Module 14

Bash Shell Scripting - 2

"Talk is cheap. Show me the code."

-Linux Torvalds

Learning Objectives:

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

- Performing calculations within shell scripts
- Use test conditions in shell scripts
- Use Conditional Statements like if-else, case-esac within shell scripts
- Use Loop Constructs like while, until and for within shell scripts

1. Performing Calculations in Shell Scripting

In the previous module, we have seen how to define a variable, assign values to the variable and retrieve the stored values. The values assigned to variables were treated as strings. When we want to perform numerical operations on these values, it cannot be done in a straight forward way like c=a+b because these variables contain strings. Let us see what happens if we perform simple addition operation on string.

```
a=10
b=15
c=$a+$b
echo "Total is: $c"

Output:
Total is:10+15
```

In the following sections, we will explore how to perform arithmetic operations on shell variables. There are four different ways used to perform calculations in shell scripting. These commands convert strings into numbers and perform the desired arithmetic operation. A programmer is free to use any of the following commands to perform arithmetic operations:

- expr (compiled application for integer arithmetic operations)
- Using $((...))
- let (built-in shell command)
- bc (compiled application for floating point operations)

We can use the above techniques on command prompt as follows:

```
$expr 2 + 5
$echo $((2+5))
$let x=( 2 + 5 )
$echo $x
$echo "2 + 5" | bc
```

Commands like expr and bc are widely used by shell programmers. Techniques like $((...)) and let command are not used frequently, the main reason is probably they are not supported by other shells. Let us explore expr, let and bc commands in detail with sample shell scripts.

expr : Evaluation Expressions

expr command evaluates arithmetic expressions and helps to perform integer calculations. The syntax of expr is:

`expr arg1 operator arg2`

Few Examples:

```
$expr 2 + 3

$a=5
$b=10
$expr $a + $b
$expr $a \* $b
```

It must be noted that the operator used for multiplication, asterisk (*) is treated as a wildcard character by shell, that is why we have to use escape sequence (\) while performing multiplications.

The following shell script performs basic arithmetic calculations using expr command over two variables. Here we have used command substitution so that the expression gets evaluated first and the results are stored in the variable "ans".

```
#!/bin/bash
#Script to perform basic calculations
#Works only with integer numbers
echo "Enter First Number  : "
read a
echo "Enter Second Number  : "
read b
ans=`expr $a + $b`
echo "Addition : " $ans
ans=`expr $a - $b`
echo "Subtraction : " $ans
ans=`expr $a / $b`
echo "Division : " $ans
ans=`expr $a \* $b`
echo "Multiplication : " $ans
```

Note: expr command can also be used to find the substring of a given string or length of a given string. We request you to read the manual of expr for more details. (use man expr)

let : Assigning and Evaluating expressions

The let command performs integer calculations as we did with expr command. However, when we use let command, there is no need to use the '$' sign with variables and use command substitution as with expr. let command evaluates the expression and stores the values in the variable. The syntax of let command is:

`let variable=value/expression [variable2=value/expression...]`

```
#!/bin/bash
#Script to perform basic calculations
#Works only with integer numbers
echo "Enter First Number: "
read a
echo "Enter Second Number: "
read b
let ans=a+b
echo "Addition : " $ans
let ans=a-b
echo "Subtraction : " $ans
```

```
let ans=a/b
echo "Division : " $ans
let ans=a\*b
echo "Multiplication : " $ans
```

bc : Base Conversion

The bc calculator can be used within shell scripts to perform floating point operations. However, while using in shell scripts, we have to pass the expression to bc using pipe (|) symbol. As we know that bc can be invoked from command prompt and allows us to perform calculations. We can also specify the variable scale to get precise answers while working with floating point numbers. The following script demonstrates the use of bc:

```
#!/bin/bash
#Script to perform basic calculations
#bc supports floating point operations
echo "Enter First Number: "
read a
echo "Enter Second Number: "
read b
ans=`echo "scale = 2; $a + $b" | bc`
echo "Addition : " $ans
ans=`echo "scale = 2; $a - $b" | bc`
echo "Subtraction : " $ans
ans=`echo "scale = 2; $a / $b" |bc`
echo "Division : " $ans
ans=`echo "scale = 2; $a * $b" | bc`
echo "Multiplication : " $ans
```

2. Decision Making

Decision making is a key part of any programming language. Every language provides if-else statement for decision making within the programs. However, for each language the syntax and keywords may vary. Bash shell programming provides two different constructs for decision making. The programmer has a choice to use either of the followings:

- if-then-else-fi statement
- case-esac statement

The conditional statements in shell programming have far more capabilities when compared to those of programming languages like C, C++ etc. The main reason being, it allows us to use wildcard characters within the statement. Let us explore the syntax of each one of the above.

If Statement

The if-then-else statements are considered as control statements. Depending on the condition, selected statements are executed and hence it decides the flow of script. The if-then-else construct takes three different forms. Let us first see the syntax and later explore the scripts using these forms.

```
# First form
if condition
then
    commands
fi

# Second form
if condition
then
    commands
else
    commands
fi


# Third form
if condition
then
    commands
elif condition
then
    commands
fi
```

In the first form, if the condition is true, then commands are executed. If the condition is false, nothing is done and the control comes out of the if statement.

In the second form, if the condition is true, then the first set of commands is executed and if the condition is false, then second set of commands is executed.

In the third form, if the condition is true, then the first set of commands is executed, if the condition is false, and if the second condition is true, then the second set of commands is executed. In the third form, the if construct checks for two conditions.

It must be noted that each if block ends with a fi. Let us see a simple script that demonstrates the working of if-else statement

```
if cat /etc/passwd
then
  clear
  echo "File /etc/passwd displayed"
else
  echo "Some error occured..."
fi
```

The above script will try to display the contents of /etc/passwd file. If the cat command executes successfully (that is, exit status of cat is 0) then the “File /etc/passwd displayed” message will be seen on terminal. In case if the cat command does not execute properly due to insufficient permissions or some other reason (exit status is 1)then the “Some error occurred...” message will be seen on terminal.

test and [] command: companion of if

The test command is used often with if statements to perform true/false decisions. The test command can be used to compare numbers, check files and directories or check strings. The test command checks the expression and exists with status either 0 (expression is true) or 1 (expression is false). The test command can be used with the keyword test or using the notation []. It takes two forms as shown below:

```
#First form
test expression

# Second form
[ expression ]
```

The test command can be used for three main purposes in conjunction with if statement. Test can be used to:

- Compare two numbers

- Compare two strings or a single string for null value
- Check the attributes of a file/directory

test command can also be used with loop constructs within shell scripts, however, we will explore its usage with if statements. As such, test command does not display any output but simply returns a status of $?. We must take care that there has to be blank space between expression and square brackets ('[' and ']'). Let us explore how test can be used with the above conditions.

<u>Numeric Comparison</u>

The test command can be used to compare two numbers. The comparison operators always begin with a '–' (hyphen) as shown in below table:

Operator	**Meaning**
`-eq`	Equal to
`-ne`	Not Equal to
`-gt`	Greater than
`-ge`	Greater than or equal to
`-lt`	Less than
`-le`	Less than or equal to

```
#!/bin/bash
# Compare two numbers
echo "Enter first number : "
read a
echo "Enter second number : "
read b
if [ $a -gt $b ]
then
echo $a is greater
elif [ $b -gt $a ]
then
echo $b is greater
else
echo $a and $b are equal
fi
```

The above script displays the larger number from the two numbers input by the user. We can use the following form:

`if test $a -gt $b` instead of `if [ $a -gt $b ]`

String Comparisons

test command can be used to compare two strings for equality, it can also be used to check whether a string is null or not. The following table summarizes the operators used for string comparison.

Test	**Returns true if...**
`s1 = s2`	Strings s1 and s2 are equal
`s1 == s2`	Strings s1 and s2 are equal
`s1 != s2`	String s1 is not equal to string s2
`str`	String str is assigned and not null
`-n str`	String str is not null
`-z str`	String str is null

The following program compares two strings to check whether they are same or not.

```
#!/bin/bash
#Script to compare two strings
s1=Government
s2=government
if [ $s1 = $s2 ]
echo "Both the strings are same"
else
echo "Both the strings are different"
fi
```

Output:

```
Both the strings are different
```

File Checks

The test command can be used to test various file attributes. It also tests whether a string is a file or a directory. The following table summarizes few parameters to check file attributes

Test	**Returns true if…**
`-f filename`	File exists
`-d filename`	Is a directory
`-r filename`	File has read permissions
`-w filename`	File has write permissions
`-x filename`	File has execute permissions
`-s filename`	File has size greater than zero

The following script checks whether /etc is a directory or not. It also displays whether we have write permission to the file /etc/passwd.

```
#!/bin/bash
#File checks
if [ -d /etc ]
then
echo "/etc is a directory"
fi
if [ -w /etc/passwd ]
then
echo "/etc/passwd has write permissions "
else
echo "/etc/passwd does not have write permissions"
fi
```

Case – Esac

The following is a code which helps use to check whether the user has entered yes or no. However, there are few issues in the following program. The user may enter y for yes or YES in capitals. Under such circumstances, the following program may not give expected output.

```
#!/bin/bash
# in the program
echo "Enter “yes” or “no” : "
```

```
read choice
if [ $choice = "yes" ]
then
echo "Your choice is yes..."
else
echo “Your choice is no...”
fi
```

The above program has few logical bugs. What if the user enters Yes or YES or yes or Y or y. If we try to incorporate each test condition, the if-else statements will become too complicated. Such kind of issues can be well addressed by using case-esac construct rather than if-else statements. The following code looks much cleaner:

```
#!/bin/bash
#sample code of case
read choice
case $choice in
y|Y|yes|YES|Yes) echo "You entered Yes..."
;;
n|N|no|NO|No) echo "You entered No..."
;;
*) echo "You entered wrong choice"
;;
esac
```

The case-esac statements are generally used when there are multiple conditions to be checked. It also supports wildcard characters to match appropriate patterns. The syntax is:

```
case $variable-name in
    patterns ) commands ;;
    patterns ) commands ;;
.....
esac
```

The following script demonstrates pattern matching with case-esac construct. Using regular expressions within case statements, we can check whether the word entered by a user starts with a vowel or a consonant.

```
#!/bin/bash
# Word begins with a vowel or not
echo "Enter any word : "
read word
case $word in
[AEIOU]* ) echo "Your word begins with a vowel"
```

```
             ;;
[aeiou]*) echo "Your word begins with a vowel"
             ;;
*) echo "Your word begins with a consonant"
             ;;
esac
```

Let us see one more script where we check whether the single character entered by the user is a letter or a digit. Patterns like [:lower:] and [:upper] will classify the typed character as a lower case or upper case

```
#!/bin/bash
#Checks if Character is a letter or a digit
echo -n "Type a single digit or a letter > "
read character
case $character in
# Check for letters
    [[:lower:]]  ) echo "You typed the small letter $character"
                   ;;
    [[:upper:]] ) echo "You typed the capital letter $character"
                   ;;
# Check for digits
    [0-9] )  echo "You typed the digit $character"
             ;;
# Check for anything else
    * ) echo "You did not type a single letter or a digit"

esac
```

3. Loop Constructs

Loops are important part of any programming language. They are used to perform iterative operations. Usually, a loop has a condition to identify the terminating criteria. A loop may iterate for 'n' number of times depending on some condition. In bash shell scripting, we can use three types of loops. The following loop constructs are explored in this section:

- While loop
- Until loop
- For loop

<u>While Loop</u>

The while command causes a block of code to be executed over and over, as long as the exit status of a specified expression is true.

Syntax of while loop

```
while condition is true
do
   Statement(s) to be executed if condition is true
done
```

Here is a simple example of a program that displays from zero to ten:

```
#!/bin/bash
#While loop demo
#Initialize a variable
number=0
while [ "$number" -le 10 ]
do
    echo "Number = $number"
#Increment the variable in each iteration
    number=$((number + 1))
done
```

In the above program, we have initialized the number with zero. The loop tests for condition whether number is less that or equal to (le) 10 in each iteration. We have also used the statement number=$((number + 1)) to increment the number variable in each iteration.

Until Loop

The until loop is reverse of while loop. The until command causes a block of code to be executed over and over, as long as the exit status of a specified expression is false.
Syntax of until loop is:

```
until condition is false
do
   Statement(s) to be executed if condition is false
done
```

Following is a sample script that uses until loop.

```
#!/bin/sh
# Until loop demo
number=0

until [ ! $number -le 10 ]
do
   echo $number
   a=`expr $number + 1`
done
```

In the above script, notice that we have reversed the condition. The condition is reversed by using '!' sign. We can use either while or until loop. Both the loops behave in similar manner except the interpretation of condition.

For Loop

The for loop of shell programming is different from conventional for loops used in C/C++ or Java programming languages. In shell scripting, for loop iterates for the list of values specified. We provide a list of values within for loop construct, the looping variable picks up each value and iterates for 'n' number of values provided in the construct. Syntax of for loop is:

```
for variable_name in list
do
    commands
done
```

The following script demonstrates the use of for loop. In this script the loop will iterate 3 times, in each iteration it will select word1, word2 and word3 respectively.

```
#!/bin/bash
#For loop demo
for i in word1 word2 word3
do
    echo $i
done
```

For loop is extremely useful when we combine it with command substitution. Let us explore a shell script that combines command substitution with for loop. In the following script, we display the line count of each file that is present in the current directory.

```
#!/bin/bash
#Command substitution with for loop
for files in `ls`□do□if [ -f $files ]□then□wc -l $files□fi□done
```

In the above script, the ls command is executed first to generate a list of files and directories. This list is used by the for loop to display line count for each file. We have also used file condition check to avoid the line count of directories, which may generate errors.
Let us modify the above script where we accept file names as command line

arguments (script parameters). The following script has to be executed by specifying command line arguments at $ prompt.

```
#!/bin/bash
# Demo of for loop with command line arguments
echo "Total number arguments are : $#"
echo "Arguments are : $*"
for args in "$@"
do
if [ -f $args ]
then
 wc -l $args
fi
done
```

In the above script, we have used $* and $@ to list all the parameters passed at command line. As we have discussed in previous module, $* will treat all the arguments as a single string whereas “$@” will treat each argument as separate string. Here, we have to generate a list of file names in for loop which is to be processed separately, it is advisable to use “$@” instead of $*.

Let us discuss another script that combines loop and case statement together. The following script will iterate until user enters '0' to exit the script. It uses until loop to repetitively ask for a choice and uses case to perform the operation according to the choice entered by the user.

```
#!/bin/bash
choice=
until [ "$choice" = "0" ]
do
    echo "
    PROGRAM MENU
    1 - Display free disk space
    2 - Display free memory
    3 – Display online users
    0 - exit program
"
    echo -n "Enter choice: "
    read choice
    echo ""
    case $choice in
        1 ) df ;;
        2 ) free ;
        3 ) who;;
        0 ) exit ;;
        * ) echo "Please enter 1, 2, 3 or 0"
    esac
done
```

We can enhance the functionality of shell scripts when we combine features of bash like command substitution, meta-characters, script parameters with loops and if statements. Looping constructs and if statements can be used to perform file processing. In the next module we will explore few features of file processing along with loops.

Summary:

- Numerical calculations can be performed using expr, let or bc command.
- The test command provides conditions to compare numbers, check strings and check file attributes.
- The while loop and until loop iterate depending on some condition. while and until loop have similar syntax, however they are reverse of each other while checking for condition.
- The for loop iterates for a given list of words. The number of iterations depends on the count of words that are listed.
- We can enhance the functionality of shell scripts when we combine features of bash like command substitution, script parameters with loops and if statements.

Keywords:
conditional statements, loop construct, test expression, commands:expr, let, bc