Module 7

Introduction to shell script

Introduction to Linux Shell and Shell Scripting

If you are using any major operating system you are indirectly interacting to **shell**. If you are running Ubuntu, Linux Mint or any other Linux distribution, you are interacting to shell every time you use terminal. Today we will discuss about linux shells and shell scripting so before understanding shell scripting we have to get familiar with following terminologies:

- Kernel
- Shell
- Terminal

What is Kernel

The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system. It manages following resources of the Linux system –

- File management
- Process management
- I/O management
- Memory management
- Device management etc.

What is Shell

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a command language interpreter that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.

Command Line Shell

Shell can be accessed by user using a command line interface. A special program called Terminal in linux/macOS or Command Prompt in Windows OS is provided to type in the human readable commands such as "cat", "ls" etc. and then it is being execute.

Shell Scripting

Usually shells are interactive that mean, they accept command as input from users and execute them. However some time we want to execute a bunch of commands routinely, so we have type in all commands each time in terminal.

As shell can also take commands as input from file we can write these commands in a file and can execute them in shell to avoid this repetitive work. These files are called **Shell Scripts** or **Shell Programs**. Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with **.sh** file extension eg. **myscript.sh**

A shell script have syntax just like any other programming language. If you have any prior experience with any programming language like Python, C/C++ etc. it would be very easy to get started with it.

A shell script comprises following elements –

- Shell Keywords – if, else, break etc.
- Shell commands – cd, ls, echo, pwd, touch etc.
  Functions Control flow – if..then..else, case and shell loops etc.

Why do we need shell scripts

There are many reasons to write shell scripts:

- To avoid repetitive work and automation .
- System admins use shell scripting for routine backups.
- System monitoring .
- Adding new functionality to the shell etc.

Advantages of shell scripts

- The command and syntax are exactly the same as those directly entered in command line, so programmer do not need to switch to entirely different syntax.
- Writing shell scripts are much quicker .
- Quick start .
- Interactive debugging etc.

Disadvantages of shell scripts

- Given to costly errors, a single mistake can change the command which might be harmful.
- Not well suited for large and complex task .
- Provide minimal data structure unlike other scripting languages. etc

Simple shell scripts, Interactive shell script :

- A simple shell script typically refers to a script that performs a straightforward task or a series of simple tasks. It may not involve complex logic or extensive error handling. Simple shell scripts are often used for tasks like file manipulation, text processing, or running a sequence of commands.

```
echo “This is CCLMS”
Cal
Date
```

Interactive shell script :

- an interactive shell script is one that interacts with the user, typically by prompting for input, displaying information, and responding to user actions. Interactive shell scripts often incorporate features such as user input validation, menu systems, and interactive command-line interfaces (CLIs). These scripts allow users to interact with the script in real-time, providing input and receiving immediate feedback.

```
echo "What is your name?"

# Read the user's input
read name

# Greet the user
echo "Hello, $name! Nice to meet you."
```

There are **5** basic operators in bash/shell scripting:

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- Bitwise Operators
- File Test Operators

1. **Arithmetic Operators**: These operators are used to perform normal arithmetics/mathematical operations. There are 7 arithmetic operators:
   - **Addition (+)**: Binary operation used to add two operands.
   - **Subtraction (-)**: Binary operation used to subtract two operands.
   - **Multiplication (*)**: Binary operation used to multiply two operands.
   - **Division (/)**: Binary operation used to divide two operands.
   - **Modulus (%)**: Binary operation used to find remainder of two operands.
   - **Increment Operator (++)**: Unary operator used to increase the value of operand by one.
   - **Decrement Operator (- -)**: Unary operator used to decrease the value of a operand by one

2. **Relational Operators**: Relational operators are those operators which define the relation between two operands. They give either true or false depending upon the relation. They are of 6 types:
   - **'==' Operator**: Double equal to operator compares the two operands. Its returns true is they are equal otherwise returns false.
   - **'!=' Operator**: Not Equal to operator return true if the two operands are not equal otherwise it returns false.
   - **'<' Operator**: Less than operator returns true if first operand is less than second operand otherwise returns false.
   - **'<=' Operator**: Less than or equal to operator returns true if first operand is less than or equal to second operand otherwise returns false
   - **'>' Operator**: Greater than operator return true if the first operand is greater than the second operand otherwise return false.
   - **'>=' Operator**: Greater than or equal to operator returns true if first operand is greater than or equal to second operand otherwise returns false

3. **Logical Operators** : They are also known as boolean operators. These are used to perform logical operations. They are of 3 types:
    - **Logical AND (&&)**: This is a binary operator, which returns true if both the operands are true otherwise returns false.
    - **Logical OR (||)**: This is a binary operator, which returns true is either of the operand is true or both the operands are true and return false if none of then is false.
    - **Not Equal to (!)**: This is a unary operator which returns true if the operand is false and returns false if the operand is true.

4. **Bitwise Operators**: A bitwise operator is an operator used to perform bitwise operations on bit patterns. They are of 6 types:
    - **Bitwise And (&)**: Bitwise & operator performs binary AND operation bit by bit on the operands.
    - **Bitwise OR (|)**: Bitwise | operator performs binary OR operation bit by bit on the operands.
    - **Bitwise XOR (^)**: Bitwise ^ operator performs binary XOR operation bit by bit on the operands.
    - **Bitwise complement (~)**: Bitwise ~ operator performs binary NOT operation bit by bit on the operand.
    - **Left Shift (<<)**: This operator shifts the bits of the left operand to left by number of times specified by right operand.
    - **Right Shift (>>)**: This operator shifts the bits of the left operand to right by number of times specified by right operand.

They manipulate individual bits within binary representations of data. These operators are typically used in low-level programming, such as systems programming, device drivers, cryptography, and optimization algorithms. Understanding bitwise operators can be beneficial for understanding low-level programming concepts and optimizing performance-critical code.

5. **File Test Operator**: These operators are used to test a particular property of a file.
    - **-b operator**: This operator check whether a file is a block **special file or not**. It returns true if the file is a block special file otherwise false.
    - **-c operator**: This operator checks whether a file is a **character special file or not**. It returns true if it is a character special file otherwise false.
    - **-d operator**: This operator checks if the given **directory exists or not**. If it exists then operators returns true otherwise false.
    - **-e operator**: This operator checks whether the given **file exists or not**. If it exits this operator returns true otherwise false.
    - **-r operator**: This operator checks whether the given file has **read access or not**. If it has read access then it returns true otherwise false.
    - **-w operator**: This operator check whether the given file has **write access or not**. If it has write then it returns true otherwise false.
    - **-x operator**: This operator check whether the given file has **execute access or not**. If it has execute access then it returns true otherwise false.

- **-s operator**: This operator checks the size of the given file. If the size of given file is greater than 0 then it returns true otherwise it is false.

Conditional Statements: There are total 5 conditional statements which can be used in bash programming

1. if statement
2. if-else statement
3. if..elif..else..fi statement (Else If ladder)
4. if..then..else..if..then..fi..fi..(Nested if)
5. switch statement

Their description with syntax is as follows:

if statement

This block will process if specified condition is true.

Syntax:

```
if [ expression ] then
   statement
fi
```

if-else statement

If specified condition is not true in if part then else part will be execute.

Syntax

```
if [ expression ]
then
statement1
 else
statement2
fi
```

if..elif..else..fi statement (Else If ladder)

To use multiple conditions in one if-else block, then elif keyword is used in shell. If expression1 is true then it executes statement 1 and 2, and this process continues. If none of the condition is true then it processes else part.

Syntax if [expression1] then statement1 statement2

```
  .
  .
elif [ expression2 ]
then
statement3
statement4
  .
  .
else
statement5
fi
```

if..then..else..if..then..fi..fi..(Nested if)

Nested if-else block can be used when, one condition is satisfies then it again checks another condition. In the syntax, if expression1 is false then it processes else part, and again expression2 will be check.

```
echo "Enter a number:"
read number

# Check if the number is positive, negative, or zero
if [ "$number" -gt 0 ]; then
    echo "The number is positive."

    # Nested if statement to check if the number is even or odd
    if [ "$((number % 2))" -eq 0 ]; then
        echo "The number is even."
    else
        echo "The number is odd."
    fi
elif [ "$number" -lt 0 ]; then
    echo "The number is negative."
else
    echo "The number is zero."
fi
```

switch statement

case statement works as a switch statement if specified value match with the pattern then it will execute a block of that particular pattern

When a match is found all of the associated statements until the double semicolon (;;) is executed.

A case will be terminated when the last command is executed.

If there is no match, the exit status of the case is zero.

Syntax:

```
case expression  in
  Pattern 1) Statement 1;;
  Pattern 2) Statement 2;;
   Pattern n) Statement n;;
        *) default statement ;;
esac
```

Note: `;;` (Double Semicolon):

- `;;` is used to terminate each pattern and its associated code block within the `case` statement.
- It signifies the end of the code block for the matched pattern and indicates to Bash that it should move to the next pattern.

`*)` (Wildcard Pattern):

- `*)` is a wildcard pattern that matches any value.
- It serves as the default case to execute if the expression matches none of the specified patterns.

- `esac` (Reverse of `case`):

  - `esac` is the reverse of the `case` keyword.
  - `esac` is used to terminate the `case` block and maintain the structure of the `case` statement.

Example Programs
Example 1:
Implementing if statement

```
#Initializing two variables
a=10
b=20

#Check whether they are equal
if [ $a == $b ]
then
   echo "a is equal to b"
fi

#Check whether they are not equal
if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

Output
$bash -f main.sh

a is not equal to b

Example 2:
Implementing if.else statement

```
#Initializing two variables
a=20
b=20

if [ $a == $b ] then
   #If they are equal then print this
echo "a is equal to b" else

   #else print this
   echo "a is not equal to b"
fi
```

Output
$bash -f main.sh

a is equal to b

Example 3:
Implementing switch statement

```
CARS="bmw"

#Pass the variable in string
case "$CARS" in
   #case 1
   "mercedes") echo "Headquarters - Affalterbach, Germany" ;;

   #case 2
   "audi") echo "Headquarters - Ingolstadt, Germany" ;;

   #case 3
   "bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;
esac
```

Output
$bash -f main.sh

Headquarters - Chennai, Tamil Nadu, India.