Lab 02 – Shell (BASH) Scripting in Linux Part I

Objectives

- 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

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.

Using Multiple Commands

So far you've seen how to use the command line interface (CLI) prompt of the shell to enter commands and view the command results. The key to shell scripts is the ability to enter multiple commands and process the results from each command, even possibly passing the results of one command to another. The shell allows you to chain commands together into a single step. If you want to run two commands together, you can enter them on the same prompt line, separated with a semicolon:

```
$ date ; who
Mon Feb 21 15:36:09 EST 2014
Christine tty2         2014-02-21 15:26
Samantha tty3         2014-02-21 15:26
Timothy  tty1         2014-02-21 15:26
user     tty7         2014-02-19 14:03 (:0)
```

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.

Determine Your Shell Type

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

Shell Scripting Part 1

Displaying Messages

Most shell commands produce their own output, which is displayed on the console monitor where the script is running. Many times, however, you will want to add your own text messages to help the script user know what is happening within the script. You can do this with the echo command.

Example:

```
#!/bin/bash
# This script displays the date and who's logged on
echo  The time and date are:
date
echo "Let's see who's logged into the system:"
who
```

That's nice, but what if you want to echo a text string on the same line as a command output? You can use the -n parameter for the echo statement to do that. Just change the first echo statement line to this:

```
echo -n "The time and date are: "
```

Using Variables

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 the 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

```
#!/bin/bash
# testing variables
days=10
guest="Katie"
echo "$guest checked in $days days ago"
days=5
guest="Jessica"
echo "$guest checked in $days days ago"
$
```

Each time the variable is referenced, it produces the value currently assigned to it. It's important to remember that when referencing a variable value you use the dollar sign, but when referencing the variable to assign a value to it, you do not use the dollar sign.

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:

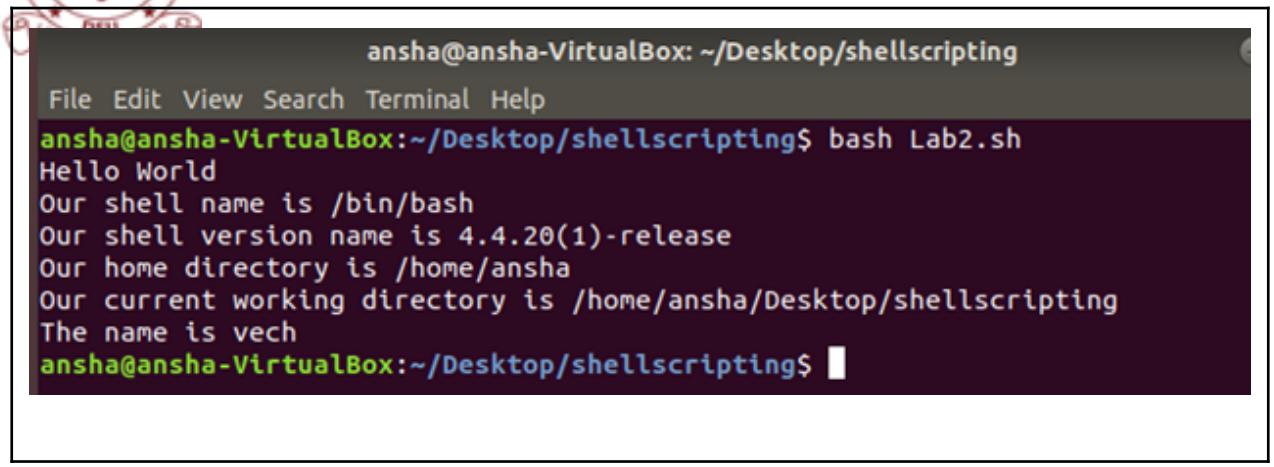
this is a comment

Example

```
#! /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:



Reading User Input

To get input from the keyboard, a 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 1

```
#! /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 2

```
#! /bin/bash

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

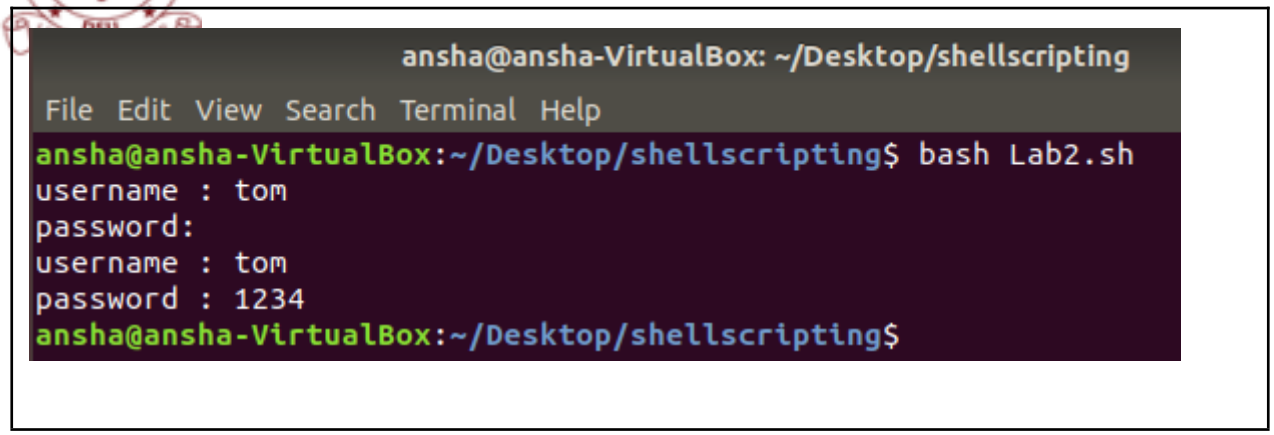
Output:

```
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 3

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

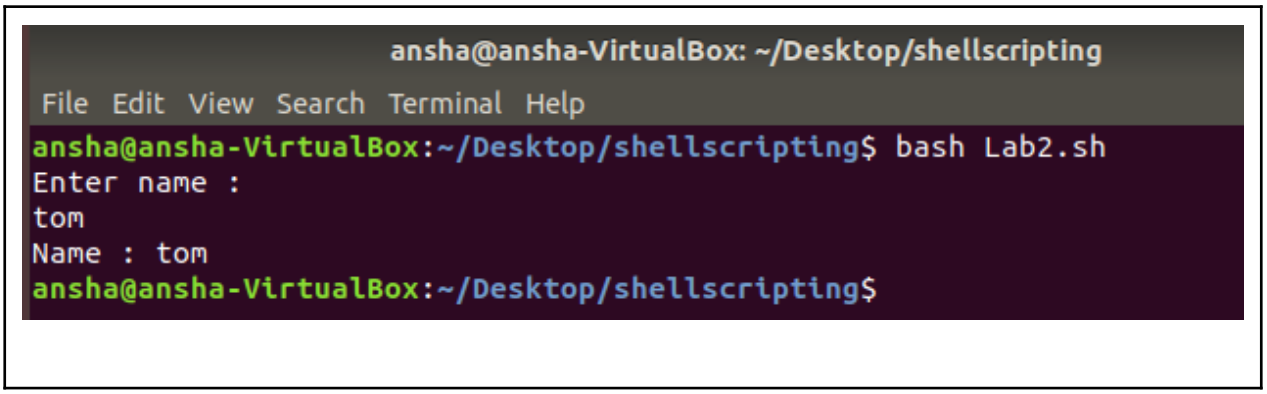
Output:



Example 4

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

Output:



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.

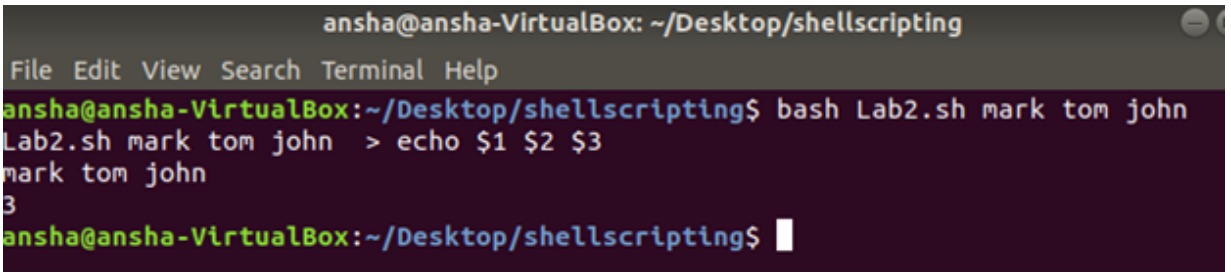
- The given arguments are accessed as $1, $2, $3 ... $n, corresponding to the position of the arguments after the function's name.

- the $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

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

Output:



Arithmetic Operations

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

Example

```
#! /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

You can use several solutions for overcoming the bash integer limitation. The most popular solution uses the built-in bash calculator, called bc.

The bash calculator is actually a programming language that allows you to enter floating point expressions at a command line and then interprets the expressions, calculates them,

and returns the result. The bash calculator recognizes these:

- Numbers (both integer and floating point)
- Variables (both simple variables and arrays)
- Comments (lines starting with a pound sign or the C language /* */ pair)

- Expressions
- Programming statements (such as if-then statements)
- Functions

Example

```
#! /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.

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 $b] is true.
-gt(>)	Checks if the value of the left operand is greater than the value of the right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt(<)	Checks if the value of the left operand is less than the value of the right operand; if yes, then the condition becomes true.	[$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

```
#! /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

```
#! /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

```
#! /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:

```
ansha@ansha-VirtualBox: ~/Desktop/shellscripting
File  Edit  View  Search  Terminal  Help
ansha@ansha-VirtualBox:~/Desktop/shellscripting$ bash Lab2.sh
Enter a number
4
The number is positive
ansha@ansha-VirtualBox:~/Desktop/shellscripting$
```

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 ;;*

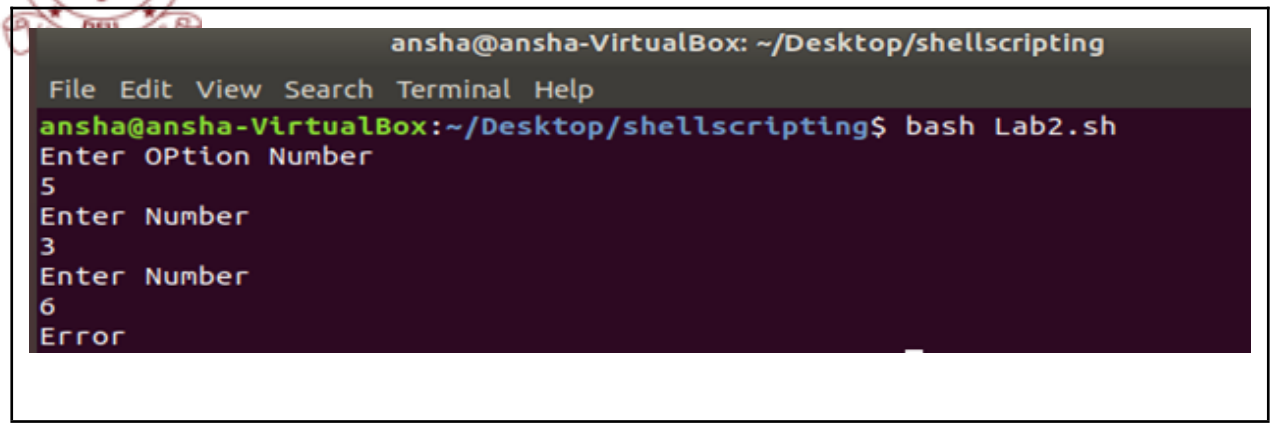
esac

Example

```
#! /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:



Lab Task

1. Practice Floating Point Operations Example
2. Practice examples of If/then/else, if/elif/else and case statements.

How to Submit

1. Copy/paste examples with ss in Google Docs.
2. Rename file as RollNo_LabTask1
3. Submit URL of Google Doc with “Viewer Only” settings.