Module 13

Bash Shell Scripting - 1

"We will encourage you to develop the three great virtues of a programmer: laziness, impatience and hubris."

- Larry Wall

Learning Objectives

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

- Understand the basics of shell and types of shell
- Able to create interactive shell scripts
- Able to use functions and variables within shell scripts
- Understand the use of command substitution
- Able to use script parameters.

1. Motivation

Suppose we want to backup the work of few selected users at regular time intervals, we may need a sophisticated backup program that takes care of the requirement. Alternatively, we can write small scripts that can perform similar task. In Linux and UNIX, we can write scripts to automate tasks depending on the need. Such kind of programming is known as "shell programming" or "shell scripting". Let us consider another example from academia. Every year a college enrolls 120 students, it is the job of system administrator to create 120 users in the server, create their profiles and assign separate passwords for each user. This mammoth task may require one full day, however, with the help of shell scripting it can be achieved in a matter of few minutes. The only requirement being, system administrator must be good in writing shell scripts. The following are few benefits of shell scripting:

a) Automate various tasks
b) Provide a controlled user interface
c) Reduce rate of errors
d) Create new/custom commands
e) Share procedures among different users
f) Combine repetitive tasks/sequences by combining various commands.

In this module, we will learn how to write simple shell scripts. In subsequent modules we will deal with more features of shell scripting that can be used to automate various tasks.

2. Introduction to Shell

As we have discussed in earlier modules, a shell is a command line interpreter which provides the user interface for terminal. Usually, shell interacts with the kernel on behalf of users. Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file. For instance, when we issue a command to create a file or directory, the shell will forward the request to Linux kernel to carry out the desired operation. As the kernel understands system calls, it will be difficult for a user to directly interact with kernel that is why, shell acts as an interpreter to interpret user's commands.

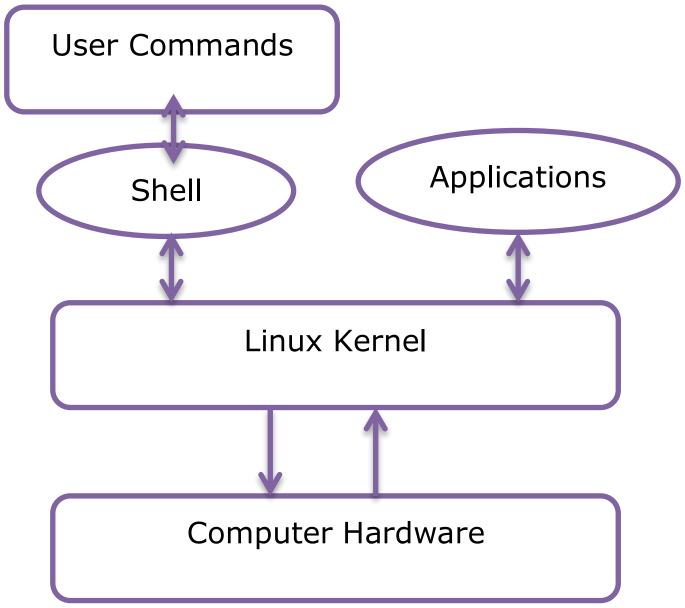

Figure: Role of Shell in executing User's commands

Linux provides wide variety of shells. Depending on the type of work, a user has freedom to use a particular shell. Some widely used shells are bash, csh, ksh, sh, and tcsh. Each shell has its own syntax and has different features. For instance, those who are comfortable with 'C' programming syntax may prefer using C shell (csh). However, most of the users use **bash** shell. We will focus on learning bash shell as it is widely used, apart from it, lot of books, good documentation and sample scripts are available online. A brief summary of various shells available with Linux is listed in following table:

Shell Name	**Developed by**	**Developed At**	**Year**
sh (Bourne shell)	Steve Bourne	AT & T	1977
bash (Bourne Again shell)	Brian Fox and Chet Ramey	Free Software Foundation (GNU Project)	1987
csh (C shell)	Bill Joy	University of California, Berkeley	1978
ksh (Korn shell)	David Korn	AT & T	1982
tcsh (TENEX C shell)	Ken Greer	Carnegie Mellon University	Early 1980's

Linux provides various shells. List of shells available in the system can be known by following command:

```
$cat /etc/shells
```

The default shell of command line interface can be known by following command:

```
$echo $SHELL
```

Usually shells are interactive, the shell accepts commands from user (via keyboard) and executes them. But in case if the user has to execute multiple commands in a sequence, then he/she can store this sequence of command in some text file and let the shell execute this text file. Shell script is series of commands along with the programming language constructs like if statements, loop constructs etc. written in plain text file. Shell scripting is somewhat similar to batch programming of MS-DOS.

3. Creating first shell script

We will learn how to create bash scripts in this module. As discussed earlier, each shell has its own syntax. As such, there is no need to learn scripting for different shells. In this course, we will focus on bash shell scripting. Let us create a simple script that displays two line messages on the screen. We can type the script in any editor like gedit, kwrite, vim, emacs etc., however, it is advisable to use either vim or emacs when the scripts become large. Create a file "sample.sh" and type the following two lines:

```
echo "Today is: "
date +%D
```

The output of above script is:

```
Today is:
05/22/15
```

To execute the above script, we can use either of the following alternatives:

a) Using bash command :
   `$bash sample.sh`
b) Make the sample.sh file executable using chmod command (chmod +x sample.sh) and then execute from the current directory indicated by dot (.) :
   `$./sample.sh`

As we have discussed earlier, shell is an interpreter. Usually a shell program is a series of commands that are interpreted by the shell. That is why, shell programs are never compiled. As the html programs run in browser without compiling, similarly shell scripts are interpreted by the shell without any need of compilation. It must also be noted that the file extension ".sh" is not required. We can create a file named "sample" and make it executable. However, for the sake of distinguishing the file as a shell program from other text files, it is a convention to use the extension of ".sh" to shell scripts.

4. Creating Interactive Scripts

Let us create another script that has more statements in the program. The following script is interactive that asks to enter the name and displays the same on the screen.

```
#!/bin/bash
#Interactive script that reads variable
echo "Enter your name"
read name
echo "You entered: $name"
```

The output of above script is:

```
Enter your name:
Richard Davis
```

```
You entered: Richard Davis
```

In the above script, the hash-tag or number-sign (#) is used to start comments. Comments can be placed anywhere in the script. It is recommended to put the first line that indicates the path of the shell for which the script is written. The above script asks to enter the name; the text entered by user will be stored in a variable "name" using the read command. The last statement of the script displays the name entered by the user. To display the content of a variable, we must use '$' sign prefixed with the variable name. Three important points that must be noted are:

a) A variable name consists of alphabets, digits or underscore. The first character must be either an alphabet or an underscore.
b) In shell scripting there is no need to mention type of variable like int, float, etc. In fact, all variables are treated as strings.
c) To display the content of any variable, the variable name is preceded by $ sign.
d) Variable names are case-sensitive.

Let us see how variables can be assigned values. We will modify the above script to make it non-interactive. The variables will be assigned values within the script. In the following script, we have used two variables, name and age. It is clear that both the variables are assigned some value. Although age is assigned the value 45, shell script will treat it like string. Curious learners may investigate by adding two lines (`age=$age+1 and echo $age`) to the below mentioned script:

```
#!/bin/bash
#Use of Variables
name = "Richard Davis"
age=45
echo "Name: $name"
echo "Age: $age"
```

The output of above script is:

```
Name: Richard Davis
Age: 45
```

The following script is another example with a better formatted output. It also demonstrates combining multiple commands in a single file. In the following script, we have used $USER variable to display the user who executes the script, it displays the current date using echo statement. The '-e' option with echo will display date in the same line, similarly the count of online users is displayed in the same line of the message. In the later sections, we will explore how to store the count of users in a variable.

```
#!/bin/bash
#Better formatting
echo "Hello $USER"
echo -e "Today is: \c"
date +%D
echo -e "Number of online user: \c"
who | wc -l
```

Output of the above script:

```
Hello Richard
Today is: 05/22/15
Number of online user: 20
```

Note: If the shell is unable to understand a word as a variable, it will interpret it as a linux command which may lead to an error.

Exporting Variables

By default, the variables being used within a shell script has its scope upto the execution of that script. These variables cannot be accessed outside the script. Once the script terminates, content of these variables are lost. For instance, after the termination of the script that displays the name of a user, if we try to display the value of variable name, it may not show any value. All the variables declared in the shell script have their life-time till the script executes. Moreover, the variable declared in one shell cannot be used in another shell. The export command allows us to make the variable global so that it can be accessed in child processes of the existing shell. Let us experiment with the variables across different shells. By typing the following sequence of commands on shell prompt, we will be able appreciate the use of export command.

Command	**Meaning**
$a=15	Create new local variable 'a' with 15 as value in first shell
$echo $a	Print the contents of variable ‘a’ Output is: 15
	Let us now load another shell in memory (Which ignores all old shell's variables
$/bin/bash	#Start another shell
$echo $a	Print the contents of variable ‘a’ (empty line printed...) Output is: <blank>
$a=20	Create new local variable 'a' with 20 as value in second shell
$echo $a	Print the contents of variable ‘a’

	Output is: 20
$exit	Exit from second shell and return to first shell
$echo $a	Print the content Output is:15

In the above case, variable 'a' was a local variable. Once we start another shell (/bin/bash), the contents of variable 'a' disappeared. To make the contents of variable 'a' visible in the new shell, we need to export the variable. The syntax of exporting a variable can be:

```
export variable_name=value
```
OR
```
variable_name=value
export variable_name
```

So, we export the variable 'a' as follows:
```
$export a
```

Exported variables are copied into child processes. The values of exported variables can be modified by child process. Let us try to export the variable 'a' in above sequence. Let us experiment with the above sequence of execution after exporting the variables.

Command	**Meaning**
$a=15	Create new local variable 'a' with 15 as value in first shell
$echo $a	Print the contents of variable ‘a’ Output is: 15
$export a	Export the variable ‘a’
	Let us now load another shell in memory (Which ignores all old shell's variables
$/bin/bash	#Start another shell
$echo $a	Print the contents of variable ‘a’ Output is:15
$a=20	Create new local variable 'a' with 20 as value in second shell
$echo $a	Print the contents of variable ‘a’ Output is:20
$exit	Exit from second shell return to first shell
$echo $a	Print the content Output is: 15

From the above execution, it might be clear that exporting a variable provides the copy of values to the child process.

5. Functions

A function is a block of code that implements a set of operations. Usually, functions are helpful in executing the procedure multiple times. Functions are also called procedures or sub-routines. Using functions in shell scripts require two steps:

a) Declaring a function
b) Calling a function

The syntax of declaring a function is:

```
function_name() {
   commands….
}
```

Functions are invoked by writing the name of function within shell script

Let us create a function that displays fancy line made up of asterisks (*). First we have to declare the function with some name, let us assume the name of our user defined function is "Myline". Within the shell script, we invoke the function in between the execution of commands. A script demonstrating the use of functions is given below:

```
#!/bin/bash
# Define function here
Myline () {
   echo "************************************"
}
# Invoke your function
Myline
echo -e "Today is :\c" ; date
Myline
echo -e "No. of online users: \c" ; who | wc -l
Myline
```

Output of the above script is:

```
************************************
Today is :Fri May 22 19:33:28 EDT 2015
************************************
No. of online users: 20
************************************
```

As seen in the above output, the function displays fancy lines wherever the function "Myline" was invoked. It must be noted that the function is called multiple times to perform the same procedure.

Passing Parameters to Functions

We can define a function which would accept parameters; these parameters are represented by $1, $2 and so on. The parameters passed to any function must follow the function call. Following is an example where we pass two parameters, "Richard" and "Davis" and then we capture and print them through the function.

```
#!/bin/bash
# Define function here
Hello () {
   echo "Hello  $1 $2"
}
# Invoke function
Hello Richard Davis
```

Output of the above script is:

```
Hello Richard Davis
```

Returning values from functions

We can return values from functions. The "return" keyword is used within function body with some value that is to be returned. In the following script, we have returned the value 12 from function Hello. This returned value is captured by $? that follows the function call.

```
#!/bin/bash
# Define function here
Hello () {
   echo "Hello $1 $2"
   return 12
}
# Invoke function
Hello Richard Davis

# Capture value returnd by last command
ret=$?
echo "Return value is $ret"
```

Output of the above script is:

```
Hello Richard Davis
Return value is 12
```

Function calls within functions

In shell scripting, functions can call other functions or can call themselves from within the function body. In the following script, the “first” function is invoking the “second” function.

```
#!/bin/bash

# Calling one function from another
first () {
   echo "This is the first function..."
   second
}

second () {
   echo "This is now the second function..."
}

# Calling the first functions
first
```

Output of the above script:

```
This is the first function...
This is now the second function...
```

Functions are an important feature of any programming language, they are generally used for validity checks like whether a file exists or not or to perform set of operations pertaining to any task.

Built-In Shell Commands

Shell scripts are used to execute sequence of commands and other types of statements like loop constructs, if statements etc. The commands that are executed by shell script can be divided into three categories:

a) Built-In bash commands: Bash shell provides a set of commands that are executed by the shell itself. Various commands like cd, pwd, echo, read, let etc. are bash commands. A complete list of bash commands can be viewed by the command `$man bash`
b) Compiled Applications: Various applications like rm, ls, df, vi etc. are present in the form of binary executable files. The shell script can invoke such type of compiled applications.
c) Other scripts: A shell script can invoke other shell scripts provided there are adequate access privileges. Such kind of shell scripts behave as a custom command.

Command Substitution

We may come across situations where there is a need of substituting the result of one command as a portion of another command. Suppose we want to display the statement in the following form: “Today is …day”, here we need to execute the date command (date +%A) first and the output of this command will become portion of echo command.

Command substitution can be done in two ways:
a) By enclosing the inner command with backticks/grave accent (`)
b) By enclosing the inner command in $( )

Following are examples of command substitution:
$echo “Today is `date +%A` ”
In the above example, the date command is enclosed with backticks. Another way of using command substitution to get the same output is:
$echo “Today is $(date +%A)”

Let us explore a shell script that uses command substitution. In the following script, we are storing the number of online users in a variable “cnt”. The value of variable cnt is displayed on screen.

```
#!/bin/bash
#Command Substitution
cnt=`who | wc -l`
echo “Number of online users: $cnt”
```

We must take care that no blank spaces must be left before or after the ‘=’ sign.

Script Parameters

Script parameters are often known as positional parameters or command line arguments. Using this facility, we can pass parameter values to a script. For instance, a script may require accepting the file names on command prompt as shown below:
$bash myscript.sh <file1> <file2>
In the above case, <file1> and <file2> are passed as command line arguments. These arguments can be accessed from within the script. Such kind of arguments are represented with a $ sign and a number. Following are some of the positional parameters used in shell programming:

Parameter	Meaning
$0	Name of the script
$1	First argument passed to the script
$2, $3, … $9	Second argument, third argument and so on… passed to the script ($3, $4 and so on ... upto $9 for second, third etc)
"$*"	All parameters (All parameters treated as single string)
"$@"	All parameters (All parameters treated as separate strings)
$*	All parameters
$@	All parameters
$#	Number of parameters

Let us explore script parameters by writing a shell script. The following shell script displays the positional parameters and count of number of parameters.

```
#!/bin/bash
#Script to read from command line
# Script file: parameters.sh
echo "The name of this program is: $0"
echo "The first argument passed from the command line is: $1"
echo "The second argument passed from the command line is: $2"
echo "The third argument passed from the command line is: $3"
echo "All of the arguments passed from the command line are : $*"
echo "All arguments in another way : $@"
echo "Total number of arguments : $#"
```

Execution and Output of the above script

To execute the script, type the bash command as given below:

```
$ bash parameters.sh one two three four
```

The output will be as follows:

```
The name of this program is: parameters.sh
The first argument passed from the command line is: one
The second argument passed from the command line is: two
The third argument passed from the command line is: three
All of the arguments passed from the command line are : one two
three four
All arguments in another way : one two three four
Total number of arguments : 4
```

At the most 9 values can be displayed starting from $1 upto $9. A curious learner may experiment by passing 12-15 parameters and see what happens by displaying upto 15 variables using `echo $15` command. Shell scripting provides functionality to use all the parameters passed. We can use the shift command to shift parameters by some specific number and display the later ones. By shifting the parameters we can display the values of script parameters from $10 and onwards. To display the content of tenth parameter, we have to start from $1 after performing the shift operation. The syntax of shift command is:

```
$shift [number of parameters to shift]
```

Features of Bourne Again Shell (BASH)

Bash being the GNU version of the standard bourne shell found in UNIX, it provides various facilities to help the programmers. It also incorporates popular features of csh, tcsh and ksh. Bash is the standard Linux shell, that gets loaded by default when most of the user accounts are created.

Programmers and system administrators can harness the following techniques within shell scripts:

a) Support for Wildcard characters (meta characters)
b) Support for I/O redirection, Piping
c) Support for Standard Error Redirection
d) Command Grouping
e) Command Separation
f) Running processes in background
g) Conditional execution using && and || operators
h) Support for command line arguments (positional parameters)
i) Access to Environment variables
j) Access to previous commands
k) Command Substitution
l) Command name abbreviation (aliasing)

Keywords

bash, csh, ksh, sh, tcsh, command substitution, positional parameters

Summary

Let us summarize the key concepts covered in this module

- Shell is a command interpreter.
- Shell scripting includes series of commands and programming constructs to perform particular tasks.

- Variables can be used in shell scripts and can be made global using export command. However, variables are treated as strings.
- Shell scripting supports use of functions, command line arguments (script parameters) and command substitution.