**BASH SCRIPTING**

**WHAT IS SHELL?**

A Shell provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

**TYPES OF SHELLS**

There are two major types of shells in Unix They are

* **BOURNE SHELL:** This is a default shell for version 7 Unix. The Character $ is the default prompt for the bourne shell. The different subcategories in the shell are korn shell, ***Bourne Again Shell (BASH),*** POSIX shell etc.
* **C SHELL:** This is a Unix shell and a command processor that is run in a text window. The character % is the default prompt for the c shell. File commands can also read easily by the C shell which is known as a script.

**CAPABILITIES OF SHELL SCRIPT**

* Batch Jobs
* Programming
* Generalization
* Shortcuts

**WHAT IS BASH SCRIPTING?**

A bash script is a file containing a sequence of commands that are executed by the bash program line by line. It allows you to perform a series of actions, such a s navigating to a specific directory, creating a folder, and launching a process using the command line. By saving these commands in a script, you can repeat the same sequence of steps multiple times and execute them by running the script.

**ADVANTAGES OF BASH SCRIPTING**

Bash scripting is a powerful and versatile tool for automating system administration tasks, managing system resources, and performing other routine tasks in Unix or Linux system. Some advantages of shell scripting are:

* **Automation:** It allows you to automate repetitive tasks and processes saving time and reducing the risk of errors.
* **Portability:** It can be run on various platforms and OS including Unix, Linux and macOS.
* **Flexibility:** They are highly customizable and can be easily modified to suit specific requirements.
* **Accessibility:** They are easy to write and don’t require any special tools or software.
* **Integration:** They can be integrated with other tools and applications such as databases, web servers and cloud services.
* **Debugging:** The are easy to debug and most shells have built-in debugging and error-reporting tools that can help identify and fix issues quickly.

**FEATURES**

* Shell scripting
* Variables
* Control Structures
* Command execution
* Input/Output
* Functions
* Environment Variables
* Command-line arguments
* String manipulation
* Conditional Expressions
* Error handling
* Operations

**What is SHEBANG LINE**?

The sign #**!** is called she-bang and is written at top of the script. It passes instruction to program **/bin/sh.** To run your script in a certain shell (shell should be supported by your system), start your script with #! followed by the shell name.

**HOW TO WRITE AND EXECUTE A SCRIPT**

we create a test.sh script. Note all the scripts would have the .sh extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct. For example

#!/bin/sh

This tells the system that the commands that follow are to be executed by the bourne shell

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT

1.

#!/bin/sh

echo "what is your name?"

read PERSON

echo "hello, $PERSON"

Save the above content and make the script executable

**$ chmod +x 1.sh**

The shell script is now ready to be executed

./1.sh or sh 1.sh

**VARIABLES**

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data

**VARIABLE NAMES**

The name of a variable can contain only letters (a to z or A to Z), numbers ( 0 to 9) or the underscore character ( \_). By convention, Unix shell variables will have their names in UPPERCASE

**TYPES OF VARIABLES**

* **System variables**

They are created & defined by Unix or Linux OS .They are predefined variables defined by OS

EX: $BASH,$BASH\_VERSION,$HOME,$PWD etc.

* **User defined variables**

They are created and maintained by us

**DEFINING VARIABLES**

Variables are defined as follows-

Variable\_name =variable\_value

Ex: NAME = “Raghu”

The above example defines the variable NAME and assigns the value “Raghu” to it

**ACCESSING VALUES**

To access the value stored in a variable, prefix its name with the dollar sign ($).

For example, the following script will access the value of defined variable NAME and print it on STDOUT

2.

#!/bin/sh

NAME="Raghu"

echo $NAME

**READ\_ONLY VARIABLES:**

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked ***read-only***, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME

3.

#!/bin/sh

NAME="Raghu"

readonly NAME

NAME="chandu"

**UNSETTING VARIABLES:**

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable

4.

#!/bin/sh

NAME="Raghu"

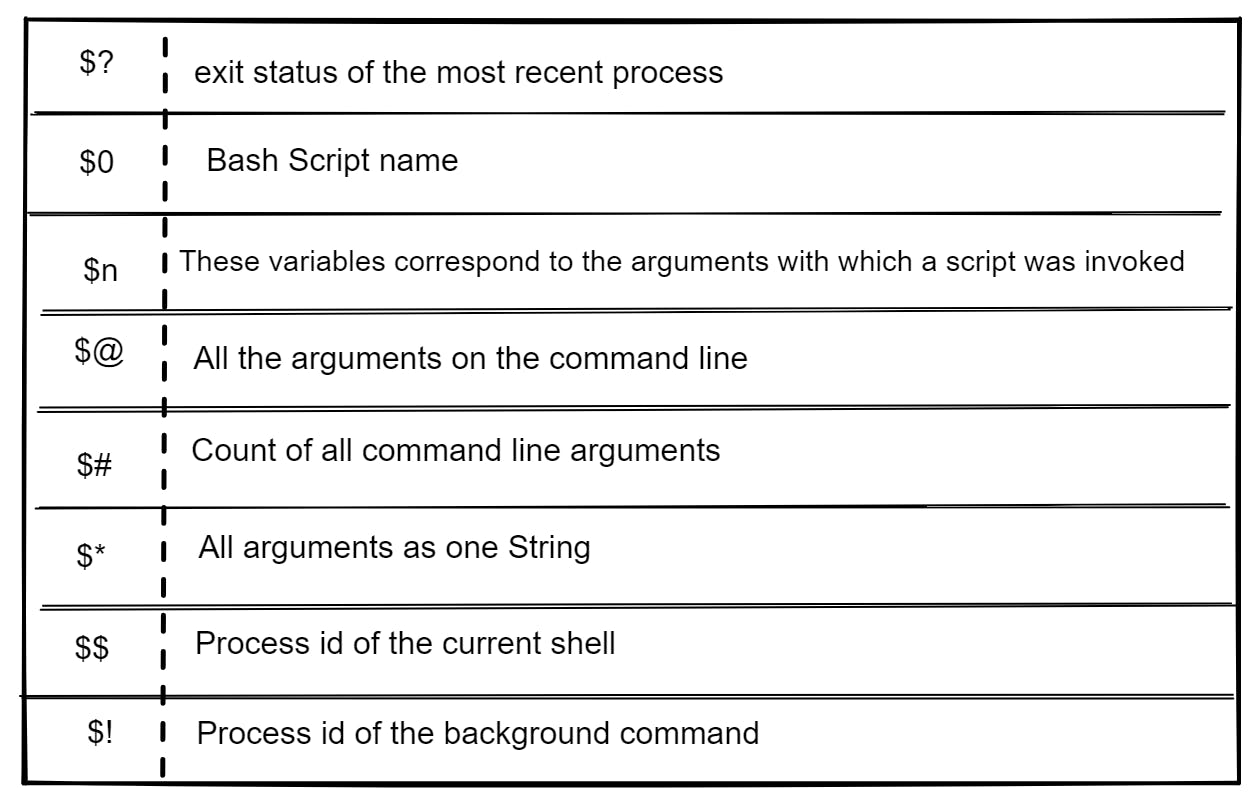
unset NAME

echo $NAME

**SPECIAL VARIABLES:**

These variables are reserved for specific functions

The following table shows a number of special variables that you can use in your shell scripts



5.

#!/bin/sh

echo "file anme: $0"

echo "first paramaeter: $1"

echo "second parameter: $2"

echo "quoted values: $@"

echo "Quoted values: $\*"

echo "Total number of parameters: $#"

**SHELL ARRAYS:**

A shell variable is capable enough to hold a single value. These variables are called scalar variables. Shell supports a different type of variable called an array variable. This an hold multiple values at the same time. Arrays provide a method of grouping a set of variables. we

**Defining ARRAY VALUES:**

The difference between an array variable and a scalar variable can be explained as follows. Each of the individual variables is a scalar variable as follows

NAME01=”Raghu”

NAME02=”chandu”

Instead of this we can use a single array to store all the above mentioned names

array\_name[index]=value

array\_name -> name of the array

index -> it is the index of the item in the array that you want to set

value-> value is the value you want to set

If you are using the **bash** shell, here is the syntax of array initialization

Array\_name=(value1..valuen)

**ACCESSING ARRAY VALUES:**

After you have set any array variable, you access it as follows

${array\_name[index]}

6.

#!/bin/sh

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

NAME[3]="Ayan"

NAME[4]="Daisy"

echo "First Index: ${NAME[2]}"

echo "Second Index: ${NAME[1]}"

You can access all the items In an array in one of the following ways

${array\_name[\*]}

${array\_name[@]}

7.

#!/bin/sh

NAME[0]="Zara"

NAME[1]="Qadir"

NAME[2]="Mahnaz"

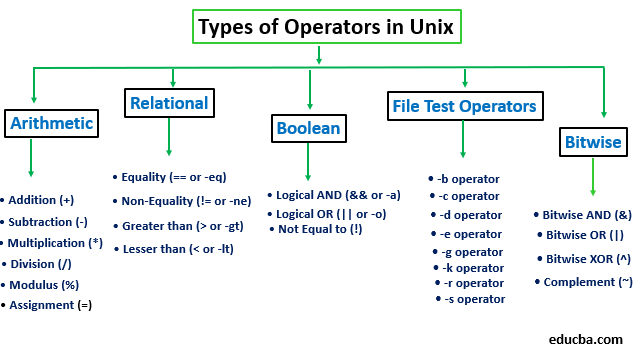
NAME[3]="Ayan"

NAME[4]="Daisy"

echo "First Method: ${NAME[\*]}"

echo "Second Method: ${NAME[@]}"

**BASIC OPERATORS:**



Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**

8.

#!/bin/sh

Val = ‘expr 2 + 2’

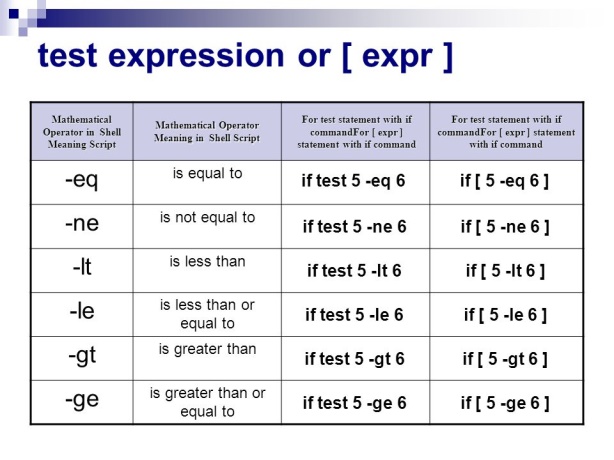
Echo “total value : $val”

Note:There must be spaces between operators and expressions.The complete expression should be enclosed between ‘’ backtics

Arithematic operators

Ex: `expr $a + $b`

[ $a == $b ] -> All the conditional expressions should be inside square braces with spaces around them



**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

Ex: [ $a <= $b ] -> all the conditional exxpressions should be placed inside square braces

**FILE TEST OPERATORS**

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

|  |  |  |
| --- | --- | --- |
| **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. |

9.

#!/bin/sh

a=10

b=20

if [ $a -eq $b ]

then

echo "$a -eq $b : a is equal to b"

else

echo "$a -eq $b: a is not equal to b"

fi

if [ $a -ne $b ]

then

echo "$a -ne $b: a is not equal to b"

else

echo "$a -ne $b : a is equal to b"

fi

if [ $a -gt $b ]

then

echo "$a -gt $b: a is greater than b"

else

echo "$a -gt $b: a is not greater than b"

fi

if [ $a -lt $b ]

then

echo "$a -lt $b: a is less than b"

else

echo "$a -lt $b: a is not less than b"

fi

if [ $a -ge $b ]

then

echo "$a -ge $b: a is greater or equal to b"

else

echo "$a -ge $b: a is not greater or equal to b"

fi

if [ $a -le $b ]

then

echo "$a -le $b: a is less or equal to b"

else

echo "$a -le $b: a is not less or equal to b"

fi

**STRING OPERATORS:**

10.

#!/bin/sh

a="abc" b="efg"

if [ $a = $b ]

then

echo "$a = $b : a is equal to b" else

echo "$a = $b: a is not equal to b"

fi

if [ $a != $b ]

then

echo "$a != $b : a is not equal to b" else

echo "$a != $b: a is equal to b"

fi

if [ -z $a ]

then

echo "-z $a : string length is zero" else

echo "-z $a : string length is not zero"

fi

if [ -n $a ]

then

echo "-n $a : string length is not zero" else

echo "-n $a : string length is zero"

fi

if [ $a ]

then

echo "$a : string is not empty" else

echo "$a : string is empty"

fi

**FOR LOOP:**

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

11.

#!/bin/sh

for i in {1..10}

do

echo " hai"

done

12.

#!/bin/sh

for (( i=10; i>=1; i--))\

do

echo "hai Raghu"

done

**IF:**

The if..fi statement is the fundamental control statement that allows shell to make decisions and execute statements conditionally

**SYNTAX:**

if[expression]

then

staments to be executed if expression is true

fi

13.

#!/bin/sh

if [ $1 –gt 100]

then

       echo “ hey that is a large number”

      pwd

fi

**IF ELSE:**

The if..else..fi statement is the next form of control statement that allows shell to execute statements in a controlled way and make the right choice

14.

#!/bin/bash

#a=20

#b=40

if [ $a == $b ]

then

    echo "a is equal to b"

else

    echo " a is not equal to b"

fi

**IF ELIF:**

The if...elif...fi statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

15. #!/bin/sh

a=30

b=40

c=20

if [ $a -gt $b ]

then

echo "a is greater than b"

elif [ $a -gt $c ]

then

echo "a is greater than c"

else

echo "a is not a greater number"

fi

**WHILE LOOP:**

The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly

16.

#!/bin/bash

c=1

while [ $c -le 10 ]

do

        echo "Welcome $c times"

        (( c++ ))

done

17. #!/bin/sh

a=0

while [ "$a" -lt 10 ]    # this is loop1

do

   b="$a"

   while [ "$b" -ge 0 ]  # this is loop2

   do

      echo  "$b " or echo  -n “$b”

      b=`expr $b - 1`

   done

   echo

   a=`expr $a + 1`

done

Note:  Here **-n** option lets echo avoid printing a new line character.

**WHAT IS SUBSTITUTION?**

The shell performs substitution when it encounters an expression that contains one or more special characters

Ex: Here the printing value of the variable is substituted by its value .same time .”\n” is substituted by a new line

18.

#!/bin/sh                                  #!/bin/sh

a=10                                       a=10

Echo -e “value of a is $a \n”              echo “value of a is $a \n”

o/p: value of a is 10                      o/p: value of a is 10\n

Here the **-e** option enables the interpretation of backslash escapes.

The following are the escape sequences which can be used in echo command

1. \\ backslash
2. \a alert(BEL)
3. \b backspace
4. \c suppress trailing newline
5. \f form feed
6. \n new line
7. \r carriage return
8. \t horizontal tab
9. \v vertical tab

You can use the **-E** option to disable the interpretation of the backslash escapes (default).

You can use the **-n** option to disable the insertion of a new line.

**Command substitution**

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

**SYNTAX:**

The command substitution is performed when a command is given as

`command`

When performing the command substitution make sure that you use the backquote, not the single quote character.

#!/bin/sh

DATE=`date`

echo "Date is $DATE"

**VARIABLE SUBSTITUTION:**

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

19.#!/bin/sh

echo ${var:-"Variable is not set"}

echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}

echo "2 - Value of var is ${var}"

unset var

echo ${var:+"This is default value"}

echo "3 - Value of var is $var"

var="Prefix"

echo ${var:+"This is default value"}

echo "4 - Value of var is $var"

echo ${var:?"Print this message"}

echo "5 - Value of var is ${var}"

**SHELL FUNCTIONS:**

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual tasks when needed.

Using functions to perform repetitive tasks is an excellent way to create **code reuse**. This is an important part of modern object-oriented programming principles.

**Creating functions**

function\_name () {

List of commands

}

The name of your function is function\_name and that’s what you will use to call it from elsewhere in your scripts.The function name must be followed by parentheses followed by a list of commands enclosed within braces

## Pass Parameters to a Function

You can define a function that will accept parameters while calling the function. These parameters would be represented by **$1**, **$2** and so on.

20.#!/bin/sh

# Define your function here

Hello () {

   echo "Hello World $1 $2"

}

# Invoke your function

Hello Zara Ali

**RETURNING VALUES FROM FUNCTIONS:**

If you execute an **exit** command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

21.

#!/bin/sh

# Define your function here

Hello () {

   echo "Hello World $1 $2"

   return 10

}

# Invoke your function

Hello Zara Ali

# Capture value returnd by last command

ret=$?

echo "Return value is $ret"

**NESTED FUNCTIONS:**

One of the more interesting features of functions is that they can call themselves and also other functions. A function that calls itself is known as a ***recursive function***.

22.

#!/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

# SHELL SCRIPTING CASE:

# A case construct helps us to simplify nested if statement. You can match several variables against one variable. Each case is an expression matching a certain pattern.

23.

#!/bin/bash

echo "Enter color:"

read color

case $color in

    Red)

        echo "Roses are red"

        ;;

    Green)

        echo "Grass is green"

        ;;

    Blue)

        echo "Sky is blue"

        ;;

    \*)

        echo "Color not recognized"

        ;;

esac

**FAQ’s**

**1. Find Given Number is Even or Odd**

#!/bin/sh

echo "enter a number: "

read n

if [ $((n%2)) -eq 0 ]

then

  echo "number is even"

else

  echo "number is odd"

fi

**2. Print Fibonacci Series**

#!/bin/bash

echo "Enter a number"

read num

a=0

b=1

echo "The Fibonacci series is : "

for (( i=0; i<num; i++ ))

do

    echo -n "$a "

    fn=$((a + b))

    a=$b

    b=$fn

done

echo ""

**3. Print given String in Reverse**

#!/bin/bash

echo "Enter a string"

read str

echo "$str" | rev

**4. Find given Directory exist or not if exist create a empty file detail.txt under same folder else create directory with file newdata.txt**

#!/bin/sh

echo "enter a directory name"

read dir

if [ -d $dir ]

then

   touch "$dir/detail.txt"

else

   mkdir new

   cd new

   touch new.txt

fi

**5. Read array of inputs and find min, max, average**

#!/bin/bash

echo "Enter space-separated integers:"

read -a arr

sum=0

for i in "${arr[@]}"

do

    sum=$((sum+i))

done

avg=$(echo "scale=2; $sum/${#arr[@]}" | bc)

echo "Minimum: $(echo "${arr[@]}" | tr ' ' '\n' | sort -n | head -1)"

echo "Maximum: $(echo "${arr[@]}" | tr ' ' '\n' | sort -nr | head -1)"

echo "Average: $avg"

**6. Write example for case "Red, Green, Blue"**

#!/bin/bash

echo "Enter color:"

read color

case $color in

    Red)

        echo "Roses are red"

        ;;

    Green)

        echo "Grass is green"

        ;;

    Blue)

        echo "Sky is blue"

        ;;

    \*)

        echo "Color not recognized"

        ;;

Esac

**7. Find given number is prime number or not**

#!/bin/bash

echo "Enter a number"

read num

is\_prime=true

if [ $num -lt 2 ]

then

    is\_prime=false

else

    for (( i=2; i<=$num/2; i++ ))

    do

        if [ $((num%i)) -eq 0 ]

        then

            is\_prime=false

            break

        fi

    done

fi

if [ $is\_prime = true ]

then

    echo "Prime"

else

    echo "Not prime"

fi

**8. Find sum of digits, input integer**

#!/bin/sh

echo "Enter a number"

read num

sum=0

while [ $num -gt 0 ]

do

    digit=$((num%10))

    sum=$((sum+digit))

    num=$((num/10))

done

echo "Sum of digits: $sum"

**9. Find Max number for given 3 integers**

#!/bin/sh

echo "Enter three numbers"

read num1 num2 num3

max=$num1

if [ $num2 -gt $max ]

then

    max=$num2

fi

if [ $num3 -gt $max ]

then

    max=$num3

fi

echo "Maximum number: $max"

**10. Print content of input file if exist else create a new file with content "Hello World"**

#!/bin/sh

echo "enter a filename:"

read name

if [ -f $name ]

then

echo " file is found "

cat $name

else

echo hello world >hello.txt

fi

**11. Print Current date in dd-MMM-YYYY, MM-dd-YYYY HH:mm:ss format**

#!/bin/bash

echo "Current date in dd-MMM-YYYY format: $(date +'%d-%b-%Y')"

echo "Current date in MM-dd-YYYY HH:mm:ss format: $(date +'%m-%d-%Y %H:%M:%S')"

**12. Print Current Process and Bakcground Process details**

#!/bin/bash

echo "Current process ID: $$"

echo "Current script file name: $0"

echo "Arguments passed to script: $@"

echo "Parent process ID: $PPID"

echo "Background processes: " $(jobs)

**13. Write a Scheduling code to read log files, compress and zip, write to Log Server host.**

#!/bin/bash

log\_dir="/var/log"

log\_server="example.com"

log\_username="username"

log\_password="password"

for log\_file in $(ls $log\_dir/\*.log)

do

    gzip $log\_file

    scp "${log\_file}.gz" ${log\_username}@${log\_server}:/logs/

done;wq

**14. Write a script to print a multiplication number for a given number**

#!/bin/bash

# Input from user

echo "Enter the number -"

read n

# initializing i with 1

i=1

# Looping i, i should be less than

# or equal to 10

while [ $i -le 10 ]

do

res=`expr $i \\* $n`

# printing on console

echo "$n \* $i = $res"

# incrementing i by one

((++i))

# end of while loop

done

**ASSESSMENT:**

1. Write a Bash script to create a new user account on a Linux system. The Script should take the username, password and an optional comment as input. It should perform the following tasks:

* Check if the script is run with root privileges. If not, display an error message and exit.
* Check if the username already exists. If yes, prompt the user to choose a different username
* Create a new user with the given username and set the provided password.

1. Write a Bash script to analyze a log file named app.log and display the following information:

* The total number of lines in the log file
* The number of ERROR and WARNING messages in the log.

1. Write a bash script that takes a search string and replace string as arguments. The script should perform the following tasks:

* Search for the given search string in all text files in the current directory.
* If found, replace all occurrences of the search string with the replace string
* Print the number of replacements made for each file

1. Write a bash script that performs a system checkup. The script should:

* Delete all the files with a “.tmp” extension in the user’s home directory and its subdirectories.
* Empty the contents of the system’s temporary directory (/tmp)