

# 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. In this article I 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 –

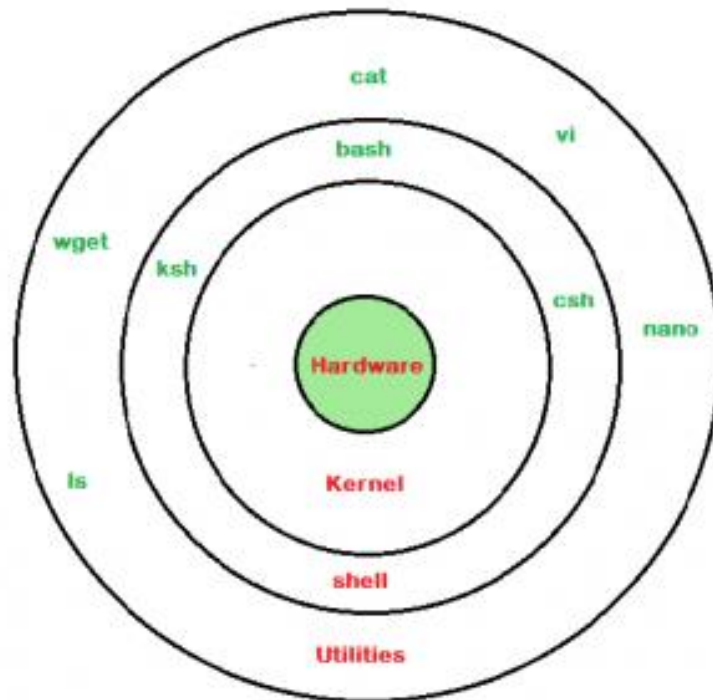
- File management
- Process management
- I/O management
- Memory management
- Device management etc.

It is often mistaken that [Linus Torvalds](#) has developed Linux OS, but actually he is only responsible for development of Linux kernel.

Complete Linux system = Kernel + [GNU](#) system utilities and libraries + other management scripts + installation scripts.

## 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.



### *linux shell*

Shell is broadly classified into two categories –

- Command Line Shell
- Graphical shell

### **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. The result is then displayed on the terminal to the user. A terminal in Ubuntu 16.4 system looks like this –

```

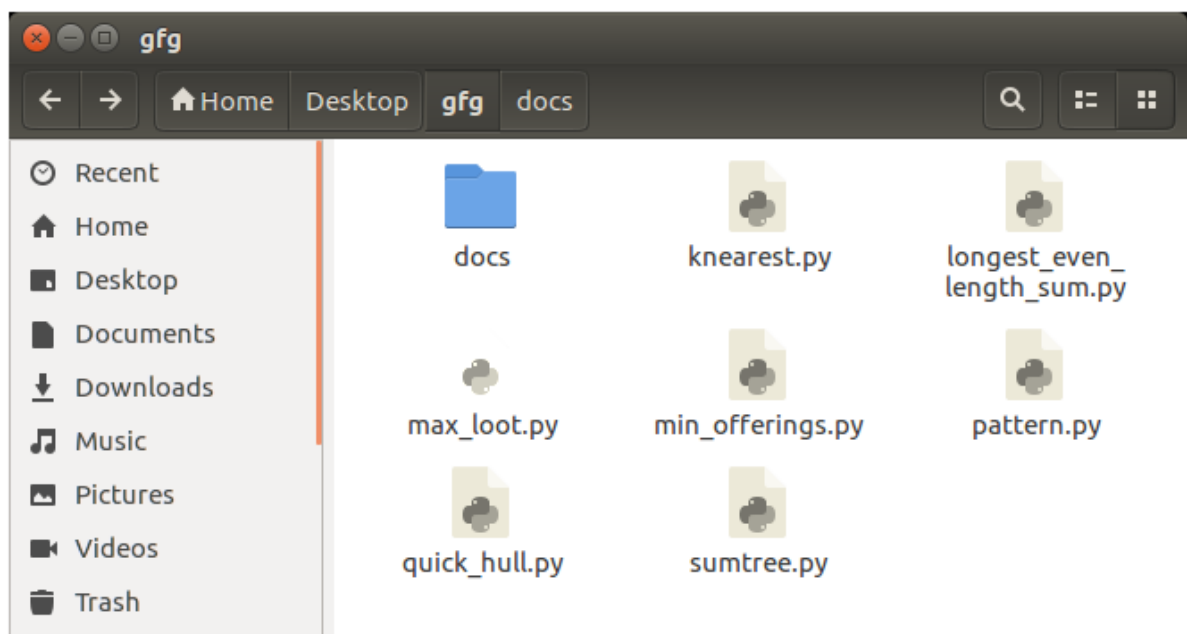
override@Atul-HP: ~
override@Atul-HP:~$ ls -l
total 212
drwxrwxr-x 5 override override 4096 May 19 03:45 acadenv
drwxrwxr-x 4 override override 4096 May 27 18:20 acadview_demo
drwxrwxr-x 12 override override 4096 May 3 15:14 anaconda3
drwxr-xr-x 6 override override 4096 May 31 16:49 Desktop
drwxr-xr-x 2 override override 4096 Oct 21 2016 Documents
drwxr-xr-x 7 override override 4096 Jun 1 13:09 Downloads
-rw-r--r-- 1 override override 8980 Aug 8 2016 examples.desktop
-rw-rw-r-- 1 override override 45005 May 28 01:40 hs_err_pid1971.log
-rw-rw-r-- 1 override override 45147 Jun 1 03:24 hs_err_pid2006.log
drwxr-xr-x 2 override override 4096 Mar 2 18:22 Music
drwxrwxr-x 21 override override 4096 Dec 25 00:13 Mydata
drwxrwxr-x 2 override override 4096 Sep 20 2016 newbin
drwxrwxr-x 5 override override 4096 Dec 20 22:44 nltk_data
drwxr-xr-x 4 override override 4096 May 31 20:46 Pictures
drwxr-xr-x 2 override override 4096 Aug 8 2016 Public
drwxrwxr-x 2 override override 4096 May 31 19:49 scripts
drwxr-xr-x 2 override override 4096 Aug 8 2016 Templates
drwxrwxr-x 2 override override 4096 Feb 14 11:22 test
drwxr-xr-x 2 override override 4096 Mar 11 13:27 Videos
drwxrwxr-x 2 override override 4096 Sep 1 2016 xdm-helper
override@Atul-HP:~$

```

In above screenshot “ls” command with “-l” option is executed. It will list all the files in current working directory in long listing format. Working with command line shell is bit difficult for the beginners because it’s hard to memorize so many commands. It is very powerful, it allows user 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 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 good example which provide GUI to user for interacting with program. User do not need to type in command for every actions. A typical GUI in Ubuntu system –



### GUI shell

There are several shells are available for Linux systems like –

- [BASH \(Bourne Again SHell\)](#) – It is 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 usage are very similar to the C programming language.

- [KSH \(Korn SHell\)](#) – The Korn Shell also was the base for the POSIX Shell standard specifications etc.

Each shell does the same job but understand different commands and provide different built in functions.

## 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**

### 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

- 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

## Shell scripting

# Writing a Bash Script

To start with Bash scripting, create a new file using a text editor. If you're using Vim, run the following command:

```
vim script.sh
```

The extension for Bash scripts is `.sh`. However, the extension is not necessary. Adding the `.sh` makes the file easy to identify and maintain.

## Adding the "shebang"

The first line in Bash scripts is a character sequence known as the "shebang." The shebang is the program loader's first instruction when executing the file, and the characters indicate which interpreter to run when reading the script.

Add the following line to the file to indicate the use of the Bash interpreter:

```
#!/bin/bash
```

The shebang consists of the following elements:

- `#!` directs the program loader to load an interpreter for the code in the file.
- `/bin/bash` the Bash interpreter's location.

Some typical shebang lines for different interpreters are in the table below.

Shebang	Interpreter
<code>#!/bin/bash</code>	Bash
<code>#!/bin/sh</code>	Bourne shell
<code>#!/usr/bin/env &lt;interpreter&gt;</code>	Uses the <b>env</b> program to locate the interpreter. Use this shebang for other scripting languages, such as Perl, Python, etc.
<code>#!/usr/bin/pwsh</code>	Powershell

## Variables

- Variables are the containers which store data
- The value can be a number, string, filename, device or any other type of data.
- 

### Types of variables

1. Local Variable
2. Environmental variable
3. Shell Variable

### Local variable :

- A local variable is a variable which is present in the current instance of a shell script.
- A variable declared as local is one that is visible only within the block of code in which it appears. It has local "scope". In a function, a local variable has meaning only within that function block.

Eg 1:- Func () {

NAME=SHANTANU # here the name variable is just limited to this particular function block but not in the program

}

Eg2-

If (int i=10;i<=5;i++) # here the value of i has a local scope just for this particular If condition

### Environmental variable :

- The Environment Variables form a simple and effective way to pass information about the current operating environment to the program being executed.
- Two common examples of Linux environment variables are the \$PATH and \$HOME variables.
- We can set env variables using export command.
- To simplify we can say that we can use the current shell variables in our script

## SHELL variables

- A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly
- A shell variable can be env. variable or a local variable.
- A shell variable is created with the following syntax:  
"variable\_name=variable\_value".

## Special variables

Variable	Description
<b>\$0</b>	The filename of the current script.
<b>\$n</b>	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is \$1, the second argument is \$2, and so on).
<b>\$\$</b>	The process ID of the current shell. For shell scripts, this is the process ID under which they are executing.
<b>\$#</b>	The number of arguments supplied to a script.
<b>\$@</b>	All arguments passed to script or function.
<b>\$*</b>	All arguments passed to script or function.
<b>\$?</b>	The exit status of the last command executed.
<b>!</b>	The process ID of the last background command.
<b>_</b>	The last argument of the previous command.

## Examples 1. Print output using echo command

### In the file vk.sh, included below things

```
#!/bin/bash
```

```
name=vishal    #variable is name and vishal is a data which is stored in name  
name2=manoj    #variable is name2 and manoj is the data which is stored in name2
```

```
echo "$name and $name2 both are friends"
```

### after executing

bash vk.sh or ./vk.sh

```
[ec2-user@ip-172-31-36-214 ~]$  
[ec2-user@ip-172-31-36-214 ~]$ ls  
vk.sh  
[ec2-user@ip-172-31-36-214 ~]$ bash vk.sh  
vishal and manoj both are friends  
[ec2-user@ip-172-31-36-214 ~]$
```



## Examples 2 : using special variable

```
#!/bin/sh

echo "File Name: $0"
echo "First argument : $1"
echo "Second argument : $2"
echo "All arguments passed: $@"
echo "All argument passed : $"
echo "Total Number of arguments passed to this script : $#"
```

echo " process id of curent shell : \$\$"

echo "exits status of last command excuted if pass return true value will be 0 and if f  
ailed there would be random value : \$?"

echo "The process id of the last background command : \$!"

echo " The last arguments of the previous command : \$\_"

#####

echo " who are childhood friends..?"

echo "\$1 and \$2 both are friends from childhood "

### output after executing

bash special\_variable.sh

```
[ec2-user@ip-172-31-36-214 ~]$
[ec2-user@ip-172-31-36-214 ~]$ ls
special_variables.sh
[ec2-user@ip-172-31-36-214 ~]$ bash special_variables.sh manoj vishal
File Name: special_variables.sh
First argument : manoj
Second argument : vishal
All arguments passed: manoj vishal
All argument passed : manoj vishal
Total Number of arguments passed to this script : 2
 process id of curent shell : 4829
exits status of last command excuted if pass return true value will be 0
The process id of the last background command :
The last arguments of the previous command : The process id of the last
who are childhood friends..?
manoj and vishal both are friends from childhood
[ec2-user@ip-172-31-36-214 ~]$
```

**Manoj and vishal both are arguments pass while running script**

## Basic operators in shell

- **Arithmetic Operators**
- **Relational operators**
- **Boolean operators**
- **String operators**
- **File Test operators**

### Arithmetic Operators :

assuming a=10 and b=20

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr \$a + \$b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr \$a - \$b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr \$a \* \$b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr \$b / \$a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr \$b % \$a` will give 0
= (Assignment)	Assigns right operand in left operand	a = \$b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[ \$a == \$b ] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[ \$a != \$b ] would return true.

### Examples 3 : using arithmetic operators print values

Created file arithmetic.sh and saved following things:

```
#!/bin/bash

echo "enter value of x"

read x    # here " read " command take input from user and stored value in x variable

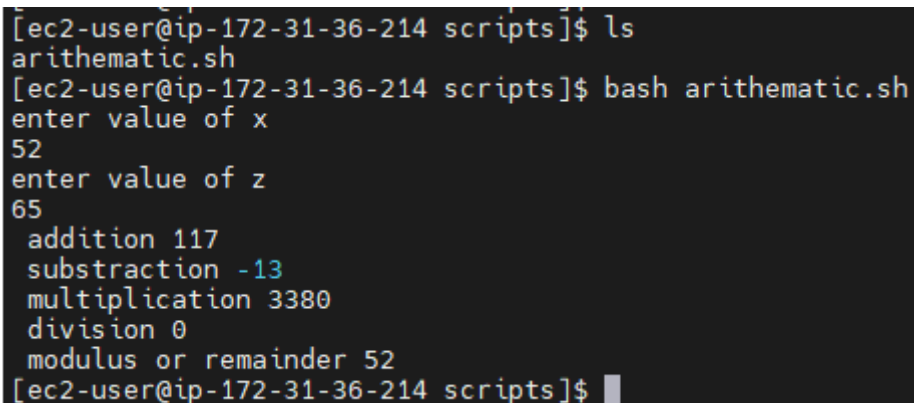
echo "enter value of z"

read z    # here " read " command take input from user and stored value in z variable


echo " addition `expr $x + $z`"
echo " substraction `expr $x - $z`"
echo " multiplication `expr $x \* $z`"
echo " division `expr $x / $z`"
echo " modulus or remainder `expr $x % $z`"
```

### output

bash arithmetic.sh

A terminal window showing the execution of the arithmetic.sh script. The user runs 'ls' and 'bash arithmetic.sh'. The script prompts for 'x' (52) and 'z' (65), then displays the results of addition (117), subtraction (-13), multiplication (3380), division (0), and modulus (52).

```
[ec2-user@ip-172-31-36-214 scripts]$ ls
arithmetic.sh
[ec2-user@ip-172-31-36-214 scripts]$ bash arithmetic.sh
enter value of x
52
enter value of z
65
addition 117
substraction -13
multiplication 3380
division 0
modulus or remainder 52
[ec2-user@ip-172-31-36-214 scripts]$
```