SHELL SCRIPTING

- Create a file with .sh extension.
- hash (#) ➔ Any line starting with a hash (#) becomes (comment). Comment means, that line will not take part in script execution. It will not show up in the output

1. 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 (_).

Valid variable names :–

_ALI

TOKEN_A

VAR_1

VAR_2

Invalid variable names: –

2_VAR

-VARIABLE

VAR1-VAR2

VAR_A!

2. Defining Variables:-

VAR1="positive"

VAR2=108

- Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

3. Accessing Values:-

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

  NAME="Zara Ali"

  $NAME

  Eg :- echo $NAME

4. Read-only Variables:-

- It 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:-

  NAME="Zara Ali"

  readonly NAME

  NAME="Qadiri"

result :–

$: NAME: This variable is read only.

5. Unsetting Variables:-

- Unsetting or deleting, remove the variable from the list of variables. Once you unset a variable, you cannot access the stored value in the variable.
- You cannot use the unset command to unset variables that are marked readonly.

```
NAME="Zara Ali"
unset NAME
echo $NAME
```

6. Variable Types:-

- **Local Variables** – It present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- **Environment Variables** –It available to any child process of the shell. Some programs need environment variables in order to function correctly.
- **Shell Variables** – It a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these are environment variables others are local variables.

$0 ➔ The filename of the current script.

$n ➔ correspond to the arguments with a script was invoked. n is a positive decimal number corresponding to an argument (the first argument is $1, the second argument is $2, and so on).

$# ➔ The number of arguments supplied to a script.

$* ➔All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.

$@➔All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.

$? ➔The exit status of the last command executed.

$$ ➔The process number of the current shell. For shell scripts, this is the process ID under which they are executing.

$! ➔The process number of the last background command.

Shift ➔ command is a built-in command. Command takes number as argument. Arguments shift down by this number.

Eg :- if number is 5, then $5 become $1, $6 become $2 and so on.

Read ➔ command allows a user to provide the runtime input.

Eg :- enter ur name :

```
read name
```

source ➔Use of external configuration files prevents a user from making changes to a script. Config file is added with the help of source command.

- If a script is shared in many users and every user need a different configuration file, then instead of changing the script each time simply include the config files.

getopts ➔ options are used in shell scripts to parse arguments passed to them. arguments are passed on the command line, getopts parse those arguments instead of command lines.(like switch case in java)

eg without argument:-

```
OPTSTRING=":ab"
while getopts ${OPTSTRING} opt;
do
        case ${opt} in
        a)
        echo "Option -a was triggered."
         ;;
         b)
        echo "Option -b was triggered."
        ;;
         ?)
```

```
        echo "Invalid option: -${OPTARG}."
        exit 1
        ;;
    esac
done
```

eg with argument:-

```
OPTSTRING=":x:y:"
while getopts ${OPTSTRING} opt;
do
    case ${opt} in
      x)
      echo "Option -x was triggered, Argument: ${OPTARG}"
      ;;
      y)
      echo "Option -y was triggered, Argument: ${OPTARG}"
      ;;
      :)
      echo "Option -${OPTARG} requires an argument."
      exit 1
      ;;
      ?)
      echo "Invalid option: -${OPTARG}."
      exit 1
```

```
        ;;
    esac
done
```

7. if then else:-

&& ➔ Logical AND

$0 ➔ Argument 0 i.e. the command that's used to run the script

$1 ➔ First argument (change number to access further arguments)

-eq ➔ Equality check

-ne ➔ Inequality check

-lt ➔ Less Than

-le ➔ Less Than or Equal

-gt ➔ Greater Than

-ge ➔ Greater Than or Equal

Eg:-

```
m=1
n=2

if [ $n -eq $m ]
then
```

```
        echo "Both variables are the same"
    else
        echo "Both variables are different"
    fi
```

(fi ➔ terminate entire loop , without fi loop run every single time)

8. if then elif:-

```
a=10
b=20
if [ $a == $b ]
then
    echo "a is equal to b"
elif [ $a -gt $b ]
then
     echo "a is greater than b"
elif [ $a -lt $b ]
then
    echo "a is less than b"
else
     echo "None of the condition met"
fi
```

9. for loop:-

- for i in 1 2 3 4 5

  do

  echo "Welcome $i times"

  done

- for ((c=1; c<=5; c++))

  do

  echo "Welcome $c times"

  done

10. while loop:-

- a=7

  while [$a -gt 4];

  do

  echo $a

  ((a–))

  Done

11. until loop:-

The ‘until’ loop is a unique looping mechanism that runs a block of code repeatedly until a specified condition becomes true. It essentially works in the opposite manner of the ‘while’ loop

- i=10

```
until [ $i == 1 ]
do
        echo "$i is not equal to 1";
        i=$((i-1))
done
```

12. Shell Functions:-

a function can be divided into smaller or logical parts, which can be called to perform their task. It helps to check our program part by part. We can reuse the function where ever we want.

1. The functions with return:-

```
find_avg(){
        len=$#
        sum=0
        for x in "$@"
        do
                sum=$((sum + x))
        done
        avg=$((sum/len))
        return $avg
        }
        find_avg 30 40 50 60
        printf "%f" "$?"
        printf "\n"
```

2. The functions with ‘exit’ keyword:-

```
is_odd(){
        x=$1
        if [ $((x%2)) == 0 ];
        then
                echo "Invalid Input"
        exit 1
        else
        echo "Number is Odd"
        fi
        }
        is_odd 64
```

3. The functions with variable:-

```
a=1
increment(){
        a=$((a+1))
        return
        }
increment
echo "$a"
```

4. The functions with echo standard output:-

```
hello_world(){
  echo "Hello World"
  return
}
hello_world
```

13. Shell Scripting case:-

- You can match several variables against one variable. Each case is an expression matching a certain pattern.

```
DEPARTMENT="Computer Science"
echo -n "Your DEPARTMENT is "
case $DEPARTMENT in
    "Computer Science")
     echo -n "Computer Science"
     ;;
    "Electrical and Electronics Engineering" | "Electrical
    Engineering")
   echo -n "Electrical and Electronics Engineering or Electrical
    Engineering"
    ;;
  "Information Technology" | "Electronics and
   Communication")
   echo -n "Information Technology or Electronics and
   Communication"
   ;;
  *)
   echo -n "Invalid"
```

```
    ;;
esac
```

14. Shell Scripting eval command:-

- eval is a built-in Linux command which is used to execute arguments as a shell command. It combines arguments into a single string and uses it as an input to the shell and execute the commands.

Eg:-

```
command="echo \$(date)"
eval "$command"
```

15. Shell Scripting let command:-

- The let command is an arithmetic operator. It is almost same as (( )). Only difference is that, let is an arithmetic command while (( )) is a compound command.

Eg:-

```
let "myvar =2" "myvar1=1" "myvar2=myvar1+myvar";
echo $myvar2
let "myvar =2" "myvar1=1" "myvar2=myvar1-myvar";
echo $myvar2
```

```
let "myvar =2" "myvar1=1" "myvar2=myvar1*myvar";
echo $myvar2
let "myvar =2" "myvar1=1" "myvar2=myvar1/myvar";
echo $myvar2
let "myvar =2" "myvar1=1" "myvar2=myvar1%myvar";
echo $myvar2
```