

Introduction to Linux Shell and Shell Scripting

If you are using any major operating system you are indirectly interacting to shell. If you are running Ubuntu, Linux Mint or any other Linux distribution, you are interacting to shell every time you use terminal. In this article I will discuss about linux shells and shell scripting so before understanding shell scripting we have to get familiar with following terminologies –

- Kernel
- Shell
- Terminal

What is Kernel

The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages following resources of the Linux system –

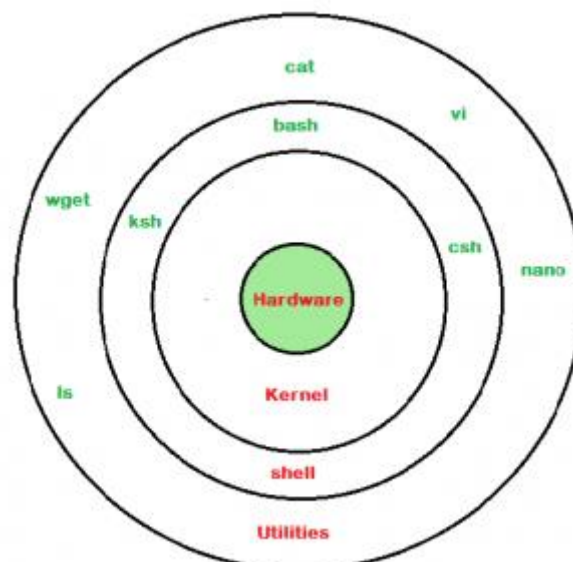
- File management
- Process management
- I/O management
- Memory management
- Device management etc.

It is often mistaken that Linus Torvalds has developed Linux OS, but he is only responsible for development of Linux kernel.

Complete Linux system = Kernel + GNU system utilities and libraries + other management scripts + installation scripts.

What is Shell

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.

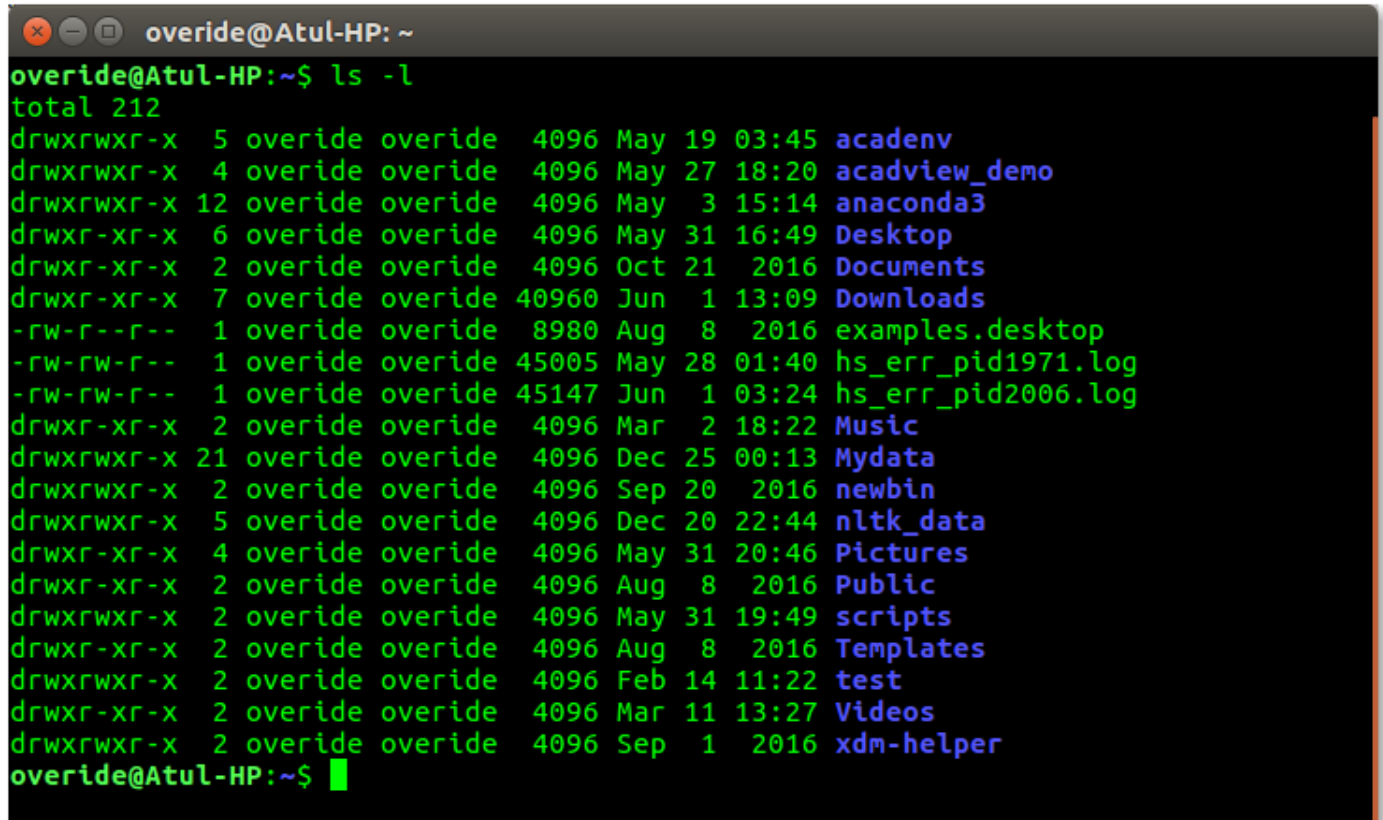


Shell is broadly classified into two categories –

- Command Line Shell
- Graphical shell

Command Line Shell

Shell can be accessed by user using a command line interface. A special program called **Terminal** in linux/macOS or **Command Prompt** in Windows OS is provided to type in the human readable commands such as “cat”, “ls” etc. and then it is being execute. The result is then displayed on the terminal to the user. A terminal in Ubuntu 16.4 system looks like this –

A screenshot of a terminal window titled 'override@Atul-HP: ~'. The prompt is 'override@Atul-HP:~\$'. The command 'ls -l' has been executed, displaying a long listing of files and directories. The output shows permissions, owner, group, size, date, time, and filename for each item. Files like 'acadenv', 'acadview_demo', 'anaconda3', 'Desktop', 'Documents', 'Downloads', 'examples.desktop', 'hs_err_pid1971.log', 'hs_err_pid2006.log', 'Music', 'Mydata', 'newbin', 'nltk_data', 'Pictures', 'Public', 'scripts', 'Templates', 'test', 'Videos', and 'xdm-helper' are listed.

```
override@Atul-HP:~$ ls -l
total 212
drwxrwxr-x  5 override override 4096 May 19 03:45 acadenv
drwxrwxr-x  4 override override 4096 May 27 18:20 acadview_demo
drwxrwxr-x 12 override override 4096 May  3 15:14 anaconda3
drwxr-xr-x  6 override override 4096 May 31 16:49 Desktop
drwxr-xr-x  2 override override 4096 Oct 21  2016 Documents
drwxr-xr-x  7 override override 4096 Jun  1 13:09 Downloads
-rw-r--r--  1 override override 8980 Aug  8  2016 examples.desktop
-rw-rw-r--  1 override override 45005 May 28 01:40 hs_err_pid1971.log
-rw-rw-r--  1 override override 45147 Jun  1 03:24 hs_err_pid2006.log
drwxr-xr-x  2 override override 4096 Mar  2 18:22 Music
drwxrwxr-x 21 override override 4096 Dec 25 00:13 Mydata
drwxrwxr-x  2 override override 4096 Sep 20  2016 newbin
drwxrwxr-x  5 override override 4096 Dec 20 22:44 nltk_data
drwxr-xr-x  4 override override 4096 May 31 20:46 Pictures
drwxr-xr-x  2 override override 4096 Aug  8  2016 Public
drwxrwxr-x  2 override override 4096 May 31 19:49 scripts
drwxr-xr-x  2 override override 4096 Aug  8  2016 Templates
drwxrwxr-x  2 override override 4096 Feb 14 11:22 test
drwxr-xr-x  2 override override 4096 Mar 11 13:27 Videos
drwxrwxr-x  2 override override 4096 Sep  1  2016 xdm-helper
override@Atul-HP:~$
```

In above screenshot “ls” command with “-l” option is executed.

It will list all the files in current working directory in long listing format. Working with command line shell is bit difficult for the beginners because it’s hard to memorize so many commands. It is very powerful, it allows user to store commands in a file and execute them together. This way any repetitive task can be easily automated. These files are usually called **batch files** in Windows and **Shell Scripts** in Linux/macOS systems.

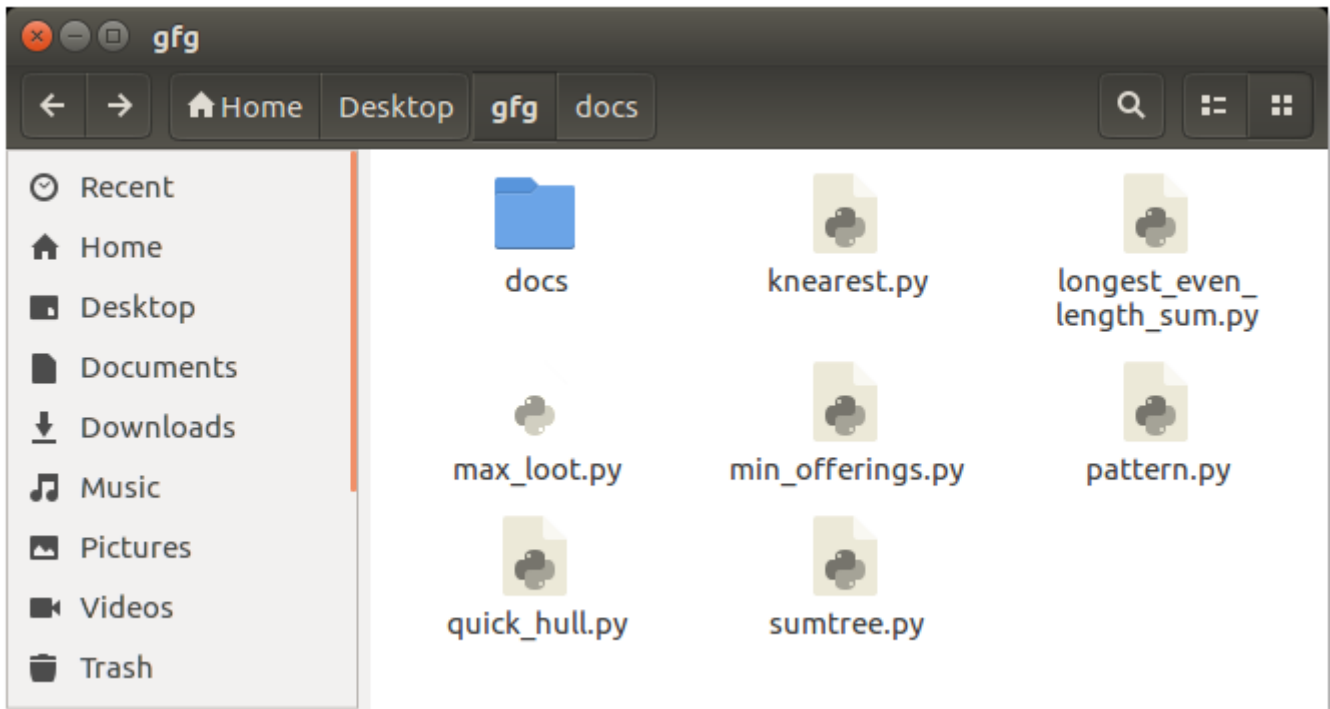
Graphical Shells

Graphical shells provide means for manipulating programs based on graphical user interface (GUI), by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every actions. A typical GUI in Ubuntu system is shown on next page.

There are several shells are available for Linux systems like –

- [BASH \(Bourne Again SHell\)](#) – It is most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.

- [CSH \(C SHell\)](#) – The C shell's syntax and usage are very similar to the C programming language.
 - [KSH \(Korn SHell\)](#) – The Korn Shell also was the base for the POSIX Shell standard specifications etc.
- Each shell does the same job but understand different commands and provide different built-in functions.



Shell Scripting

Usually shells are interactive that mean, they accept command as input from users and execute them. However some time we want to execute a bunch of commands routinely, so we have to type in all commands each time in terminal.

As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called **Shell Scripts** or **Shell Programs**. Shell scripts are similar to the [batch file](#) in MS-DOS. Each shell script is saved with **.sh** file extension eg. **myscript.sh**

A shell script have syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

A shell script comprises following elements –

- Shell Keywords – if, else, break etc.
- Shell commands – cd, ls, echo, pwd, touch etc.
- Functions
- Control flow – if..then..else, case and shell loops etc.

Why do we need shell scripts?

There are many reasons to write shell scripts –

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups
- System monitoring
- Adding new functionality to the shell etc.

Advantages of shell scripts

- The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax

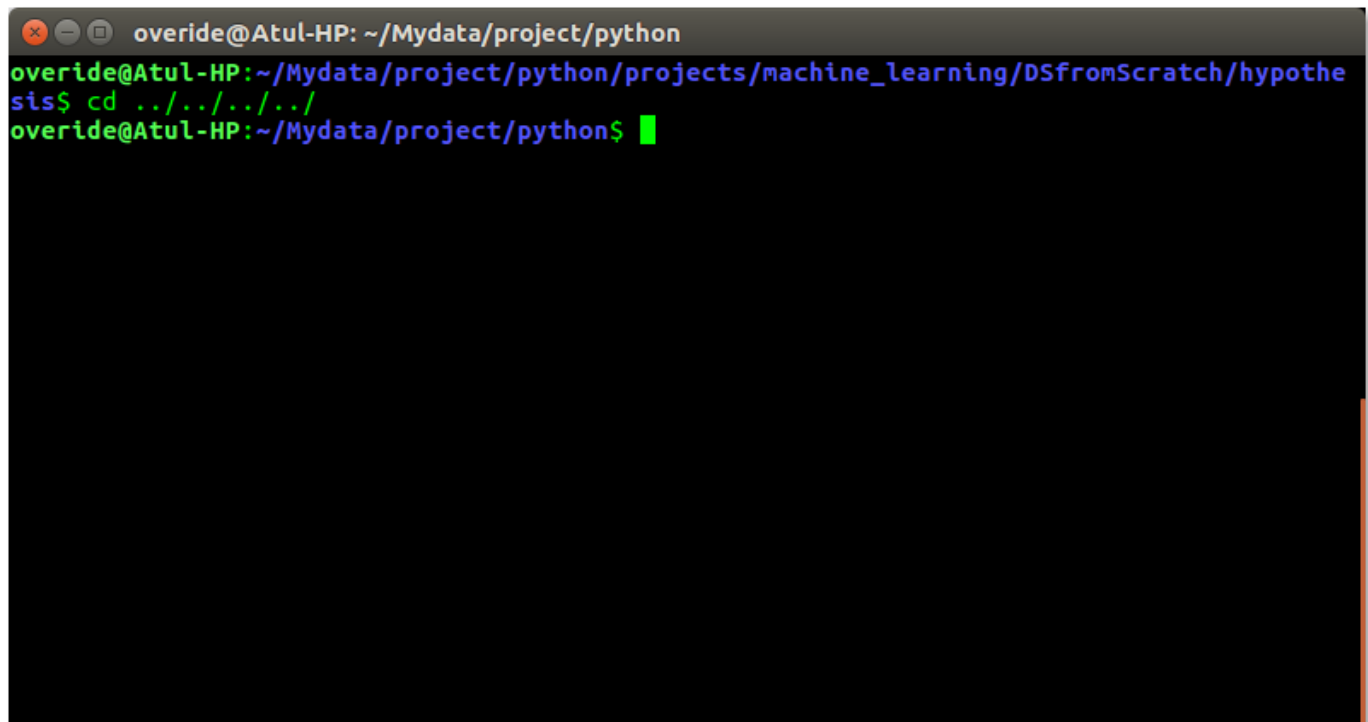
- Writing shell scripts are much quicker
- Quick start
- Interactive debugging etc.

Disadvantages of shell scripts

- Prone to costly errors, a single mistake can change the command which might be harmful
- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. etc

Simple demo of shell scripting using Bash Shell

If you work on terminal, something you traverse deep down in directories. Then for coming few directories up in path we must execute command like this as shown below to get to the “python” directory –



```

override@Atul-HP: ~/Mydata/project/python
override@Atul-HP:~/Mydata/project/python/projects/machine_learning/DSfromScratch/hypothesis$ cd ../../../../
override@Atul-HP:~/Mydata/project/python$

```

Any thing (command or variable with some value) that we wish to make available on every terminal session, we have to put this in “**.bashrc**” file.

“**.bashrc**” is a shell script that Bash shell runs whenever it is started interactively. The purpose of a .bashrc file is to provide a place where you can set up variables, functions and aliases, define our prompt and define other settings that we want to use whenever we open a new terminal window.

Operators & Programming constructs in shell programming

There are **5** basic operators in bash/shell scripting:

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- Bitwise Operators
- File Test Operators

1. Arithmetic Operators: These operators are used to perform normal arithmetic/mathematical operations. There are 7 arithmetic operators:

- **Addition (+):** Binary operation used to add two operands.

- **Subtraction (-):** Binary operation used to subtract two operands.
- **Multiplication (*):** Binary operation used to multiply two operands.
- **Division (/):** Binary operation used to divide two operands.
- **Modulus (%):** Binary operation used to find remainder of two operands.
- **Increment Operator (++):** Unary operator used to increase the value of operand by one.
- **Decrement Operator (--):** Unary operator used to decrease the value of a operand by one

Examples

```
#!/bin/bash
```

```
#reading data from the user
```

```
read -p 'Enter a : ' a
```

```
    read
```

```
-p 'Enter b : ' b
```

```
    add
```

```
= $((a + b))
```

```
    echo Addition of a and b are $add
```

```
    sub
```

```
= $((a - b))
```

```
    echo Subtraction of a and b are $sub
```

```
    mul
```

```
= $((a * b))
```

```
    echo Multiplication of a and b are $mul
```

```
    div
```

```
= $((a / b))
```

```
    echo division of a and b are $div
```

```
    mod
```

```
= $((a % b))
```

```
    echo Modulus of a
```

```
and b are $mod
```

```
((++a))
```

```
    echo Increment
```

```
operator when applied on "a" results into a = $a
```

```
((--b))
```

```
    echo Decrement
```

```
operator when applied on "b" results into b = $b
```

2. Relational Operators: Relational operators are those operators which define the relation between two operands. They give either true or false depending upon the relation. They are of 6 types:

- **'==' Operator:** Double equal to operator compares the two operands. Its returns true is they are equal otherwise returns false.
- **'!=' Operator:** Not Equal to operator return true if the two operands are not equal otherwise it returns false.

- '**<**' **Operator**: Less than operator returns true if first operand is less than second operand otherwise returns false.
- '**<=**' **Operator**: Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false
- '**>**' **Operator**: Greater than operator return true if the first operand is greater than the second operand otherwise return false.
- '**>=**' **Operator**: Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

Examples:

```
#!/bin/bash
```

```
#reading data from the user
```

```
read -p 'Enter a : ' a
```

```
read -p 'Enter b : ' b
```

```
if(( $a==$b )) or ((a==b)) //preceded with or without dollar "$", both work
then
```

```
    echo a is equal to b.
```

```
else
```

```
    echo a is not equal to b.
```

```
fi
```

```
if(( $a!=$b ))
```

```
then
```

```
    echo a is not equal to b.
```

```
else
```

```
    echo a is equal to b.
```

```
fi
```

```
if(( $a<$b ))
```

```
then
```

```
    echo a is less than b.
```

```
else
```

```
    echo a is not less than b.
```

```
fi
```

```
if(( $a<=$b ))
```

```
then
```

```
    echo a is less than or equal to b.
```

```
else
```

```
    echo a is not less than or equal to b.
```

```
fi
```

```
if(( $a>$b ))
```

```
then
```

```
    echo a is greater than b.
```

```
else
```

```
    echo a is not greater than b.
```

```
fi
```

```
if(( $a>=$b ))
```

```
then
```

```
    echo a is greater than or equal to b.
```

```
else
```

```
    echo a is not greater than or equal to b.
```

```
fi
```

. **Logical Operators** : They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:

- **Logical AND (&&)**: This is a binary operator, which returns true if both the operands are true otherwise returns false.
- **Logical OR (||)**: This is a binary operator, which returns true if either of the operand is true or both the operands are true and return false if none of them is false.
- **Not Equal to (!)**: This is a unary operator which returns true if the operand is false and returns false if the operand is true.

Examples

```
#!/bin/bash
```

```
#reading data from the user
```

```
read -p 'Enter a : ' a
```

```
read -p 'Enter b : ' b
```

```
if(($a == "true" && $b == "true" ))
```

```
then
```

```
    echo Both are true.
```

```
else
```

```
    echo Both are not true.
```

```
fi
```

```
if(($a == "true" || $b == "true" ))
```

```
then
```

```
    echo At least one of them is true.
```

```
else
```

```
    echo None of them is true.
```

```
fi
```

```
if(( ! $a == "true" ))
```

```
then
```

```
    echo "a" was initially false.
```

```
else
```

```
    echo "a" was initially true.
```

```
fi
```

4. Bitwise Operators: A bitwise operator is an operator used to perform bitwise operations on bit patterns. They are of 6 types:

- **Bitwise And (&)**: Bitwise & operator performs binary AND operation bit by bit on the operands.
- **Bitwise OR (|)**: Bitwise | operator performs binary OR operation bit by bit on the operands.
- **Bitwise XOR (^)**: Bitwise ^ operator performs binary XOR operation bit by bit on the operands.
- **Bitwise complement (~)**: Bitwise ~ operator performs binary NOT operation bit by bit on the operand.
- **Left Shift (<<)**: This operator shifts the bits of the left operand to left by number of times specified by right operand.
- **Right Shift (>>)**: This operator shifts the bits of the left operand to right by number of times specified by right operand.

Examples

```
#!/bin/bash
```

```
#reading data from the user
```

```
read -p 'Enter a : ' a
```

```
read -p 'Enter b : ' b
```

```
bitwiseAND=$(( a&b ))
```

```
echo Bitwise AND of a and b is $bitwiseAND
```

```
bitwiseOR=$(( a|b ))
```

```
echo Bitwise OR of a and b is $bitwiseOR
```

```
bitwiseXOR=$(( a^b ))
```

```
echo Bitwise XOR of a and b is $bitwiseXOR
```

```
bitwiseComplement=$(( ~a ))
```

```
echo Bitwise Compliment of a is $bitwiseComplement
```

```
leftshift=$(( a<<1 ))
```

```
echo Left Shift of a is $leftshift
```

```
rightshift=$(( b>>1 ))
```

```
echo Right Shift of b is $rightshift
```

5. File Test Operator: These operators are used to test a particular property of a file.

- **-b operator:** This operator check whether a file is a block special file or not. It returns true if the file is a block special file otherwise false.
- **-c operator:** This operator checks whether a file is a character special file or not. It returns true if it is a character special file otherwise false.
- **-d operator:** This operator checks if the given directory exists or not. If it exists then operators returns true otherwise false.
- **-e operator:** This operator checks whether the given file exists or not. If it exists this operator returns true otherwise false.
- **-r operator:** This operator checks whether the given file has read access or not. If it has read access then it returns true otherwise false.
- **-w operator:** This operator check whether the given file has write access or not. If it has write then it returns true otherwise false.
- **-x operator:** This operator check whether the given file has execute access or not. If it has execute access then it returns true otherwise false.
- **-s operator:** This operator checks the size of the given file. If the size of given file is greater than 0 then it returns true otherwise it is false.

```
#!/bin/bash
```

```
#reading data from the user
```

```
read -p 'Enter file name : ' FileName
```

```
if [ -e $FileName ]
```

```
then
```

```
    echo File Exist
```

```
else
```

```
    echo File doesnot exist
```



```
fi
```

```
if [ -s $FileName ]
then
    echo The given file is not empty.
else
    echo The given file is empty.
fi
```

```
if [ -r $FileName ]
then
    echo The given file has read access.
else
    echo The given file does not has read access.
fi
```

```
if [ -w $FileName ]
then
    echo The given file has write access.
else
    echo The given file does not has write access.
fi
```

```
if [ -x $FileName ]
then
    echo The given file has execute access.
else
    echo The given file does not has execute access.
fi
```

Conditional Statements: There are total 5 conditional statements which can be used in bash programming

1. if statement
2. if-else statement
3. if..elif..else..fi statement (Else If ladder)
4. if..then..else..if..then..fi..fi..(Nested if)
5. switch statement

Their description with syntax is as follows:

if statement

This block will process if specified condition is true.

Syntax:

```
if (( expression ))
then
    statement
fi
```

if-else statement

If specified condition is not true in if part then else part will be execute.

Syntax

```
if (( expression ))
then
    statement1
else
    statement2
```

```
fi
```

if..elif..else..fi statement (Else If ladder)

To use multiple conditions in one if-else block, then elif keyword is used in shell. If expression1 is true then it executes statement 1 and 2, and this process continues. If none of the condition is true then it processes else part.

Syntax

```
if [ expression1 ] or (( expression1 )) //but only in case of arithmetic or logical operators
```

```
then
```

```
    statement1
```

```
    statement2
```

```
    .
```

```
    .
```

```
elif [ expression2 ]
```

```
then
```

```
    statement3
```

```
    statement4
```

```
    .
```

```
    .
```

```
else
```

```
    statement5
```

```
fi
```

if..then..else..if..then..fi..fi..(Nested if)

Nested if-else block can be used when, one condition is satisfies then it again checks another condition. In the syntax, if expression1 is false then it processes else part, and again expression2 will be check.

Syntax:

```
if [ expression1 ]
```

```
then
```

```
    statement1
```

```
    statement2
```

```
    .
```

```
else
```

```
    if [ expression2 ]
```

```
    then
```

```
        statement3
```

```
        .
```

```
    fi
```

```
fi
```

switch statement

case statement works as a switch statement if specified value match with the pattern then it will execute a block of that particular pattern

When a match is found all of the associated statements until the double semicolon (;;) is executed.

A case will be terminated when the last command is executed.

If there is no match, the exit status of the case is zero.

Syntax:

```
case in
```

```
    Pattern 1) Statement 1;;
```

```
    Pattern n) Statement n;;
```

```
esac
```

Example Programs //in following examples, we can use (()) instead of [] for arithmetic and logical operations on numeric quantities

Example 1:

Implementing if statement

#Initializing two variables

a=10

b=20

#Check whether they are equal

if [\$a == \$b] **or** ((a==b)) **or** ((\$a==\$b))

then

echo "a is equal to b"

fi

Example-2

Implementing `if.else` statement

#Check whether they are not equal

if [\$a != \$b]

then

echo "a is not equal to b"

fi

#Initializing two variables

a=20

b=20

if [\$a == \$b]

then

#If they are equal then print this

echo "a is equal to b"

else

#else print this

echo "a is not equal to b"

fi

Example-3:

Implementing `switch` statement

CARS="bmw"

#Pass the variable in string

case "\$CARS" in

#case 1

"mercedes") echo "Headquarters - Affalterbach, Germany" ;;

#case 2

"audi") echo "Headquarters - Ingolstadt, Germany" ;;

#case 3

"bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;

esac

Looping Statements in Shell Scripting: There are total 3 looping statements which can be used in bash programming

1. while statement

2. for statement

3. until statement

To alter the flow of loop statements, two commands are used they are,

1. break
2. continue

Their descriptions and syntax are as follows:

- **while statement**

Here command is evaluated and based on the result loop will be executed, if command raises to false then loop will be terminated

Syntax

```
while command
do
    Statement to be executed
done
```

- **for statement**

The for loop operates on lists of items. It repeats a set of commands for every item in a list.

Here var is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Syntax

```
for var in word1 word2 ...wordn
do
    Statement to be executed
done
```

- **until statement**

The until loop is executed as many as times the condition/command evaluates to false. The loop terminates when the condition/command becomes true.

Syntax

```
until command
do
    Statement to be executed until command is true
done
```

Example Programs

Example 1:

Implementing for loop with break statement

#Start of for loop

```
for a in 1 2 3 4 5 6 7 8 9 10
```

```
do
```

```
    # if a is equal to 5 break the loop
```

```
    if [ $a == 5 ]
```

```
    then
```

```
        break
```

```
    fi
```

```
    # Print the value
```

```
    echo "Iteration no $a"
```

```
done
```

Example-2:

Implementing for loop with continue statement

```
for a in 1 2 3 4 5 6 7 8 9 10
```

```
do
```

```
    # if a = 5 then continue the loop and
```

```
    # don't move to line 8
```

```
    if [ $a == 5 ]
```

```

        then
            continue
        fi
        echo "Iteration no $a"
done

```

Example3:

Implementing while loop

a=0

-lt is less than operator

#Iterate the loop until a less than 10

```
while [ $a -lt 10 ]
```

```
do
```

```
    # Print the values
```

```
    echo $a
```

```
    # increment the value
```

```
    a=`expr $a + 1`
```

```
done
```

Example4:

Implementing until loop

a=0

-gt is greater than operator

#Iterate the loop until a is greater than 10

```
until [ $a -gt 10 ]
```

```
do
```

```
    # Print the values
```

```
    echo $a
```

```
    # increment the value
```

```
    a=`expr $a + 1`
```

```
done
```

Shell Programming examples

#Program-1 (input through read)

```
#!/bin/bash
```

```
read -p "Your first name: " fname
```

```
read -p "Your last name: " lname
```

```
echo "Hello $fname $lname ! I am learning how to create shell scripts"
```

```
-----
```

#program-2 (input through command line)

```
#!/bin/csh
```

```
echo "The entered inputs are $1 & $2"
```

```
((c=$1+$2))
```

```
echo "Sum of $1 and $2 is $c"
```

```
-----
```

#program-3 for usage of file operators

```
#!/bin/bash
```

```
cd
```

```
ls -l
```

```

read -p "Enter a file name: " filename
if [ -e $filename ]
then
    echo "file exists!"
    if [ -r $filename ]
    then
        status="readable "
    fi
    if [ -w $filename ]
    then
        status=$status"writable "
    fi
    if [ -x $filename ]
    then
        status=$status"executable"
    fi
    echo "file permission: $status "
else
    echo "file does not exist"
fi

```

#Program-4 (usage of string operators)

```

#!/bin/bash
read -p "First String: " str1
read -p "Second String: " str2
if [ -z "$str1" ]
then
    echo "The 1st string is null"
elif [ -z "$str2" ]
then
    echo "The 2nd string is null"
else
    if [ $str1 == $str2 ]
    then
        echo "The strings are equal"
    else
        echo "The strings are not equal"
    fi
fi

```

#Program-5

```

#!/bin/bash
read -p "Enter an integer: " int1
if((int1==0))
then
    echo "Zero"
elif((int1<0))
then
    echo "Negative"
else
    if (((int1%2)==0))

```

```

then
    echo "Even"
else
    echo "Odd"
fi
fi

```

```

#Program-6 to show case-esac
#!/bin/bash
clear
read -p "Integer1: " int1
read -p "Integer2: " int2
echo "=====
printf "Menu: \n[a] Addition\n[b]Subtraction\n[c]Multiplication\n[d]Division\n"
echo "=====
read -p "Your choice: " choice
res=0
case $choice in
a) res=$((int1+int2)) ;;
b) ((res=int1-int2)) ;;
c) res=$((int1*int2)) ;;
d)
    res=$((int1/int2))
;;
*)
    echo "Invalid input"
esac
echo "The result is: " $res

```

```

#porgram-7 for usage of new type of for loop
#!/bin/bash
result=0;
input=0;
for var in 1 2 3 4 5
do
    printf "Input integer %d : " $var
    read input
    result=$((result+input))
done
echo "the result is " $result

```

```

#program-8 check palindrome
#!/bin/bash
read -p "Enter string to be checked: " str
rev=""
len=${#str}
while((len>=1))
do
ch=$(echo $str|cut -c$len-$len)
rev=$rev$ch
((len--))

```



```
done
if [ $str == $rev ]
then
echo "string $str is plaindrome"
else
echo "string $str is not palindrome"
fi
```

```
-----
#program-9 create multiple directoris using while
#!/bin/bash
cd sample
echo "creating directories..."
while read var
do
    mkdir $var
done<list
-----
```

```
#program-10 create multiple directoris using special for
#!/bin/bash
list="var1 var2 var3 var4"
var=""
mkdir sample
cd sample
echo creating the "directories...."
for var in $list
do
    mkdir $var
done
-----
```

```
#program-11 use of array
declare -a my_array=( "Hydrogen gas" "Helium" "Lithium" "Beryllium" )
## Array Loop
for i in "${my_array[@]}"
do
    echo "$i"
done
-----
```

```
#program-12 to check armstrong number
set -x
read -p "Enetr number to be checked:" num
new=$num
s=0
while((num!=0))
do
    ((t=num%10))
    ((t=t*t*t))
    ((s=s+t))
    ((num=num/10))
done
echo $s
if((s!=new))
```

```
then
echo "Number $new is not armstrong"
else
echo "Number $new is armstrong"
fi
```

```
-----
#program-13
hournow=$(date +%H)
user=$USER
case $hournow in
1[0]0[1-9]) echo "Good Morning Mr/Ms : $user";;
1[2-5]) echo "Good Afternoon $user";;
1[6-9]) echo "Good evening";;
*) echo "Good Night"
esac
```

```
-----
#program-14
read -p "Enter Name" str
if test -f $str
then echo "file exists n it is an ordinary file"
elif [ -d $str ]
then echo "directory file"
elif [ -c $str ]
then echo "character device files"
elif [ -b $str ]
then echo "block device files"
else
echo "file not exists"
fi
```

```
-----
#program-15 for sorting a list of names (strings) in ascending order
read -p "How many elements: " n
for((i=0;i<n;i++))
do
echo -e "Element[$i]: \c"
read a[i]
done
echo "The unsorted array-"
for((i=0;i<n;i++))
do
echo ${a[$i]}
done
#sorting
for((i=0;i<n-1;i++))
do
for((j=i+1;j<n;j++))
do
if [[ ${a[$i]} > ${a[$j]} ]]
#if((${a[$i]}>${a[$j]}))
then
t=${a[$i]}
```

```

a[$i]=$(a[$j])
a[$j]=$t
fi
done
done
echo "The sorted array is-"

```

```

#program-16 for creation of function

```

```

#!/bin/bash
addfun()
{
#local a
((a=$1+$2))
echo "The sum of function is $a"
}
read -p "enter input in main program" a b
echo "the value of variables a and b in main are $a , $b"
addfun $a $b

```

```

#program-17 for creation of function

```

```

#!/bin/bash
inputs(){
    read -p "Enter first integer: " int1
    read -p "Enter second integer: " int2
}
while(true)
do
clear
printf "Choose from the following operations: \n"
printf "[a]Addition\n[b]Subtraction\n[c]Multiplication\n[d]Division\n"
printf "#####\n"
read -p "Your choice: " choice
case $choice in
[aA]) inputs
    res=$((int1+int2));

[bB]) inputs
    res=$((int1-int2));

[cC]) inputs
    res=$((int1*int2));

[dD]) inputs
    res=$((int1/int2));

*)    res=0
    echo "wrong choice!"
esac
echo "The result is: " $res
read -p "Do you wish to continue? [y]es or [n]o: " ans
if [ $ans == 'n' ]

```

```

then
    echo "Exiting the script. Have a nice day!"
    echo "this script was executed by $$"
    break
else
    continue
fi
done
-----
#program-18 for creation of conversion function
#!/bin/bash
bin(){
    bin1=$(echo "obase=2;ibase=10;$1"|bc)
    #echo "this is not local variable of function op=$op"
    #declare op
    #op=11
    #echo "this is local variable of function op=$op"
    echo "The binary equivalent of $1 is $bin1"
}
dec(){
    dec1=$(echo "obase=10;ibase=2;$1"|bc)
    return $dec1
}
any(){
    out=$(echo "obase=$1;ibase=$2;$3"|bc)
    echo "The equivalent of $3 is $out"
}
#####Main#####
printf "Choose from the following operations:\n[1]Decimal to Binary  Conversion\n"
printf "[2]Binary to Decimal Conversion\n"
printf "[3]Any base to any other base\n"
read -p "Your choice: " op
case $op in
1)
    read -p "Enter integer number: " int
    bin $int
;;
2)
    read -p "Enter binary number: " int
    dec $int
    echo "The decimal equivalent of $int is $?"
;;
3)
    read -p "Enter input base: " ib
    read -p "Enter output base: " ob
    read -p "Enter number: " int
    any $ob $ib $int
;;
*)
    echo "Wrong Choice!"
clear

```

esac

#porgram-19 sorting of numbers in the list
program to sort n numbers

echo -e "How many numbers in the list:\c"

read n

count=1

while((\$count<=\$n))

do

echo "Enter Number [\$count]:"

read list[\$count]

count=\$((count+1))

done

count=1

while((\$count<=(\$n-1)))

do

i=\$((count+1))

while((\$i<=\$n))

do

echo "number \${list[\$count]} and \${list[\$i]} are compared"

if((\${list[\$count]}>\${list[\$i]}))

then

t=\${list[\$count]}

list[\$count]=\${list[\$i]}

list[\$i]=\$t

fi

i=`expr \$i + 1`

done

count=\$((count+1))

done

i=1

echo "The Sorted List-"

while((\$i<=\$n))

do

echo "\${list[\$i]}"

i=`expr \$i + 1`

done