# Introduction

This tutorial is designed for beginners who wish to learn the basics of shell scripting/programming plus introduction to power tools such as awk, sed, etc. It is not help or manual for the shell; while reading this tutorial you can find manual quite useful (type man bash at $ prompt to see manual pages). Manual contains all necessary information you need, but it won't have that much examples, which makes idea more clear. For this reason, this tutorial contains examples rather than all the features of shell.

# Audience for this tutorial

I assumes you have at least working knowledge of Linux i.e. basic commands like how to create, copy, remove files/directories etc or how to use editor like vi or mcedit and login to your system. But not expects any programming language experience. If you have access to Linux, this tutorial will provide you an easy-to-follow introduction to shell scripting.

# What's different about this tutorial

Many other tutorial and books on Linux shell scripting are either too basic, or skips important intermediate steps. But this tutorial, maintained the balance between these two. It covers the many real life modern example of shell scripting which are almost missed by many other tutorials/documents/books. I have used a hands-on approach in this tutorial. The idea is very clear "*do it yourself or learn by doing*" i.e. trying things yourself is the best way to learn, so examples are presented as complete working shell scripts, which can be typed in and executed

# Chapter Organization

Chapter 1 to 4 shows most of the useful and important shell scripting concepts. Chapter 5 introduction to tools & utilities which can be used while programming the Linux shell smartly. Chapter 6 and 7 is all about expression and expression mostly used by tools such as sed and awk. Chapter 8 is loaded with tons of shell scripting examples divided into different categories. Chapter 9 gives more resources information which can be used while learning the shell scripting like information on Linux file system, common Linux command reference and other resources.

Chapter 1 introduces to basic concepts such as what is Linux, where Linux can used and continue explaning the shell, shell script and kernel etc.

Chapter 2 shows how to write the shell script and execute them. It explains many basic concepts which requires to write shell script.

Chapter 3 is all about making decision in shell scripting as well as loops in shell. It explains what expression are, how shell understands the condition/decisions. It also shows you nesting concept for if and for loop statement and debugging of shell script.

Chapter 4 introduces the many advanced shell scripting concepts such as function, user interface, File Descriptors, signal handling, Multiple command line arguments etc.

Chapter 5 introduces to powerful utility programs which can be used variety of purpose while programming the shell.

Chapter 6 and 7 gives more information on patterns, filters, expressions, and off course sed and awk is covered in depth.

Chapter 8 contains lot of example of shell scripting divided into various category such as logic development, system administration etc.

Note that □ indicates advanced shell scripting concepts, you can skip this if you are really new to Linux or Programming, though this is not RECOMMENDED by me.

I hope you get as much pleasure reading this tutorial, as I had writing it. After reading this tutorial if you are able to write your own powerful shell scripts, then I think the purpose of writing this tutorial is served and finally if you do get time after reading this tutorial drop me an e-mail message about your comment/suggestion/questions and off course bugs (errors) you find regarding this tutorial.

---

# What Linux is?

- Free
- Unix Like
- Open Source
- Network operating system

# Who developed the Linux?

In 1991, Linus Torvalds studying Unix at the University, where he used special educational experimental purpose operating system called Minix (small version of Unix and used in Academic environment). But Minix had it's own limitations. Linus felt he could do better than the Minix. So he developed his own version of Minix, which is now know as Linux. Linux is Open Source From the start of the day. For more information on Linus Torvalds, please visit his home page.

# How to get Linux?

Linux available for download over the net, this is useful if your internet connection is fast. Another way is order the CD-ROMs which saves time, and the installation from CD-ROM is fast/automatic. Various Linux distributions available. Following are important Linux distributions.

| Linux distributions. | Website/Logo |
|---|---|
| Red Hat Linux: http://www.redhat.com/ |  |
| SuSE Linux: http://www.suse.com/ |  |
| Mandrake Linux: http://www.mandrakesoft.com/ |  |
| Caldera Linux: http://www.calderasystems.com/ |  |
| Debian GNU/Linux: http://www.debian.org/ |  |
| Slackware Linux: http://www.slackware.com/ |  |

**Note:** If you are in India then you can get Linux Distribution from the Leading Computer magazine such as PC Quest (Even PCQuest has got its own Linux flavour) or if you are in Pune, India please visit our web site to obtained the Red Hat Linux or any other official Linux distribution. Note that you can also obtained your Linux distribution with Linux books which you purchase from local book store.

Prev
Who developed the Linux?

Home
Up

Next
How to Install Linux

# How to Install Linux ?

Please visit the LESSBS Project home page for Quick Visual Installation Guide for Red Hat Linux version 6.2 and 7.2.

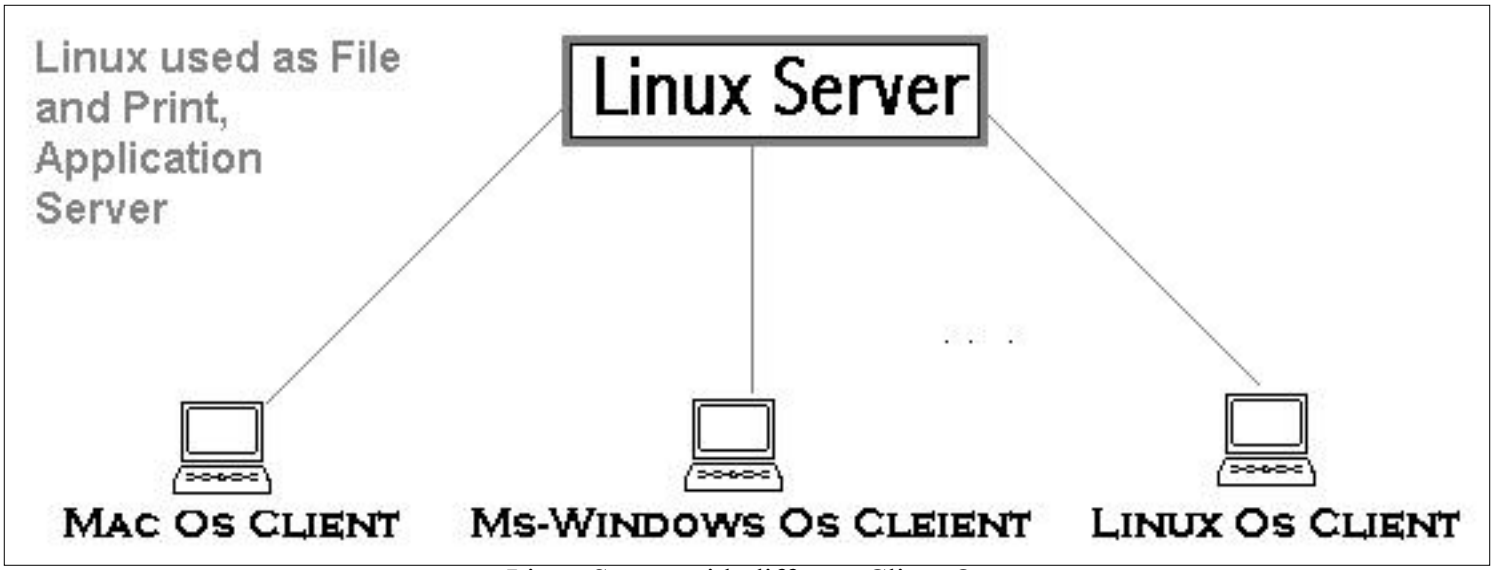
# Where I can use Linux?

You can use Linux as Server Os or as stand alone Os on your PC. (But it is best suited for Server.) As a server Os it provides different services/network resources to client. Server Os must be:

- Stable
- Robust
- Secure
- High Performance

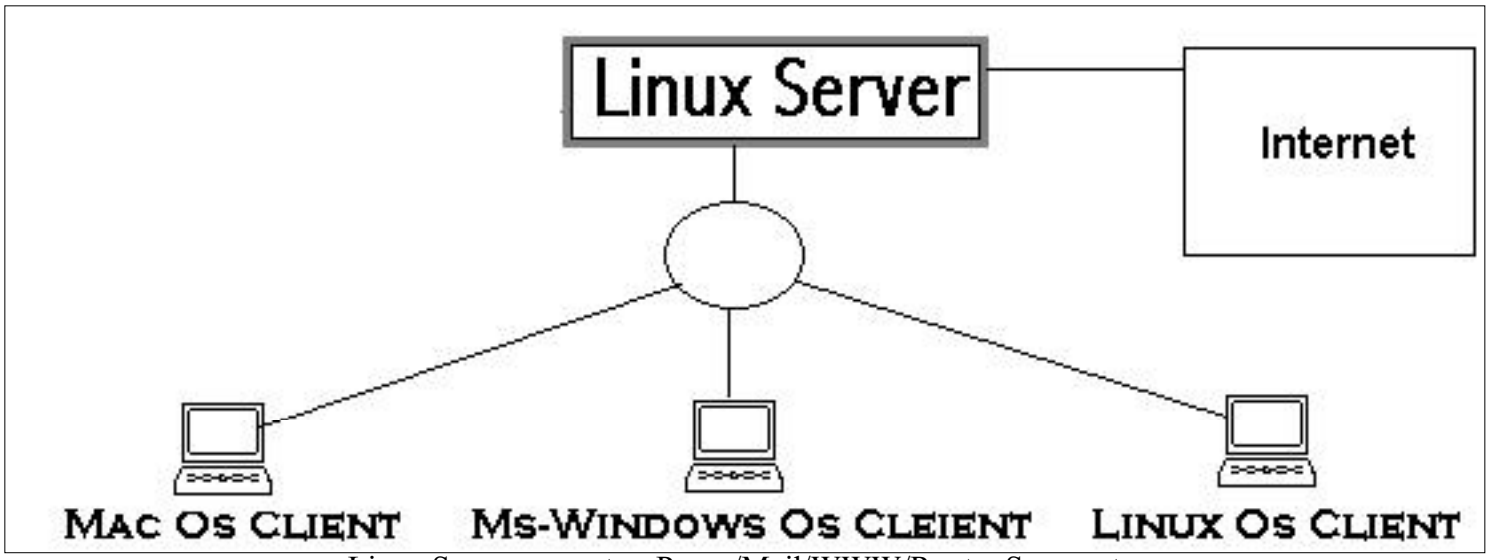
Linux offers all of the above characteristics plus its Open Source and Free OS. So Linux can be used as:

(1) On *stand alone workstation*/PC for word processing, graphics, software development, internet, e-mail, chatting, small personal database management system etc.
(2) In *network environment* as:
(A) *File and Print or Application* Server
Share the data, Connect the expensive device like printer and share it, e-mail within the LAN/intranet etc are some of the application.




Linux Server with different Client Os

(B) Linux sever can be connected to Internet, So that PC's on intranet can share the internet/e-mail etc. You can put your web sever that run your web site or transmit the information on the internet.




Linux Server can act as Proxy/Mail/WWW/Router Server etc.

So you can use Linux for:

- Personal Work
- Web Server
- Software Development Workstation
- Workgroup Server
- In Data Center for various server activities such as FTP, Telnet, SSH, Web, Mail, Proxy, Proxy Cache Appliance etc

See the LESSBS project for more information on Linux Essential Services (as mentioned above) and how to implement them in easy manner for you or your organization.

---

Prev
How to Install Linux

Home
Up

Next
What Kernel Is?

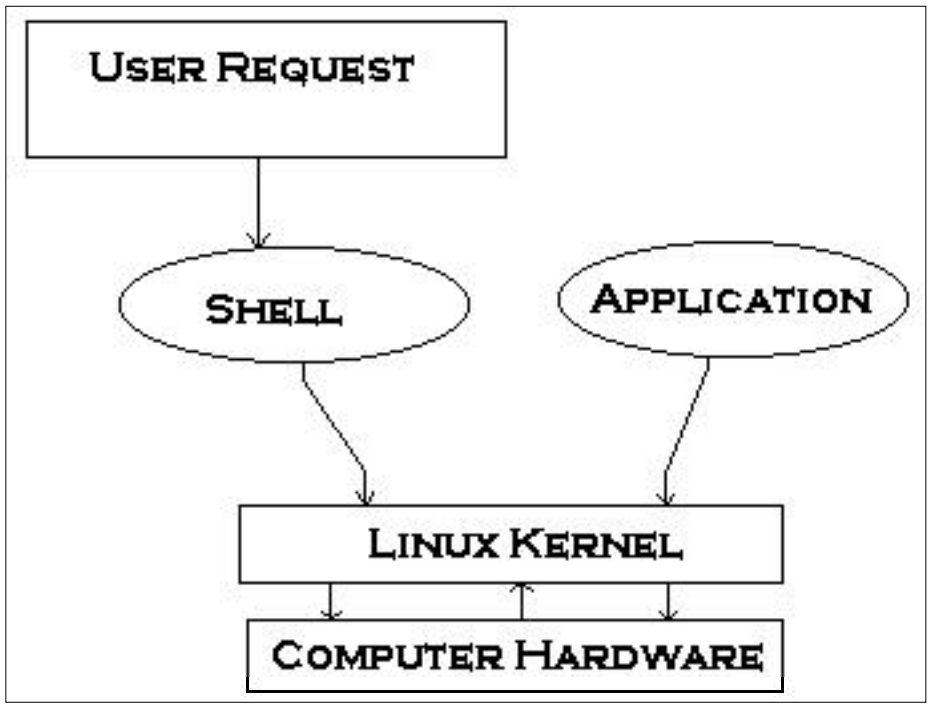
# What Kernel Is?

Kernel is heart of Linux Os.

It manages resource of Linux Os. Resources means facilities available in Linux. For e.g. Facility to store data, print data on printer, memory, file management etc .

Kernel decides who will use this resource, for how long and when. It runs your programs (or set up to execute binary files).

The kernel acts as an intermediary between the computer hardware and various programs/application/shell.




It's Memory resident portion of Linux. It performance following task :-

- I/O management
- Process management
- Device management
- File management
- Memory management

# What is Linux Shell ?

Computer understand the language of 0's and 1's called binary language.

In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in Os there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if its a valid command, it is passed to kernel.

Shell is a user program or it's a environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Several shell available with Linux including:

| Shell Name | Developed by | Where | Remark |
|---|---|---|---|
| BASH ( Bourne-Again SHell ) | Brian Fox and Chet Ramey | Free Software Foundation | Most common shell in Linux. It's Freeware shell. |
| CSH (C SHell) | Bill Joy | University of California (For BSD) | The C shell's syntax and usage are very similar to the C programming language. |
| KSH (Korn SHell) | David Korn | AT & T Bell Labs | -- |
| TCSH | See the man page. Type $ man tcsh | -- | TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH). |

***Tip:*** To find all available shells in your system type following command:
**$ cat /etc/shells**

***Note*** that each shell does the same job, but each understand a different command syntax and provides different built-in functions.

In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are!

Any of the above shell reads command from user (via Keyboard or Mouse) and tells Linux Os what users want. If we are giving commands from keyboard it is called command line interface ( Usually in-front of $ prompt. This prompt is depend upon your shell and Environment that you set or by your System Administrator, therefore you may get different prompt ).

***Tip:*** To find your current shell type following command
**$ echo $SHELL**

---

Prev
What Kernel Is?

Home
Up

Next
How to use Shell

# How to use Shell

To use shell (You start to use your shell as soon as you log into your system) you have to simply type commands.

See common Linux Command for syntax and example, this can be used as quick reference while programming the shell.

# What is Shell Script ?

Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands) , the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is know as ***shell script***.

Shell script defined as:
"*Shell Script is **series of command** written **in plain text file**. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file.*"

# Why to Write Shell Script ?

- Shell script can take input from user, file and output them on screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day today life.
- System Administration part can be also automated.

# Which Shell We are using to write Shell Script ?

In this tutorial we are using bash shell.

# Objective of this Tutorial (LSST v.1.5)

Try to understand Linux Os
Try to understand the basics of Linux shell
Try to learn the Linux shell programming

# What I need to learn this Tutorial (LSST v.1.5)

Linux OS ( I have used Red Hat Linux distribution Version 6.x+)
Web Browse to read tutorial. (IE or Netscape) For PDF version you need PDF reader.
Linux - bash shell. (Available with almost all Linux Distributions. By default bash is default shell for Red Hat Linux Distribution). All the scripts are also tested on Red Hat Linux version 7.2.

# Getting started with Shell Programming

In this part of tutorial you are introduced to shell programming, how to write script, execute them etc. We will be getting started with writing small shell script, that will print "Knowledge is Power" on screen. Before starting with this you should know

- How to use text editor such as vi, see the common vi command for more information.
- Basic command navigation

# 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 |
| #<br># My first shell script<br># | # followed by any text is considered as comment. Comment gives more information about script, logical explanation about shell script.<br>*Syntax:*<br># 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<br>*syntax:*<br>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 it's 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.

# 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 enclose 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.

---

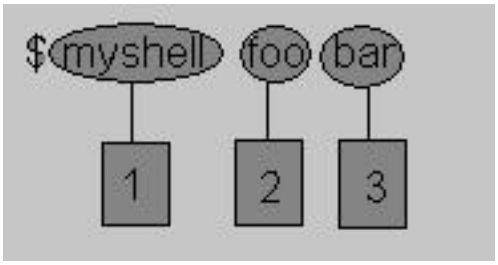
Prev
More commands on one command line
Home
Up
Next
Why Command Line arguments required

# 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 tail 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.

Lets take ls command
**$ Ls -a /***
This command has 2 command line argument -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) | | | | |
|---|---|---|---|---|---|---|
| *$0* | *$#* | *$1* | *$2* | *$3* | *$4* | *$5* |
| 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 $#"
echo "$0 is script name"
echo "$1 is first argument"
echo "$2 is second argument"
echo "All of them are :- $* or $@"
```

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
Command Line Processing

Home
Up

Next
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 sname 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

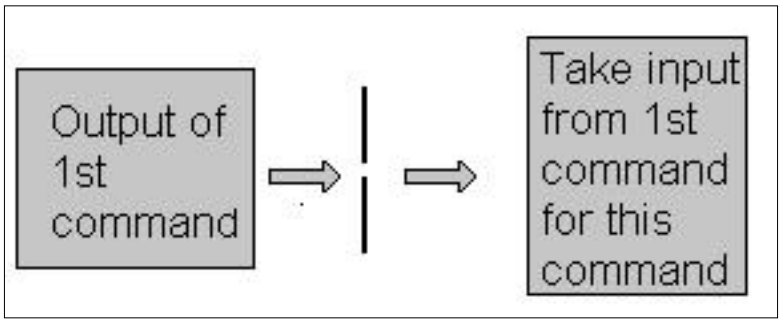
tr command is used to translate all lower case characters to upper-case letters. It take input from sname file, and tr's output is redirected to cap_names file.

**Future Point :** Try following command and find out most important point:
**$ sort > new_sorted_names < sname**
**$ cat new_sorted_names**

---

# 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

*Examles:*

| Command using Pipes | Meaning or Use of Pipes |
|---|---|
| **$ ls \| more** | Output of ls command is given as input to more command So that output is printed one screen full page at a time. |
| **$ who \| sort** | Output of who command is given as input to sort command So that it will print sorted list of users |
| **$ who \| sort > user_list** | Same as above except output of sort is send to (redirected) user_list file |
| **$ who \| wc -l** | Output of who command is given as input to wc command So that it will print number of user who logon to system |
| **$ ls -l \| wc -l** | 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<br>**$ ps ax \| grep httpd** |
| To see currently running processes and other information like memory and CPU usage with real time updates. | top<br>See the output of top command. | **$ top**<br>**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 equivalnt 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.

---

# 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.

Expreession 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*

**$ isnump_n**
*./ispos_n : You must give/supply one integers*

**$ isnump_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 **$ isnump_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 of 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* constuct 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 *if*s 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 condtion,condtion1,condtion2 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 "Opps! $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
Opps! 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 "*Opps! 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 equel 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**
**$ ./nestefor.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 varible **j** taking values from 1 to 5. The inner for loop terminates when the value of **j** exceeds 5, and the outer loop terminets when the value of **i** exceeds 5.

Following script is quite intresting, 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 produec the output as follows:


Above shell script cab be explained as follows:

| Command(s)/Statements | Explanation |
|---|---|
| for (( i = 1; i <= 9; i++ ))<br>do | Begin the outer loop which runs 9 times., and the outer loop terminets when the value of **i** exceeds 9 |
| for (( j = 1 ; j <= 9; j++ ))<br>do | Begins the inner loop, for each value of **i** the inner loop is cycled through 9 times, with the varible **j** taking values from 1 to 9. The inner for loop terminates when the value of **j** exceeds 9. |
| tot=`expr $i + $j`<br>tmp=`expr $tot % 2` | See for even and odd number positions using these statements. |
| if [ $tmp -eq 0 ]; then<br>echo -e -n "\033[47m "<br>else<br>echo -e -n "\033[40m "<br>fi | If even number posiotion print the white colour block (using **echo -e -n "\033[47m "** statement); otherwise for odd postion print the black colour box (using **echo -e -n "\033[40m "** statement). This statements are responsible to print entier chess board on screen with alternet 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 terminted by printing the chess board. |

***Exercise***

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

---

Prev Home Next
Loops in Shell Scripts Up 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<br>7 * 1 = 7<br>7 * 2 = 14<br>....<br>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)<br>**Caution:** 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<br>7 * 1 = 7<br>...<br>...<br>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. |