What is Script ?

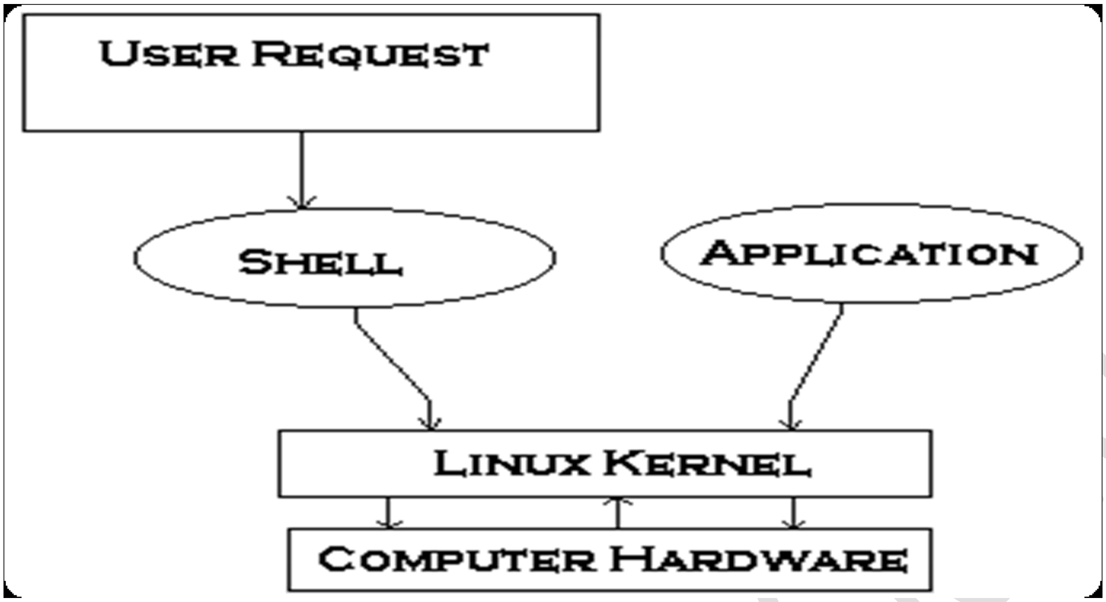
- A **scripting** or **script language** is a programming language for a special run-time environment that automates the execution of tasks.
- the tasks could alternatively be executed one-by-one by a human operator using CLI on command prompt screen.

What is Shell Scripting ?

- shell script is a computer program designed to be run by the Unix/Linux shell.
- *Shell Script is **series of command** written **in plain text file**. Shell script is just like batch file is MS-DOS*

Why we use shell scripting ?

1. Shell scripts can be used to prepare input files, job monitoring, and output processing.
2. Useful to create own commands.
3. Save lots of time on file processing.
4. To automate some task of day to day life.
5. System Administration part can be also automated.
6. Daily application backup and monitoring
7. Reduce admin work load
8. Can manage lot of server from any single place using shell script
9. Smart work ….etc



What is Shell ?

- Shell accepts your instruction or commands in English (mostly) and if it is a valid command, it is pass to kernel.

- Shell Basically is an interface between user and kernel.

- Shell is an environment in which we can run our commands, programs, and shell scripts.

- A Shell provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input.

- Shell reads your input after you press Enter. It determines the command you want executed by looking at the first word of your input.

- 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 is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

Types of Supported Shell in Linux ?

- Bourne shell (sh) ➔ /bin/sh
- Bourne Again shell (bash) ➔ /bin/bash
- Korn shell (ksh) ➔ /bin/ksh
- C shell (csh) ➔ /bin/csh
- Turbo C shell (tcsh) ➔ /bin/tcsh

```
# which   sh

# which bash

# which perl

# which python

# echo $SHELL

# echo $BASH

# echo   $BASH_VERSION
```

What is Kernel ?

- Kernel is an heart of Linux OS.
- The **Linux kernel** is the main component of a **Linux** operating system (OS) and is the core interface between a computer's hardware and its processes.
- It is an interface between Application and computer hardware as well as shell.
- It manages resource of Linux OS. Resources means facilities available in Linux.
- 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

What the kernel does

- **Memory management:** Keep track of how much memory is used to store what, and where
- **Process management:** Determine which processes can use the central processing unit (CPU), when, and for how long
- **Device drivers:** Act as mediator/interpreter between the hardware and processes
- **System calls and security:** Receive requests for service from the processes

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

How we can check the kernel version ?

```
# uname

# uname   -r

#   ls /boot
```

Echo command ?

It is use to print any message on the screen ?

```
# echo   hello
# echo   “hello”
```

My first Shell Scripting

```
[root @ krnetworkcloud ~] # vim    demo.sh
```

```
#! /bin/bash
# My first shell script
clear
echo “welcome to KR Network Cloud”
```

```
:wq
```

```
# sh demo.sh          OR          # ./demo.sh
# chmod    +x      demo.sh
```

Variable

Variables are symbolic names that represent values stored in memory.

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

1- System defined variables (SDV)

- 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.

Examples of variables ?

Echo command Advance Options:

Options	Description
-n	do not print the trailing newline.
-e	enable interpretation of backslash escapes.
\b	Backspace Remove the spance between words
\\	backslash
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

```
# echo    “Hello KR Network Cloud”

# x=10

# echo $x
```

The ‘-e‘ option in Linux acts as interpretation of escaped characters that are backslashed.

```
  # echo -e "KR \bis \ba \bcenter \bof \bLinux \btechnology"

# echo -e    "KR \nis \na \ncenter \nof \nLinux \ntechnology"

# echo -e "KR \tis \ta \tcenter \tof \tLinux \tTechnology"

# echo -e "\n\tKR \n\tis \n\ta \n\tcenter \n\tof \n\tLinux \n\tTechnology"

# echo -e "\vKR \vis \va \vcenter \vof \vLinux \vTechnology"

# echo -e "\n\vKR \n\vis \n\va \n\vcenter \n\vof \n\vLinux \n\vTechnology"

# echo -e "KR \ris a center of Technology"

# echo -e "KR is a center \cof Linux Technology"

# echo -n    "KR is a center of Linux Technology"

# echo -e    "KR is a center of \aLinux Technology"

# echo    *
```

****

Example-1

```
#!/bin/bash
echo "Printing text"
echo -n "Printing text without newline"
echo -e "\nRemoving \t special \t characters\n"
```

Example-2

```
#!/bin/bash
# Adding two values
((sum=25+35))
#Print the result
echo $sum
```

Example-3

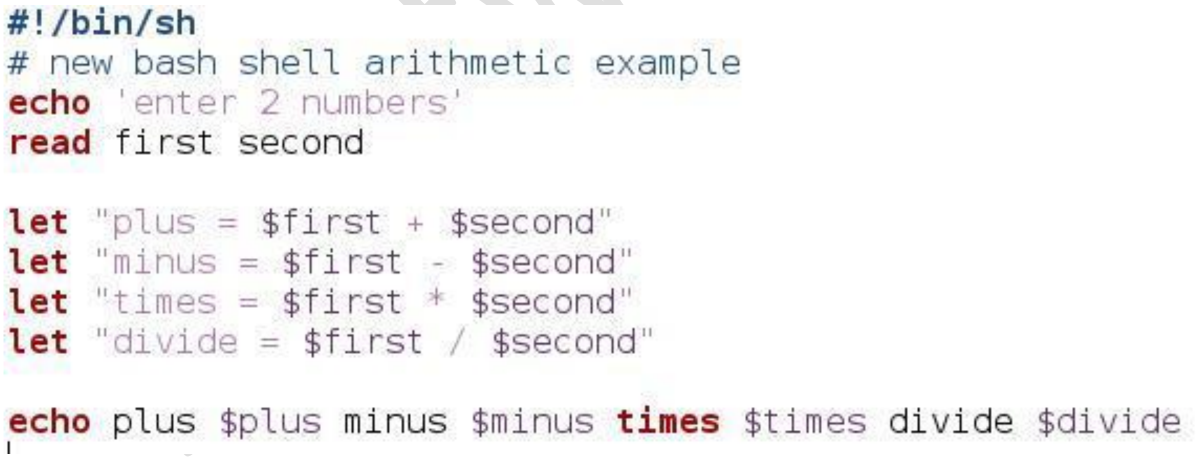
```
#!/bin/bash
echo -n "Enter Something:"
read something
echo "You Entered: $something"
```

What is the use LET command in shell scripting ?

let is a builtin function of Bash that allows us to do simple arithmetic. It follows the basic format:

Example:-1

```
#!/bin/bash

# Basic arithmetic using let

let a=5+4

echo $a

let "a = 5 + 4"

echo $a # 9

let a++

echo $a # 10

let "a = 4 * 5"

echo $a # 20
```



```
#!/bin/sh
# new bash shell arithmetic example
echo 'enter 2 numbers'
read first second

let "plus = $first + $second"
let "minus = $first - $second"
let "times = $first * $second"
let "divide = $first / $second"

echo plus $plus minus $minus times $times divide $divide
```

what is the use EXPR command in shell ?

- expr is similar to let except instead of saving the result to a variable it instead prints the answer. Unlike let you don't need to enclose the expression in quotes.
- You also must have spaces between the items of the expression.

- It is also common to use expr within command substitution to save the output to a variable.

Syntax:

```
expr item1 operator item2
```

Example:

```
#!/bin/bash

# Basic arithmetic using expr

expr 5 + 4

expr "5 + 4"

expr 5+4

expr 5 \* $1

expr 11 % 2

a=$( expr 10 - 3 )

echo $a # 7
```

Double Parentheses

Double Parentheses Example:

```
#!/bin/bash
# Basic arithmetic using double parentheses
a=$(( 4 + 5 ))
echo $a # 9
a=$((3+5))
echo $a # 8
b=$(( a + 3 ))
echo $b # 11
b=$(( $a + 4 ))
echo $b # 12
(( b++ ))
echo $b # 13
(( b += 3 ))
echo $b # 16
a=$(( 4 * 5 ))
echo $a # 20
```

Exit Status

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Example-1

```
# cat test.sh
```

```
#!/bin/bash
echo "This is a test."
# Terminate our shell script with success message
exit 0
```

```
# sh    test.sh
```

```
# echo $?
```

Example-2

```
#!/bin/bash
echo "This is a test."
# Terminate our shell script with failure message
exit 1
```

```
# sh test.sh
```

```
# echo $?
```

Example-3

```
#!/bin/bash
echo hello
echo $?
lskdf
echo $?
echo
exit 113
```

Example-4

```
#!/bin/sh
cp /foo /bar && echo Success || echo Failed
```

Example-5

```
#!/bin/sh
# First attempt at checking return codes
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
if [ "$?" -ne "0" ]; then
    echo "Sorry, cannot find user ${1} in /etc/passwd"
    exit 1
fi
NAME=`grep "^${1}:" /etc/passwd|cut -d":" -f5`
HOMEDIR=`grep "^${1}:" /etc/passwd|cut -d":" -f6`
echo "USERNAME: $USERNAME"
echo "NAME: $NAME"
echo "HOMEDIR: $HOMEDIR"
```

Uniq Command ?

Uniq command in unix or linux system is used to suppress the duplicate lines from a file. It discards all the successive identical lines except one from the input and writes the output.

The syntax of uniq command is

```
# uniq      [option]     filename
```

The options of uniq command are:

- c : Count of occurrence of each line.
- d : Prints only duplicate lines.
- D : Print all duplicate lines
- f : Avoid comparing first N fields.
- i : Ignore case when comparing.
- s : Avoid comparing first N characters.
- u : Prints only unique lines.
- w : Compare no more than N characters in lines

```
# cat    example.txt
```

```
Unix operating system
unix operating system
unix dedicated server
linux dedicated server
```

Example-1 Suppress duplicate lines

```
# uniq  example.txt
```

Example-2 Count of lines

```
#  uniq -c example.txt
```

Example-3 Display only duplicate lines.

You can print only the lines that occur more than once in a file using the -d option.

```
#   uniq -d example.txt

#   uniq -D example.txt
```

The -D option prints all the duplicate lines.

Example-4 Skip first N fields in comparison.

The -f option is used to skip the first N columns in comparison. Here the fields are delimited by the space character.

```
#  uniq -f2 example.txt
```

Example-5 Print only unique lines.

You can skip the duplicate lines and print only unique lines using the -u option

```
#   uniq -u example.txt
```

Example-6 Ignore Case

```
# cat example.txt
Hello
hello
How are you?
How are you?
Thank you
thank you
```

```
# uniq   example.txt
# uniq   -i   example.txt
```

Example-7   Ignore characters

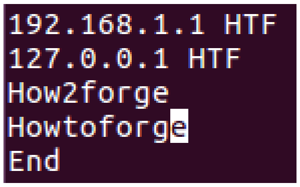
In order to ignore few characters at the beginning you can use -s parameter, but you need to specify the number of characters you need to ignore

```
# cat     example.txt
1apple
2apple
3pears
4banana
5banana
# uniq -s 1 file1
1apple
3pears
4banana
```

Example-8 Checking a Certain Number of Characters

By default, `uniq` checks the entire length of each line. If you want to restrict the checks to a certain number of characters, however, you can use the `-w` (check chars) option.

In this example, we'll repeat the last command, but limit the comparisons to the first three characters. To do so, we type the following command:



```
192.168.1.1 HTF
127.0.0.1 HTF
How2forge
Howtoforge
End
```

```
# uniq -w 3  example.txt
```

Since first 3 characters of the third and fourth lines are same, so these lines were considered as repeated. Hence, only third one is displayed in the output.

Example-9

```
# uniq      --all-repeated=prepend    example.txt
```

Example-10   How to make uniq avoid comparing first few fields

Sometimes, depending on the situation, the similarity of two lines is defined by a small part of those lines

```
#    cat example.txt
```

192.168.0.1 HTF

127.0.01 HTF

Linux FF

Android FF

```
#    uniq     -f    1      example.txt
```

Conditional Statement ?

if else statements are useful decision-making statements which is use to verify whether this statement or condition is true or false.

The if...else statements

Shell supports following forms of **if…else** statement −

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

1- Syntax if ….fi

```
If   [ condition   ]
then
      {

      Set of command OR Do this

      }
fi
```

2- Syntax if ….else..fi

```
if   [   condition ]
then
      {
        Do this   OR set of commands
        }
else
     {
      Do this
      }
fi
```

3- Syntax if …elif ..elif …else..fi

```
if   [ condition1 ]
then
     {
     Do this
      }
elif [ condition2 ]
then
     {
       Do this
     }
elif   [ condition3 ]
```

```
then
    {
    Do this
     }
…..
….
….
else
     {
       Do this
      }
fi
```

Note: What is the meaning of if [condition] in if statement

if [condition or expr]

Note: test command use to pass the expression in condition.

Syntax:

test expr OR [ ] ←==== It is equivalent of test expr command

Example:

if [$a -lt 50] OR if test $a -lt 50

Both syntax as same

Types of Test command operator use in if statement ?

We will now discuss the following operators −

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

1- Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

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

Show Examples

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=(Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]** is correct whereas, **[$a==$b]** is incorrect.

2- Relational Operators

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.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

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 <= $b] is correct whereas, **[$a <= $b]** is incorrect.

3- Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

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

Show Examples

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

4- String Operators

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then −

Show Examples

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.

5- File Test Operators

We have a few operators that can be used to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on −

Show Examples

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.

-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Example of Every Operator with if statement ?

Example-1

```
#!/bin/bash
echo -n "Enter a number: "
read num
if [ $num -gt 10 ]
then
echo "Number is greater than 10."
fi
```

Example-2

```
#!/bin/bash
echo -n "Enter first number:"
read x
echo -n "Enter second number:"
read y
(( sum=x+y ))
echo "The result of addition=$sum"
```

Example-3

```
#!/bin/bash
read n
if [ $n -lt 10 ];
then
echo "It is a one digit number"
else
echo "It is a two digit number"
fi
```

Example-4

```
#!/bin/bash
echo -n "Enter Number:"
read num
if [[ ( $num -lt 10 ) && ( $num%2 -eq 0 ) ]]; then
echo "Even Number"
else
echo "Odd Number"
fi
```

Example-5

```
#!/bin/bash

echo -n "Enter any number:"

read n

if [[ ( $n -eq 15 || $n -eq 45 ) ]]

then

echo "You won"

else

echo "You lost!"

fi
```

Example-6

```
#!/bin/bash

echo -n "Enter a number: "

read num

if [[ $num -gt 10 ]]

then

echo "Number is greater than 10."

elif [[ $num -eq 10 ]]

then

echo "Number is equal to 10."

else

echo "Number is less than 10."

fi
```

Example-7

```
#!/bin/bash

echo "Enter username"

read username

echo "Enter password"

read password

if [[ ( $username == "admin" && $password == "secret" ) ]]; then

echo "valid user"

else

echo "invalid user"

fi
```

Example-8

```
#!/bin/bash

echo "Enter any number"

read n

if [[ ( $n -eq 15 || $n   -eq 45 ) ]]

then

echo "You won the game"

else

echo "You lost the game"

fi
```

Example-9

```
#!/bin/bash

echo "Enter your lucky number"

read n

if [ $n -eq 101 ];

then

echo "You got 1st prize"

elif [ $n -eq 510 ];

then

echo "You got 2nd prize"

elif [ $n -eq 999 ];

then

echo "You got 3rd prize"

else

echo "Sorry, try for the next time"

fi
```

Check Whether You’re Root

Example-10

```
#!/bin/bash

ROOT_UID=0

if [ "$UID" -eq "$ROOT_UID" ]

then

echo "You are root."

else

echo "You are not root"

fi
```

Example-11

Removing Duplicate Lines from Files

```
#! /bin/sh

echo -n "Enter Filename-> "

read filename

if [ -f "$filename" ]; then

sort $filename | uniq | tee sorted.txt

else

echo "No $filename in $pwd...try again"

fi
```

Nested if examples:

Example-12

```
#!/bin/bash
# Nested if statements
if [ $1 -gt 100 ]
then
echo Hey that\'s a large number.
if (( $1 % 2 == 0 ))
then
echo And is also an even number.
fi
fi
```

example-13

```
#!/bin/bash
echo -n "Enter the first number: "
read VAR1
echo -n "Enter the second number: "
read VAR2
echo -n "Enter the third number: "
read VAR3
if [[ $VAR1 -ge $VAR2 ]]
then
  if [[ $VAR1 -ge $VAR3 ]]
  then
     echo "$VAR1 is the largest number."
  else
    echo "$VAR3 is the largest number."
  fi
else
  if [[ $VAR2 -ge $VAR3 ]]
  then
    echo "$VAR2 is the largest number."
  else
    echo "$VAR3 is the largest number."
  fi
fi
```

Example-14

```
#!/bin/bash
#Initializing two variables
a=10
b=20
#Check whether they are equal
if [ $a == $b ]
then
    echo "a is equal to b"
fi
#Check whether they are not equal
if [ $a != $b ]
then
    echo "a is not equal to b"
fi
```

Example-15

```
#!/bin/bash
value=$( grep -ic   "sachin" /etc/passwd )
if [ $value -eq 1 ]
then
   echo "I found sachin"
else
   echo "I didn't find sachin"
fi
```

Example-16

```
#!/bin/bash
value=$( grep -ic "sachin" /etc/passwd )
if [ $value -eq 1 ]
then
   echo "I found one sachin”
elif [ $value -gt 1 ]
then
   echo "I found multiple sachin"
else
   echo "I didn't find any sachin"
fi
```

Example-17

```
#!/bin/bash

echo "This scripts checks the existence of the messages file."

echo "Checking..."

if [ -f /var/log/messages ]

   then

      echo "/var/log/messages exists."

fi

echo

echo "...done."
```

Example-18

```
#!/bin/bash

grep $USER /etc/passwd

if [ $? -ne 0 ]

then

echo "not a local account"

fi
```

Example-19

This script is executed by cron every Sunday. If the week number is even, it reminds you to put out the garbage cans:

```
#!/bin/bash

# Calculate the week number using the date command:

WEEKOFFSET=$[ $(date +"%V") % 2 ]

# Test if we have a remainder.   If not, this is an even week so send a message.

# Else, do nothing.

if [ $WEEKOFFSET -eq "0" ]; then

echo "Sunday evening, put out the garbage cans." | mail -s "Garbage cans out"   root@localhost

fi
```

Example-20

```
#!/bin/bash

if [ "$(whoami)" != 'root' ]

then

        echo "You have no permission to run $0 as non-root user."

fi
```

Example-21

```
#!/bin/bash
# This script does a very simple test for checking disk space.
space=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 | cut -d "%" -f1 -`
alertvalue="80"
if [ "$space" -ge "$alertvalue" ]
then
   echo "At least one of my disks is nearly full!" | mail -s "daily diskcheck"   root
else
   echo "Disk space normal" | mail -s "daily diskcheck" root
fi
```

Example-22

```
#!/bin/bash
# This script will test if we're in a leap year or not.
year=`date +%Y`
if [ $[$year % 400] -eq "0" ]; then
   echo "This is a leap year.   February has 29 days."
elif [ $[$year % 4] -eq 0 ]; then
           if [ $[$year % 100] -ne 0 ]; then
              echo "This is a leap year, February has 29 days."
           else
              echo "This is not a leap year.   February has 28 days."
           fi
else
   echo "This is not a leap year.   February has 28 days."
fi
```

Example-23

```
#!/bin/bash
read -p "Enter value of i :" i
read -p "Enter value of j :" j
read -p "Enter value of k :" k

if [ $i -gt $j ]
then
    if [ $i -gt $k ]
    then
        echo "i is greatest"
    else
        echo "k is greatest"
    fi
else
    if [ $j -gt $k ]
    then
        echo "j is greatest"
    else
  echo "k is greatest"
    fi
fi
```

Example-24

```
#!/bin/bash
echo "Enter a Natural Number :"
read n
i=$(expr $n % 2)
if [ $i -eq 0 ]
then
    echo "Its Even!"
else
    echo "Its Odd!"
fi
```

Example-25

```
#!/bin/bash
if [ $1 -ge 18 ]
then
echo You may go to the party.
elif [ $2 == 'yes' ]
then
echo You may go to the party but be back before midnight.
else
echo You may not go to the party.
fi
```

Sleep Command :-

The sleep command allows your shell script to pause between instructions. It is useful in a number of scenarios such as performing system-level jobs.

Example-1

```
#!/bin/bash

echo "How long to wait?"

read time

sleep $time

echo "Waited for $time seconds!"
```

Example-2

```
#!/bin/bash

for i in 1 2 3 4 5

do

echo   “hello KR Network”

sleep 5
```

```
done
```

Debuging A Bash Script

Bash scripting provides an option to debug your script at runtime. You using "**set -xv**" command inside shell script or using -xv on command line while executing script.

Syntax:

```
# sh   -xv   script.sh
```

OR

Example – Enable Debug in Script

This is useful to enable debugging for some part of the script.

```
#!/bin/bash

set -xv     # this line will enable debug

cd /var/log/

for i in "*.log"; do

  du -sh $i

done
```

```
# sh   myscript.sh
```

Bash Exit Codes

The exit code is a number between 0 and 255. This is the value returns to parent process after completion of a child process. In other words, it denotes the exit status of the last command our function.

The exit code value return based on a command or program will successfully execute or not.

- **Success –** A zero (0) value represents success.
- **failure –** A non-zero exit-code represents failure.

Example -1

```
#!/bin/bash

echo "hi" > /tmp/tesfile.txt

if [ $? -eq 0 ]; then

  echo "Hurrey. it works"

else

  echo "Sorry, can't write /tmp/tesfile.txt"

fi
```

Example-2

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

```
STRING="root"

if grep ${STRING} /etc/passwd

then

   echo "Yeah! string found"

else

   echo "Ooooh, no matching string found"

fi
```

Include Files in Bash

Similar to other programming languages which allow to include other files to a file, Bash scripting also allows to include (source) another shell script file to script file.

For example, to include config.sh script to current script file use following syntax, where config.sh is available in the same directory of the current script.

For this example, First, I have created a data.sh file with the following content.

```
#!/bin/bash

id=100

name="KR Network"
```

Now create another file show.sh, which includes data.sh file.

```
#!/bin/bash

source data.sh
```

```
echo "Welcome $name"

echo "Your id is $id"
```

Execute script show.sh in a terminal and view the results:

```
#sh   show.sh

Welcome KR Network

Your id is 100
```

User Input In Bash

Linux **read** command is used for interactive user input in bash scripts. This is helpful for taking input from users at runtime.

Syntax:

```
read [options] variable_name
```

Example-1

```
#!/bin/bash

echo "Enter your name:"

read myname

echo "Hello" $myname
```

Example-2

```
#!/bin/bash

read -p "Enter your username: " myname

read -sp "Enter your password: " mypassword
```

```
echo -e "\nYour username is $myname and Password is $mypassword"
```

Note:

Use -s to input without displaying on the screen. This is helpful to take password input in the script.

Shell Commands

Basically, a shell script is a collection of shell commands. You can simply run any bash shell command inside a shell script.

Example-1

```
#VAR=`date +"%d%b%Y"`

# echo $VAR
```

Example-2

```
#   VAR=`date +"%d%b%Y"`

#   mkdir /backup/db/$VAR

#    ls   /backup/db

              OR      we can run it in same like this


# mkdir /backup/db/`date +"%d%b%Y"`
```

Bash Arithmetic Operations Example

Example-1

```
#!/bin/bash

read -p "Enter numeric value: " n1

read -p "Enter numeric value: " n2

echo "Addition of $n1 + $n2 is          = " $((n1+n2))

echo "Subtraction of $n1 - $n2 is       = " $((n1-n2))

echo "Division of $n1 / $n2 is              = " $((n1/n2))

echo "Multiplication of $n1 * $n2 is      = " $((n1*n2))

echo "Modulus of $n1 % $n2 is              = " $((n1%n2))
```

Increment and Decrement Operator:

Bash also used increment (++) and decrement (–) operators.

Both uses in two types pre-increment/post-increment and pre-decrement/post-decrement.

```
## Post-increment example

$ var=10

$ echo $((var++))   ## First print 10 then increase value by 1
```

```
## Pre-increment example
$ var=10

$ echo $((++var))   ## First increase value by 1 then print 11
```

```
## Post-increment example

$ var=10

$ echo $((var++))   ## First print 10 then increase value by 1
```

```
## Pre-increment example

$ var=10

$ echo $((++var))   ## First increase value by 1 then print 11
```

What is the use of Loop in shell Scripting ?

shell scripts also support for loops to do the repetitive tasks.

Types Of   Loop :-

1- FOR Loop

2- SELECT Loop

3- While Loop

4- Until Loop

What is the use of   FOR LOOP ?

- A ‘for loop’ is a bash programming language statement which allows code to be repeatedly executed.
- What it does is say for each of the items in a given list, perform the given set of commands.

Syntax number-1

```
for VARIABLE in 1 2 3 4 5 .. N
do
    {
        command1
        command2
        commandN
    }
done
```

Syntax number-2

```
for VARIABLE in file1 file2 file3
do
        command1 on $VARIABLE
        command2
        commandN
done
```

Syntax Number-3

```
for OUTPUT in $(Linux-Or-Unix-Command-Here)
do
        command1 on $OUTPUT
        command2 on $OUTPUT
        commandN
done
```

What is SELECT Loop ?

The **select** mechanism allows you to create a simple menu system. It has the following format:

Syntax:-

```
select var in value1 value2 value3…. Value-n
do
<commands>
done
```

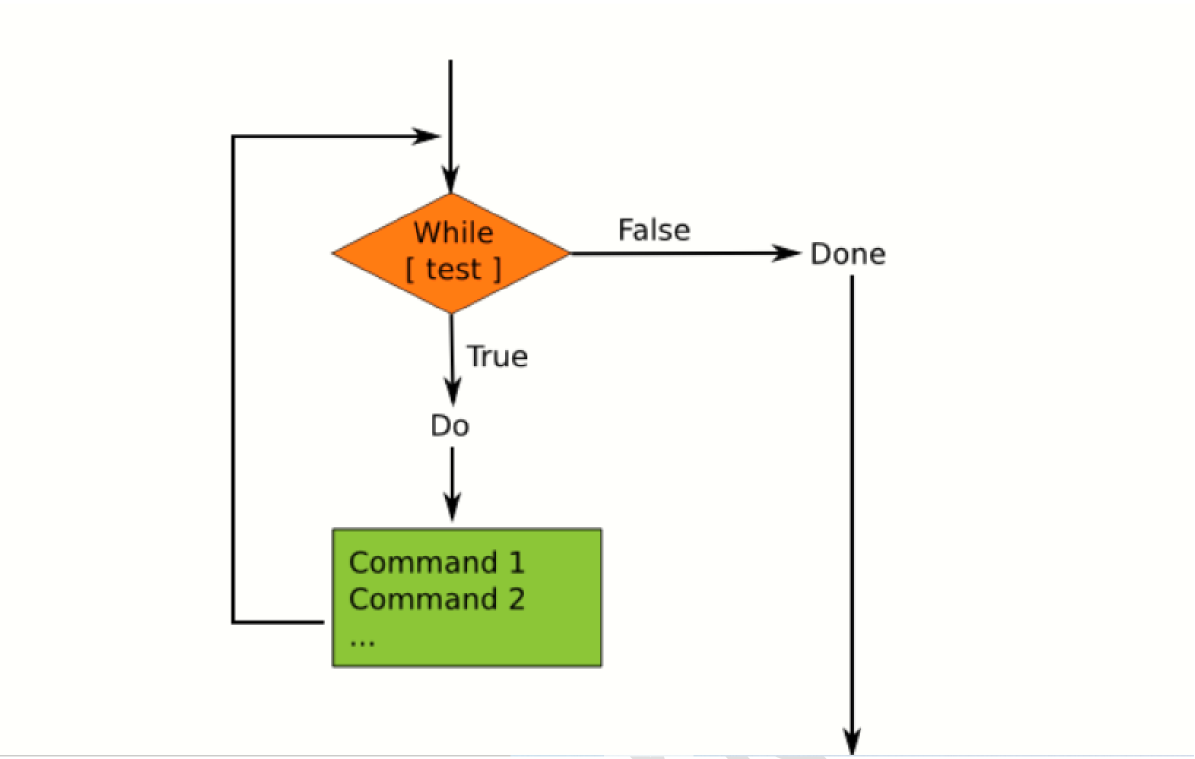
What is While Loop ?

While Loop say, while an expression is true, keep executing these lines of code.

They have the following format:

Syntax:

```
while [ expression ]
do
<set of commands>
done
```



What is the use of UNTIL LOOP ?

The difference is that it will execute the commands within it until the test becomes true.

Syntax:

```
until [ expression ]

do

<set of commands>

done
```

Examples Of FOR Loop ?

Example-1

```
#!/bin/bash
for i in 1 2 3 4 5
do
    echo "$i"
done
```

Example-2

```
#!/bin/bash
for i in 1 2 3 4 5
do
   {
     echo   “hello”
   }
done
```

Example-3

```
#!/bin/bash
for day in SUN MON TUE WED THU FRI SUN
do
    echo "$day"
done
```

Example-4

```
#!/bin/bash
for ((i=1; i<=10; i++))
do
  echo "numbers are = $i"
done
```

Example-5

```
#!/bin/bash
for   filename   in   *
do
  ls -l   $filename
done
```

Example-6

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

```
for color in Blue Green Pink White Red
do
echo "Color = $color"
done
```

Example-7

```
ColorList=("Blue Green Pink White Red")
for color in $ColorList
do
if [ $color == 'Pink' ]
then
echo "My favorite color is $color"
fi
done
```

Example-8

```
for (( n=1; n<=5; n++ ))
do
if (( $n%2==0 ))
then
echo "$n is even"
else
echo "$n is odd"
fi
done
```

Example-9

```
#!/bin/bash

for i in $(seq 1 2 20)

do

    echo "Welcome $i times"

done
```

Example-10

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

```
for (( ; ; ))
do
     echo "infinite loops [ hit CTRL+C to stop]"
done
```

Example-11

```
i=1
for username in `awk -F: '{print $1}' /etc/passwd`
do
  echo "Username $((i++)) : $username"
done
```

Example-12

```
#!/bin/bash
for a in 1 2 3 4 5 6 7 8 9 10
do
    if [ $a == 5 ]
    then
        break
    fi
    echo "Iteration no $a"
```

```
done
```

Example-13

```
for a in 1 2 3 4 5 6 7 8 9 10
do
    if [ $a == 5 ]
    then
        continue
    fi
    echo "Iteration no $a"
done
```

Example-14

```
i=1
for day in Mon Tue Wed Thu Fri Sat Sun
do
  echo -n "Day $((i++)) : $day"
  if [ $i -eq 7 -o $i -eq 8 ]; then
      echo " (WEEKEND)"
      continue;
  fi
  echo " (weekday)"
done
```

Examples of Select Loop:-

Example-1

```
#!/bin/bash

select i in red green yellow blue

do

   echo   “you have select   $i   color”

done
```

Example-2

```
#!/bin/bash

echo "Which is Your Favorite Linux Distribution..?"

select os in Ubuntu LinuxMint CentOS RedHat Fedora

do

      echo "I also like $os !"

done
```

Example-3

```
#!/bin/bash
PS3=’Please enter a number from above list: ‘
select var1 in abc ced efg hij
do
echo “Present value of var1 is $var1”
done
```

If you see you are prompted with a prompt: ‘#?’, This is default prompt used by select which is assigned to PS3 variable.

Examples of While Loop :-

Example-1

```
#!/bin/bash

num=1

while [ $num -le 5 ]

do

    echo "$num"

    ((num++))

done
```

Example-2 Infinite Loop

```
#!/bin/bash

while true

do

   echo "Press CTRL+C to Exit"

done
```

Example-3

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

```
# Basic while loop

counter=1

while [ $counter -le 10 ]

do

echo $counter

((counter++))

done

echo All done
```

Example-4

```
#!/bin/bash

INPUT_STRING=hello

while [ "$INPUT_STRING" != "bye" ]

do

   echo "Please type something in (bye to quit)"

   read INPUT_STRING

   echo "You typed: $INPUT_STRING"

done
```

Example-5

```
a=0
while [ $a -lt 10 ]
do
    echo $a
    a=`expr $a + 1`
done
```

Until Loop Examples:

Example-1

```
#!/bin/bash
# Basic until loop
counter=1
until [ $counter -gt 10 ]
do
echo $counter
((counter++))
done
echo All done
```

Example-2

```
#!/bin/bash
i=0
until [ $i -gt 20 ]
do
    i=$(expr $i + 1)
    j=$(expr $i % 2)
    if [ $j -ne 0 ]
    then
        continue
    fi
    echo "$i"
done
```

Function in Linux ?

They are particularly useful if you have certain tasks which need to be performed several times.

Instead of writing out the same code over and over you may write it once in a function then call that function every time.

A **function** is a group of commands that are assigned a name that acts like a handle to that group of commands.

To execute this group of commands defined in the **function**, you simply call the **function** by the name you provided.

Syntax-1 Bash Function Declaration

```
functionname  ()  {

  commands

}
```

OR

Syntax-2

```
function   functionname   {

                          Commands

                          }
```

How to Call Function in program ?

```
functionname
```

****

Example-1

```
#!/bin/bash

hello_world () {

                echo 'hello, world'

                }

 hello_world
```

Example-2

```
#!/bin/bash

lines_in_file () {

                cat $1 | wc -l

                   }

num_lines=$( lines_in_file $1 )

echo The file $1 has $num_lines lines in it.


~ ] # sh   hello.sh   myfile.txt
```

Example-3

```
#!/bin/bash

function Add()   {

echo -n "Enter a Number: "

read x

echo -n "Enter another Number: "

read y

echo "Adiition is: $(( x+y ))"

}

Add
```

Return Values

- Most other programming languages have the concept of a return value for functions, a means for the function to send data back to the original calling location.
- Bash functions don't allow us to do this. They do however allow us to set a return status.
- Similar to how a program or command exits with an exit status which indicates whether it succeeded or not.
- We use the keyword return to indicate a return status.

Example-1

```
#!/bin/bash

# Setting a return status for a function

print_something () {

echo Hello $1

return 5

}

print_something Mars

print_something Jupiter

echo The previous function has a return value of $?
```

Example-2

```
#!/bin/bash

# Setting a return value to a function

lines_in_file () {

cat $1 | wc -l

}

num_lines=$( lines_in_file $1 )

echo The file $1 has $num_lines lines in it.
```

Note: create any file with content to run this program.

```
# cat   myfile.txt

Tamato

Banana

Apple

Mango

# sh   hello.sh     myfile.txt          ==> will give the exact output
```

Next Example

```
#cat    function.sh
```

```
#!/bin/bash
f1 () {
        echo "Hello $name"
}

f2 () {
        echo "Enter your name: "
        read name
        f1
}
f2
```

Nested Function

```
#!/bin/sh

# Calling one function from another

number_one   ()   {

     echo "This is the first function speaking..."

     number_two

}

number_two   ()   {

     echo "This is now the second function speaking..."

}

# Calling function one.

number_one
```

BASH SHELL SCRIPT TO CHECK RUNNING PROCESS

```
#!/bin/bash

SERVICE="httpd"

if pgrep -x "$SERVICE" >/dev/null

then

      echo "$SERVICE is running"

else

      echo "$SERVICE stopped"

fi
```

Case Statement

- The `case` statement executes any one block of commands, based on the outcome of a pattern match.
- We have a variable that stores a value to be matched and a number of patterns in the order they are arranged, which may or may not be regular expressions, against which the value is matched
- The case statement is useful and processes faster than an else-if ladder. Instead of checking all if-else conditions, the case statement directly select the block to execute based on an input

Syntax : Case without Loop

```
case $varName in

    pattern1)

        Block of Commands

        ;;

    pattern2)

        Block of Commands

        ;;

    patternN)

        Block of Commands

        ;;

    *)  Default Block of Commands

        ;;

esac
```

Example-1

```
#!/bin/bash
echo "Which is your Favorite Operating System..?"
read os
case $os in
    "Linux") echo "Woww!! I am also a Linux Fan!!"   ;;
    "Mac") echo "You must be very Rich!"   ;;
    "Windows") echo "You Should Try Linux Once.. You would love it!" ;;
    *) echo "I've never used that one!"
esac
```

Example-2

```
#!/bin/bash
echo "Which is your Favourite Operating System..?"
read os
case $os in
    [lL]inux) echo "Woww!! I am also a Linux Fan!!" ;;
    [mM]ac) echo "You must be very Rich!" ;;
    [wW]indows) echo "You Should Try Linux Once.. You would love it!" ;;
      *) echo "I've never used that one!"
esac
```

Example-3

```
#!/bin/bash

echo "Which is your Favorite Operating System..?"

read os

case $os in

   "Linux" | "Ubuntu" | "Linux Mint" | "CentOS") echo "Woww!! I am also a Linux Fan!!"   ;;

    "Mac") echo "You must be very Rich!" ;;

    "Windows") echo "You Should Try Linux Once.. You would love it!" ;;

    *) echo "I've never used that one!"

esac
```

Example-4

```
#!/bin/bash

echo "Which Operating System Do You Use..?"

PS3="Enter your choice (must be a number): "

select os in Ubuntu LinuxMint Windows8 Windows7 WindowsXP Mac

do

    case $os in

        "Ubuntu" | "LinuxMint")   echo "I also use $os ..!”;;

        "Windows8" | "Windows7" | "WindowsXP") echo "Why don't you try Linux..?" ;;

      "Mac")   echo "You must be Very Rich..!" ;;

        *)   echo "Invalid option. Program will exit now."

            break ;;

    esac

done
```

Example-5

```
#!/bin/bash
read -p "Enter your choice [yes/no]:" choice
case $choice in
     yes)   echo "Thank you"
              echo "Your type: Yes" ;;
      no)   echo "Ooops"
              echo "You type: No” ;;
     *)    echo "Sorry, invalid input" ;;
esac
```

Example-6

```
#!/bin/bash

read -p "Enter your choice [yes/no]:" choice

case $choice in
     Y/y/Yes/YES/yes)
       echo "Thank you"
       echo "Your type: Yes"
       ;;
     N/n/No/NO/no)
       echo "Ooops"
       echo "You type: No"
       ;;
     *)
       echo "Sorry, invalid input"
       ;;
esac
```

Example-7

```
#!/bin/bash
FRUIT="kiwi"
case "$FRUIT" in
        "apple") echo "Apple pie is quite tasty." ;;
        "banana") echo "I like banana nut bread." ;;
        "kiwi") echo "New Zealand is famous for kiwi." ;;
esac
```

Example-8

```
#!/bin/bash

# if no command line arg given

# set rental to Unknown

if [ -z $1 ]

then

   rental="*** Unknown vehicle ***"

elif [ -n $1 ]

then

# otherwise make first arg as a rental

   rental=$1

fi

case $rental in

    "car") echo "For $rental rental is Rs.20 per k/m.";;

    "van") echo "For $rental rental is Rs.10 per k/m.";;

    "jeep") echo "For $rental rental is Rs.5 per k/m.";;

    "bicycle") echo "For $rental rental 20 paisa per k/m.";;

    "enfield") echo "For $rental rental Rs.3   per k/m.";;

    "thunderbird") echo "For $rental rental Rs.5 per k/m.";;

    *) echo "Sorry, I can not get a $rental rental   for you!";;

esac
```

```
# sh     hello.sh   car
```

Example-9

```
#!/bin/bash

NOW=$(date +"%a")

case $NOW in

          Mon)    echo "Full backup";;

          Tue|Wed|Thu|Fri)   echo "Partial backup";;

          Sat|Sun)       echo "No backup";;

          *) ;;

esac
```

Example-10

```
#!/bin/bash

case $1 in

start)   echo starting ;;

stop)   echo stopping ;;

restart)   echo restarting ;;

*) echo don\'t know ;;

esac
```

```
# sh   case.sh    start
```

Example-11

```
#!/bin/bash

# Print a message about disk utilization.

space_free=$( df -h | awk '{ print $5 }' | sort -n | tail -n 1 | sed 's/%//' )

case $space_free in

[1-5]*) echo Plenty of disk space available ;;

[6-7]*) echo There could be a problem in the near future ;;

8*) echo Maybe we should look at clearing out old files ;;

9*) echo We could have a serious problem on our hands soon ;;

*) echo Something is not quite right here ;;

esac
```

Example-12

```
#!/bin/sh

while read f

do

   case $f in

           hello)           echo English ;;

           howdy)           echo American       ;;

           gday)            echo Australian     ;;

           bonjour)         echo French ;;

           "guten tag")     echo German;;

            *)       echo Unknown Language: $f ;;

esac

done < myfile
```

Example-13

```
#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)

input=/home/$user

output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

tar -czf $output $input

echo "Backup of $input completed! Details about the output backup file:"

ls -l $output
```

Shell Colors: colorizing shell scripts

Shell scripts commonly used ANSI escape codes for colour output. Following table shows Numbers representing colours in Escape Sequences.

Color	Foreground	Background
Black	30	40
Red	31	41
Green	32	42
Yellow	33	43
Blue	34	44
Magenta	35	45
Cyan	36	46
White	37	47

ANSI CODE	Meaning
0	Normal Characters
1	Bold Characters
4	Underlined Characters
5	Blinking Characters
7	Reverse video Characters

```
# echo -e "\033[COLORm Sample text"

# echo -e "\033[32m Hello World"
             or
# printf "\e[32m Hello World"
```

Examples

The Wait Command

The wait command is used for pausing system processes from Linux bash scripts. Check out the following example for a detailed understanding of how this works in bash.

Example-1

```
#!/bin/bash

echo "Testing wait command"

sleep 5 &

pid=$!

kill $pid

wait $pid

echo $pid was terminated.
```

Input, Output and Error Redirections

Example-1

```
#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

tar -czf $output $input 2> /dev/null
echo "Backup of $input completed! Details about the output backup file:"
ls -l $output
```

Example-2

```
#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)

input=/home/$user

output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

# The function total_files reports a total number of files for a given
directory.

function total_files {

        find $1 -type f | wc -l

}

# The function total_directories reports a total number of directories

# for a given directory.

function total_directories {

        find $1 -type d | wc -l

}
```

```
tar -czf $output $input 2> /dev/null

echo -n "Files to be included:"

total_files $input

echo -n "Directories to be included:"

total_directories $input

echo "Backup of $input completed!"

echo "Details about the output backup file:"

ls -l $output
```

```
#!/usr/bin/bash

/usr/bin/expect <<EOD

spawn /usr/bin/scp -l 200 file.ext   user@server://path/on/server/

expect "password:"

send "YourPassword\r"

expect "\r"

send "\r\n"

EOD
```

Command Line Arguments in Bash

You can pass command line arguments to bash shell script. These are helpful to make a script more dynamic. Learn more about Bash command arguments.

```
#!/bin/bash

DBS=`mysql -uroot   -e"show databases"`

for b in $DBS

do

mysql -uroot -e"show tables from $b"

done
```

https://tecadmin.net/tutorial/bash-scripting/bash-debugging/

http://www.yourownlinux.com/2016/12/bash-scripting-arrays-examples.html