**SHELL SCRIPTING**

INTRODUCTION TO LINUX SHELL AND SHELL SCRIPTING:

If we are using any major operating system, we are indirectly interacting with the shell. While running Ubuntu, Linux Mint, or any other Linux distribution , we are interacting with the shell by using the terminal.

* KERNEL
* SHELL
* TERMINAL

Table of Contents

* What is Kernel?
* What is Shell?
* Command Line Shell?
* Graphical Shell
* What is Terminal
* Shell Scripting

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 the following resources of the Linux System:

* File Management
* Process Management
* I/O Management
* Memory management
* Device management

It is often mistaken that Linus Torvalds has developed Linux OS , but actually, he is only responsible for the development of the Linux kernel.

What is a Shell?

A shell is a special user program that provides an interface for the user to use operating system services. Shell accepts human-readable commands from users and converts them into something which that kernel can understand. It is a command language interpreter that executes commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or starts the terminal.

**Command Line Shell**

Shell can be accessed by users 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 executed.

The “**ls**” command with “**-l**” option is executed. It will list all the files in the current working directory in a long listing format.  
Working with a command line shell is a bit difficult for beginners because it’s hard to memorize so many commands.

It is very powerful; it allows users to store commands in a file and execute them together. This way any repetitive task can be easily automated. These files are usually called batch files in Windows and **Shell**Scripts in Linux/macOS systems.

**Graphical Shells**

Graphical shells provide means for manipulating programs based on the graphical user interface (GUI), by allowing for operations such as opening, closing, moving, and resizing windows, as well as switching focus between windows.

Window OS or Ubuntu OS can be considered as a good example which provides GUI to the user for interacting with the program. Users do not need to type in commands for every action.

There are several shells are available for Linux systems like –

* BASH (Bourne Again Shell) – It is the most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.
* CSH (C Shell) – The C shell’s syntax and its usage are very similar to the C programming language.
* KSH (Korn Shell) – The Korn Shell was also the base for the POSIX Shell standard specifications etc.

Each shell does the same job but understands different commands and provides different built-in functions.

What is a Terminal?

* A program which is responsible for providing an interface to a user that he/she can access the shell.
* It basically allows users to enter commands to see the output of those commands in a text-based interface.
* Large scripts that are written to automate and perform complex tasks are executed in the terminal.

**Shell Scripting**

As a shell can take commands as inputs from the 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 AScripts or Shell Programs . Shell scripts are similar to the batch files in MS-DOS. Each shell scripts are similar to the batch files in MS-DOS. Each shell script is saved with `.sh` file extension e.g., myscript.sh .

A shell script has syntac just like any other programming languages.

A shell script has comprise the following elements.

* Shell Keyword- 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.

**Some Advantages of shell scripts**

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

**Some Disadvantages of shell scripts**

* Prone to costly errors, a single mistake can change the command which might be harmful.
* Slow execution speed
* Design flaws within the language syntax or implementation
* Not well suited for large and complex task
* Provide minimal data structure unlike other scripting languages. etc.

COMPARISON OPERATORS

Integer comparison

|  |  |
| --- | --- |
| **Operator** | **Description** |
| -eq | Is equal to |
| -ne | Is not equal to |
| -gt | Is greater than |
| -ge | Is greater than or equal to |
| -lt | Is less than |
| -le | Is less than or equal to |
| == | Is equal to |
| != | Is not equal to |
| \< | Is less than, in ASCII alphabetical order |
| \> | Is greater than, in ASCII alphabetical order |

CONDITIONAL STATEMENTS:  
if statement:  
This block will process if specified condition is true.  
Syntax:

if [ expression ]

then

statement

fi

**Example:**

#!/bin/sh

x=10

y=11

If [ $x -ne $y ]

Then

Echo “Not Equal”

fi

**IF-Else Statement:**

In an if-else statement you can specify a set of commands to run if the condition is not met.

Syntax:

if [ expression ]

then

statement1

else

statement2

fi

Example:

#!/bin/sh

x=10

y=11

If [ $x -ne $y ]

then

echo “Not Equal”

else

echo “They are equal”

fi

**LOOPS**

**While loop:**

It starts running the specified commands if the condition is true and repeats them until the condition is false.

Syntax:

while [ condition ]

do

#set of statements

done

**Example:**

#!/bin/sh

x=2

while [ $x -lt 6 ]

do

echo $x

x=`expr $x + 1`

done

For Loop:

In a loop the variable iterates over a list of values and ends when there are no more values to iterate over.

**Syntax:**

For var in val1 val2 val3

do

#statements

done

**Example:**

#!bin/sh

For var in 2 3 4 5

do

echo $var

done

Positional Arguments:

Positional agumets are the arguments or values which we pass to the shell script while executing the script. They are accessed by variables $0, $1,$2,…..$9. Beyond that, they are referenced by ${10}, ${11} and so on. $# stores the no of passed arguments and $0 stores the no of passed arguments and $0 stores the name of the script.

#!/bin/sh

echo "No of arguments is $#"

echo "Name of the script is $0"

echo "First argument is $1"

echo "Second argument is $2"

Storing the output of commands

You can store the output of commands inside a variable in a shell script. There are two ways to do so.

SYNTAX:

#Syntax 1

var=$(a valid linux command)

#Syntax 2

var2=`a valid linux command`

**Let’s see an example:**

#!/bin/sh

b=$(pwd)

c=`pwd`

echo $b

echo $c

d=$(ls /bin | grep bash)

echo $d

**Exit Codes of shell commands**

Whenever a command ends and returns the control to the parent process, it returns exit codes between 0 and 255. Exit code 0 means the command was successful, and any other exit code means, the command was unsuccessful. You can view the exit code after running any command by accessing the $? variable.

You can manually set an exit code for your shell script. This can be used with conditional statements to convey if the script’s purpose was achieved or not.

**Example:**

#!/bin/sh

read x

if [ $x -ne 10 ]

then

echo failed

exit 1

else

echo passed

exit 0

fi