He currently lerarns linux.
```

```
:8 s/lerarns/learn/
```

```
:p
```

```
He currently learn linux.
```

**Note** Using above command, you are substituting the word "learn" for the word "lerarns".

Above command can be explained as follows:

Command	Explanation
<b>8</b>	Goto line 8, address of line.
<b>s</b>	Substitute
<b>/lerarns/</b>	Target pattern
<b>learn/</b>	If target pattern found substitute the expression (i.e. <b>learn/</b> )

Considered the following command:

```
:1,$ s/Linux/Unix/
```

```
Rani my sister never uses Unix
```

```
:1,$ p
```

```
Hello World.
```

```
This is vivek from Poona.
```

```
....
```

```
..
```

```
.....
```

*. (DOT) is special command of linux.*

*Okay! I will stop.*

Using above command, you are substituting all lines i.e. s command will find all of the address line for the pattern "Linux" and if pattern "Linux" found substitute pattern "Unix".

Command	Explanation
<b>:1,\$</b>	Substitute for all line
<b>s</b>	Substitute

<b>/Linux/</b>	Target pattern
<b>Unix/</b>	If target pattern found substitute the expression (i.e. <b>Unix/</b> )

Even you can also use **contextual address** as follows

```
:/sister/ p  
Rani my sister never uses Unix  
:g /sister/ s/never/always/  
:p  
Rani my sister always uses Unix
```

Above command will first find the line containing pattern "sister" if found then it will substitute the pattern "always" for the pattern "never" (It mean find the line containing the word sister, on that line find the word never and replace it with word always.)

Try the following and watch the output very carefully.

```
:g /Unix/ s/Unix/Linux  
3 substitutions on 3 lines
```

Above command finds all line containing the regular expression "Unix", then substitute "Linux" for all occurrences of "Unix". Note that above command can be also written as follows

```
:g /Unix/ s//Linux
```

Here // is replace by the last pattern/regular expression i.e. Unix. Its shortcut. Now try the following

```
:g /Linux/ s//UNIX/  
3 substitutions on 3 lines  
:g/Linux/p  
Linux is coool.  
Linux is now 10 years old.  
Rani my sister always uses Linux
```

```
:g /Linux/ s//UNIX/  
3 substitutions on 3 lines  
:g/UNIX/p  
UNIX is coool.  
UNIX is now 10 years old.  
Rani my sister always uses UNIX
```

By default substitute command only substitute first occurrence of a pattern on a line. Let's take another example, give command

```
:/brother/p  
My brother Vikrant also loves linux who also loves unix.
```

Now in above line "also" word is occurred twice, give the following substitute command

```
:g/brother/ s/also/XYZ/  
:/brother/p  
My brother Vikrant XYZ loves linux who also loves unix.
```

Make sure next time it works

**:g/brother/ s/XYZ/also/**

Note that "also" is only once substituted. If you want to **s** command to work with all occurrences of pattern within a address line give command as follows:

**:g/brother/ s/also/XYZ/g**

**:p**

*My brother Vikrant XYZ loves linux who XYZ loves unix.*

**:g/brother/ s/XYZ/also/g**

**:p**

*My brother Vikrant also loves linux who also loves unix.*

The **g** option at the end instruct **s** command to perform replacement on all occurrences of the target pattern within a address line.

---

[Prev](#)

[Home](#)

[Next](#)

Searching the words

[Up](#)

Replacing word with confirmation from user

# Replacing word with confirmation from user

Give command as follows

**:g/Linux/ s//UNIX/gc**

After giving this command ex will ask you question like - *replace with UNIX (y/n/a/q/^E/^Y)?*  
Type **y** to replace the word or **n** to not replace or **a** to replace all occurrence of word.

---

# Finding words

Command like

```
:g/the/p
```

*It is different from all other Os*

*My brother Vikrant also loves linux who also loves unix.*

Will find word like theater, the, brother, other etc. What if you want to just find the word like "the" ? To find the word (Let's say Linux) you can give command like

```
:/^<Linux\>
```

*Linux is coool.*

```
:g/^<Linux\>/p
```

*Linux is coool.*

*Linux is now 10 years old.*

*Rani my sister never uses Linux*

The symbol \< and \> respectively match the empty string at the beginning and end of the word. To find the line which contain Linux pattern at the beginning give command

```
:/^Linux
```

*Linux is coool.*

As you know \$ is end of line character, the ^ (caret) match beginning of line. To find all occurrence of pattern "Linux" at the beginning of line give command

```
:g/^Linux
```

*Linux is coool.*

*Linux is now 10 years old.*

And if you want to find "Linux" at the end of line then give command

```
:/Linux $
```

*Rani my sister never uses Linux*

Following command will find empty line:

```
:/^$
```

To find all blank line give command:

```
:g/^$
```

To view entire file without blank line you can use command as follows:

```
:g/[^\^$]
```

*Hello World.*

*This is vivek from Poona.*

*I love linux.*

*It is different from all other Os*

*My brother Vikrant also loves linux who also loves unix.  
He currently learn linux.  
Linux is coool.  
Linux is now 10 years old.  
Next year linux will be 11 year old.  
Rani my sister never uses Linux  
She only loves to play games and nothing else.  
Do you know?  
. (DOT) is special command of linux.  
Okay! I will stop.*

Command	Explanation
<b>g</b>	All occurrence
<b>/[^</b>	<b>[^]</b> This means not
<b>/^\$</b>	Empty line, Combination of ^ and \$.

To delete all blank line you can give command as follows

**:g/^\$/d**  
*Okay! I will stop.*

**:1,\$ p**  
*Hello World.  
This is vivek from Poona.  
I love linux.  
It is different from all other Os  
My brother Vikrant also loves linux who also loves unix.  
He currently learn linux.  
Linux is coool.  
Linux is now 10 years old.  
Next year linux will be 11 year old.  
Rani my sister never uses Linux  
She only loves to play games and nothing else.  
Do you know?  
. (DOT) is special command of linux.  
Okay! I will stop.*

Try u command to undo, to undo what you have done it, give it as follows:

**:u**  
**:1,\$ p**  
*Hello World.  
This is vivek from Poona.  
....  
...  
....  
Okay! I will stop.*

[Prev](#)

Replacing word with confirmation from user

[Home](#)

[Up](#)

[Next](#)

Using range of characters in regular expressions

# Using range of characters in regular expressions

Try the following command

```
:g/Linux/p
```

*Linux is coool.*

*Linux is now 10 years old.*

*Rani my sister never uses Linux*

This will find only "Linux" and not the "linux", to overcome this problem try as follows

```
:g/[Ll]inux/p
```

*I love linux.*

*My brother Vikrant also loves linux who also loves unix.*

*He currently learn linux.*

*Linux is coool.*

*Linux is now 10 years old.*

*Next year linux will be 11 year old.*

*Rani my sister never uses Linux*

*. (DOT) is special command of linux.*

Here a list of characters enclosed by [ and ], which matches any single character in that range. if the first character of list is ^, then it matches any character not in the list. In above example [Ll], will try to match L or l with rest of pattern. Let's see another example. Suppose you want to match single digit character in range you can give command as follows

```
:/[0123456789]
```

Even you can try it as follows

```
:g/[0-9]
```

*Linux is now 10 years old.*

*Next year linux will be 11 year old.*

Here range of digit is specified by giving first digit (0-zero) and last digit (1), separated by hyphen. You can try [a-z] for lowercase character, [A-Z] for uppercase character. Not just this, there are certain named classes of characters which are predefined. They are as follows:

Predefined classes of characters	Meaning
<b>[:alnum:]</b>	Letters and Digits (A to Z or a to z or 0 to 9)
<b>[:alpha:]</b>	Letters A to Z or a to z
<b>[:cntrl:]</b>	Delete character or ordinary control character (0x7F or 0x00 to 0x1F)



<b>[[:digit:]]</b>	Digit (0 to 9)
<b>[[:graph:]]</b>	Printing character, like print, except that a space character is excluded
<b>[[:lower:]]</b>	Lowercase letter (a to z)
<b>[[:print:]]</b>	Printing character (0x20 to 0x7E)
<b>[[:punct:]]</b>	Punctuation character (ctrl or space)
<b>[[:space:]]</b>	Space, tab, carriage return, new line, vertical tab, or form feed (0x09 to 0x0D, 0x20)
<b>[[:upper:]]</b>	Uppercase letter (A to Z)
<b>[[:xdigit:]]</b>	Hexadecimal digit (0 to 9, A to F, a to f)

For e.g. To find digit or alphabet (Upper as well as lower) you will write  
**:/[0-9A-Za-Z]**

Instead of writing such command you could easily use predefined classes or range as follows  
**:/[[:alnum:]]**

**The . (dot) matches any single character.**

For e.g. Type following command

**:g/\<.o\>**

*She only loves to play games and nothing else.*

*Do you know?*

This will include lo(ves), Do, no(thing) etc.

**\* Matches the zero or more times**

For e.g. Type following command

**:g/L\***

*Hello World.*

*This is vivek from Poona.*

....

....

**:g/Li\***

*Linux is coool.*

*Linux is now 10 years old.*

*Rani my sister never uses Linux*

**:g/c.\*and**

*. (DOT) is special command of linux.*

Here first **c** character is matched, then any single character (.) followed by n number of single character (1 or 100 times even) and finally ends with and. This can found different word as follows command or catand etc.

In the regular expression metacharacters such as . (DOT) or \* loose their special meaning if we use as \  
or \\*. The backslash removes the special meaning of such meatcharacters and you can use them as  
ordinary characters. For e.g. If u want to search . (DOT) character at the beginning of line, then you can't

use command as follows

**:g/^.**

*Hello World.*

*This is vivek from Poona.*

....

..

...

*. (DOT) is special command of linux.*

*Okay! I will stop.*

Instead of that use

**:g/^.**

*. (DOT) is special command of linux.*

---

[Prev](#)

Finding words

[Home](#)

[Up](#)

[Next](#)

Using & as Special replacement character

# Using & as Special replacement character

Try the following command:

```
:1,$ s/Linux/&-Unix/p
```

*3 substitutions on 3 lines*

*Rani my sister never uses Linux-Unix*

```
:g/Linux-Unix/p
```

*Linux-Unix is coool.*

*Linux-Unix is now 10 years old.*

*Rani my sister never uses Linux-Unix*

This command will replace, target pattern "Linux" with "Linux-Unix". & before - Unix means use "last pattern found" with given pattern, So here last pattern found is "Linux" which is used with given -Unix pattern (Finally constructing "Linux-Unix" substitute for "Linux").

Can you guess the output of this command?

```
:1,$ s/Linux-Unix/&Linux/p
```

# Converting lowercase character to uppercase

Try the following command

```
:1,$ s/[a-z]/^u &/g
```

Above command can be explained as follows:

Command	Explanation
<b>1,\$</b>	Line Address location is all i.e. find all lines for following pattern
<b>s</b>	Substitute command
<b>/[a-z]/</b>	Find all lowercase letter - Target
<b>^u&amp;/</b>	Substitute to Uppercase. <b>^u&amp;</b> means substitute last patter (&) matched with its UPPERCASE replacement ( <b>^u</b> ) <u>Note</u> : Use <b>^l</b> (small L) for lowercase character.
<b>g</b>	Global replacement

Can you guess the output of following command?

```
:1,$ s/[A-Z]/^l&/g
```

**Congratulation**, for successfully completion of this tutorial of regular expressions.

I hope so you have learn lot from this. To master the expression you have to do lot of practice. This tutorial is very important to continue with rest of tutorial and to become power user of Linux. Impress your friends with such expressions. Can you guess what last expression do?

```
:1,$ s/^_ _*$//
```

Note : **\_ \_** indicates two black space.