Shell Scripting

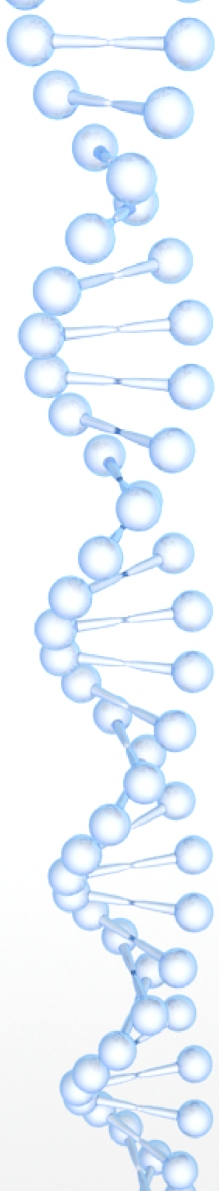
GnuGroup India

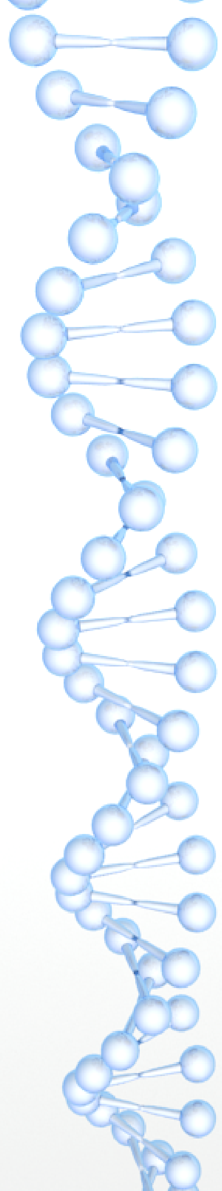
Discovering infinite possibilities

Insight GNU/Linux Group

Reinventing the way you
Think,
Learn,
Work

Email:info@gnugroup.org





Why Shell Scripting ?

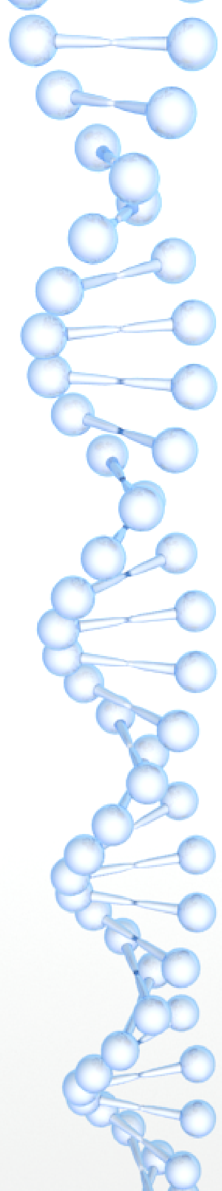
A shell script is computer program designed to be run by unix / linux shell, a command interpreter.

A shell script is a quick-and-dirty method of prototyping a complex application.

- Writing a shell script is much quicker than writing the equivalent code in other programming languages.

- It’s for automating the routine tasks.

Reference - Absolute bash scripting guide.by Mendel Cooper
website:http://www.tldp.org



When not to shell program

When your applications are way to complex.

When you link it with various libraries.

When you have heavy duty processing.

When you have floating point operations.

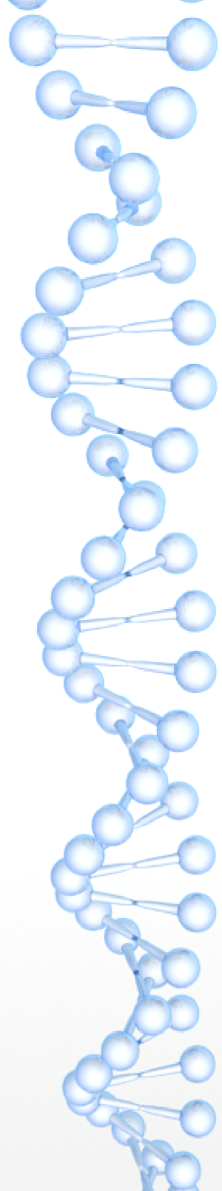
When execution speed of the programs matters.

What is required to learn shell scripting?

A working unix / linux system

A terminal, with bash shell

Knowledge of using editors like vim, nano, gedit



Developing habits to become a good programmer

Reading and Observing skills

Attitude of problem solving

Always looking for problem to be solved

Gets inspired from good programmers around...

Learns new programming languages...

for beginners - Linux and Shell scripting is a good start

Develop proficiency in Unix/Linux like OS.

Use frustratration, but uses it as fuel for solving problems.

Practice is the key....practice....practice..Spend 25 mins a day for coding at least....

How to write shell script

(1) Use any editor like vi or vim or nano 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).

Execute your script

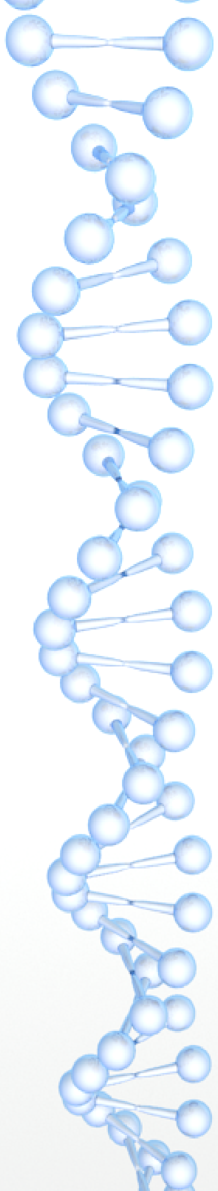
(3) Execute your script as

```
$ bash your-script-name
OR
$ sh your-script-name          (deprecated)
OR
$ ./script-name    ( After making it executable)
( if executing from your current directory.)
```

Create a file using any editor – vim or emacs or gedit

```
#!/bin/bash
## Script to print user information who is currently logged on , 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
```

# = comments



Bash shell has two categories of variable:

***System defined variables* (SDV)-**

Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.

***User defined variables* (UDV) -**

Created and maintained by user. This type of variable defined in lower letters.

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

USER , PATH , HOME

_myvariable , greetings, greet123

(2) Don't put spaces on either side of the equal sign when assigning value to variable.

For e.g.

In following variable declaration acceptable:-

```
x=10
```

Below variable declaration are not acceptable:

```
x =112
x= 12
x = 24
```

Spaces......no no

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

For e.g.

```
xo=10
Xo=11
XO=20
xO=2
```

Above all are different variable name, so to print value 20 we have to use $ echo $XO and not any of the following

```
echo $xo        # will print 10 but not 20
echo $Xo        # will print 11 but not 20
echo $xO        # will print 2 but not 20
```

(4) You can define NULL variable as follows:

Note: NULL variable is variable which has no value at the time of definition.

For e.g.

```
$ vehicle=

$ vehicle=""
```

Try to print it's value by issuing following command

```
$ echo $vehicle
```

No output will appear on stdout as variable has no value i.e. it's NULL variable.

Printing variables

To access UDV use following syntax, prefix a $ sign before the variable name.

Syntax:

```
$variablename
```

Define variable vehicle and x as follows:

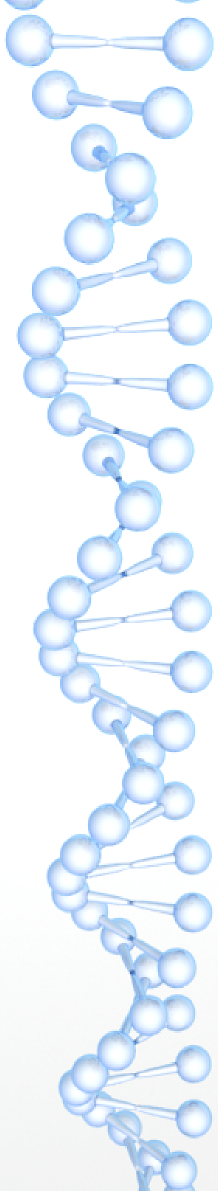
```
vehicle=car
x=10
```

To print contains of variable 'vehicle' type

```
echo $vehicle                          #     It will print 'car’
```

To print contents of variable 'x' type command as follows:

```
echo $x
```



Shell Arithmetic

Use to perform arithmetic operations.

Syntax:

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

Examples:

Shell prompt

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

Or

```
echo $((1+3))
echo $((2-1))
echo $((10*3))
echo  $((10/2))
echo $((20%3))
 echo $((2**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.

Set the variables on the shell prompt and try these on shell prompt:

```
x=2
y=2

echo  $[x+y]

echo  $[x-y]

echo  $[x*y]

echo  $[x/y]

echo  $[x%y]

echo  $[x**y]
```

For the last statement note 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
$ echo `expr 6 + 3`        # It will print 9
$ n=6/3
$ echo $n
6/3
```

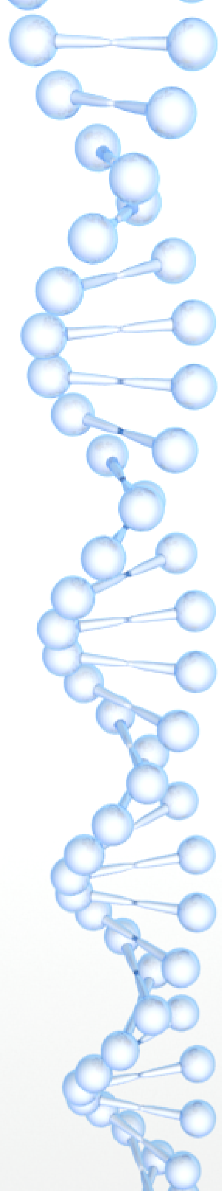
declare

```
$ declare -i n
$ n=6/3
$ echo $n
2
$ z=5
$ z=`expr $z+1`      ---- Need spaces around + sign.
$ echo $z
5+1

$ z=`expr $z + 1`
$ echo $z
6
```

expr

Evaluates



```
$ let z=5
$ echo $z
5

$ let z=$z+1
$ echo $z
6

$ let z=$z + 1   # --- Spaces around + sign are
                    bad with let -bash: let: +:
                 syntax error:operand expected
                 (error token is "+")



$ let z=z+1           # --- look dear, no $ to read
                 a variable.
$ echo $z
7
```

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

```
echo -e "An apple a day keeps away \a\t\tdoctor\n"
```

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 ILG 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

```
$ vim variscript

## Script to test MY knowledge about variables!

myname=ilg

myos = TroubleOS

myno=5

echo "My name is $myname"

echo "My os is $myos"

echo "My number is myno, can you see this number"
```

Quotes

There are three types of quotes

"Double Quotes" - Anything enclose in double quotes removed meaning of that characters (except \ and $).

'Single quotes' - Enclosed in single quotes remains unchanged.

`Back quote` - To execute command

E.g:

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

Can't print message with today's date.

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

Exit Status

By default in Linux if particular command/shell script is executed, it returns 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?

To determine this exit Status you can use $? special variable of shell.

e.g

```
$ ls
$ echo $?
```

It will print 0 to indicate command is successful.

The ‘read’ Statement

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

Syntax:

read variable1 variable2 ...variableN

Create a script as follows in vim editor / nano editor.

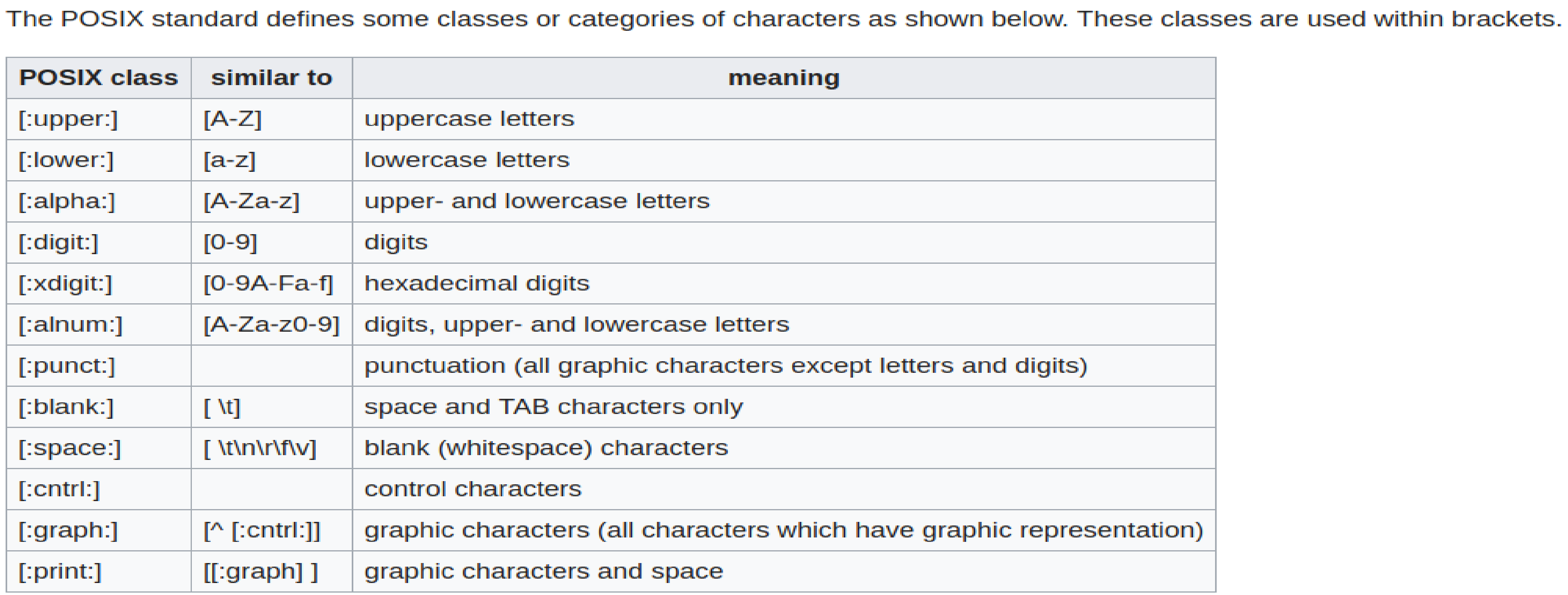
```
$ vim kbinput.sh
#!/bin/bash
#Script to read your name from key-board
echo "Your first name please:"
read fname
echo "Hello $fname, Lets play game of chess !"
```

Run it as follows:

```
$ chmod 755 kbinput.sh
$ ./kbinput.sh
Your first name please: Jagjit
Hello Jagjit, Lets play game of chess !
```

Try these on the shell prompt

```
read -p "Dir : " dirname; echo $dirname;ls -l $dirname;
read -t 5  -s -p   "Enter pass within 5 sec :  "   pass
```


The POSIX standard defines some classes or categories of characters as shown below. These classes are used within brackets.

POSIX class	similar to	meaning
[:upper:]	[A-Z]	uppercase letters
[:lower:]	[a-z]	lowercase letters
[:alpha:]	[A-Za-z]	upper- and lowercase letters
[:digit:]	[0-9]	digits
[:xdigit:]	[0-9A-Fa-f]	hexadecimal digits
[:alnum:]	[A-Za-z0-9]	digits, upper- and lowercase letters
[:punct:]		punctuation (all graphic characters except letters and digits)
[:blank:]	[\t]	space and TAB characters only
[:space:]	[\t\n\r\f\v]	blank (whitespace) characters
[:cntrl:]		control characters
[:graph:]	[^ [:cntrl:]]	graphic characters (all characters which have graphic representation)
[:print:]	[[:graph]]	graphic characters and space

More command on one command line

Syntax:

```
command1;command2
```

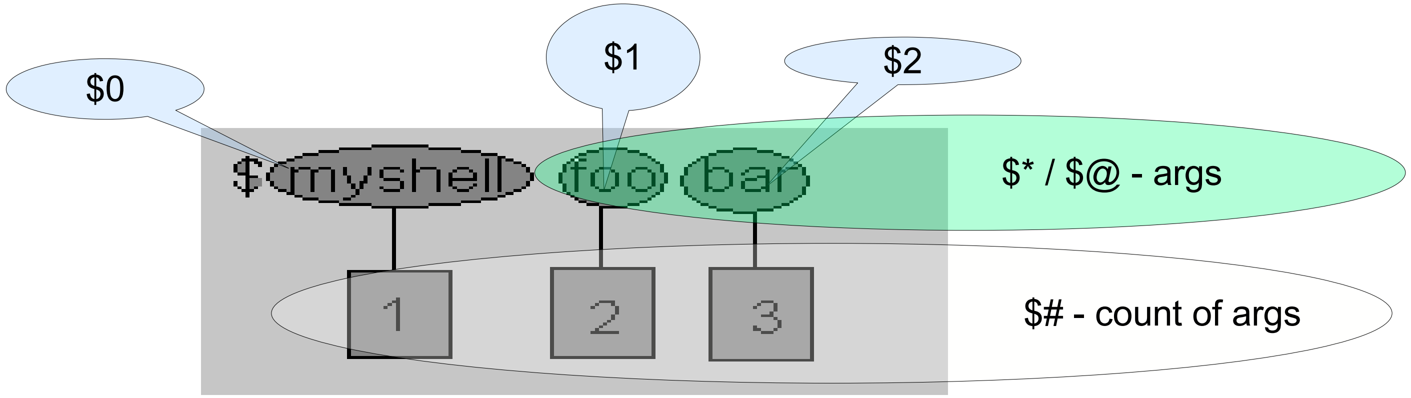
To run two command with one command line.

E.g:

```
$ date;who
```

Will print today's date followed by users who are currently login.

Command Line arguments



$# holds number of arguments specified on command line.

$* or $@ refer to all arguments passed to script.

$1, $2, $3....actual arguments.

$0 represents script name.

Shell Script name	-	myshell
First command line argument passed to myshell	-	foo
Second command line argument passed to myshell	-	bar

```
$ vim demo.sh
#!/bin/bash

## 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 $@"
```

Shift

The shift command reassigns the positional parameters, in effect shifting them to the left one notch.

$1 <--- $2, $2 <--- $3, $3 <--- $4, etc.

The old $1 disappears, but $0 (the script name) does not change.

If you use a large number of positional parameters to a script, shift lets you access those past 10, although {bracket} notation also permits this.

Using shift

```
#!/bin/bash

#Reference – Absolute bash scripting guide – Mendel Cooper.

#./shft.sh a b c def 83 barndoor

until [ -z "$1" ]  # Until all parameters used up
do
  echo -n "$1 "
  shift
done

echo                  # Extra linefeed.

exit 0
```

The shift command can take a numerical parameter indicating *how many positions to shift.*

```
#!/bin/bash
# shift-past.sh
shift 3          # Shift 3 positions


#  n=3; shift $n
#  Has the same effect.


echo "$1"
exit 0


# ======================== #


$ ./shift-past.sh 1 2 3 4 5
4
```

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 comparison between two values."

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]

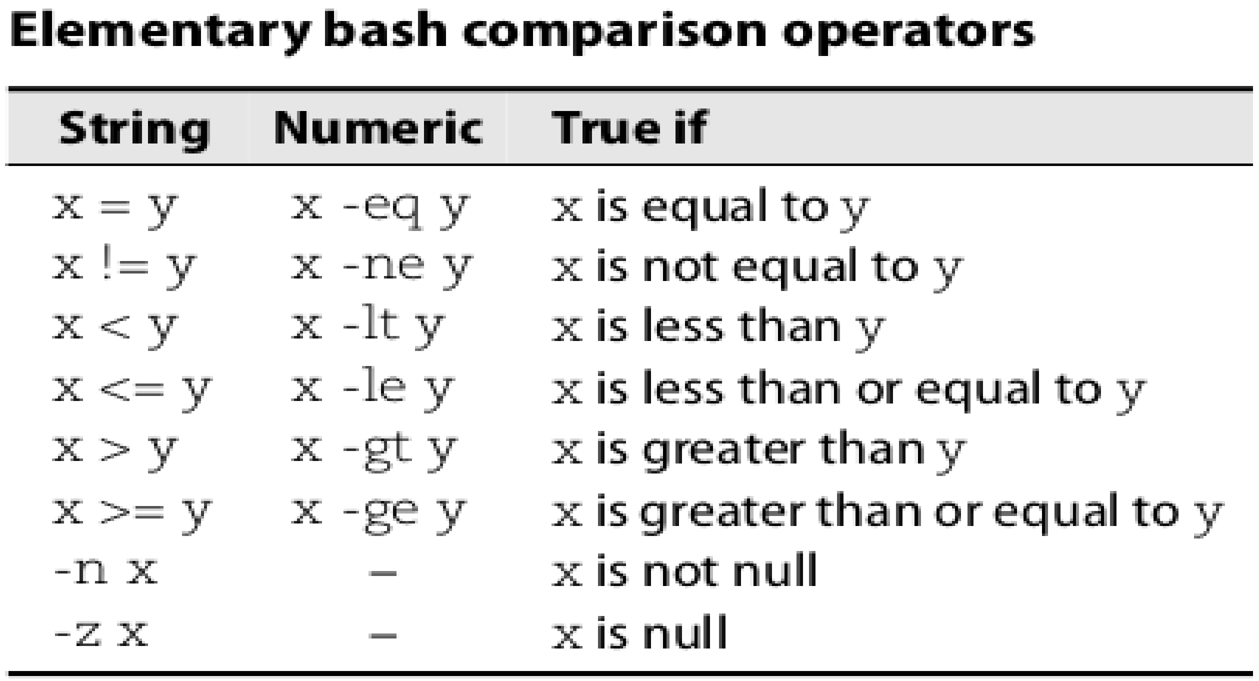
E.g:

Following script determine whether given argument number is positive.

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

Mathematics,	Meaning	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]

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 is defined
-n string1	True if string is not empty.
-z string1	True if string is empty.



Elementary bash comparison operators

String	Numeric	True if
x = y	x -eq y	x is equal to y
x != y	x -ne y	x is not equal to y
x < y	x -lt y	x is less than y
x <= y	x -le y	x is less than or equal to y
x > y	x -gt y	x is greater than y
x >= y	x -ge y	x is greater than or equal to y
-n x	–	x is not null
-z x	–	x is null

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
file1 -nt file2	file1 is newer than file2
file1 -ot file2	file1 is older than file2

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

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
```

```
$ vim isnump_n.sh

#!/bin/bash

# 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
```

Nested if-else-fi

```
$ vim nestedif.sh

#!/bin/bash

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
```

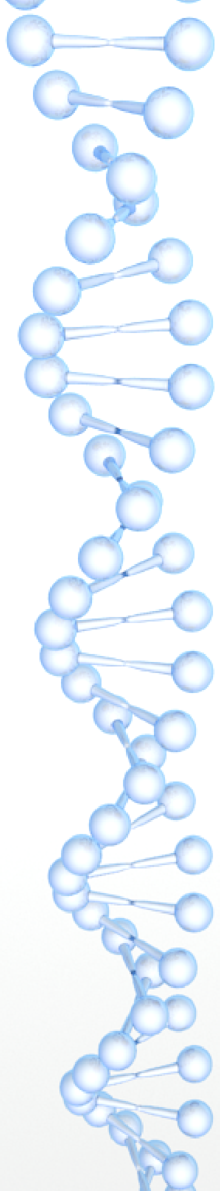
You can write the entire if-else construct within either the body of the if statement of the body of an else statement. This is called the nesting of ifs.

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
              aretrue(i.e.all of the above nonzero or false)
            execute all commands up to fi
fi
```

```
$ cat > elf.sh
#!/bin/bash
# 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
```



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
```

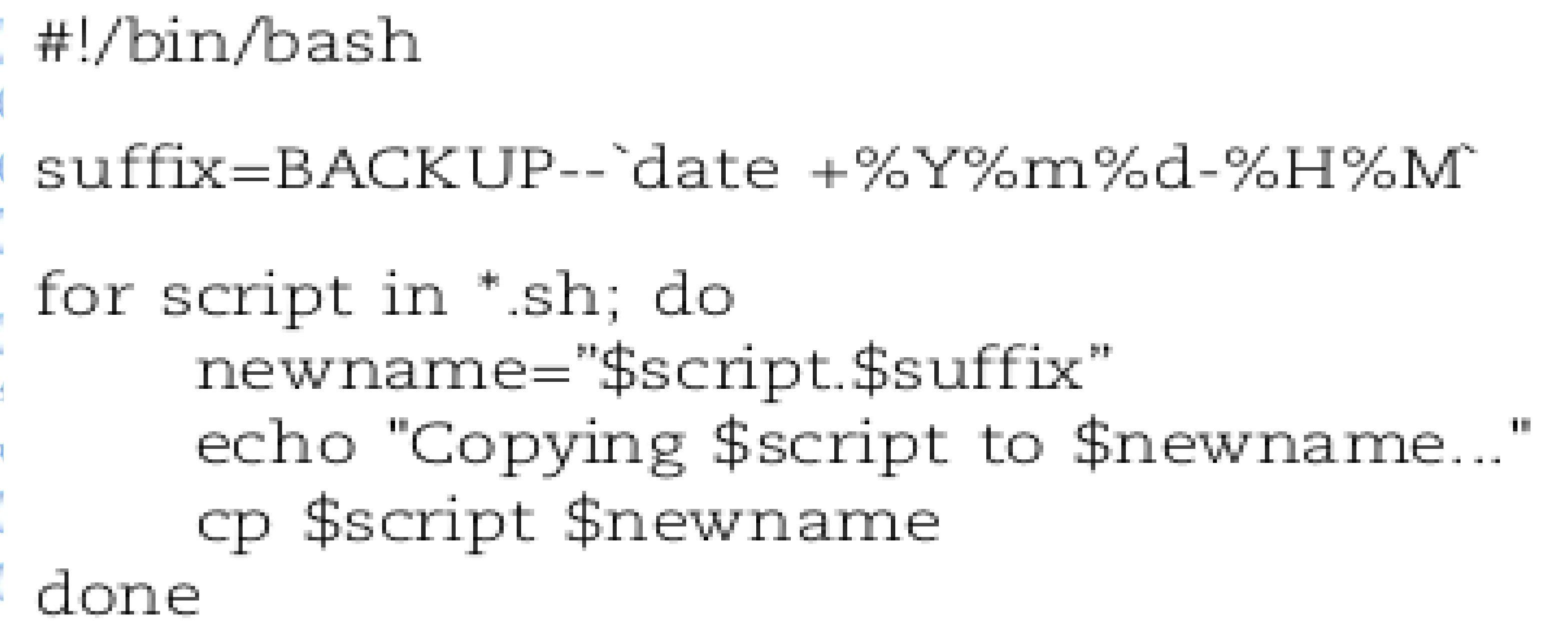
```
#!/bin/bash
#Script to test for loop
if [ $# -eq 0 ]
then
        echo "Error - Number missing form command line argument"
        echo "Syntax : $0 number"
        echo "Use to print multiplication table for given number"
        exit 1
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
do
        echo "$n *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

E.g 1

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

E.g 2

```
$ 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
```



```
#!/bin/bash

suffix=BACKUP--`date +%Y%m%d-%H%M`

for script in *.sh; do
        newname="$script.$suffix"
        echo "Copying $script to $newname..."
        cp $script $newname
done
```

Following shell script will go though all files stored in /etc directory.

The for loop will be abandoned when /etc/resolv.conf file found.

```
#!/bin/bash
for file in /etc/*
do
        if [  "${file}" == "/etc/resolv.conf" ]
        then
                countNameservers=$(grep -c nameserver /etc/resolv.conf)
                echo "Total ${countNameservers} nameservers defined in ${file}"
                break
        fi
done
```

This script make backup of all file names specified on command line.

If .bak file exists, it will skip the cp command.

```
#!/bin/bash
FILES="$@"
for f in $FILES
do
        # if .bak backup file exists, read next file
        if [ -f ${f}.bak ]
        then
                echo "Skiping $f file..."
            # read next file and skip the cp command
                continue
        fi
# we are here means no backup file exists, just use cp command to copy file
        /bin/cp $f $f.bak
done
```

Infinite for loop can be created with empty expressions, such as:

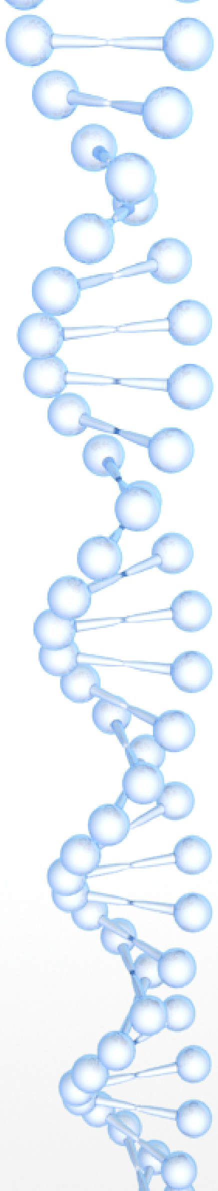
```
#!/bin/bash
for (( ; ; ))
do
   echo "infinite loops [ hit CTRL+C to stop]"
done
```

while loop

Syntax:

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

```
$ vim table2.sh
#!/bin/bash                              ##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
```



```
#!/bin/bash
# set n to 1
n=1

# continue until $n equals 5
while [ $n -le 5 ]
do
	echo "Welcome $n times."
	n=$(( n+1 ))	# increments		echo $n
done
```

```
#!/bin/bash
n=1

while (( $n <= 5 ))
do
        echo "Welcome $n times."
        n=$(( n+1 ))
done
```

Reading A Text File

```
#!/bin/bash

file=/etc/resolv.conf

while IFS=’ ‘ read -r line
do
        # echo line is stored in $line
        echo $line
done < "$file"
```

Reading A Text File With Separate Fields

```
#!/bin/bash

file=/etc/resolv.conf

# set field separator to a single white space
while IFS=' ' read -r f1 f2
do
	echo "field # 1 : $f1 ==> field #2 : $f2"
done < "$file"
```

```
#!/bin/bash
file=/etc/passwd
# set field delimiter to :
# read all 7 fields into 7 vars
while IFS=: read -r user enpass uid gid desc home shell
do
    # only display if UID >= 500
    [ $uid -ge 500 ] && echo "User $user ($uid) assigned \"$home\" home directory with $shell shell."
done < "$file"
```

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
```

```
$ vim car.sh
# 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
```

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.

```
$ 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 +x dsh1.sh
```

Run for debugging

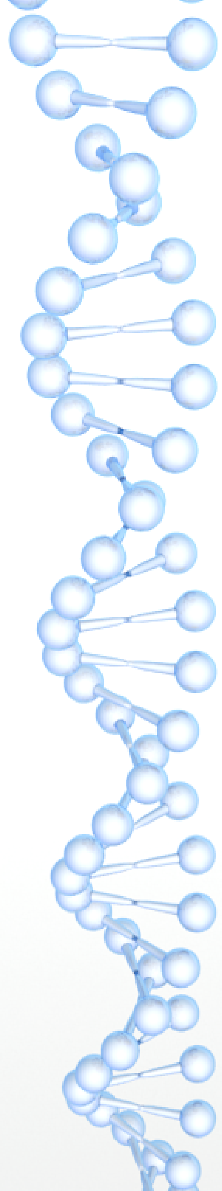
```
$ ./dsh1.sh 4 5              #Executing output
9
$ bash -x dsh1.sh 4 5  #debugging
```

Functions – how to define

```
function func1
    {
        Command
            }

            OR

    function_name()
    {
            commands
    }
```



Function defined

```
#!/bin/bash
clear

be_happy()
        {
                echo "I am always happy"
        }
be_happy
```

Calling the function

```
function hello
        {
                echo "Hello!How are you?"
        }

hello

exit 0
```

Reference Book:-

Absolute bash scripting guide

Website http://www.tldp.org

Questions

&

Answers