Role	Name	Affiliation
Principal Investigator	Dr. Savita Gandhi	Professor, Dept. of Computer Science, Gujarat University, Ahmedabad
Content Writer	Mr. Hardik Joshi	Asst. Professor, Dept. of Computer Science, Gujarat University, Ahmedabad
Content Reviewer	Dr. Hiren Joshi	Professor, Dept. of Computer Science, Gujarat University, Ahmedabad

Item	Description
Subject Name	Information Technology
Paper Name	Open Source Software
Module No	15
Module Name	Bash Shell Scripting - 3
Pre-requisite	Basics of Linux
Objectives	Create shell scripts that uses arrarys, files and learn to debug shell scripts
Keywords	Shell scripting

Bash Shell Scripting - 3

"When the superior programmer refrains from coding, his force is felt for a thousand miles."

-Eric S. Raymond

Learning Objectives:

By the end of this module, you should be able to:

- Learn to produce effective output using echo and tput command
- Learn how to debug shell scripts
- Learn how to discard intermediate output using /dev/null file
- Learn different methods to read from text file
- Explore the usage of arrays in shell scripting

1. Formating output using echo and tput commands in Shell Scripting

echo command

In the previous modules, we have used echo in the simplest form. The echo command supports escape sequences to manipulate the output of shell scripts. For instance, we can change the colour of text, make it bold or underlined and also insert tab space or newline using echo. Let us go through certain tricks that can be applied while using echo command within shell script. Table 1 lists various options available with echo command and its brief description. While using these options, -e must be used.

Options	Description
-n	do not print the trailing newline.
-e	use to enable interpretation of backslash escapes.
\b	use backspace
\\	use backslash
\n	use new line
\r	use carriage return
\t	use horizontal tab
\v	use vertical tab

Table 1: Options of echo command

The following is an example command followed by its output that uses backspace, tab and newline with echo:

```
$echo -e "A quick brown\b\b\b fox jumps \t\t over the lazy \n dog"
```

Output:

```
A quick br fox jumps          over the lazy
dog
```

Special escape sequences can be used with echo command to display text in different colours or to apply formatting to the output text. The following script demonstrates the use of special sequence characters to format the output text. The escape sequences are stored in meaningful variable names. For instance, to make the text underlined, we have used `UNDERLINE='\033[4m'. Here, '\033[4m'` indicates the escape sequence used for underlining the output.

```
#!/bin/bash
PATH=/bin:/usr/bin:
NONE='\033[00m'
RED='\033[01;31m'
```

```
GREEN='\033[01;32m'
YELLOW='\033[01;33m'
PURPLE='\033[01;35m'
CYAN='\033[01;36m'
WHITE='\033[01;37m'
BOLD='\033[1m'
UNDERLINE='\033[4m'
REVERSE='\033[7m'

echo -e "This text is ${RED}red${NONE} and ${GREEN}green${NONE} and
${BOLD}bold${NONE} and ${UNDERLINE}underlined${NONE} and finally ${REVERSE}
reverse video."
```

tput command

Apart from echo, there is a tput command that can control the way the output is displayed on the screen. The tput command is designed to work for different terminal types of Unix and Linux systems. Few options that are available with tput command are listed in table 2

Option	**Action**
clear	Clears the screen
cup r c	Places the cursor to row r and column c
bold	Makes the display bold
rev	Makes the display in reverse video
smul	Starts underline mode
rmul	Ends the underline mode
lines	Displays the number of lines on the screen
cols	Displays the number of columns on the screen
sgr0	Restores to the original mode

Table 2: Options of tput command

The following shell script demonstrates the use of tput command with various options. Such options will be helpful when we want to create menu driven scripts that are interactive and which requires the formatting to highlight parts of screen. It is required to use `tput sgr0` occasionally to restore the default screen.

```
#!/bin/bash
#Demonstration of tput command
tput clear
echo -e "Total no. of columns on screen = \c"
tput cols
echo -e "Total no. of lines on screen = \c"
```

```
tput lines
echo "This is normal text"
tput bold
echo "This is in bold"
tput sgr0
tput rev
echo "This is in reverse video"
tput sgr0
tput smul
echo "This is underlined"
tput rmul
echo "Now, the cursor will be positioned at 10,20"
tput cup 10 20
echo "My new place..."
tput sgr0
```

2. Debugging Shell scripts and dealing with errors

At times, while working with scripts and commands, we may face runtime errors. These may be due to errors in the script, such as an incorrect syntax, or missing file or insufficient permission to do an operation. In most of the cases, the errors may be reported with a specific error code or some message, but often just yield incorrect or confusing output. In shell scripting, there are ways and means to identify whether a command has executed successfully or not. The process of identifying errors while executing the shell script is known as debugging.

Debugging helps us to troubleshoot and resolve errors, it is one of the most important task that is usually performed by programmers and system administrators.

There are two popular ways to debug in shell scripting. They are as follows:

- Executing bash script in debug mode:

  We can execute a bash script in debug mode by issuing the bash command as:

  ```
  $bash -x ./script_filename.sh
  ```

- Bracketing part of script using "set" command:

  Within the script, we can bracket the parts of the script with "`set -x`" and "`set +x`" that are to be debugged.

The above technqiue helps to debug because it prefixes each command with the '+' character that is within the script, it displays each command before executing it and it can debug only selected block of a script which can be as follows:

```
set -x    # debugging is turned on
...
set +x    # debugging is turned off
```

The following is a sample shell script that uses debugging mode that uses bracketing technique.

```
#!/bin/bash
#A script that demonstrates debugging
echo "Error debugging turned on ..."
set -x
date
ps
cal
echo "Error debugging turned off..."
set +x
date
```

In Linux and Unix, all programs that run are given three open file streams as listed in table 3:

File stream	Description	File Descriptor
stdin	Standard Input, by default it is the keyboard or terminal for programs run from the command line	0
stdout	Standard output, by default is the screen	1
stderr	Standard error, where output error messages are shown or saved	2

Table 3: File streams and various descriptors

Using redirection, we can save the stdout and stderr output streams or files for later analysis after a program or command is executed. Let us take an example of script that may run into errors. The following script may enter into a runtime error if the file with .java extension is not present in the current directory. Let us presume that the script generates error.

```
#!/bin/bash
#A script that demonstrates error
#Usage: ./script_filename.sh 2> error.txt
cnt=0
for i in `ls *.java`
     do
     echo $i
     cnt=`expr $cnt + 1`
done
```

Under such circumstances, the program can be executed as shown below:

```
$./script_filename.sh 2> error.txt
```

In the presence of any error, the error message will be stored in the file error.txt that can be displayed afterwards using the following command:

```
$cat error.txt
```

3. Discarding Output with `/dev/null`

Certain commands like find or grep will produce huge amount of output, such outputs can overwhelm the console. Such commands may create problems (by producing output that is not required) while executing scripts, if they are used within the scripts. To avoid this, we can redirect the unwanted output to a special file (a device node) called `/dev/null`. This pseudofile is also called the bit bucket or black hole.

The following script is an example where we discard unwanted output to the `/dev/null` file while the program is executing. In the following script, user is asked to enter the name of friend(user-id) whom he wants to check whether the user has logged in or not. The script can run in background till the user wants to keep on checking whether the friend has logged-in or not in that time frame. In this script, the grep command may produce unnecessary output after each interval of one minute. So, to hide this output, we redirect it into the `/dev/null` file.

```
#!/bin/bash
#Check user
echo -n "Enter the login id of your friend: "
read name
interval=0
echo -n "Enter the unit of time in minutes: "
read min
until who | grep -w "$name" > /dev/null
do
sleep 60
interval=`expr $interval + 1`
if [ $interval -gt $min ]
then
echo "$name has not logged in since $min minutes"
exit
fi
done
echo "$name has now logged in..."
```

4. Reading from files

At times, it is required to process files from the shell scripts. Online transaction systems or billing systems usually process sequence of records from files. Small shell scripts can be helpful in processing voluminous amount of data and such scripts can be created within minutes. Let us explore few ways of reading files line by line from shell scripts. Although there are many different methods to read files, we will explore few of them which are relatively simple.

Method 1: Piped while-read loop

In the following shell script, the while loop accepts output of the command `cat $FILENAME` and processes each line, the program calculates number of lines of the file.

```
#!/bin/bash
#Process a file line by line using while-read loop.

FILENAME=$1
cnt=0
cat $FILENAME | while read LINE
do
let cnt++
echo "$cnt $LINE"
done
echo -e "\nTotal $cnt lines read from the file $FILENAME"
```

Method 2: Redirected "while-read" loop

In the following shell script, the while loop accepts input from the file using the statement `done < $FILENAME`, each line of the file is read using while loop. This script counts the number of lines of the given file.

```
#!/bin/bash
#Process a file line by line using redirected while-read loop.

FILENAME=$1
count=0
while read LINE
do
let count++
echo "$count $LINE"
done < $FILENAME
echo -e "\nTotal $cnt lines read from the file $FILENAME"
```

Method 3: Using file descriptors

exec is a shell command that replaces the current shell process with the command specified after exec. The first exec command redirects stdin to file descriptor 3. The use

of second exec command redirects the $FILENAME into stdin, which is indicated by the file descriptor 0. When the while loop executes, the loop takes input from the file and reads into the LINE variable. When the while loop exits, to restore the file descriptors, the exec restores it back to the original file descriptor 0.

```
#!/bin/bash
#Process a file line by line using while read LINE Using
#File Descriptors


FILENAME=$1
count0=
exec 3<&0
exec 0< $FILENAME

while read LINE
do
let count++
echo "$count $LINE"
done
exec 0<&3
echo -e "\nTotal $cnt lines read from the file $FILENAME"
```

Method 4: Using combination of head and tail commands

Use of head and tail commands can be combined to point to a specific line from a given file. These commands when used within loop can help in reading entire file as shown in the following script.

```
#!/bin/bash
#Process a file line by line by combining head and tail commands

FILENAME=$1
Lines=`wc -l < $FILENAME`
count=0
while [ $count -lt $Lines ]
do
let count++
LINE=`head -n $count $FILENAME | tail -1`
echo "$count $LINE"
done
echo -e "\nTotal $cnt lines read from the file $FILENAME"
```

5.Using arrays in Shell Scripting

An array is a variable containing multiple values that may be of same type or of different type. In Linux, there is no maximum limit to the size of an array, nor any requirement that member

variables be indexed or assigned contiguously. An array index starts with zero. Let us explore how to use arrays within shell scripts.

Declaring an Array and Assigning values

In bash, array is created automatically when a variable is used in the format as shown below:

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

In the above format, the name can be any variable name, index values must be greater than or equal to zero, values that are stored in an array usually takes string format. To access an element of an array, the syntax uses curly brackets like ${name[index]}.

```
#! /bin/bash
Linux[0]='CentOS'
Linux[1]='Red hat'
Linux[2]='Ubuntu'
Linux[3]='Suse'
Linux[4]='Trisquel'
#Display the element at position 1
echo ${Linux[1]}
```

Instead of initializing each element of an array separately, we can declare and initialize an array by specifying the list of elements (separated by white space) within curly braces. The syntax of the same is as follows:

```
declare -a array_name=(element1 element2 element3)
```

While declaring the array and initializing with strings, if the string (elements) has white sace character, we must enclose it with quotes.

To print the whole array, following syntax is used:

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

To display the length of an array, the following syntax is used:

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

To display element at index position 2, the following syntax is used:

```
echo ${array_name[2]}
```

To display the length of an element at index position 2, the following syntax is used:

```
echo ${#array_name[2]}
```

The following shell script illustrates the use of array, it initializes the array with 4 elements, further we display the entire array, length of the array, display the element at position 3 and to display the length of element at position 3 of that array.

```
#!/bin/bash
#Initilize an array with 4 elements
declare -a Linux=('Centos' 'Red Hat' 'Suse' 'Trisquel')
echo "Length of the array: "
echo ${#Linux[@]}
echo "Elements of the array: "
echo ${Linux[@]}
echo "Length of the element at index 3 of the array: "
echo ${#Linux[3]}
echo "Element at index 3 of the array: "
echo ${Linux[3]}
```

Keywords:
array, debugging, file descriptors, stdin, stdout, stderr
commands: tput, echo, exec

Summary

- Producing effective output using options of echo and tput command
- Debugging of shell scripts using set command
- Reading files line by line using different methods like redirection of file descriptors, combination of head and tail commands and inputting file to while loop
- Using arrays in shell scripting