

Lab No. 11

Writing & Executing Shell Scripts, I/O, Variables, and Operators

Objective

This lab introduces the fundamental concepts of shell scripting along with its basic constructs.

Activity Outcomes:

On completion of this lab students will be able to:

- Understand the process of writing and executing shell scripts
- Use input/output commands
- Use variables and operators

Instructor Notes

As pre-lab activity, read Chapter 24, 26, 28 & 34 from the book “The Linux Command Line”, William E. Shotts, Jr.

1) Useful Concepts

Introduction to Shell

Shell is an Interface between the user and the operating system. It is simply a program that is used to start other programs. It takes the commands from the keyboard and gives them to the operating system to perform the particular task. There are many different shells, but all derive several of their features from the Bourne shell, a standard shell developed at Bell Labs for early versions of UNIX. Linux uses an enhanced version of the Bourne shell called bash or the “Bourne-again” shell. The bash shell is the default shell on most Linux distributions, and /bin/sh is normally a link to bash on a Linux system. Other examples of Linux shells include: Korn, C Shell, tcsh, bash, zsh, etc.

Shell Script

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line. Shell scripts are the equivalent of batch files in MS-DOS, and can contain long lists of commands, complex flow control, arithmetic evaluations, user-defined variables, user-defined functions, and sophisticated condition testing. The basic advantage of shell scripting includes:

- Shell script can take input from user, file and output them on screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day today life.
- System Administration part can be also automated.

Writing and Executing Shell Script

We need to perform following steps to create and execute shell script

1. **Write Script:** Shell scripts are ordinary text files. We can use any text editor such as gedit to write shell script

2. **Make the shell script executable:** next step is to make the script executable. We can do it using the chmod command. For example if the script file is saved with the name test then we can make it executable in the following way:

```
sudo  chmod  777  test
```

3. **Place the script at some suitable place:** The shell automatically searches certain directories for executable files when no explicit pathname is specified. For maximum convenience, we will place our scripts in these directories. For example, we can move the executable file test to /usr/bin directory using the following command:

```
sudo  mv test /usr/bin
```

Now, we can execute the test file just like normal commands.

Writing First Shell Script:

Now, we write a simple Shell script to demonstrate the above mention steps. This script just shows a “Hello World” message on the terminal screen. Open the gedit and write the following code.

```
1 #!/bin/bash
2 #this is the first script
3 echo "Hello World"
```

The first line of the code tells about the shell for which we are writing the script. The second line is a comment while the third line displays the message on the terminal screen. After writing this code save it with the name test. Now, make it executable using the chmod command and move it to /usr/bin directory. Now, we can run this script by just writing test on the command line.

Shell Variables

Shell variables can be categorized as system-defined, parameter or user-defined. The details are given below:

1. **System defined** variables are already defined and included in the environment when we login. Their names are in upper case letters. Some of them are as follows:
 - HISTFILE – filename of the history file. Default is \$HOME/.sh history.
 - HISTSIZE – Maximum number of commands retained in the history file
 - HOME – The pathname of your home directory
 - IFS – Inter Filed Separator. Zero or more characters treated by the shell as delimiters in parsing a command line into words. Default – Blank, Tab and Newline in succession
 - PATH – List of directories separated by colon that are searched for executable
 - PWD – The present working directory
 - RANDOM – This is a random number from 0 – 32767. Its value will be different every time you examine it.
 - SHELL – The pathname of the shell

\$# – The number of parameters passed
\$\$ – The PID of the parent process i.e. shell
\$? – exit status of last command run

2. Parameter variables contain the values of the parameters passed to a shell script.

\$0 Path of the program
\$1, \$2 ... Store the parameters given to the script
\$* A list of all parameters, separated by the character defined in IFS
\$@ Same as \$* except the parameters are separated by space character

3. User defined variables are subject to the following rules:

- Variable names can begin with a letter or an underscore and can contain an arbitrary number of letters, digits and underscores
- No arbitrary upper limit to the variables you can define or use
- It is not necessary to declare a variable before using it
- Variables are of loosely typed
- A variable retains its value from the time it is set – whether explicitly by you or implicitly by the shell – until its value is changed or the shell exits
- To retrieve the value, precede the variable name with a **dollar sign (\$)**
- Quotes are used to specify values which contain spaces or special characters.
 - Double quotes (") prevent shell interpretation of special characters except \$ and `.
 - Single quotes (') prevent shell interpretation of all special characters.
 - The backslash (\) prevents the shell from interpreting the next character specially.

Input/Out-put Commands

echo- The echo command is used to display a line of text/string on standard output or a file. We use quotation marks to display text/string. To display the value of a variable, we need to place \$ symbol before the variable name.

```
echo "The value of x is $x"
```

read command- The read built-in command is used to read a single line of standard input i.e. keyboard. This command can be used to read keyboard input or, when redirection is employed, a line of data from a file. The syntax of read command is:

```
read -options arguments
```

Where options is one or more of the available options listed below and variable is the name of one or more variables used to hold the input value. If no variable name is supplied, the shell variable REPLY contains the line of data.

```
#!/bin/bash
echo "Please Enter a number"
read num
echo "You Entered $num"
```

Common options for read command are:

Option	Description
-a	<i>array</i> Assign the input to <i>array</i> , starting with index zero.
-n num	Read <i>num</i> characters of input, rather than an entire line.
-p prompt	-p <i>prompt</i> Display a prompt for input using the string <i>prompt</i> .
-t second	Timeout. Terminate input after <i>seconds</i> . read returns a non-zero exit status if an input times out
-u fd	Use input from file descriptor <i>fd</i> , rather than standard input

Operators

An operator is a symbol that usually represents an action or process. There various types of operators supported by bash shell.

Arithmetic Operators- Arithmetic operators are used to perform arithmetic operations on variables.

Following arithmetic operators are available in bash shell

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Integer Division
%	Remainder
**	Exponentiation

Writing Arithmetic Expressions- We use arithmetic expansion to perform arithmetic operations. It can be done using the following syntax

```
$((Arithmetic Expression))
```

In the following example, we perform some basic arithmetic operations

```
#!/bin/bash
echo "Please Enter First Number"
read num1
echo "Please Enter the Second Number"
read num2
echo "The sum is $((num1+num2))"
echo "The multiplication is $((num1*num2))"
echo "Square of sum of number is $((($num1+$num2)**2))"
```

We can also use expr command to write arithmetic expressions. The syntax is as given below:

```
echo `expr $num1 + $num2`
```

Note: There must be a space between operator and operands.

Assignment Operators

Assignment operators are used to assign values to variables. The following assignment operators are available in bash shell

Operator	Operation
=	Simple assignment. Assigns value to parameter. Example: a=\$b Note: there should be no space between = operator and operands
parameter+=value	Addition. Equivalent to parameter = parameter + value.
parameter-=value	Subtraction. Equivalent to parameter = parameter - value.
parameter*=value	Multiplication. Equivalent to parameter = parameter * value.
parameter/=value	Integer Division. Equivalent to parameter = parameter / value.
parameter%=value	Remainder. Equivalent to parameter = parameter % value.
parameter++	Post increment
Parameter--	Post decrement
++parameter	Pre increment
-- parameter	Post increment

Comparison Operators

These operators are used to compare the values of variables. Assume variable a holds 10 and variable b holds 20 then:

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

Logical Operators

These operators are used to perform logical operations. Suppose the values of variable a and b are 10 and 20 respectively then:

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.

-o	This is logical OR. If one of the operands is true, then the condition becomes true.	[\$a -lt 20 -o \$b -gt 100] is true.
-a	This is logical AND. If both the operands are true, then the condition becomes true otherwise false.	[\$a -lt 20 -a \$b -gt 100] is false.

File Operators

We have several operators that can be used to test file properties. Assume a variable file holds an existing file name "test" the size of which is 100 bytes and has read, write and execute permission on.

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.









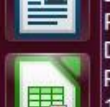

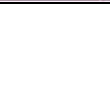
2) Solved Lab Activities

<i>Sr.No</i>	<i>Allocated Time</i>	<i>Level of Complexity</i>	<i>CLO Mapping</i>
1	10	Low	CLO-6
2	10	Low	CLO-6
3	10	Low	CLO-6
4	10	Low	CLO-6
5	10	Low	CLO-6

Activity 1:

Write a shell script that gets two numbers from user and perform basic arithmetic operations (+, -, *, /, %, **) on these numbers.










Solution:

Code	
     	<pre>*scripts (/usr/bin) - gedit #!/bin/bash echo "Please Enter the First Number" read num1 echo "Please Enter the Second Number" read num2 s=\$((num1+num2)) #addition echo "Sum of \$num1 and \$num2 is \$s" d=\$((num1-num2)) #subtraction echo "Dfiference of \$num1 and \$num2 is \$d" m=\$((num1*num2)) #multiplication echo "Product of \$num1 and \$num2 is \$m" div=\$((num1/num2)) #division echo "Division of \$num1 and \$num2 is \$div" r=\$((num1%num2)) #remainder echo "Remainder of \$num1 and \$num2 is \$r" e=\$((num1**num2)) #exponentiation echo "\$num1 raise to power \$num2 is \$e"</pre>
Out-put	
    	<pre>test_fa21 [Running] - Oracle VM VirtualBox File Machine View Input Devices Help Terminal File Edit View Search Terminal Help ubuntu@ubuntu:~\$ sudo chmod 777 scripts ubuntu@ubuntu:~\$ sudo mv scripts /usr/bin ubuntu@ubuntu:~\$ scripts Please Enter the First Number 18 Please Enter the Second Number 6 Sum of 18 and 6 is 24 Dfiference of 18 and 6 is 12 Product of 18 and 6 is 108 Division of 18 and 6 is 3 Remainder of 18 and 6 is 0 18 raise to power 6 is 34012224 [2]+ Done ubuntu@ubuntu:~\$ gedit /usr/bin/scripts</pre>

Activity 2:

Write a shell script that gets two numbers at the command line. First it displays the path of the script and perform basic arithmetic operations (+, -, *, /, %, **) on these numbers.



Solution:

Code	
     	<pre>scripts (/usr/bin) - gedit #!/bin/bash echo "Path of the Script is \$0" num1=\$1 num2=\$2 s=\$((\$num1+\$num2)) #addition echo "Sum of \$num1 and \$num2 is \$s" d=\$((\$num1-\$num2)) #subtraction echo "Dfiference of \$num1 and \$num2 is \$d" m=\$((\$num1*\$num2)) #multiplication echo "Product of \$num1 and \$num2 is \$m" div=\$((\$num1/\$num2)) #division echo "Division of \$num1 and \$num2 is \$div" r=\$((\$num1%\$num2)) #remainder echo "Remainder of \$num1 and \$num2 is \$r" e=\$((\$num1**\$num2)) #exponentiation echo "\$num1 raise to power \$num2 is \$e"</pre>
Out-put	
  	<pre>ubuntu@ubuntu: ~ ubuntu@ubuntu:~\$ scripts 18 6 Path of the Script is /usr/bin/scripts Sum of 18 and 6 is 24 Dfiference of 18 and 6 is 12 Product of 18 and 6 is 108 Division of 18 and 6 is 3 Remainder of 18 and 6 is 0 18 raise to power 6 is 34012224 ubuntu@ubuntu:~\$</pre>

Activity 3:

Write a shell script that creates a long list of all directories exists at the current location


Solution:

Code	
	<pre>scripts (/usr/bin) - gedit #!/bin/bash echo "Following Files Exist at the Current Location" ls -l</pre>
Out-put	
	<pre>ubuntu@ubuntu: ~ ubuntu@ubuntu:~\$ scripts Following Files Exist at the Current Location total 16 drwxr-xr-x 2 ubuntu ubuntu 80 May 16 08:15 Desktop drwxr-xr-x 2 ubuntu ubuntu 40 May 16 08:16 Documents drwxr-xr-x 2 ubuntu ubuntu 40 May 16 08:16 Downloads drwxr-xr-x 2 ubuntu ubuntu 40 May 16 08:16 Music drwxr-xr-x 2 ubuntu ubuntu 40 May 16 08:16 Pictures drwxr-xr-x 2 ubuntu ubuntu 40 May 16 08:16 Public -rw-r--r-- 1 ubuntu ubuntu 303 May 16 09:52 scripts drwxr-xr-x 2 ubuntu ubuntu 40 May 16 08:16 Templates -rwxr-xr-x 1 ubuntu ubuntu 8088 May 16 09:23 thread -rw-r--r-- 1 ubuntu ubuntu 1467 May 16 09:23 thread.cpp drwxr-xr-x 2 ubuntu ubuntu 40 May 16 08:16 Videos ubuntu@ubuntu:~\$</pre>

Activity 4:

Write a shell script that asks the user to enter the file name and deletes that file. After deleting the file it shows a success message and also displays the list of the remaining files

Solution:

Code	
	<pre>scripts (/usr/bin) - gedit #!/bin/bash echo "Following files exist at the current location" ls echo "Write the name of the file you want to delete" read fname rm -r \$fname echo "The remainig files are:" ls</pre>

```
Out-put
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ scripts
Following files exist at the current location
Desktop Downloads Pictures scripts test-dir thread.cpp
Documents Music Public Templates thread Videos
Write the name of the file you want to delete
test-dir
The remainig files are:
Desktop Downloads Pictures scripts thread Videos
Documents Music Public Templates thread.cpp
ubuntu@ubuntu:~$
```

Activity 5:

Write a shell script that modifies the basic mkdir command as: first it show the list of existing files on the current location. Then it asks the users to enter the name of the directory; and then creates that directory and display a success message.

Solution:

```
Code
scripts (/usr/bin) - gedit
#!/bin/bash
echo "Following files exist at the current location"
ls
echo "Write the name of the directory you want to create"
read dname
mkdir $dname
echo "$dname directory is created successfully"
ls

Out-put
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ scripts
Following files exist at the current location
Desktop Downloads Pictures scripts thread Videos
Documents Music Public Templates thread.cpp
Write the name of the directory you want to create
Mydir
Mydir directory is created successfully
Desktop Downloads Mydir Public Templates thread.cpp
Documents Music Pictures scripts thread Videos
ubuntu@ubuntu:~$
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write a shell script that asks the user to enter the name of the directory and the destination; and then creates a directory at the given destination. Also display a success message along with the list of directories only.

Task 2:

In this task you have to re-write the cp command in such a way that your script asks the users to enter the path of files to be copied and the destination; and then copies all of the files at the destination. Also display a success message along with the list of directories only.

Task 3:

Write a shell script that takes a file path and a search string as input; and displays all of the lines that contain that string.

Task 4:

Write a shell script that takes a list of text files as input and merges all those files into a single file; and displays its contents in the less application.

Lab No. 12

Writing Shell Scripts using Conditional-Statements, and Loops

Objective

The objective of this lab is to familiarize the students with the use of conditional and looping statements in shell scripting.

Activity Outcomes:

On completion of this lab students will be able to

- Write shell scripts that uses conditional statements
- Use looping statements available in bash shell

Instructor Notes

As pre-lab activity, read Chapter 27 &31 from the book “The Linux Command Line”, William E. Shotts, Jr.

1) Useful Concepts

Conditional Statements

A conditional statement tells a program to execute an action depending on whether a condition is true or false. It is often represented as an if-then or if-then-else statement.

if Statement

if statement is provided in many variant in bash shell. The most commonly used syntax is as given below:

```
if [ condition ]  
    then  
        commands  
fi
```

Note that there must be a space between condition and both opening and closing brackets. The syntax of else-if command is

```
if [ condition ]  
then  
    commands  
elif [ condition ]
```

```

then
    commands
    .
    .
    .
else
    commands
fi

```

The following example shows the use of if statement. In this example, we take a number as input from user and check whether it is even or odd.

```

#!/bin/bash
echo "Please Enter a Number"
read num
if [ $(num%2) -eq 0 ]
then
    echo "$num is an even
number"
else
    echo "$num is an odd
number"
fi

```

Exit Status

Commands issue a value to the system when they terminate, called an exit status. This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command's execution. By convention, a value of zero indicates success and any other value indicates failure. The exit status of a command is saved in a system variable \$?.

Other syntax for if statement

Recent versions of bash include a compound command that acts as an enhanced replacement for test. It uses the following syntax:

```

if [[ condition ]]
then
    commands
fi

```

Similarly, (()) syntax can also be used which is designed for arithmetic expressions.

Case statement

The bash case statement is generally used to simplify complex conditionals when you have multiple different choices. Using the case statement instead of nested if statements will help you make your bash scripts more readable and easier to maintain. The syntax of case statement is:

```
case    EXPRESSION
in
Pattern-1)

Commands
;;
Pattern-2)

Commands
;;
.
.
.
Pattern-n)

Commands
;;
*)

Commands
;;
esac
```

Following are some examples of some valid patterns for the case statement.

Pattern	Description
a)	Matches if <i>word</i> equals “a”.
[[:alpha:]]	Matches if <i>word</i> is a single alphabetic character.
???)	Matches if <i>word</i> is exactly three characters long
*.txt)	Matches if <i>word</i> ends with the characters “.txt”.
*)	Matches any value of <i>word</i> . It is good practice to include this as the last pattern in a case command, to catch any values of <i>word</i> that did not match a previous pattern; that is, to catch any possible invalid values.

Looping Statements

A program loop is a series of statements that executes for a specified number of repetitions or until specified conditions are met. While, until and for are the common looping statement provided by bash shell.

For Loop

The original syntax of for loop is:

```
for    variable    in
values
do
        statements
done
```

Following example shows the working of for loop

<pre>#!/bin/bash for i in A B C D E do echo \$i done</pre>	Out-Put A B C D E
--	--

We can give a range of values as {0..9}. bash shell also supports a C like syntax of for loop which is given below:

<pre>for ((Expression; Expression2; Expression)) do statements done</pre>

While Loop

The syntax of while loop is as given below

<pre>While ((Condition)) do statements done</pre>

Following example shows the working of while loop

<pre>#!/bin/bash count=1 while [[\$count -le 5]] do echo \$count count=\$((count + 1)) done</pre>	Out-Put 1 2 3 4 5
---	--

Breaking the while loop: bash provides two built-in commands that can be used to control program flow inside loops. The break command immediately terminates a loop, and program control resumes with the next statement following the loop. The continue command causes the remainder of the loop to be skipped, and program control resumes with the next iteration of the loop.

Until Loop

The until command is much like while, except instead of exiting a loop when a nonzero exit status is encountered, it does the opposite. An until loop continues until it receives a zero exit status i.e. the condition becomes true.

<pre>#!/bin/bash count=1 until [[\$count -gt 5]] do echo \$count count=\$((count + 1)) done</pre>	Out-Put 1 2 3 4 5
---	--

2) Solved Lab Activities

Sr.No	Allocated Time	Level of Complexity	CLO Mapping
1	10	Low	CLO-6
2	15	Medium	CLO-6
3	15	High	CLO-6

Activity 1:

Write a shell script that modifies the mkdir command as: first it takes the directory name from the user as input and checks if a directory with same name exists then shows an error message otherwise creates the directory and show the success message.

Solution:

Code

```

sh_ex (~/) - gedit
1 #!/bin/bash
2 echo "Please Enter the Name of the Directory You Want to Create"
3 read dname
4 if [[ -e $dname ]]
5 then
6 echo "$dname Already Exists"
7 else
8 mkdir $dname
9 ls
10 fi

```

Out-put

```

ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./sh_ex
Please Enter the Name of the Directory You Want to Create
testdir
testdir Already Exists
ubuntu@ubuntu:~$ ./sh_ex
Please Enter the Name of the Directory You Want to Create
testdir2
abc Documents even odd script Templates test_file Videos
cd Downloads Music Pictures scripts testdir thread
Desktop ef Mydir Public sh_ex testdir2 thread.cpp
ubuntu@ubuntu:~$

```

Activity 2:

Write a shell script that takes a word as input and finds whether it is a single alphabet, ABC followed by a digit, of length 3, ends with .txt or it is something else.

Solution:

Code	Out-put
#!/bin/bash	Out-Put
read -p "enter word > "	1
case \$REPLY in	2
[:alpha:]) echo "is a single alphabetic character." ;;	3
[ABC][0-9]) echo "is A, B, or C followed by a digit." ;;	4

<pre> ???) echo "is three characters long." ;; *.txt) echo "is a word ending in '.txt'" ;; *) echo "is something else." ;; esac </pre>	5
--	---

Activity 3:

Write a shell script that, given a filename as the argument will write the even numbered line to a file with name even file and odd numbered lines in a text file called odd file.

Solution:

Code

Open

```

1 #!/bin/bash
2 echo "Please Enter the Name of the File You Want to Read"
3 read fname
4 line_num=1
5 while read line
6 do
7 if [[ $((line_num%2)) -eq 0 ]]
8 then
9 echo $line >> even
10 else
11 echo $line >> odd
12 fi
13 line_num=$((line_num+1))
14 done < $fname

```

Output

ubuntu@ubuntu: ~

ubuntu@ubuntu:~\$./sh_ex

Please Enter the Name of the File You Want to Read

test_file

ubuntu@ubuntu:~\$ cat even

drwxr-xr-x 2 root root 3406 Apr 12 2017 bin

dr-xr-xr-x 1 root root 2048 Apr 12 2017 cdrom

drwxr-xr-x 146 root root 620 May 16 08:16 etc

lrwxrwxrwx 1 root root 33 Apr 12 2017 initrd.img -> boot/initrd.img-4.10.0-19-generic

drwxr-xr-x 3 root root 80 May 17 05:20 media

drwxr-xr-x 2 root root 3 Apr 12 2017 opt

drwxr-xr-x 21 root root 325 Apr 12 2017 rofs

drwxr-xr-x 29 root root 900 May 18 08:19 run

drwxr-xr-x 2 root root 3 Apr 6 2017 snap

dr-xr-xr-x 13 root root 0 May 24 06:07 sys

drwxr-xr-x 15 root root 120 May 16 09:52 usr

lrwxrwxrwx 1 root root 30 Apr 12 2017 vmlinuz -> boot/vmlinuz-4.10.0-19-generic

ubuntu@ubuntu:~\$ cat odd

total 2

drwxr-xr-x 4 root root 60 May 16 08:15 boot

drwxr-xr-x 19 root root 3920 May 18 08:19 dev

drwxr-xr-x 3 root root 60 May 16 08:15 home

drwxr-xr-x 26 root root 60 May 16 08:15 lib

drwxr-xr-x 2 root root 3 Apr 12 2017 mnt

dr-xr-xr-x 171 root root 0 May 16 08:15 proc

drwx----- 3 root root 60 Apr 12 2017 root

drwxr-xr-x 2 root root 4553 Apr 12 2017/sbin

drwxr-xr-x 2 root root 3 Apr 12 2017/srv

drwxrwxrwt 12 root root 300 May 24 05:17 tmp

drwxr-xr-x 20 root root 160 May 16 08:15 var

ubuntu@ubuntu:~\$

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write a shell script that asks the user to enter the marks for three subjects and calculates the GPA for each subject along with the CGPA.

Task 2:

Write a menu-driven shell script that gives four options A, B, C and Q to the user to select one of them. If user enters A then it displays the host-name and uptime, if user enters B then it gives information about disk and memory space, if user enter C then it gives information about home space utilization and if user enters Q then it quits the program.

Task 3:

Write a shell script that, given a filename as the argument will count vowels, blank spaces, characters, number of line and symbols.

Lab No. 13

Using Arrays, and Functions in Shell Scripts

Objective

This lab is designed to introduce the usage of arrays and functions in shell scripting.

Activity Outcomes:

On completion of this lab students will be able to:

- Write shell scripts using array
- Write functions

Instructor Notes

As pre-lab activity, read Chapter 35 & 31 from the book “The Linux Command Line”, William E. Shotts, Jr.

1) Useful Concepts

Arrays

Arrays are variables that hold more than one value at a time. Arrays are organized like a table. Arrays in bash are limited to a single dimension.

Creating an Array

Array variables are named just like other bash variables, and are created automatically when they are accessed. Here is an example:

<pre>#!/bin/bash a[1]=5 echo "\${a[1]}"</pre>	Out-put 5
---	---------------------

We can also use the declare command to declare an array. The syntax is given below

```
declare -a array_name
```

Adding Values to an Array

New values can be added to an array using the following syntax

```
array_name[index]=value
```

To add multiple values, we use the following syntax

```
array_name=(value1 value2 ... )
```

These values are assigned sequentially to elements of the array, starting with element zero. It is also possible to assign values to a specific element by specifying a subscript for each value:

```
array_name=[index]=value1 [index]=value2 ... )
```

Accessing Array Elements

Array elements can be accessed as follows

```
array_name=([index]=value1 [index]=value2 ... )
```

Operations on Arrays

Out-putting Entire Array: by using * or @ as index, we can output an entire array.

```
#!/bin/bash
animals=("a dog" "a cat" "a fish")
for i in ${animals[*]}
do echo $i;
done
```

Out-put

```
a dog
a cat
a fish
```

Note: we can also use \${animals[@]} instead of \${animals[*]}. If we use "" marks i.e. "\${animals[@]}" then contents are displayed on single line

Determining the Number of Array Elements: we can find the total number of elements in an array by using following

```
${#array_name[@]}
```

While the length of an element can be found as

```
${#array_name[index]}
```

The following example shows the usage of these

```
#!/bin/bash
a[100]=foo
echo ${#a[@]}          # number of array elements
echo ${#a[100]}        # length of element 100
```

Out-put

```
1
3
```

Finding the Index Used by an Array: As bash allows arrays to contain “gaps” in the assignment of subscripts, it is sometimes useful to determine which elements actually exist. This can be done with a parameter expansion using the following forms:

```
${!array_name[@]} or ${#array_name[*]}
```

The following example shows the usage of this

```
#!/bin/bash
foo=( [2]=a [4]=b [6]=c)
for i in "${!foo[@]}"
do
    echo $i
done
```

Out-put

```
2
4
6
```

Adding Elements to the End of an Array:

```
#!/bin/bash
foo=(a b c)
echo ${foo[@]}
foo+=(d e f)
echo ${foo[@]}
```

Out-put

```
a b c
a b c d e f
```

Sorting an Array:

<pre>#!/bin/bash a=(f e d c b a) echo "Original array: \${a[@]}" a_sorted=(\$(for i in "\${a[@]}"; do echo \$i; done sort)) echo "Sorted array: \${a_sorted[@]}"</pre>	Out-put Original array: f e d c b a Sorted array: a b c d e f
--	--

Deleting an Array:

<pre>#!/bin/bash foo=(a b c d e f) echo \${foo[@]} unset foo echo \${foo[@]}</pre> <p>Note: to delete a specific index, we can use unset 'foo[index]'</p>	Out-put a b c d e f
--	-------------------------------

Writing Functions

A Bash function is essentially a set of commands that can be called multiple times. The purpose of a function is to help you make your bash scripts more readable and to avoid writing the same code repeatedly. Compared to most programming languages, Bash functions are somewhat limited.

The syntax for declaring a bash function is straightforward. Functions may be declared in two different formats:

<pre>function-name(){ Commands }</pre> <p>Or</p> <pre>function function-name(){ Commands }</pre>
--

Functions can be called by name.

<pre>#!/bin/bash hello_world () { echo 'hello, world' } hello_world</pre>	Out-put hello world
---	-------------------------------

We can define local variables within the function using the **local** keyword. To return a value, we can use return statement. Following example shows the use of return command.

<pre>#!/bin/bash my_function () { echo "hello world" return 55 } my_function</pre>	Out-put hello world 55
---	-------------------------------------

echo \$?	
----------	--

Arguments can be passed to functions in the following way.

<pre>#!/bin/bash greeting () { echo "Hello \$1" } greeting "Ali"</pre>	Out-put Hello Ali
---	-----------------------------

2) Solved Lab Activities

Sr.No	Allocated Time	Level of Complexity	CLO Mapping
1	20	Medium	CLO-6
2	25	Medium	CLO-6
3	25	Medium	CLO-6

Activity 1:

Write a shell script that accepts 10 numbers from user as input and finds: maximum, minimum, odd/even of them.

Solution:

Code
 <pre>#!/bin/bash count=0 while [[\$count -lt 10]] do echo "Please Enter a Number at Index \$count" read num num_array[\$count]=\$num count=\$((count+1)) done max=0 for val in \${num_array[@]} do if [[\$val -gt \$max]] then max=\$val fi done echo "Max is \$max "</pre>
Out-put

```
ubuntu@ubuntu:~$ ./sh_ex
Please Enter a Number at Index 0
23
Please Enter a Number at Index 1
33
Please Enter a Number at Index 2
22
Please Enter a Number at Index 3
33
Please Enter a Number at Index 4
334
Please Enter a Number at Index 5
33
Please Enter a Number at Index 6
23
Please Enter a Number at Index 7
23
Please Enter a Number at Index 8
56
Please Enter a Number at Index 9
54
Max is 334
ubuntu@ubuntu:~$
```

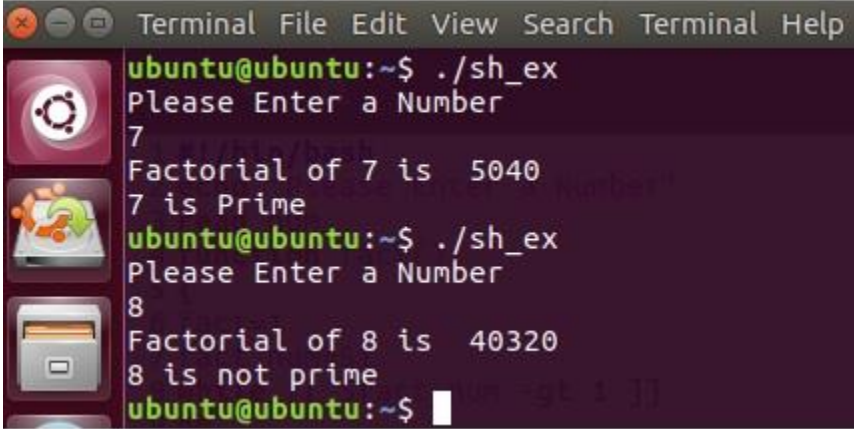
Activity 2:

Write a shell script that contains two function `fact()` and `is_prime()`. `fact()` function finds the factorial of a number while `is_prime()` checks whether a number is prime or not. Your script should take an integer as input from user and pass this number to `fact()` and `is_prime()` functions as argument.

Solution:

Code	
	<code>1 #!/bin/bash</code>
	<code>2 echo 'Please Enter a Number'</code>
	<code>3 read num</code>
	<code>4 function fact()</code>
	<code>5 {</code>
	<code>6 fact=1</code>
	<code>7 fact_num=\$1</code>
	<code>8 while [[\$fact_num -gt 1]]</code>
	<code>9 do</code>
	<code>10 fact=\$((fact*fact_num))</code>
	<code>11 fact_num=\$((fact_num-1))</code>
	<code>12 done</code>
	<code>13 echo "Factorial of \$num is \$fact"</code>
	<code>14 }</code>
	<code>15</code>
	<code>16 function is_prime()</code>
	<code>17 {</code>
	<code>18 p_num=\$1</code>
	<code>19 upper_limit=\$((p_num/2))</code>
	<code>20 count=2</code>
	<code>21 ans=1</code>
	<code>22</code>
	<code>23 for((count=2; count<=\$upper_limit; count++))</code>
	<code>24 do</code>
	<code>25 if [[\$((p_num%count)) -eq 0]]</code>
	<code>26 then</code>
	<code>27 ans=0</code>
	<code>28 break</code>
	<code>29 fi</code>
	<code>30 done</code>
	<code>31</code>
	<code>32 if [[\$ans -eq 1]]</code>
	<code>33 then</code>
	<code>34 echo "\$p_num is Prime"</code>
	<code>35 else</code>
	<code>36 echo "\$p_num is not prime"</code>
	<code>37 fi</code>
	<code>38 }</code>
	<code>39 fact \$num</code>
	<code>40 is_prime \$num</code>

Out-put



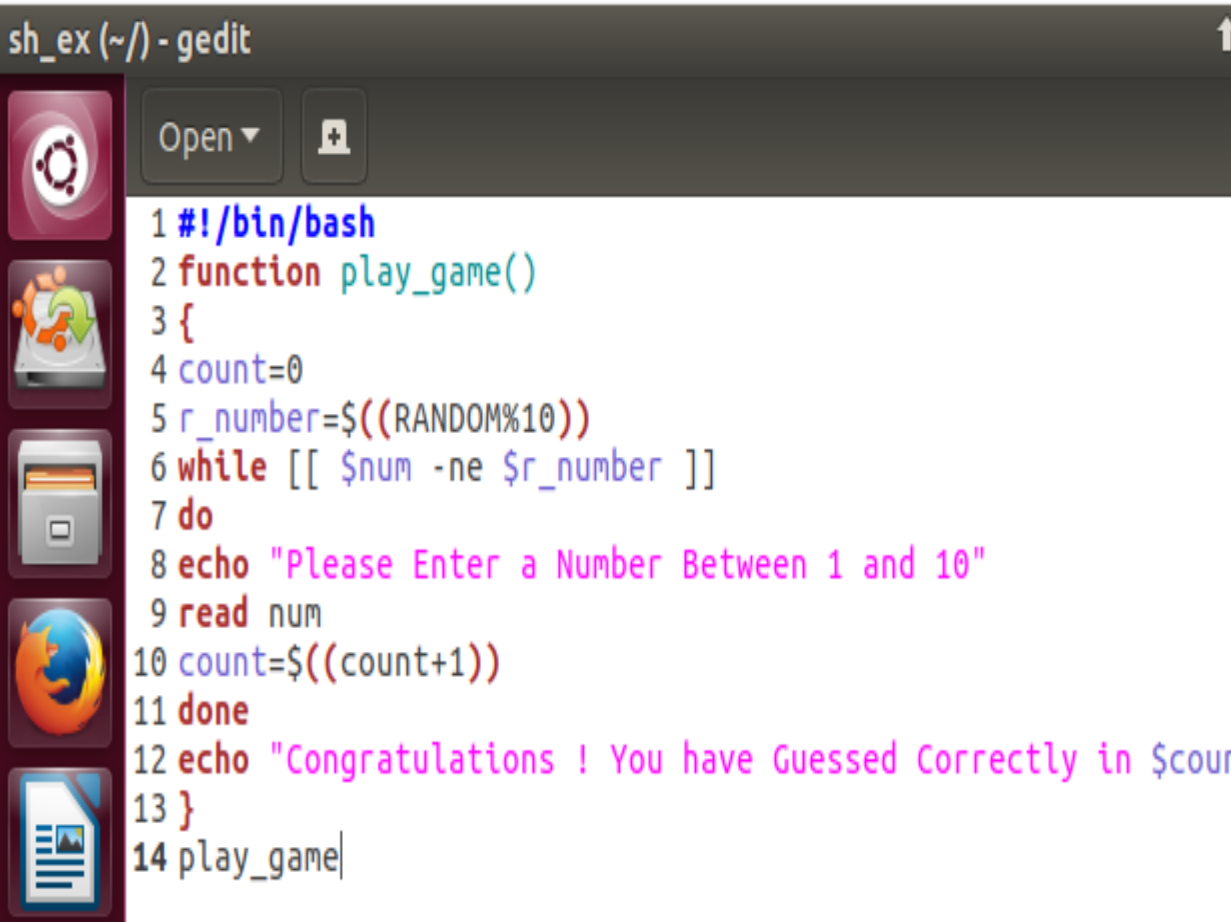
```
Terminal File Edit View Search Terminal Help
ubuntu@ubuntu:~$ ./sh_ex
Please Enter a Number
7
Factorial of 7 is 5040
7 is Prime
ubuntu@ubuntu:~$ ./sh_ex
Please Enter a Number
8
Factorial of 8 is 40320
8 is not prime
ubuntu@ubuntu:~$
```

Activity 3:

Write a shell script that contains a function `play_game`. This function generates a random number between 1 and 10; and keeps on asking the user to guess the number as long as user enters a number which is equal to the random number. In the end, the total number of attempts made by the user to enter correct guess is displayed.

Soltuion:

Code



```
sh_ex (~/) - gedit
1 #!/bin/bash
2 function play_game()
3 {
4     count=0
5     r_number=$((RANDOM%10))
6     while [[ $num -ne $r_number ]]
7     do
8         echo "Please Enter a Number Between 1 and 10"
9         read num
10        count=$((count+1))
11    done
12    echo "Congratulations ! You have Gussed Correctly in $count attempts"
13 }
14 play_game
```

```
Out-put
ubuntu@ubuntu: ~
ubuntu@ubuntu:~$ ./sh_ex
Please Enter a Number Between 1 and 10
3
Please Enter a Number Between 1 and 10
4
Please Enter a Number Between 1 and 10
5
Congratulations ! You have Guessed Correctly in 3 Attempts
ubuntu@ubuntu:~$
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write a function that finds the sum of digits in an integer

Task 2:

Write a function that finds whether an integer is a palindrome or not

Task 3:

Write a function that writes an integer in reverse (for example writes 123 as 321)