What is Shell?

Shell is a UNIX term for **an interface between a user and an operating system** service. Shell provides users with an interface and accepts human-readable commands into the system and **executes those commands which can run automatically and give the program's output in a shell script**.

What is 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 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.
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 is a text file that contains a sequence of commands for a UNIX - based operating system. It is called a shell script because it combines a sequence of commands, that would
otherwise have to be typed into the keyboard one at a time, into a single script.

A shell script comprises following elements –

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

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

How to Write Shell Script in Linux/Unix

Shell Scripts are written using text editors. On your Linux system, open a text editor program, open a new file to begin typing a shell script or shell programming, then give the shell permission to execute your shell script and put your script at the location from where the shell can find it.

Let us understand the steps in creating a Shell Script:

1. **Create a file using** a **vi** editor(or any other editor). Name script file with **extension .sh**
2. **Start** the script with **#! /bin/bash**
3. Write some code.
4. Save the script file as filename.sh
5. For **executing** the script type **bash filename.sh**

"#!" is an operator called <u>**shebang**</u> which directs the script to the **interpreter location**. So, if we use"#! /bin/sh" the script gets directed to the bourne-shell.

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a <u>shebang</u> because the **#** symbol is called a hash, and the ! symbol is called a bang*.

Let's create a small script -

```
#!/bin/sh
ls
Pwd
```

```
Date
echo “job over”
```

Adding shell comments

Commenting is important in any program. In Shell programming, the syntax to add a comment is

```
#comment line
```

What are Shell Variables?

Variables store data in the form of characters and numbers. Similarly, **Shell variables are used to store information and they can by the shell only.**

For example, the following creates a shell variable and then prints it:

```
A=5
B=6
echo $a $b
msg=”welcome”
echo $msg
```

read command :

The Linux read command is a bash builtin that is typically used to accept user input in a shell script. You can assign that input to a variable to be used for processing.

By default the read command will take input from stdin (standard input) and store it in a variable.

Prompt User for Input:

The read command comes with the -p (prompt) option, which displays a prompt to allow user input.

```
read -p "Enter two numbers: " n1 n2
```

Below is a small script which will use a variable.

```
#!/bin/sh
echo "what is your name?"
read name
echo "What is your phone no $name?"
```

```
read ph_no
echo "My phone number is $ph_no”
```

Check from terminal/command prompt:

```
debasis@LAPTOP-H3N6JCNE:~$ echo Enter value

enter value

debasis@LAPTOP-H3N6JCNE:~$ read n

34

debasis@LAPTOP-H3N6JCNE:~$ echo $n

34

debasis@LAPTOP-H3N6JCNE:~$ read -p "Enter two numbers: " n1 n2

Enter two numbers: 23 45

debasis@LAPTOP-H3N6JCNE:~$ echo $n1 $n2

23 45
```

The above code write in a file and save it with extension .sh and run the file by sh filename

$vi prog.sh

```
echo “Enter value: “

read n

echo “Number is : “ $n

read -p "Enter two numbers: " n1 n2

echo “Numbers are : “ $n1 $n2
```

Run script:

$sh prog.sh

expr command in Linux with examples:

The **expr** command in Unix evaluates a given expression and displays its corresponding output. It is used for:

- Basic operations like **addition, subtraction, multiplication, division, and modulus on integers.**
- Evaluating **regular expressions, string operations like substring, length of strings etc**

NAME

expr - evaluate expressions

SYNOPSIS/Syntax:

- `expr expression`
- expr OPTION

Below are some examples to demonstrate the use of "expr" command:

Using expr for basic arithmetic operations :

Example: Addition

- `` sum=`expr 12 + 8` ``
- `echo $sum`

Example: Multiplication

- `` m=`expr 12 \* 2` ``
- `echo $m`

Note: The multiplication operator * must be escaped when used in an arithmetic expression with *expr*.

Note: *expr* is an external program used by Bourne shell. It uses *expr* external program with the help of backtick. The *backtick(`)* is actually called command substitution

2. Performing operations on variables inside a shell script

Example: Adding two numbers in a script

```
#!/bin/bash

echo "Enter two numbers"

read x

read y

sum=`expr $x + $y`

echo "Sum = $sum"
```

#EX: Write a shell script to input two numbers and Find Addition, Subtraction, Multiplication, Division and Remainder.

#Script

```
#!/bin/bash

echo -n "Enter first number: "

read a

echo -n "Enter second number: "

read b

s=`expr $a + $b`

sub=`expr $a - $b`

m=`expr $a \* $b`    # Return error if  use  only *

d=`expr $a / $b`

rr=`expr $a % $b`

echo "sum= " $s

echo "subtraction: " $sub

echo "Multiplication= " $m

echo "Division= " $d

echo "Remainder= " $rr

echo "job over”
```

output:

To Run Script:

```
debasis@LAPTOP-H3N6JCNE:~$ sh test1.sh

Enter first number: 9

Enter second number: 5

sum=  14

subtraction:  4
```

```
Multiplication=  45

Division=  1

Remainder=  4

job over
```

Performing numerical operations by using following operators :

Operator	Meaning
-lt	Less than
-le	Less than or equal to
-gt	Greater than
-ge	Greater than or equal to
-eq	Equal to
-nq	Not equal to
-a	AND
-o	OR
==	Equal to (Equality)
!= -ne	Not equal to(Not Equality)
=	Assignment(No Space)

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

Assume variable **a** holds 10 and variable **b** holds 20 then −

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.

-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them.

For example, **[$a -nq $b]** is correct

whereas, **[$a -nq $b]** is incorrect.

Conditional Statements | Shell Script:

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. case statement(switch)

(a)if statement

This block will process if specified condition is true.

Syntax:

```
if [ expression ]

then

   statement

fi
```

```
#Example:  To check whether a number is EVEN

#!/bin/bash
echo -n "Enter number : "
read n
rr=`expr $n % 2`
echo "remainder' " $rr
echo " %%%%%%%%%%%%%%%%%%%%%%%%%%%%%% "
if [ $rr -eq 0 ]
        then
                echo "$n is even number."
fi
echo job over
```

(b)if-else statement

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

Syntax

```
if [ expression ]

then

    statement1

   statement2

else

   statement3

   statement4

fi
```

```
Example: Check whether a input number is EVEN or ODD.
#!/bin/bash
echo -n "Enter number : "
read n
rr=`expr $n % 2`
echo "remainder' " $rr
echo " %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%"
if [ $rr -eq 0 ]
        then
                echo "$n is even number."
```

```
      else
            echo "$n is ODD number."
fi
echo job over
```

```
#Input two numbers and Find Maximum value.

#!/bin/bash
echo -n "Enter two numbers: "
read a b
if [ $a -gt $b ]
then
      max=$a
else
      max=$b
fi
echo " maximum value= " $max
```

©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 ]

then

   statement1

   statement2

   .

   .

elif [ expression2 ]

then

   statement3

   statement4

   .

   .

else

   statement5
```

```
fi
```

```
#Input three numbers and Find maximum
#!/bin/bash

 echo -n "Enter three numbers: "
read a b c
if [ $a -gt $b -a $a -gt $c ]
then
        max=$a
elif [ $b -gt $a -a $b -gt $c ]
then
        max=$b
else
        max=$c
fi
echo " maximum value= " $max
```

```
#Input marks of 3 subjects and calculate Sum, Average and
Grade(Marksheet).

#!/bin/bash

echo -n "Enter marks of three subjects: "
read m1 m2 m3

sum=`expr $m1 + $m2 + $m3`
echo "Total marks= " $sum

avg=`expr $sum / 3`
echo " Average marks : " $avg

if [ $avg -gt 100 -o $avg -lt 0 ]
then
        grade="Invalid Marks."
elif [ $avg -ge 90 -a  $avg -le 100 ]
then
        grade='O'
elif [ $avg -ge 80 -a $avg -lt 90 ]
then
        grade='E'
elif [ $avg -ge70 -a $avg -lt 80 ]
then
        grade='A'
elif [ $avg -ge 60 -a $avg -lt 70 ]
then
```

```
        grade='B'
elif [ $avg -ge 50 -a $avg -lt 60 ]
 then
        grade='C'
elif [ $avg -ge 40 -a $avg -lt 50 ]
then
        grade='C'
else
        grade='F'
fi
echo "Grade is = " $grade
```

```
#Mark sheet

#!/bin/bash
echo -n "Enter marks of three subjects: "
read m1 m2 m3
sum=`expr $m1 + $m2 + $m3`
avg=`expr $sum / 3`
if [  $avg -lt 0 -o  $avg -gt 100  ]
then
        grade="Invalid Marks."
elif [ $avg -le 100 -a  $avg -gt 90 ]
then
        grade='O'
elif [ $avg -le 90 -a $avg -gt 80 ]
then
        grade='E'
elif [ $avg -le 80 -a $avg -gt 70 ]
then
        grade='A'
elif [ $avg -lt 70 -a $avg -ge 60 ]
then
        grade='B'
elif [ $avg -lt 60 -a $avg -ge 50 ]
then
        grade='C'
elif [ $avg -lt 50 -a $avg -gt 40 ]
then
        grade='C'
else
        grade='F'
fi
echo "Total marks= " $sum
echo " Average marks : " $avg
echo "Grade is = " $grade
```

(d)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
```