DHA SUFFA UNIVERSITY
Department of Computer Science
CS-2004L
Operating Systems
Fall 2020

LAB 02 – Shell Scripting in Linux

OBJECTIVE(S)

- Learn about Variables and Comments
- Learn about Reading User Input
- Learn about Passing arguments to a Bash-Script
- Learn about Arithmetic and Floating Point Operations
- Learn about Conditional Statements
- Learn about Array Variables
- Learn about Loops
- Learn about Functions

What is a Shell?

A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it. The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name Shell.

What is Shell Scripting?

Shell scripting is writing a series of commands for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script, which can be stored and executed anytime. This reduces the effort required by the end user.
Bourne Again Shell (bash) is the most popular shell.

Steps for creating a Shell Script

(i) Create a file using an editor and name the script file with extension .sh.
(ii) Start the script with #! /bin/sh.
(iii) Write some code.
(iv) Save the script file as filename.sh.
(v) For executing the script, type bash filename.sh or ./filename.sh.

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

DHA SUFFA UNIVERSITY
Department of Computer Science
CS-2004L
Operating Systems
Fall 2020

Shell Scripting

To know which shell types your OS supports, type the following command into the terminal:

$ cat /etc/shells

And to know where bash is located in your OS, type the below command and you will get the specific location:

$ which bash

Using Variables and Comments

Variables are named memory locations that store data or values. There are two types of variables namely System variables and User defined variables.

System Variables

These are created and maintained by the Linux OS. These are some predefined variables that are defined by the OS. The standard convention is that these are defined in Capital letters.
Some of the system-defined variables as following:

Variable	Description
BASH	It represents the Bash Shell location.
BASH_VERSION	It specifies the Shell version which the Bash holds.
COLUMNS	It specifies the number of columns for our screen.
HOME	It specifies the home directory for the user.
LOGNAME	It specifies the logging user name.
OSTYPE	It tells the type of OS.
PWD	It represents the current working directory.
USERNAME	It specifies the name of currently logged in user.

Caution: Do not modify System variable this can sometimes create problems

User defined Variables

These are created and maintained by the user. These are generally defined in Lowercase letters.

How to define User defined variables (UDV)

To define UDV use following syntax

Syntax:

Variable name=value

'**value**' is assigned to a given '**variable name**' and value must be on the right side of = sign.

Example:

$ no=10 # this is ok

$ 10=no # Error, NOT Ok, Value must be on the right side of = sign.

To define variable called 'vech' having value Bus

$ vech=Bus

To define variable called n having value 10

$ n=10

Comments

Comments are lines of code which are not executed by the Script but are helpful to know some information about the script. To write a single line comment, insert a *hash (#)* before the line.

Example:

this is a comment

Echo Command

Echo command is used to display text or value of variable.

echo [options] [string, variables...]

Displays text or variables value on screen.

Escape Sequence	Purpose
\n	new line
\t	Horizontal tab
\\	Backslash

\”	Double quote
\’	Single Quote
\v	Vertical tab

For e.g. **$ echo -e "An apple a day keeps away\t\t doctor\n"**

Example 1:

```
#! /bin/bash
#this is a comment
echo "Hello World"
echo Our shell name is $BASH
echo Our shell version name is $BASH_VERSION
echo Our home directory is $HOME
echo Our current working directory is $PWD

name=vech
echo The name is $name
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
Hello World
Our shell name is /bin/bash
Our shell version name is 4.4.20(1)-release
Our home directory is /home/ansha
Our current working directory is /home/ansha/Desktop/shellscripting
The name is vech
ansha@ansha-VirtualBox:~/Desktop/shellscripting$
```

Reading User Input

To get input from the keyboard *Read* command is used. It takes input from the keyboard and assigns it to a variable. If you do not define a variable then it stores the value in a default variable named *REPLY*.

- read <variable_name>

To keep the input on silent mode, such that whatever be a user input on the command line will be hidden to others, we pass a username and hide the password (silent mode) by using the command line options (-s, -p).

- read -sp PROMPT <variable_name>

Where -s allows a user to keep the input on silent mode and -p to take input on the same line.

Example 2:

```
#! /bin/bash

echo "Enter name: "
read name
echo "Entered name : $name"
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
Enter name:
ansha
Entered name : ansha
ansha@ansha-VirtualBox:~/Desktop/shellscripting$
```

Example 3:

```
#! /bin/bash

echo "Enter name: "
read name1 name2 name3
echo "Names : $name1, $name2, $name3"
```

Output:

```
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
Enter name:
max tom john
Names : max, tom, john
ansha@ansha-VirtualBox:~/Desktop/shellscripting$
```

Example 4:

```
#! /bin/bash
read -p 'username : ' user_var
read -sp 'password: ' pass_var
echo
echo "username : $user_var"
echo "password : $pass_var"
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
username : tom
password:
username : tom
password : 1234
ansha@ansha-VirtualBox:~/Desktop/shellscripting$
```

Example 5:

```
#! /bin/bash
echo "Enter name :"
read
echo "Name : $REPLY"
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
Enter name :
tom
Name : tom
ansha@ansha-VirtualBox:~/Desktop/shellscripting$
```

Passing arguments to a Bash-Script

To pass any number of arguments to the bash function, we are required to insert them just after the function's name. We must apply spaces between function names and arguments.

- The given arguments are accessed as $1, $2, $3 ... $n, corresponding to the position of the arguments after the function's name.
- $0 variable is kept reserved for the function's name.
- $# specifies the total number (count) of arguments passed to the script.
- $@ stores the list of arguments as an array.

Example 6:

```
#! /bin/bash
echo $0 $1 $2 $3 ' > echo $1 $2 $3 '
args=("$@")
echo $@
echo $#
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh mark tom john
Lab2.sh mark tom john  > echo $1 $2 $3
mark tom john
3
ansha@ansha-VirtualBox:~/Desktop/shellscripting$
```

Arithmetic Operations

Like variables, arithmetic operations are also reasonably easy to apply.

Example 7:

```
#! /bin/bash

num1=100
num2=5

echo $(( num1 + num2 ))
echo $(( num1 - num2 ))
echo $(( num1 * num2 ))
echo $(( num1 / num2 ))
echo $(( num1 % num2 ))
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
105
95
500
20
0
```

Floating Point Operations

Example 8:

```
#! /bin/bash

num1=20.5
num2=5

echo "$num1+$num2" | bc
echo "$num1-$num2" | bc
echo "20.5*5"|bc
echo "scale=2;20.5/5" | bc
echo "20.5%5" |bc
echo "scale=2;sqrt($num2)"|bc -l
echo "scale=2;3^3" | bc -l
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
25.5
15.5
102.5
4.10
.5
2.23
27
```

bc stands for basic calculator and *-l* is the Math library.

DHA SUFFA UNIVERSITY
Department of Computer Science
CS-2004L
Operating Systems
Fall 2020

Conditional statements

If then statement

Syntax

- *if [expression];*
  then
  statements
  fi

For using multiple conditions with AND operator:

- *if [expression_1] && [expression_2];*
  then
  statements
  fi

For using multiple conditions with OR operator:

- *if [expression_1] || [expression_2];*
  then
  statements
  fi

For compound expressions with AND & OR operators, we can use the following syntax:

- *if [expression_1 && expression_2 || expression_3];*
  then
  statements
  fi

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	$a=4 $b=5 [$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.

Example 9:

```
#! /bin/bash

count=10
word=abc
if [ $count -eq 10 ]
then
        echo "condition is true"
fi

if [ $word == "abc" ]
then
        echo "condition is true"
fi
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
condition is true
condition is true
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
```

If then else statement

Syntax

- *if [condition];*
  then
  <if block commands>
  else
  <else block commands>
  fi

Example 10:

```
#! /bin/bash

echo "Enter Number"
read n
rem=$(($n%2))
if [ $rem -eq 0 ]
then
        echo "$n is Even Number"
else
        echo "$n is Odd Number"
fi
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
Enter Number
5
5 is Odd Number
```

If elif else statement

Syntax

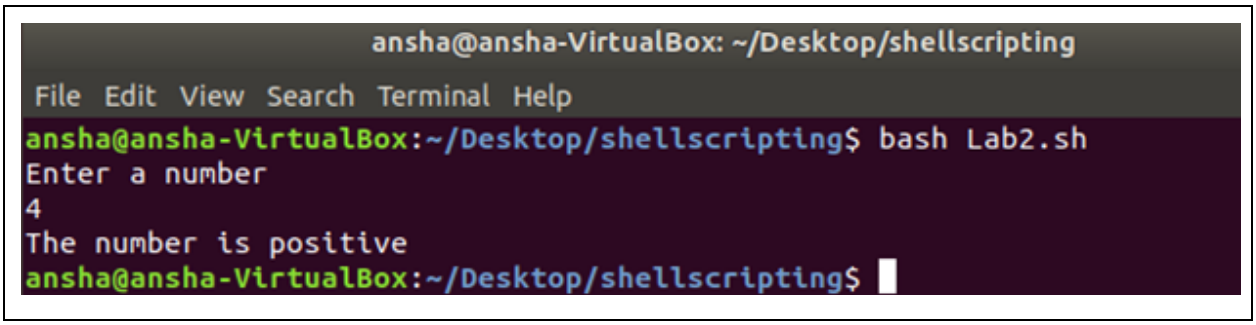
➢ *if [condition];*
then
<commands>
elif [condition];
then
<commands>
else
<commands>
fi

Example 11:

```
#! /bin/bash

echo "Enter a number"
read n
if [ $n -eq 0 ]
then
        echo "The entered number is zero"
elif [ $n -lt 0 ]
then
        echo "The number is negative"
else
        echo "The number is positive"
fi
```

Output:



Case Statements

Case statement is a good alternative for multi-level if then else conditional statements. It enables us to match several values against one value.

Syntax

- *case expression in*

pattern_1)
statements ;;
pattern_2)
statements ;;
pattern_3)
statements ;;
pattern_n)
statements ;;
**) Statements ;;*
esac

Example 12:

```
#! /bin/bash

echo "Enter OPtion Number"
read op
echo "Enter Number"
read a
echo "Enter Number"
read b
case $op in
        1) echo "sum =" $(( a + b ));;
        2) echo "sum =" $(( a - b ));;
        3) echo "sum =" $(( a * b ));;
        *) echo "Error" ;;
esac
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
Enter OPtion Number
5
Enter Number
3
Enter Number
6
Error
```

Array Variables

Bash supports a simple one dimensional array.

Example 13:

```
#! /bin/bash

os=('ubuntu' 'windows' 'kali-linux')

echo "${os[@]}"
echo "${os[0]}"
echo "${!os[@]}"
echo "${#os[@]}"
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
ubuntu windows kali-linux
ubuntu
0 1 2
3
ansha@ansha-VirtualBox:~/Desktop/shellscripting$
```

Loops

While Loop

The bash while loop can be defined as a control flow statement which allows executing the given set of commands repeatedly as long as the applied condition evaluates to true.

Syntax

- *while [expression];*
  do
  commands;

multiple commands;

done

Example 14:

```
#! /bin/bash

n=1

while [ $n -le 10 ]
do
        echo "$n"
        n=$(( n+1 ))
done
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
1
2
3
4
5
6
7
8
9
10
```

For Loop

- *for variable in list*

do

commands

done

OR

- *for ((expression1; expression2; expression3))*

do

commands

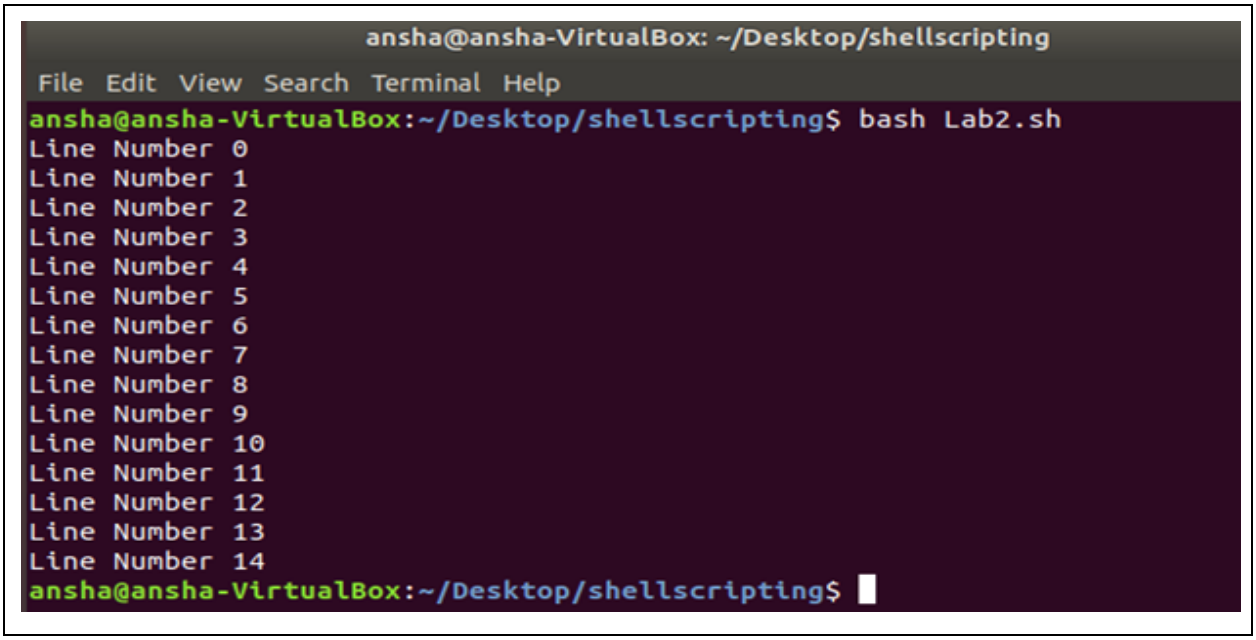
Done

Example 15:

```
#! /bin/bash


for((i=0; i<15; i+=1))
do
 echo "Line Number $i"
done
```

Output:



Example 16:

```
#! /bin/bash

for i in 1 2 3 4 5
do
        echo $i
done
```

Output:

```
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
1
2
3
4
5
```

Example 17:

```
#! /bin/bash

for i in {1..10}
do
        echo $i
done
```

Output:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
1
2
3
4
5
6
7
8
9
10
```

Nesting of for Loop

As you see the if statement can be nested similarly loop statements can be nested. You can nest the for loop. To understand the nesting of for loop see the following shell script.

```
for (( i = 1; i <= 5; i++ )) ### Outer for loop ###
do
 for (( j = 1 ; j <= 5; j++ )) ### Inner for loop ###
        do
                echo "$i "
        done
echo "" #### print the new line ###
done
```

Functions

A function in Shell can be defined as follows:

```
function_name()
{
}
```

E.g.

```
#Defining the function
Greet()
{
    echo Hello
}
#Invoking the function
Greet
```

Point to remember: You should only invoke the functions after it has been defined.

Arguments in a Shell 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.

Example 18:

```
#! /bin/bash

add()
{
        echo `expr $1 + $2`
}
#Invoking Function
add 10 20
```

Output:

```
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
30
```

Returning a value from a function

$? Depicts the value returned by the last command. Assigning that value to a variable means you are catching the value returned by the function that you invoked in the previous command.

Example 19:

```
#! /bin/bash

#Defining Function
add()
{
        temp=`expr $1 + $2`
        return $temp
}

#Invoking Function
add 10 20
sum=$?
        echo "Answer = $sum"
#Invoking Function

add 30 40
sum=$?
        echo "Answer = $sum"
```

Output:

```
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
Answer = 30
Answer = 70
ansha@ansha-VirtualBox:~/Desktop/shellscripting$
```

DHA SUFFA UNIVERSITY
Department of Computer Science
CS-2004L
Operating Systems
Fall 2020

ASSIGNMENT #02

1) Use the Bubble sort algorithm to sort an array in Bash.
2) Write a shell script to find the last prime number that occurs before the entered number.
3) Write a shell script to validate password strength. It should follow the following rules:

   Length can be minimum of 8 characters, contains both alphabet and number and includes both lower and uppercase letters. If the password doesn't comply with any of the above conditions, then the script should report it as a "Weak Password".

Note: Read submission guidelines carefully before submitting the assignment

SUBMISSION GUIDELINES

- Make a folder named with your Roll No e.g. **cs181xxx**
- Make separate bash files for all three questions named as question1.sh, question2.sh, question3.sh.
- Make a word file, add screenshots of code and output both in that word file
- Place all the files(.sh and word both) in the folder.
- Submit the folder at LMS