

## -----Shell Scripting-----

- 1) Shell is responsible to read commands/applications provided by user.
- 2) Shell will check whether command is valid or not and whether it is properly used or not. If everything is proper then shell interpretes (converts) that command into kernel understandable form. That interpreted command will be handover to kernel.
- 3) Kernel is responsible to execute that command with the help of hardware.

Shell acts as interface between user and kernel.

shell+kernel is nothing but operating system.

## 1) Bourne Shell:

- > It is developed by Stephen Bourne.
- > This is the first shell which is developed for UNIX.
- > By using sh command we can access this shell.

## 2) BASH Shell:

- > BASH ? Bourne Again SHell.
- > It is advanced version of Bourne Shell.
- > This is the default shell for most of the linux flavours.
- > By using bash command we can access this shell.

## 3) Korn Shell:

- > It is developed by David Korn.
- > Mostly this shell used in IBM AIX operating system.
- > By using ksh command, we can access this shell.

## 4) CShell:

- > Developed by Bill Joy.
- > C meant for California University.
- > It is also by default available with UNIX.
- > By using csh command, we can access this shell.

## 5) TShell:

- > T means Terminal.
- > It is the advanced version of CShell.
- > This is the most commonly used shell in HP UNIX.
- > By using tcsh command, we can access this shell.

## 6) Z Shell:

- > Developed by Paul.
- > By using zsh command we can access this shell.

**Note:** The most commonly used shell in linux environment is BASH. It is more powerful than remaining shells.

## How to Check Default Shell in our System?

```
jitendra@DESKTOP-ACUC1TV:~$ echo $0
-bash
jitendra@DESKTOP-ACUC1TV:~$ echo $SHELL
/bin/bash
jitendra@DESKTOP-ACUC1TV:~$ cat /etc/passwd
```

3rd way is to check passwd file where we have information about the Users

## How to check all available Shells in our System?

/etc/shells file contains all available shells information.

```
jitendra@DESKTOP-ACUC1TV:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/usr/bin/sh
/bin/dash
/usr/bin/dash
/usr/bin/tmux
/usr/bin/screen
```

## How to Switch to other Shells?

Based on our requirement we can switch from one shell to another shell.

```
jitendra@DESKTOP-ACUC1TV:~$ sh
$ echo $0
sh
$ exit
jitendra@DESKTOP-ACUC1TV:~$ dash
$ echo $0
dash
$ exit
jitendra@DESKTOP-ACUC1TV:~$ sh
$ bash
jitendra@DESKTOP-ACUC1TV:~$
```

## What is Shell Script:

A sequence of commands saved to a file and this file is nothing but shell script.

Inside shell script, we can also use programming features like conditional statements, loops, functions etc. Hence we can write scripts very easily for complex requirements also.

Best suitable for automated tasks.

## How to write and run Shell Script:

### Step - 1: Write script:

We can write our script using cat command, gedit, vi or any other editors like Vscode etc

### Step - 2: Provide execute permissions to the script:-

Using chmod command we can change permissions

### Step - 3: Run the script:-

We can run the script in multiple ways

```

jitendra@PC:~$ #step 1
jitendra@PC:~$ cat>script
echo "hello world"
jitendra@PC:~$ #step 2
jitendra@PC:~$ chmod u+x script
jitendra@PC:~$ #step3
jitendra@PC:~$ ./script
hello world
jitendra@PC:~$ bash script
hello world
jitendra@PC:~$ sh script
hello world
jitendra@PC:~$

```

**Note:** The default shell is bash. Hence bash is responsible to execute our script. Instead of bash, if we want to use Bourne shell then we need to execute the Script using sh

## Importance of Sha-Bang:

By using sha-bang, we can specify the interpreter which is responsible to execute the script.

? # Sharp  
? ! Bang

#! ? Sharp Bang or Shabang or Shebang

#! /bin/bash ? It means the script should be executed by bash

#! /bin/sh ? It means the script should be executed by Bourne Shell

#! /usr/bin/python3 ? It means the script should be executed by Python3 interpreter

If we write shabang in our script at the time of execution, we are not required to provide command to execute and we can execute script directly.

```
jitendra@PC:~/class$ cat code.py
#!/bin/python3
n = int(input("enter a number "))
if n%2==0:
    print("even")
else:
    print("odd")
jitendra@PC:~/class$ ./code.py
enter a number 4
even
jitendra@PC:~/class$
```

## Shell Variables :-

Variables are place holders to hold values.

Variables are key-value pairs.

In Shell programming, there are no data types. Every value is treated as text type/ String type.

All variables are divided into 2 types

- 1) Environment variables / predefined variables
- 2) User defined variables

### 1)Environment Variables:

These are predefined variables and mostly used internally by the system. Hence these variables also known as System variables.

But based on our requirement, we can use these variables in our scripts.

We can get all environment variables information by using either env command or set command.

```
jitendra@PC:~$ env
SHELL=/bin/bash
WSL2_GUI_APPS_ENABLED=1
WSL_DISTRO_NAME=Ubuntu
NAME=PC
PWD=/home/jitendra
LOGNAME=jitendra
HOME=/home/jitendra
LANG=C.UTF-8
```

A simple shell script that uses environment variables

```
jitendra@PC:~$ cat script
#!/bin/bash
echo "User Name: $USER"
echo "User Home Directory: $HOME"
echo "Default Shell Name: $SHELL"
jitendra@PC:~$ ./script
User Name: jitendra
User Home Directory: /home/jitendra
Default Shell Name: /bin/bash
jitendra@PC:~$
```

## 2) User defined Variables:

Based on our programming requirement, we can define our own variables also.

```
jitendra@PC:~$ cat script
#!/bin/bash
name="jitendra singh"
echo $name
```

```
jitendra@PC:~$ ./script
jitendra singh
jitendra@PC:~$
```

### Rules to define Variables:

- 1) It is recommended to use UPPER CASE characters.
- 2) If variable contains multiple words, then these words should be separated with \_ symbol.
- 3) Variable names should not start with digit.  
\$ 123A=40  
123A=40: command not found
- 4) We should not use special symbols like -, @, # etc

### How to define Readonly Variables:

We can define read only variables by using readonly keyword.

```
$A=100
```

```
$readonly A
```

```
$A=300
```

```
bash: A: readonly variable
```

If the variable is readonly then we cannot perform reassignment for that variable. It will become constant

### Variable Substitution:

Accessing the value of a variable by using \$ symbol is called variable substitution. Syntax:

```
$variablename
```

```
${variablename}
```

Recommended to use \${variablename}



```

jitendra@PC:~$ cat script
#!/bin/bash
a=10
b=20
COURSE="linux"
ACTION="SLEEP"
echo "Values of a and b are: $a and $b"
echo "My Course is: ${COURSE}"
echo "Your Favourite Action: $ACTIONING"
echo "Your Favourite Action: ${ACTION}ING"
jitendra@PC:~$ ./script
Values of a and b are: 10 and 20
My Course is: linux
Your Favourite Action:
Your Favourite Action: SLEEPING
jitendra@PC:~$

```

## Command Substitution:

We can execute command and we can substitute its result based on our requirement by using command substitution.

Syntax:

**Old style:** `command` These are backquotes but not single quotes

**New Style:** \$(command) Ex

1:-

```

jitendra@PC:~$ echo "Today date is: `date +%D`"
Today date is: 02/26/23
jitendra@PC:~$ echo "Today date is: $(date +%D)"
Today date is: 02/26/23
jitendra@PC:~$

```

Ex 2:-

```

jitendra@PC:~$ echo "my current working directory is $(pwd)"
my current working directory is /home/jitendra
jitendra@PC:~$ echo "my current working directory is 'pwd'"
my current working directory is 'pwd'
jitendra@PC:~$ echo "my current working directory is `pwd`"
my current working directory is /home/jitendra
jitendra@PC:~$

```

Do not get confused between single quotes and backquotes

# Command Line Arguments :-

The arguments which are passing from the command prompt at the time of executing our script, are called command line arguments.

**\$ ./test.sh learning linux is very easy**

The command line arguments are learning, linux, is, very, easy.  
Inside script we can access command line arguments as follows:

\$# ? Number of Arguments (5)

\$0 ? Script Name (./test.sh)

? \$1 1st Argument (learning)  
? \$2 2nd Argument (linux)  
? \$3 3rd Argument (is)  
? \$4 4th Argument (very)  
? \$5 5th Argument (easy)  
? \$\* All Arguments (learning Linux is very easy)  
? \$@ All Arguments (learning Linux is very easy)

\$? ? Represents exit code of previously executed command or script.

```
jitendra@PC:~$ cat test.sh
```

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

```
echo "number of arguments : $#"
```

```
echo "Script name : $0"
```

```
echo "first argument : $1"
```

```
echo "second argument : $2"
```

```
echo "third argument : $3"
```

```
echo "fourth argument : $4"
```

```
echo "fifth argument : $5"
```

```
echo "Total arguments : $*"
```

```
jitendra@PC:~$ ./test.sh My Name is Jitendra Singh
```

```
number of arguments : 5
```

```
Script name : ./test.sh
```

```
first argument : My
```

```
second argument : Name
```

```
third argument : is
```

```
fourth argument : Jitendra
```

```
fifth argument : Singh
```

```
Total arguments : My Name is Jitendra Singh
```

## Difference between \$@ and \$\*:

**\$@** ? All command line arguments with space separator

"\$1" "\$2" "\$3" ...

**\$\*** ? All command line arguments as single string

"\$1c\$2c\$3c.."

Where c is the first character of the Internal Field Separator (IFS). The default first character is space.

### How to Check Default IFS:

```
$ set | grep "IFS" IFS=$'\n'
```

We can set IFS during script

```
#!/bin/bash
IFS="-"
echo "number of arguments : $#"
```

```
echo "Script name : $0"
```

```
echo "first argument : $1"
```

```
echo "second argument : $2"
```

```
echo "third argument : $3"
```

```
echo "fourth argument : $4"
```

```
echo "fifth argument : $5"
```

```
echo "Total arguments seperated by IFS: $*"
echo "Total arguments seperated by space : $@"
```

```
jitendra@PC:~$ ./test.sh My Name is Jitendra Singh
number of arguments : 5
Script name : ./test.sh
first argument : My
second argument : Name
third argument : is
fourth argument : Jitendra
fifth argument : Singh
Total arguments seperated by IFS: My-Name-is-Jitendra-Singh
Total arguments seperated by space : My Name is Jitendra Singh
jitendra@PC:~$
```

**Q1. shell script to check length of given command line argument**

```
jitendra@PC:~$ cat test.sh
#!/bin/bash

len=$(echo -n "$1" | wc -c)
echo "The length of given string is $1 : $len"
jitendra@PC:~$ ./test.sh "jitendra"
The length of given string is jitendra : 8
jitendra@PC:~$ ./test.sh jitendra
The length of given string is jitendra : 8
jitendra@PC:~$
```

Another method

```
jitendra@PC:~$ cat test.sh
#!/bin/bash

first=$1
len=${#1}
echo "The length of given string is $1 : $len"
jitendra@PC:~$ ./test.sh jitendra
The length of given string is jitendra : 8
jitendra@PC:~$
```

#var        > will give length of the variable value  
#1         > will give length of first command line argument

Doing all the tasks in a single line using variable substitution

```
jitendra@PC:~$ cat test.sh
#!/bin/bash
echo "The length of given string is $1 : ${#1}"
jitendra@PC:~$ ./test.sh jitendra
The length of given string is jitendra : 8
jitendra@PC:~$
```

## How to Read Dynamic Data from the User

Using read keyword we can read the dynamic data from the user

```
jitendra@PC:~$ read a b
100 200
jitendra@PC:~$ echo "a = $a, b = $b"
a = 100, b = 200
jitendra@PC:~$
```

What if I give more than 2 values

```
jitendra@PC:~$ read a b
100 200 300 400
jitendra@PC:~$ echo "a = $a, b = $b"
a = 100, b = 200 300 400
jitendra@PC:~$
```

**With Prompt Message:**

Approach-1

```
jitendra@PC:~$ cat test.sh
#!/bin/bash
echo "Enter A Value:"
read A
echo "Enter B Value:"
read B
echo "A Value: $A"
echo "B Value: $B"
jitendra@PC:~$ ./test.sh
Enter A Value:
100
Enter B Value:
200
A Value: 100
B Value: 200
jitendra@PC:~$
```

## Approach-2

Using read -p p >

prompt

```
jitendra@PC:~$ cat test.sh
#!/bin/bash
read -p "Enter A Value:" A
read -p "Enter B Value:" B
echo "A Value: $A"
echo "B Value: $B"
jitendra@PC:~$ ./test.sh
Enter A Value:100
Enter B Value:200
A Value: 100
B Value: 200
jitendra@PC:~$
```

**read -s** > it hides the input on screen which is provided by end user

```
jitendra@PC:~$ cat test.sh
#!/bin/bash
read -p "Enter username:" username
read -s -p "Enter password:" password
echo ""
echo "username is: $username"
echo "password is : $password"
jitendra@PC:~$ ./test.sh
Enter username:jitendra
Enter password:
username is: jitendra
password is : 12345
jitendra@PC:~$
```

**Q. Write a shell script which will take input data of an employee**

**R. and add that to a file**

eid , ename, esal, erole

```
jitendra@PC:~$ cat code.sh
```

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

```
read -p "Enter employee id :" eid
```

```
read -p "Enter employee name :" ename
```

```
read -p "Enter employee salary :" esal
```

```
read -p "Enter employee role :" erole
```

```
echo "Below details are saved to the file"
```

```
echo "$eid:$ename:$esal:$erole"
```

```
echo "$eid:$ename:$esal:$erole">>empsdata.txt
```

**Q. Write a Script to Read File Name from the End User  
and display its Content?**

```
jitendra@PC:~$ cat code.sh
```

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

```
read -p "Enter filename to display content: " fname
```

```
echo "-----"
```

```
cat $fname
```

```
echo "-----"
```

```
jitendra@PC:~$ ./code.sh
```

```
Enter filename to display content: empsdata.txt
```

```
-----
```

```
1:jitendra:10k:trainer
```

```
2:kunal:20k:sde
```

```
-----
```

```
jitendra@PC:~$
```

**Q Write a Script to Read File Name from the End User and Remove  
Blank Lines Present in that File?**

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

```
read -p "Enter any File name to remove blank lines:" fname grep -v
```

```
"^$" $fname > temp.txt
```



```
mv temp.txt $fname
echo "In $fname all blank lines deleted"
```

## Write a Script to Read File Name from the End User and Remove Duplicate Lines Present in that File?

```
#!/bin/bash
read -p "Enter any File name to remove duplicate lines:" fname
sort -u $fname > temp.txt
mv temp.txt $fname
echo "In $fname all duplicate lines deleted"
```

## -----Operators-----

?

?

?

?

?

### 1)Arithmetic Operators

- + Addition
- Substraction
- \* Multiplication
- / Division
- % Modulo Operator

### 2)Relational Operators: (Numeric Comparison Operators)

?

?

?

?

?

?

- gt Greater than
- ge Greater than or equal to
- lt Less than
- le Less than or equal to
- eq Is equal to
- ne Not equal to

?

?

?

### 3)Logical Operators:

- a Logical AND
- o Logical OR
- ! Logical Not

### 4)Assignment operator =

**Note:** Except assignment operator, for all operators we have to provide space before and after operator.

## How to perform Mathematical Operations:

There are multiple ways

1) By using expr keyword

- 2) By using let keyword
- 3) By using (( ))
- 4) By using []

### 1) By using expr Keyword:

expr means expression

```
jitendra@PC:~$ cat code.sh
#!/bin/bash
read -p "Enter First Number:" a
read -p "Enter First Number:" b
sum=$(expr $a + $b)
echo $sum
jitendra@PC:~$ ./code.sh
Enter First Number:3
Enter First Number:5
8
```

**Note:** While using expr keyword, \$ symbol is mandatory. Space must be required before and after + symbol.

### 2) By using let Keyword:

```
let sum=a+b
echo "The Sum: $sum"
let sum=$a+$b
echo "The Sum: $sum"
```

Here \$ symbol is optional.

But we should not provide space before and after +

### 3) By using (( ))

```
jitendra@PC:~$ cat code.sh
#!/bin/bash
read -p "Enter First Number:" a
read -p "Enter First Number:" b
sum=$((a+b))
echo $sum
jitendra@PC:~$ ./code.sh
Enter First Number:3
Enter First Number:4
7
jitendra@PC:~$
```

Here space and \$ symbol, both are optional.

#### 4)By using []:

```
jitendra@PC:~$ cat code.sh
#!/bin/bash
read -p "Enter First Number:" a
read -p "Enter First Number:" b
sum=$((a+b))
echo "The Sum: $sum"
sum=$((a+b))
echo "The Sum: $sum"

jitendra@PC:~$ ./code.sh
Enter First Number:3
Enter First Number:5
The Sum: 8
The Sum: 8
jitendra@PC:~$
```

Here also space and \$ symbol, both are optional.

**Note:-** All the above 4 approaches will work only for integral arithmetic

(only for integer numbers).

If we want to perform floating point arithmetic then we should use **bc** command.

```
jitendra@PC:~$ cat code.sh
#!/bin/bash
read -p "Enter First Number:" a
read -p "Enter First Number:" b
sum=$(echo $a+$b | bc)
echo "The Sum: $sum"
echo "The difference is $(echo $a-$b | bc)"
echo "The Product is $(echo $a*$b | bc)"
jitendra@PC:~$ ./code.sh
Enter First Number:3.7
Enter First Number:43.2
The Sum: 46.9
The difference is -39.5
The Product is 159.8
jitendra@PC:~$
```

**Q. Write a Script to Read 4 Digit Integer Number and Print the Sum of Digits Present in that Number?**

```
jitendra@PC:~$ cat code.sh
#!/bin/bash
read -p "Enter First Number:" n
a=$(echo $n | cut -c 1)
b=$(echo $n | cut -c 2)
c=$(echo $n | cut -c 3)
d=$(echo $n | cut -c 4)

echo "sum of digits is : ${a+b+c+d}"
jitendra@PC:~$ ./code.sh
Enter First Number:5464
sum of digits is : 19
jitendra@PC:~$
```

**Q Write a Script to Read Employee Monthly Salary and Print his Bonus. The Bonus should be 20% of Annual Salary**

```
jitendra@PC:~$ cat code.sh
#!/bin/bash
read -p "Enter your monthly salary" sal
yearly=$(echo "12*$sal" | bc)
bonus=$(echo "${yearly}*20/100" | bc)
echo "your bonus is $yearly"
```

## -----Control Statements-----

- 2) case statement
- 3) while loop
- 4) for loop
- 5) until loop
- 6) break
- 7) continue
- 8) exit



# 1)if Statement:

There are 4 types of if statements

- a) simple if
- b) if-else
- c) nested if
- d) ladder if

## a) simple if:-

```
if [ condition ]  
    then  
        action  
fi
```

If condition is true then only action will be executed.

Q1. write a script to read name from the end user and if name is "jitendra" then display a special message

## b) if -else:

```
if [ condition] then  
    action if condition is true  
else  
    action if condition is false fi
```

## c) Nested if:

```

if [ condition ]
then
    .....
    .....
    if [ condition ] then
        .....
        .....
else fi
.....
.....

else fi
.....

```

#### d) ladder -if:

```

if [condition] then
    action-1
elif [ condition ]
then
    action-2
elif [ condition ]
then
    action-3 else
    default action fi

```

**Q. WAS to find greater of 2 numbers**

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

```
read -p "Enter first number " a
read -p "Enter Second number " b

```

```

if [ $a -gt $b ]
then
    echo "$a is greater"
else
    echo "$b is greater"
fi

```

**Q. Write a Script to Check whether Numbers OR Equal OR Not.  
If the Numbers are not Equal then Print Greater Number?**



```
1 #!/bin/bash
2 read -p "Enter first number " a
3 read -p "Enter Second number " b
4 if [ $a -eq $b ]
5 then
6 echo "both are equal"
7 elif [ $a -gt $b ]
8 then
9 echo "$a is greater"
10else
11echo "$b is greater"
12fi
13
```

**Q. write a script to find greater of 3 numbers**

**first method**

```
#!/bin/bash
read -p "Enter first number " a
read -p "Enter Second number " b
read -p "Enter third number " c
if [ $a -gt $b ]
then
    if [ $a -gt $c ]
    then
        echo "$a is greater"
    else
        echo "$c is greater"
    fi
elif [ $b -gt $c ]
then
    echo "$b is greater number"
else
    echo "$c is greater number"
fi
```

**second method**

```
1 #!/bin/bash
2 read -p "Enter first number " a
3 read -p "Enter Second number " b
4 read -p "Enter third number " c
5 if [ $a -eq $b -a $a -gt $c ]
6 then
7 echo "$a is greater number"
8 elif [ $b -gt $c ]
9 then
10 echo "$b is greater number"
11 else
12 echo "$c is greater number"
13 fi
```

**Q. write a script to read marks of 3 subjects if any of subject**

**R. have marks less than 33 then print student failed**

**method 1**

```
1 #!/bin/bash
2 read -p "Enter first number " a
3 read -p "Enter Second number " b
4 read -p "Enter third number " c
5
6 if [ $a -gt 33 -a $b -gt 33 -a $c -gt 33 ]
7 then
8 echo "passed"
9 else
10 echo "failed"
11 fi
```

It should be -ge

**method 2**

```
1 #!/bin/bash
2 read -p "Enter first number " a
3 read -p "Enter Second number " b
4 read -p "Enter third number " c
5
6 if [ $a -lt 33 ]
7 then
8 echo "failed"
9 elif [ $b -lt 33 ]
10 then
11 echo "failed"
12 elif [ $c -lt 33 ]
13 then
14 echo "failed"
15 else
16 echo "passed"
17 fi
```

### Exit Codes/ Status Codes:

Every command and script return some value after execution, which indicates that whether it is successfully executed or not. This return value is called exit code or status code.

We can find exit code by using "\$?".

zero means command/script executed successfully.

non-zero means command/script not executed successfully.

```
jitendra@PC:~$ echo $0
-bash
jitendra@PC:~$ echo $?
0
jitendra@PC:~$ echopico
echopico: command not found
jitendra@PC:~$
jitendra@PC:~$ echo $?
127
jitendra@PC:~$ echo $?
0
jitendra@PC:~$
```

**Q. Write a Script that takes 2 Integer Numbers as Command Line Arguments and Prints Sum**

**If Number of Arguments is not 2, then we have to get Message saying "You should provide only 2 Arguments"**

**If the Arguments are not Integer Numbers then we have to get Message saying "You should provide Integer Numbers only"**



```

1 #! /bin/bash
2 if [ $# -ne 2 ]
3 then
4 echo "you should provide exactly two args"
5 exit 1
6 fi
7 x=$1
8 y=$2
9 sum=$(( expr $x + $y ))
10 if [ $? -ne 0 ]
11 then
12 echo "you should provide ints only"
13 exit 2
14 else
15 echo "the sum is $sum"
16 fi

```

**Write a Script that Reads an Integer Number and Check whether the given Number is +ve Number OR -ve Number?**

**Write a Script that Reads an Integer Number and Checks whether it is Even Number OR not?**

**Write a Script that Reads an Integer Number and Checks whether it is 3 Digit Number OR not?**

### **File Test Options :-**

?

-e Returns true if file/directory exists

?

-s Returns true if the file is non-empty

?

?

- f Returns true if the file is a regular file
- d Returns true if the file is a directory

- ❓ -l Returns true if the file is link file
- ❓ -b Returns true if the file is block special file
- ❓ -c Returns true if the file is character special file
  
- ❓ -r Returns true if current user has read permission on the file
- ❓ -w Returns true if current user has write permission on the file
- ❓ -x Returns true if current user has execute permission on the file
- ❓ -o Returns true if current user is owner of the file.
- ❓ file1 -ot file2 Returns true if file1 is older than file2 (last modified time) file1
- ❓ -nt file2 Returns true if file1 is newer than file2(last modified time)

Q. Script to Test whether the given File is Regular File OR Directory?

```

1  #!/bin/bash
2  read -p "Enter File Name to test:" fname
3  if [ -e $fname ]; then
4      if [ -f $fname ]; then
5          echo "It is regular file"
6      elif [ -d $fname ]; then
7          echo "It is Directory file"
8      else
9          echo "It is special file"
10     fi
11 else
12     echo "$fname does not exist"
13 fi
14

```

**Write a Script that Reads a File Name and Display its Content to the Terminal**



```
1 #!/bin/bash
2 read -p "Enter File Name to test:" fname
3 if [ -e $fname ]; then
4     if [ -f $fname ]; then
5         if [ -r $fname ]; then
6             cat $fname
7         else
8             echo "User not having Read permission"
9         fi
10    else
11        echo "It is not a regular file"
12    fi
13else
14    echo "$fname does not exist"
15fi
16
```

**q. Write a Script that Reads File Name and Check whether it is Empty File OR not?**



```
1 #!/bin/bash
2 read -p "Enter File Name to test:" fname
3 if [ -e $fname ]; then
4     if [ -f $fname ]; then
5         if [ -s $fname ]; then
6             echo "$fname is not empty file"
7         else
8             echo "$fname is empty file"
9         fi
10    else
11        echo "It is not a regular file"
12    fi
13 else
14     echo "$fname does not exist"
15 fi
16
```

**Q write a script to check wheather 2 files are same or not**



```
1 #!/bin/bash
2 read -p "Enter file1: " file1
3 read -p "Enter file2: " file2
4
5 result=$(cmp $file1 $file2)
6 if [ -z "$result" ]
7 then
8 echo "The given 2 files have same content"
9 else
10echo "given 2 files have different content"
11fi
12
```

q Write a Script that Accepts a File Name and Display User Permissions?



```
1  #! /bin/bash
2  read -p "Enter First File Name: " fname
3  READ=NO
4  WRITE=NO
5  EXECUTE=NO
6  if [ -r $fname ]; then
7  READ=YES
8  fi
9  if [ -w $fname ]; then
10 WRITE=YES
11 fi
12 if [ -x $fname ]; then
13 EXECUTE=YES
14 fi
15 echo "User Permissions Summary"
16 echo "_____"
17 echo "Read Permission: $READ"
18 echo "Write Permission: $WRITE"
19 echo "Execute Permission: $EXECUTE"
20
```

Write a Script that Reads File Name and Remove the specified File?

```
1 #! /bin/bash
2 read -p "Enter file/directory name to delete:" fname
3 if [ -e $fname ]; then
4   rm -r $fname
5   echo "$fname removed successfully"
6 else
7   echo "$fname does not exist"
8 fi
9
```

## Mini Application:

**Copy all files and directories present in the first directory to the second directory. We should create compressed tar file and have to move that tar file.**

**After moving tar file to the second directory, extract all files and directories and remove that tar file.**

Tests to Perform:

- 1) The number of command line arguments should be 2
- 2) The source and destination directories should be available already
- 3) The source and destination arguments should be directories
- 4) The user should have read and execute permissions on source directory
- 5) The user should have write and execute permissions on destination directory
- 6) All error messages should be sent to error file and the file name should contain timestamp.
- 7) All intermediate steps should be displayed to the terminal

## String Test Options:

- 1) `str1 = str2` ? Returns true if both strings are same



2) `str1 != str2` ¶ Returns true if both strings are different

- 3) `-z str` Returns true if the string is empty
- 4) `-n str` Returns true if the string is not empty
- 5) `str1 > str2` Returns true if the str1 is alphabetically greater than str2
- 5) `str1 < str2` Returns true if the str1 is alphabetically less than str2

## 2)case Statement:

If multiple options are available then it is not recommended to use nested if-else statements. It increases length of the code and reduces readability.

To handle such type of requirements we should go for case statement

Syntax:

```
case $variable in option1
)
action-1
;;
option2 )
action-2
;;
option3 )
action-3
;;
* )
default action
;;
esac
```

### Note:

- 1) space is optional while defining options.
- 2) `;;` can be used to come out of case statement.
- 3) `;;` is mandatory for all options except for default option.
- 4) If we take default option at the beginning, then for any input the same default option will be executed.

Q. take a digit from 0 to 9 and print the digit number



```
#!/bin/bash
read -p "Enter any digit from 0 to 9 : " n
case $n in
0)
echo "Zero"
echo "Zero again"
;;
1) echo "One" ;;
2) echo "Two" ;;
3) echo "Three" ;;
4) echo "Four" ;;
5) echo "Five" ;;
6) echo "Six" ;;
7) echo "Seven" ;;
8) echo "Eight" ;;
9) echo "Nine" ;;
*) echo "Please enter a digit from 0 to 9 only"
esac
```


Q read a character from user input then check whether it is digit, alphabet or special symbol



```
#!/bin/bash
read -p "Enter any character to check " n
case $n in
[A-Za-z])
echo "It is an Alphabet symbol"
;;
[0-9])
echo "it is a digit"
;;
# [^a-zA-Z0-9])
*)
echo "It is a Special Symbol"
;;
esac
```

^ symbol is used for exclude

**Q. Write a Script that Accepts a Single Character and Checks whether it is Digit OR Special Character OR Vowel OR Consonant?**



```
#!/bin/bash
read -p "Enter Any Character to check: " ch
case $ch in
    [^a-zA-Z0-9])
        echo "It is a Special Character"
        ;;
    [0-9]) echo "It is a Digit"; echo "digit"s
        ;;
    [aeiouAEIOU])
        echo "It is a Vowel"
        ;;
    [^aeiouAEIOU])
        echo "It is a Consonent"
        ;;
    *)
        echo "Enter only one character"
esac
```

if we write multiple comands in same line by seperating using ;

## Iterative Statements:

If we want to execute a group of commands multiple times then we should go for iterative statements.

There are 3 types of iterative statements

- 1) while loop
- 2) until loop
- 3) for loo

### 1) **while Loop:**

If we don't know the number of iterations in advance, then we should go for while

loop.

Syntax:

```
while [ condition ]
```

```
do body
```

```
done
```

As long as condition is true,

then body will be executed. Once condition fails then only loop

will be terminated

## Q. Write a Script to Print Numbers from 1 to 10

```
js@Jitendra:~$ cat code.sh
```

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

```
i=1
```

```
while [ $i -ne 10 ]
```

```
do
```

```
echo $i
```

```
i=$((i+1))
```

```
done
```

```
js@Jitendra:~$ ./code.sh
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

Use echo -n if you want to print in same line

**Write a Script to generate Numbers until Limit which is provided by End User?**

```
1 #!/bin/bash
2 read -p "Enter Limit:" n
3 i=1
4 while [ $i -le $n ]
5 do
6 echo $i
7 sleep 2
8 let i++
9 done
```

**Q3) Write a Script to find the Sum of First n Integers, where n is provided by End User**

```
1 #! /bin/bash
2 read -p "Enter n value:" n
3 i=1
4 sum=0
5 while [ $i -le $n ]
6 do
7 let sum=sum+i
8 let i++
9 done
10 echo "The Sum of first $n numbers: $sum"
11
```

Write a Script to Display Timer (Digital Timer)



```

1 #!/bin/bash
2 while [ true ]
3 do
4   clear
5   printf "\n\n\n\n\n\n\t\t\t\t$(date +%H:%M:%S)"
6   sleep 1
7 done
8
9

```

Note: To use escape characters like \n and \t, we should not use echo and we should use printf command

Note: true and false are keywords which represents boolean values.

## break Statement

Based on some condition, if we want to break loop execution (i.e to come out of loop) then we should go for break statement

```

1 #!/bin/bash
2 i=1
3 while [ $i -le 10 ]
4 do
5   if [ $i -eq 5 ]; then
6     break
7   fi
8   echo $i
9   let i++
10 done
11

```

**Write a script to run timer till given hour and minute**

```
1 #!/bin/bash
2 while [ true ]
3 do
4   clear
5   printf "\n\n\n\n\n\n\n\t\t\t\t\t$(date +%H:%M:%S)"
6   sleep 1
7   h=$(date +%H)
8   m=$(date +%M)
9   if [ $h -eq 8 -a $m -eq 54 ]; then
10    break
11  fi
12 done
13
```

## continue Statement:

We can use continue statement to skip current iteration and continue for the next iteration.

```
1 #!/bin/bash
2 i=0
3 while [ $i -lt 10 ]
4 do
5   let i++
6   if [ ${i%2} -eq 0 ]; then
7     continue
8   fi
9   echo $i
10 done
11
```

```
js@Jitendra:~$ ./code.sh
```

```
1
```

```
3
```


```
5
```

```
7
```

```
9
```

```
js@Jitendra:~$
```

**Write a Script to Read File Name and Display its Content?**



```
1  #!/bin/bash
2
3  while [ true ]
4  do
5
6  read -p "enter file name to display data : " fname
7
8  if [ -f $fname ]
9  then
10 echo "_____ "
11 echo "The content of $fname"
12 echo "_____ "
13 cat $fname
14 else
15 echo "$fname does not exist"
16 fi
17
18 read -p "do you want to display another file [Yes|No]: " option
19 echo $option
20 case $option in
21 [Yy][eE][sS])
22 echo $option
23 ;;
24
25 [nN][oO])
26 break
27 ;;
28 esac
29
30 done
31 echo "thanks for using app"
32
```

**Take a string as an input from user and print  
it's reverse**



```
1  #! /bin/bash
2
3  read -p "Enter the string to reverse: " str
4
5  len=$(echo -n $str | wc -c)
6  ans=""
7
8  while [ $len -gt 0 ]
9  do
10 ch=$(echo -n $str | cut -c $len)
11 ans=$ans$ch
12 len=$((len-1))
13 done
14
15 echo "The original string $str"
16 echo "the reversed string $ans"
17
18 if [ $str = $ans ]
19 then
20 echo "palindrome"
21 fi
22
```

```
1  #! /bin/bash
2
3  read -p "Enter the string to reverse: " str
4
5  if [ $str = $(echo $str | rev) ]
6  then
7  echo "palindrome"
8  fi
9
```

## 2) Until loop :-

It is opposite to while loop.

Syntax:

```
until [ condition ] do
  body
done
```

The body will be executed as long as condition returns false. Once condition returns true, then loop will be terminated.

Q1. print 1 to 5 using until loop

```
js@Jitendra:~$ cat code.sh
#!/bin/bash
```

```
i=1
until [ $i -gt 5 ]
do
echo $i
let i++
done
js@Jitendra:~$ ./code.sh
1
2
3
4
5
```

## 3) For loop:-

If we want to perform some action for every item in the given list, then we should go for for loop

Syntax:-

```
for variable in item1 item2 item3... itemN
do action
done
```

Q. Print q 1 to 5 using for loop

```
1 #!/bin/bash
2
3 for i in 1 2 3 4 5
4 do
5 echo $i
6 done
7
```

Ex :-

```
js@Jitendra:~$ cat code.sh
#!/bin/bash
```

```
for course in java linux python c++ dsa
do
echo $course
done
```

```
js@Jitendra:~$ ./code.sh
java
linux
python
c++
dsa
```

Q. Write a script that display numbers from 1 to 100 , which are divisible by 10

```
1 #!/bin/bash
2
3 for i in {1..100}
4 do
5 if [ ${i%10} -eq 0 ]
6 then
7 echo $i
8 fi
9 done
```

Q. Write a script to display all file names present in current working directory

```
1 #!/bin/bash
2
3 for name in *
4 do
5 if [ -f $name ]
6 then
7 echo $name
8 fi
9 done
```

Q. Write a script to append current date and time in every .txt file present in current Working directory

```
1 #!/bin/bash
2
3 for file in *.txt
4 do
5 date>>$file
6 done
```


Q. Write a script to print all command line arguments



```
1 #!/bin/bash
2
3 if [ $# -ne 0 ]
4 then
5
6 for arg in $@
7 do
8 echo "$arg"
9 done
10
11 else
12 echo "no command line arguments passed"
13 fi
```

```
js@Jitendra:~$ ./code.sh
no command line arguments passed
js@Jitendra:~$ ./code.sh c++ dsa linux
c++
dsa
linux
js@Jitendra:~$
```

Q. Write a script to display multiple files content to the terminal and all the File names Passed as command line arguments ?




```

1  #!/bin/bash
2
3  if [ $# -ne 0 ]; then
4
5      for fname in $@; do
6          if [ -f $fname ]; then
7              echo "_____File Content_____"
8              cat $fname
9              echo "_____ "
10         else
11             echo "$fname is not regular file"
12         fi
13     done
14
15 else
16     echo "no command line arguments passed"
17 fi
18

```

Q. Write a script to append multiple files content to a single file result.txt file names are passed as command line arguments ?



```

1  #! /bin/bash
2  if [ $# -ne 0 ]; then
3      for fname in $@; do
4          if [ -f $fname ]; then
5              cat $fname >>result.txt
6          else
7              echo "$fname does not exist or it is not a regular file"
8          fi
9      done
10 else
11     echo "Please pass atleast one file name"
12 fi
13

```

Q. Write a Script to Display all Employees Information where Salary is Greater than 2500

emp.txt

```
1 102:Jitendra:100000:Bengaluru
2 121:Vikas:150000:Kanpur
3 133:Rajesh:50000:Delhi
4 111:Kartik:200000:Hyderabad
5 67:Muthu:120000:Chennai
```

```
1 #!/bin/bash
2 for record in $(cat emp.txt)
3 do
4     sal=$(echo $record | cut -d ':' -f 3)
5     if [ $sal -gt 100000 ]; then
6         echo $record
7     fi
8 done
9
```

Q. Write a script to save all employees info where salary is greater than 2500 and city is hyderabad to hyd.txt

```
1 #!/bin/bash
2 for record in $(cat emp.txt)
3 do
4     sal=$(echo $record | cut -d ':' -f 3)
5     city=$(echo $record | cut -d ':' -f 4)
6     if [ $sal -gt 100000 -a $city = "Hyderabad" ];
7     then
8         echo $record
9     fi
10 done
11
```

Q. Write a script to display minimum and maximum salaries



```
1  #!/bin/bash
2  max=$(cat emp.txt | head -1 | cut -d ":" -f 3)
3  min=$(cat emp.txt | head -1 | cut -d ":" -f 3)
4  max_record=$(cat emp.txt | head -1)
5  min_record=$(cat emp.txt | head -1)
6
7  for record in $(cat emp.txt)
8  do
9      sal=$(echo $record | cut -d ':' -f 3)
10     if [ $sal -gt $max ];
11     then
12         max=$sal
13         max_record=$record
14     fi
15
16     if [ $sal -lt $min ];
17     then
18         min=$sal
19         min_record=$record
20     fi
21 done
22
23 echo "maximum salary " $max
24 echo "Employee data " $max_record
25
26 echo "minimum salary " $min
27 echo "Employee data " $min_record
28
```

## Alternative Syntax of for loop:

```
for((i=1;i<10;i++)) do
```

```
done
```

eg :-print using for loop



```
1  #!/bin/bash
2  read -p "enter a number n " n
3  for((i=0;i<n;i++))
4  do
5  echo $i
6  done
7
```

eg :- display table using loop



```
1  #!/bin/bash
2  read -p "enter a number n " n
3  for((i=0;i≤10;i++))
4  do
5  echo "$n * $i = ${n*i}"
6  done
```

Q. check wheather a given number is prime or not



```
1  #!/bin/bash
2  read -p "enter a number n " n
3  prime=true
4  for ((i = 2; i ≤ $n / 2; i++)); do
5      if [ $((n % i)) -eq 0 ]; then
6          prime=false
7          break
8      fi
9  done
10
11 echo $prime
```

Q. print triangle pattern



```
1  #!/bin/bash
2  read -p "enter a number n " n
3  for ((i = 0; i < $n; i++)); do
4      for ((j = 0; j ≤ $i; j++)); do
5          echo -n "*"
6      done
7      echo ""
8  done
```

## -----Arrays-----

IF we want to represent a group of values with a single name then we should go for arrays

### 1. Declaring arrays

declare -a arrayname

### 2. Initializing arrays

courses=(java c++ linux python)

### 3. **Assigning values**

`courses[0]=java`



```
courses[1]=python
courses[2]=linux
```

#### 4. Accessing Elements : -

`${courses[i]}` -----> ith element  
`${courses[@]}` -----> All elements present inside array  
`${courses[@]}` -----> All elements present inside array separated by IFS  
`${!courses[@]}` -----> All elements where elements are available  
`${#courses[@]}` -----> It returns the number of elements present inside array.  
`${#courses[0]}` -----> It returns the length of first element.

```
1  #!/bin/bash
2
3  course[0]=java
4  course[1]=python
5  course[2]=c++
6  course[3]=Django
7  course[6]=Node
8
9  echo "first element : ${course[0]}"
10 echo "all elements using @ : ${course[@]}"
11 echo "all elements using * : ${course[*]}"
12 echo "all indices where elements are available : ${!course[@]}"
13 echo "total number of elements : ${#course[@]} "
14 echo "total length of first element : ${#course[0]}"
15
16 #finding variable value length using #
17 d="good book"
18 len=${#d}
19 echo $len
```

```
pc@Jitendra:~$ ./code.sh
first element : java
all elements using @ : java python c++ Django Node
all elements using * : java python c++ Django Node
all indices where elements are available : 0 1 2 3 6
total number of elements : 5
total length of first element : 4
9
pc@Jitendra:~$
```



**Q. print array elements using while loop, for loop**



```
1  #!/bin/bash
2  fruits=("Apple" "Orange" "Banana" "Mango")
3  size=${#fruits[@]}
4  i=0
5
6  echo "All elements by using while loop:"
7  echo "_____"
8  while [ $i -lt $size ]
9  do
10 echo ${fruits[$i]}
11 let i++
12 done
13
14 echo "All elements by using for loop:"
15 echo "_____"
16
17 for fruit in ${fruits[@]}
18 do
19 echo $fruit
20 done
21
22 echo "All elements by using advanced for loop:"
23 echo "_____"
24
25 for((i=0;i<${#fruits[@]};i++))
26 do
27 echo ${fruits[$i]}
28 done
```

```
pc@Jitendra:~$ ./code.sh
```

```
All elements by using while loop:
```

```
-----
```

```
Apple
```

```
Orange
```

```
Banana
```

```
Mango
```

```
All elements by using for loop:
```

```
-----
```

```
Apple
```

```
Orange
```

```
Banana
```

```
Mango
```

```
All elements by using advanced for loop:
```

```
-----
```

```
Apple
```

```
Orange
```

```
Banana
```

```
Mango
```

```
pc@Jitendra:~$
```

Q2 Write a script for accessing Array elements by using for loop indices are random



```
1  #!/bin/bash
2
3  # declare -a fruits
4  fruits[1]="mango"
5  fruits[23]="banana"
6  fruits[34]="apple"
7
8  echo "——loop on elements——"
9  for fruit in ${fruits[@]}
10 do
11 echo $fruit
12 done
13
14 echo "——loop on index——"
15 for idx in ${!fruits[@]}
16 do
17 #echo $fruits[$idx] will not work
18 #here
19 echo ${fruits[$idx]}
20 done
```

```
pc@Jitendra:~$ ./code.sh
----loop on elements----
mango
banana
apple
----loop on index----
mango
banana
apple
pc@Jitendra:~$
```

## 5. Removing array elements

Yes we can remove array elements using unset command

```
unset arr[10]
```

## 6. reading the values into array

```
1 read -p "Enter the number of values: " n
2
3 for ((i = 0, j = 1; i < n; i++)); do
4     read -p "Enter the number $((j++)): " nums[${i}]
5 done
6
7 for ((i = 0; i < n; i++)); do
8     echo ${nums[${i}]}
9 done
```

Q. read n numbers and then print sum of even and odd numbers



```
1 read -p "enter size of array " n
2 for ((i = 0; i < n; i++)); do
3     read nums[$i]
4 done
5
6 oddsum=0
7 evensum=0
8
9 for ((i = 0; i < n; i++)); do
10     if [ $((nums[$i] % 2)) -ne 0 ]; then
11         oddsum=$((oddsum + nums[$i]))
12     else
13         evensum=$((evensum + nums[$i]))
14     fi
15 done
16
17 echo "odd numbers sum = $oddsum"
18 echo "even numbers sum = $evensum"
```

Q. write a script to store all .txt file names present in current directory in to an array

```
# store name of all text file
# in our current directory
# to array
i=0
for file in *.txt;
do
    list[i++]=$file
done

for ((i = 0; i < ${#list[@]}; i++));
do
    echo ${list[$i]}
done
```

# -----Functions-----

If any group of commands are repeatedly required, then it is not recommended to write these commands separately everytime. It increases length of the code and reduces readability. Such type of repeated code we have to define inside a block and we can call that block where ever required. This block of commands is nothing but function. Hence function is nothing but a block of repeatable commands.

Advantages of Functions:

- 1) It reduces length of the code.
  - 2) It improves readability.
  - 3) It improves maintainability.
  - 4) It promotes DRY principle.
- DRY ☞ Don't Repeat Yourself

## 1. Defining a Function

**1st Way:**

```
function function_name()  
{  
  commands  
}
```

**2nd Way:**

```
function_name()  
{  
  commands  
}
```

## 2. Calling a Function

function\_name param1 param2 param3 ....



```
1  #!/bin/bash
2
3  #defining function 1
4  wish1()
5  {
6      echo "Hello how are you doing ?"
7  }
8  #defining function 2
9  function wish2()
10 {
11     echo "Hellow how are you ?, again."
12 }
13
14 #calling functions
15 wish1
16
17 wish2
18
```

eg 2:-





```
1  #!/bin/bash
2
3  f1()
4  {
5      echo "I am in function 1 1"
6  }
7
8  f2()
9  {
10     echo "I an in function 2"
11     f1
12 }
13
14 f2
15
```

```
pc@Jitendra:~$ ./code.sh
I an in function 2
I am in function 1 1
pc@Jitendra:~$
```

### 3. functions with parameters :

function can accept parameters also. with in the function we can access parameters as follows :

\$1	> first parameter
\$2	> second parameter
\$@	> All parameters
\$*	> All parameters

\$#	> Total number of parameters
\$0	> It is script name but not function name



```
1  #!/bin/bash
2
3  demo()
4  {
5      echo "first parameter : $1"
6      echo "second parameter : $2"
7      echo "total numuber of parameters : $#"
```

```
pc@Jitendra:~$ ./code.sh
first parameter : 10
second parameter : 20
total numuber of parameters : 4
all parameters with @ :  10 20 30 40
script name : ./code.sh
pc@Jitendra:~$
```


eg :-



```
1  #!/bin/bash
2
3  wish()
4  {
5      if [ $# -eq 0 ]
6      then
7          echo "Hello Guest Good Evening"
8      else
9          echo "Hello $1 good evening"
10     fi
11 }
12
13 wish
14 wish "Jitendra Singh"
15
```

```
pc@Jitendra:~$ ./code.sh
Hello Guest Good Evening
Hello Jitendra Singh good evening
pc@Jitendra:~$
```

Q. write a function to do arithmetic operators of two operators




```

1  #!/bin/bash
2
3  calc() {
4      if [ $# -ne 2 ]; then
5          echo "You should pass exactly 2 arguments"
6      else
7          a=$1
8          b=$2
9          echo "$a+$b = $((a + b))"
10         echo "$a-$b = $((a - b))"
11         echo "$a*$b = $((a * b))"
12         echo "$a/$b = $((a / b))"
13         echo
14     fi
15 }
16 calc 10
17 calc 20 10
18 calc 200 100
19 calc 2000 1000
20

```

**Q. write a function to print all parameters of a function**



```

1  #!/bin/bash
2
3  parameter_printing() {
4      if [ $# -eq 0 ]; then
5          echo "No parameters passed to this function"
6      else
7          echo "All Passed Parameters are:"
8          echo "....."
9          for p in $@; do
10             echo $p
11         done
12     fi
13 }
14

```

**Q. Find maximum of 2 using function**



```
1  #!/bin/bash
2
3  max()
4  {
5      if [ $1 -gt $2 ]
6      then
7          echo "The max of $1 and $2 is " $1
8      else
9          echo "The max of $1 and $2 is " $2
10     fi
11 }
12
13 max 10 20
14 max 200 100
15
```

**Q. write a function to find factorial**



```
1  #!/bin/bash
2  factorial() {
3      original=$n
4      fact=1
5      while [ $n -gt 1 ]; do
6          let fact=fact*n
7          let n--
8      done
9      echo "The Factorial of $original is: $fact"
10 }
11 read -p "Enter a number to find factorial:" n
12 factorial $n
13
```

Q. Find factorial of 1st 10 natural number



```
1  #! /bin/bash
2  factorial() {
3      original=$1
4      n=$1
5      fact=1
6      while [ $n -gt 1 ]; do
7          let fact=fact*n
8          let n--
9      done
10     echo "The Factorial of $original: $fact"
11 }
12 for i in {1..10}; do
13     factorial $i
14 done
15
```



Q. write a program to generate all prime numbers till given number



```
1  #!/bin/bash
2  prime_numbers() {
3      for ((n = 2; n ≤ $1; n++)); do
4          is_prime="Yes"
5
6          for ((i = 2; i < n; i++)); do
7              if [ $((n % i)) -eq 0 ]; then
8                  is_prime="No"
9                  break
10             fi
11         done
12
13         if [ $is_prime = Yes ]; then
14             echo $n
15         fi
16     done
17 }
18
19 read -p "Enter N value : " n
20 prime_numbers $n
21
```

---

```
pc@Jitendra:~$ ./code.sh
Enter N value : 10
2
3
5
7
pc@Jitendra:~$
```

## Variable Scope :-

By default every variable in shell script is global. i.e we can access anywhere in our script. But before using a variable, it should be declared already.

### Eg 1:

```
#!/bin/bash f1()
{
    echo "x value : $x"
} x=10
f1
```

### Output:

```
x value : 10
```

### Eg 2:

```
#!/bin/bash f1()
{
    x=20
    echo "x value : $x"
} x=10
f1
echo "After f1 execution x value: $x"
```

### Output:

```
x value : 20
```

After f1 execution x value: 20

**Eg 3:**

```
#!/bin/bash f1()
{
    echo "x value : $x"
}
f1 x=10
f1
```

**Output:**

```
x value :
x value : 10
```

--->The variables which are declared inside a function, can be accessed outside of that function, because every variable has global scope by default.

---->If we want a variable only within the function and should not be available outside of that function, then we have to use local keyword for the variable.  
local variables can be accessed only inside function and cannot be accessed outside of that function.



```
1  #! /bin/bash
2
3  f1()
4  {
5      local x=10
6      echo "inside function x is " $x
7  }
8
9  f1
10
11 echo "outside function x is " $x
12
```

```
pc@Jitendra:~$ ./code.sh
inside function x is 10
outside function x is
pc@Jitendra:~$
```

## Return Statement in Functions:

Every function in shell scripting returns some value. The default returned value is the exit code of the last command inside function.

But based on our requirement we can return values explicitly by using return statement.

return <exitcode>

The allowed values for exitcode are 0 to 255. 0

means successful

non-zero means unsuccessful.

We can get return value of function by using \$? . eg:-



```
1  #! /bin/bash
2  sum()
3  {
4    if [ $# ≠ 2 ]; then
5      echo "You should pass exactly two numbers"
6      return 1
7    else
8      echo "The SUM:$(( $1+$2 ))"
9    fi
10 }
11 sum 10 20
12 echo "The Return value of this function:$?"
13 echo
14 sum 10
15 echo "The Return value of this function:$?"
16
```

pc@Jitendra:~\$ ./code.sh

The SUM:30

The Return value of this function:0

You should pass exactly two numbers

The Return value of this function:1

pc@Jitendra:~\$

### Use Case:

backup()

```
{
  commands to take backup
}
```

backup

if [ \$? != 0 ]; then

something goes wrong backup failed

```
else  
  backup successful
```

## break vs exit vs return:

### 1) break:

We can use break only inside loops to break loop execution. we will come out of the loop and remaining statements after loop will be executed.

### 2) exit:

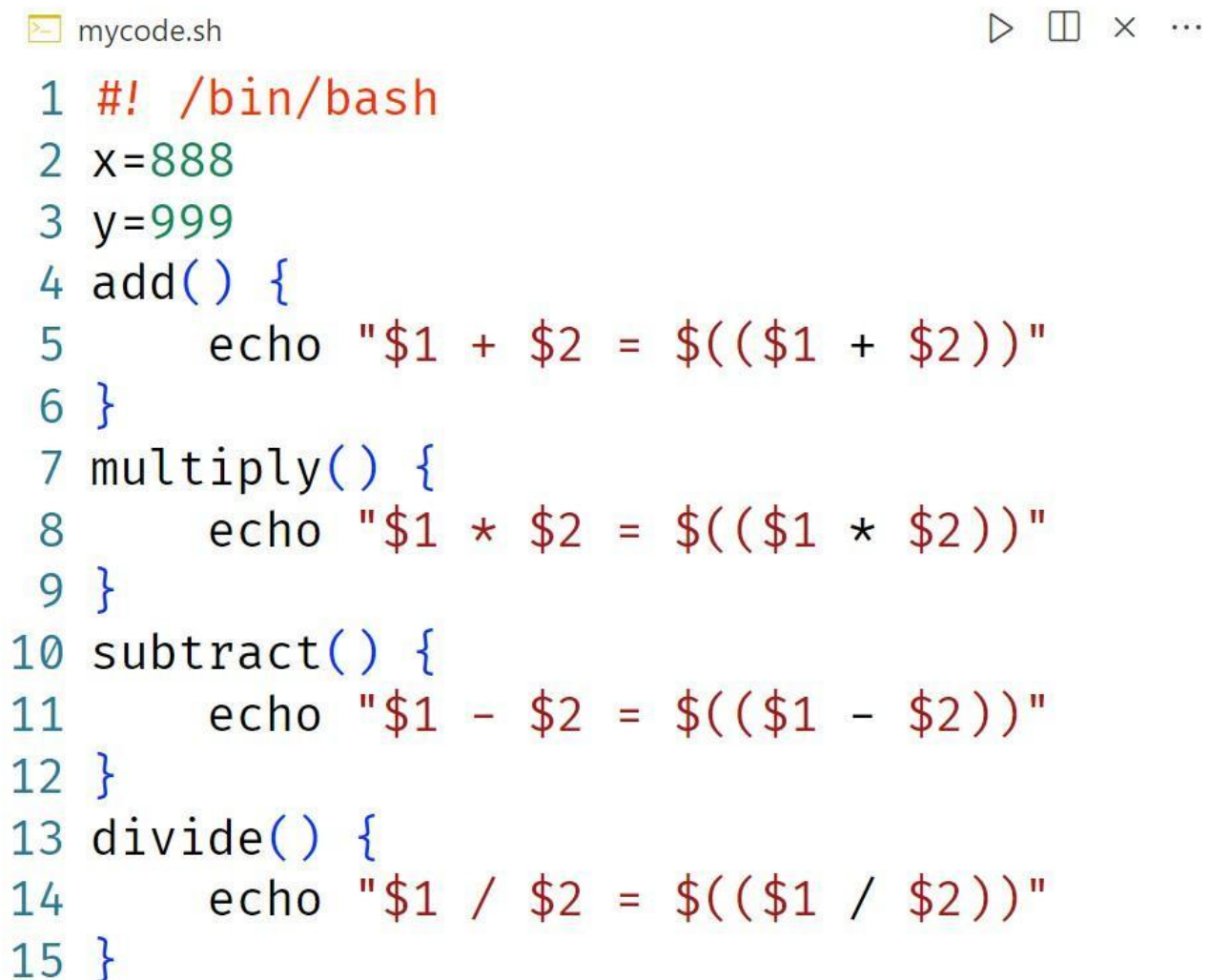
We can use anywhere exit statement to terminate script execution. The script will be terminated. No chance of executing any other statement.

### 3) return:

We can use return statement only inside function to stop function execution. After return statement, the remaining statements inside function won't be executed. But after function call the remaining statements will be executed.

## How to call functions present in another Script:

Script file 1 with functions defined



```
mycode.sh
1  #!/bin/bash
2  x=888
3  y=999
4  add() {
5      echo "$1 + $2 = $((($1 + $2)))"
6  }
7  multiply() {
8      echo "$1 * $2 = $((($1 * $2)))"
9  }
10 subtract() {
11     echo "$1 - $2 = $((($1 - $2)))"
12 }
13 divide() {
14     echo "$1 / $2 = $((($1 / $2)))"
15 }
```

```
1 #!/bin/bash
2 # syntax for including another script
3 . ./mycode.sh
4
5 add $x $y
6 add 10 20
7 subtract 20 10
8 multiply 10 20
9 divide 20 10
10
11 echo "The value of x : $x"
12 echo "The value of y : $y"
```

---

## **GCC compiler :-**

GCC, or GNU Compiler Collection, is a free and open-source compiler system developed by the GNU Project. It is widely used for compiling code written in a variety of programming languages, including C, C++, Objective-C, Fortran, Ada, and others. GCC is the default compiler for most Linux distributions.

GCC is a collection of compilers, each of which supports one or more programming languages. These compilers are invoked by the command line interface. The basic syntax for compiling a C program with GCC is:

```
gcc [options] filename.c -o executable_name
```

In this command, [options] represents any additional compiler options that you may want to use, filename.c is the name of the C source file, and executable\_name is the desired name of the output executable file.



GCC supports a wide variety of options that allow you to control the behavior of the compiler, optimize your code, and generate debugging information. Some of the most commonly used options include:

**-c:** This option tells GCC to compile the source file(s) without linking them to produce an object file. This is useful when you want to compile multiple source files separately and then link them together later. For example, if you have two source files `main.c` and `helper.c`, you can compile them separately using `gcc -c main.c` and `gcc -c helper.c`, and then link them together using `gcc main.o helper.o -o program`.

**-O:** This option enables optimization of the code generated by the compiler. The level of optimization can be specified by appending a number to the option, such as `-O1`, `-O2`, or `-O3`. The higher the optimization level, the more aggressive the optimizations performed by the compiler, but also the longer the compilation time. Optimization can result in faster and more efficient code, but can also make debugging more difficult.

While optimization can improve program performance and efficiency, there are several reasons why it may not always be desirable to use it:

**Increased compilation time:** Enabling optimization can significantly increase the time it takes to compile your code. This is because the compiler has to perform additional analysis and transformations on your code, which can be time-consuming. In some cases, the increased compilation time may be unacceptable, especially if you are developing a large project or need to iterate quickly.

**Increased code complexity:** Optimization can make your code more complex and harder to understand. This is because the optimizations performed by the compiler can change the structure and behavior of your code, making it more difficult to reason about. This can make debugging and maintenance more difficult, especially for inexperienced developers.

**Increased memory usage:** Some optimizations, such as loop unrolling and function inlining, can increase the amount of memory used by your program. This can be a problem if you are developing for a resource-constrained environment or if your program needs to run on a large number of machines.

**Possible loss of portability:** Some optimizations may be specific to a particular CPU architecture or operating system, and may not work correctly on other platforms. This can lead to bugs and compatibility issues if you need to port your code to other systems.

**Potential introduction of bugs:** Optimization can sometimes introduce subtle bugs into your code, especially if you are using non-standard or undefined behavior. This is because some optimizations may rely on assumptions about the behavior of your code that are not guaranteed by the language specification. As a result, it is important to thoroughly test your optimized code to ensure that it behaves correctly.

For these reasons, it is often a good idea to use optimization selectively and to test your code thoroughly before deploying it. You should also be aware of the trade-offs involved in optimization and weigh the benefits against the potential costs.

**-g:** This option generates debugging information in the output file. This information can be used by debugging tools to help you trace the execution of your program and find bugs. The level of debugging

information can be specified by appending a number to the option, such as -g1, -g2, or -g3. The higher the level, the more detailed the debugging information, but also the larger the output file.

**-Wall:** This option enables all warning messages produced by the compiler. Warnings are messages that indicate potential problems in your code, such as uninitialized variables, unused parameters, or type mismatches. Enabling warnings can help you catch errors early and improve the quality of your code.

**-std=<standard>:** This option specifies the version of the programming language standard that the compiler should use to compile the code. For example, -std=c11 specifies that the compiler should use the C11 standard, while -std=c++17 specifies that the compiler should use the C++17 standard. Using the correct standard can help ensure that your code is portable and compatible with different compilers and platforms.

