## **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 etc.

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

Complete Linux system = Kernel + GNUsystem utilities and libraries + other management scripts + installation scripts.

## **What is Shell?**

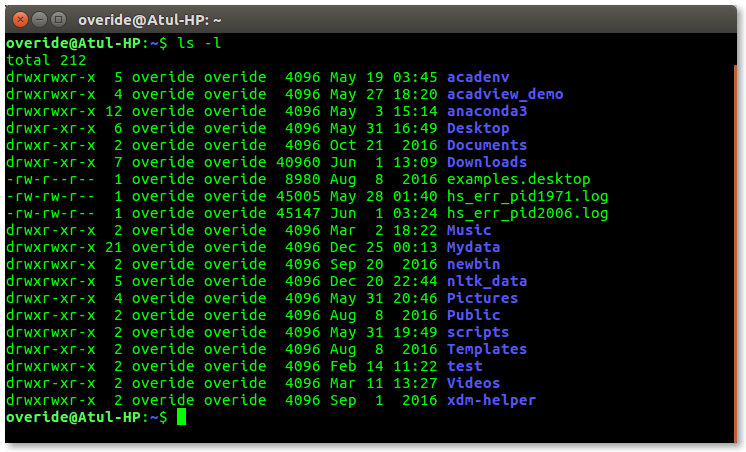
A shell is a special user program that provides an interface for the user to use operating system services. Shell accepts human-readable commands from users and converts them into something which the kernel can understand. It is a command language interpreter that executes commands read from input devices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

Shell is broadly classified into two categories –

* Command Line Shell
* Graphical shell

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

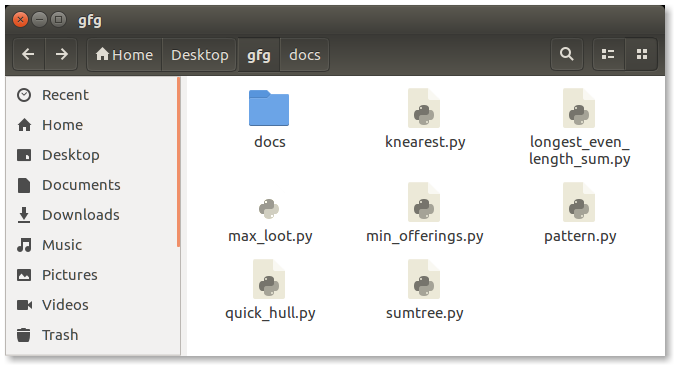


*linux command line*

In the above screenshot “**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. A typical GUI in the Ubuntu system –



*GUI Shell*

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 so that he/she can access the shell. It basically allows users to enter commands and 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**

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

As a 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 e.g., **myscript.sh.**

A shell script has 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 the 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.

### **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.

**Creating a Shell Script**

Login to your Linux machine and open the terminal, navigate to the folder where you want to store the shell script. Shell scripts end with the extension “.sh”. Let’s create our first shell script. Type in

touch script.sh

Now, this script file is not executable by default, we have to give the executable permission to this file. Type in

chmod +x script.sh

Now, we will add some commands to this shell script. Open this shell script with any text editor of your choice (command-line based or GUI based) and add some commands. We will use nano. Type in

Vim script.sh

Add the following commands to test this shell script

* echo This is my first shell script
* touch testfile
* ls
* echo End of my shell script

Save the changes, and run the shell script by typing in ./script.sh

**Comments in the shell script**

Any line which starts with “#” in the shell script is treated as a comment and is ignored by the shell during execution, except the shebang line, which we will see later in this article. Let’s see an example. A shell script is created with the following content.

# This is a comment

echo Testing comments in shell script

**Variables in Shell Script**

Yes, Shell scripts support the use of variables, and we need not define a variable’s type during its declaration. There are two types of variables:

System Defined variables

User-Defined Variables.

System-defined variables, also called environment variables, are generally Capitalised. You can view all the current environment variables using the printenv command.

User-Defined variables are set by the user, and they exist only during script execution. You can define a variable by simply typing its name and assigning a value with = sign and access a variable by adding a $ before the variable name. Variables are demonstrated in the following example script.

* # Accessing an Environment Variable
* echo $USER
* # Creating and accessing User defined Variable
* variable\_name="Geeksforgeeks"
* echo $variable\_name

### 1)Accessing variable

Variable data could be accessed by appending the variable name with ‘$’

**2) Unsetting Variables**

The unset command directs a shell to delete a variable and its stored data from list of variables. It can be used as follows:

* #!/bin/bash
* var1="Devil"
* var2=23
* echo $var1 $var2
* unset var1
* echo $var1 $var2

Output:

* DEVIL 23
* 23

## **Variable Types**

We can discuss three main types of variables:

### ****1) Local Variable:****

Variables which are specific to the current instance of shell. They are basically used within the shell, but not available for the program or other shells that are started from within the current shell.

**For example:**

`name=Jayesh`

In this case the local variable is (name) with the value of Jayesh. Local variables is temporary storage of data within a shell script.

### 2) Environment Variable:

These variables are commonly used to configure the behavior script and programs that are run by shell. Environment variables are only created once, after which they can be used by any user.

**For example:**

`export PATH=/usr/local/bin:$PATH` would add `/usr/local/bin` to the beginning of the shell’s search path for executable programs.

### 3) Shell Variables:

Variables that are set by shell itself and help shell to work with functions correctly. It contains both, which means it has both, some variables are Environment variable, and some are Local Variables.

**For example:**

`$PWD` = Stores working directory

`$HOME` = Stores user’s home directory

`$SHELL` = Stores the path to the shell program that is being used.

## Few more examples in Shell Scripting and Shell Variable

### How to Store User Data in a Variable?

#!/bin/bash  
 echo "Enter the length of the rectangle"  
 read length  
 echo "Enter the width of the rectangle"  
 read width  
 area=$((length \* width))  
 echo "The are of the rectangle is: $area"

**Defining the Shell Script interpreter**

There are many Shells available in Linux, such as The bourne shell(sh), The Korn Shell(ksh), and GNU Bourne-Again Shell(bash). Scripts written for the sh shell are called shell scripts, and they can be interpreted by both, the ksh and bash shells. ksh and Bash are improved versions of the original sh shell and they have more features than sh. Bash is generally the default shell in most of the Linux Distributions and scripts written specifically for bash shell are called bash scripts. You can specify which shell the script will use, even if the script is executed from another shell terminal. To do this, add “#!” on top of the script file, followed by the absolute path of the shell of choice. To specify bash as an interpreter, Add the following line on top of the shell script.

#!/bin/bash

This line is called the shebang line.

Note: This will only work if bash is installed on your system.

**Comparison Operators**

You can compare two variables in shell scripting. We do these comparisons to make decisions, we will see how to do that later in this article, but before that, here is a list of some 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

**String Comparison**

Operator Description

== is equal to

!= is not equal to

\< is less than, in ASCII alphabetical order

\> is greater than, in ASCII alphabetical order

We add a \ before < and > because they need to be escaped when typed in the [ ] construct. Now, let’s see where these are used.

**Conditional statements**

Conditional statements are used to execute a block of code only when certain conditions are met. Shell scripts support the use of conditional statements. We use comparison operators to check the conditions. Let’s see a few conditional statements.

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

**If statement**

It checks the condition, and if it is conditioned true, it executes the commands.

Syntax

* if [ condition ]
* then
* #statements
* Fi

Let’s see an 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 [ condition ]
* then
* #set of statements if the condition is true
* else
* #set of statements if the condition is false
* fi

Let’s see an example

* #!/Syntaxbin/sh
* x=10
* y=10
* if [ $x -ne $y ]
* then
* echo "Not equal"
* else
* echo "They are equal"
* fi

Note: Type a space after [ and before ] while specifying the condition to be checked otherwise you will get an error.

**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.  
***Syntax:***

if [ expression1 ]

then

statement1

statement2

.

else

if [ expression2 ]

then

statement3

.

fi

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 in

Pattern 1) Statement 1;;

Pattern n) Statement n;;

Esac

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

**Loops**

Using loops, you can a set of commands over and over again, until a certain condition is met. Let’s see some of the 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

Let’s see an example.

* #!/bin/sh
* x=2
* while [ $x -lt 6 ]
* do
* echo $x
* x=`expr $x + 1`
* done

**For loop**

In a for 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

Let’s see an example.

* #/bin/bash
* #Start of for loop
* for a in 1 2 3 4 5 6 7 8 9 10  
  do
* # if a is equal to 5 break the loop  
  if [ $a == 5 ]  
  then  
  break  
  fi
* # Print the value  
  echo “Iteration no $a”  
  done

**until` statement in Shell Script in Linux**

The until loop is executed as many times as the condition/command evaluates to false. The loop terminates when the condition/command becomes true.

Syntax:

* #/bin/bash
* until <condition>
* do
* <command 1>
* <command 2>
* <etc>
* Done

Example:

#/bin/bash

a=0

# -gt is greater than operator

#Iterate the loop until a is greater than 10

until [ $a -gt 10 ]

do

# Print the values

echo $a

# increment the value

a=`expr $a + 1`

done

**7) Command-line Arguments:**

Handling arguments passed to the script.

In shell scripting, you can access command-line arguments passed to the script using special variables. These arguments allow users to provide input to the script at runtime. Here's how you can handle command-line arguments in shell scripts:

Accessing Command-line Arguments:

$0: Represents the name of the script.

$1, $2, $3, ...: Represent the first, second, third, etc., command-line arguments passed to the script.

**$@**: Represents all command-line arguments as separate words.

#!/bin/bash

# Display script name

echo "Script name: $0"

# Display first and second command-line arguments

echo "First argument: $1"

echo "Second argument: $2"

# Display all command-line arguments

echo "All arguments: $@"

./script.sh arg1 arg2 arg3

Output :

Script name: script.sh

First argument: arg1

Second argument: arg2

All arguments: arg1 arg2 arg3

**Checking the Number of Arguments:**

You can check the number of command-line arguments passed to the script using the **$#** variable

#!/bin/bash

# Check the number of command-line arguments

if [ "$#" -eq 0 ]; then

echo "No arguments provided."

elif [ "$#" -eq 1 ]; then

echo "One argument provided."

else

echo "More than one argument provided."

fi

Output: More than one argument provided.

**Notes:**

1. Always quote command-line arguments to handle spaces or special characters properly. For example, use **"$@"** instead of **$@**.
2. Remember that **$0** represents the script's name, not the first argument passed to the script.
3. You can use command-line arguments to make your scripts more flexible and versatile, allowing users to customize script behavior based on their requirements.
4. Handling command-line arguments effectively in your shell scripts enables users to provide input dynamically, making your scripts more interactive and adaptable to different use cases

**8) File Handling:**

Reading from and writing to files.

In shell scripting, you can perform file handling operations such as reading from and writing to files using various commands and redirection techniques. Here's how you can read from and write to files in shell scripts:

**Reading from Files:**

You can use commands like **cat**, **head**, **tail**, **grep**, **awk**, or **sed** to read from files. Additionally, you can use input redirection (**<**) to redirect the contents of a file into a command or read line by line using a **while** loop.

**Example using** cat**:**

#!/bin/bash

# Read the contents of a file using cat

cat file.txt

Example using while loop:

#!/bin/bash

# Read the file line by line using while loop

while IFS= read -r line; do

echo "$line"

done < file.txt

**Writing to Files:**

You can use redirection operators (> for writing, >> for appending) to write output to files. Additionally, you can use commands like echo, printf, or cat to generate content to be written to files.

Example using echo:

#!/bin/bash

# Write to a file using echo

echo "Hello, world!" > output.txt

Example using printf:

#!/bin/bash

# Write to a file using printf

printf "%s\n" "Line 1" "Line 2" "Line 3" > output.txt

**Appending to Files:**

To append content to an existing file, you can use the >> redirection operator.

#!/bin/bash

# Append to a file using echo

echo "Additional line" >> output.txt

Example using cat:

#!/bin/bash

# Append to a file using cat

cat >> output.txt <<EOF

Additional line 1

Additional line 2

EOF

**Notes:**

1. Make sure the file paths are correctly specified to read from or write to the desired files.
2. Ensure appropriate permissions are set for file handling operations, especially when writing to files.
3. Use error handling techniques to handle file-related errors gracefully, such as checking for file existence or permission issues.
4. File handling is a fundamental aspect of shell scripting, enabling you to manipulate and process data stored in files efficiently. Understanding how to read from and write to files allows you to create scripts that can perform various data processing and management tasks.

**9) String Manipulation:**

Operations like concatenation, substitution, and pattern matching.

In shell scripting, string manipulation is a common task, involving operations such as concatenation, substitution, and pattern matching. Here's how you can perform these operations in shell scripts:

**Concatenation:**

You can concatenate strings using the concatenation operator (+).

#!/bin/bash

# Concatenate two strings

string1="Hello, "

string2="world!"

concatenated\_string="$string1$string2"

echo "$concatenated\_string"

**Substitution:**

You can substitute parts of a string using parameter expansion or commands like sed, awk, or grep.

**Example using parameter expansion:**

#!/bin/bash

# Substitute part of a string using parameter expansion

original\_string="Hello, world!"

substituted\_string="${original\_string/world/planet}"

echo "$substituted\_string"

**Example using sed:**

#!/bin/bash

# Substitute part of a string using sed

original\_string="Hello, world!"

substituted\_string=$(echo "$original\_string" | sed 's/world/planet/')

echo "$substituted\_string"

**Pattern Matching:**

You can perform pattern matching using commands like grep, awk, or [[ ... ]] constructs.

**Example using grep:**

#!/bin/bash

# Pattern matching using grep

string="Hello, world!"

if echo "$string" | grep -q "world"; then

echo "Pattern found"

else

echo "Pattern not found"

fi

**Example using [[ ... ]]:**

#!/bin/bash

# Pattern matching using [[ ... ]]

string="Hello, world!"

if [[ "$string" == \*"world"\* ]]; then

echo "Pattern found"

else

echo "Pattern not found"

fi

**Notes:**

1. Parameter expansion and pattern matching in shell scripting are powerful tools for manipulating strings efficiently.
2. Commands like sed, awk, and grep provide more advanced string manipulation capabilities, especially for complex pattern matching and substitution tasks.
3. Always ensure proper quoting of strings to handle spaces and special characters correctly.
4. Regular expressions can be used for more advanced pattern matching tasks.
5. String manipulation is a crucial aspect of shell scripting, enabling you to process and manipulate textual data effectively. By mastering string manipulation techniques, you can create more versatile and powerful shell scripts for various tasks.

**10) Arrays:**

Working with arrays to store and manipulate data.

In shell scripting, arrays provide a convenient way to store and manipulate multiple values under a single variable name. Shell arrays can hold strings or integers and are indexed starting from 0. Here's how you can work with arrays in shell scripts:

**Declaring Arrays:**

You can declare an array by assigning values to it within parentheses, separated by spaces.

#!/bin/bash

# Declare an array

my\_array=(value1 value2 value3)

**Accessing Array Elements:**

You can access individual elements of an array using their index enclosed in square brackets [].

#!/bin/bash

# Accessing array elements

echo "First element: ${my\_array[0]}"

echo "Second element: ${my\_array[1]}"

echo "Third element: ${my\_array[2]}"

**Length of an Array:**

You can find the length of an array using the ${#array[@]} syntax.

#!/bin/bash

# Length of an array

echo "Length of the array: ${#my\_array[@]}"

**Modifying Array Elements:**

You can modify individual elements of an array by assigning new values to them.

#!/bin/bash

# Modifying array elements

my\_array[1]="new\_value"

**Iterating Over Arrays:**

You can iterate over all elements of an array using a for loop.

#!/bin/bash

# Iterate over array elements

for element in "${my\_array[@]}"; do

echo "$element"

done

**Example:**

bash

Copy code

#!/bin/bash

# Declare an array

fruits=("Apple" "Banana" "Orange" "Mango")

# Accessing array elements

echo "First fruit: ${fruits[0]}"

echo "Second fruit: ${fruits[1]}"

echo "Number of fruits: ${#fruits[@]}"

# Modify array element

fruits[1]="Grapes"

# Iterate over array elements

echo "Updated list of fruits:"

for fruit in "${fruits[@]}"; do

echo "$fruit"

done

**Notes:**

1. Arrays in shell scripting are versatile and allow you to store and manipulate collections of data efficiently.
2. It's important to properly quote array variables to handle values containing spaces or special characters correctly.
3. Shell arrays can be multidimensional, although they are not commonly used due to their complexity.
4. Understanding how to work with arrays in shell scripting is essential for building more complex and dynamic scripts that involve processing and managing collections of data. Arrays provide a flexible and powerful mechanism for handling such tasks in shell scripts.Arrays: Working with arrays to store and manipulate data.

**Error Handling:** Handling errors and exceptions in shell scripts.

**Regular Expressions**: Using regex for pattern matching and text manipulation.

**Mathematical Operations**: Performing arithmetic operations in shell scripts.

**Debugging**: Techniques for debugging shell scripts.

**Advanced Topics:**

**Signals and Traps**: Handling signals and setting traps for graceful script termination.

**Advanced File Operations**: Working with directories, permissions, and symbolic links.

**Process Management**: Controlling processes and managing their execution.

**Networking**: Interacting with network resources (e.g., sending HTTP requests).

**System Administration**: Automating system administration tasks.

**Security:** Writing secure shell scripts and handling sensitive data.

**Writing Shell Scripts Program Examples:**

Here are some example programs you can write to practice shell scripting:

File Backup Script: Automate the backup of specified files or directories.

Log Analyzer: Parse log files and extract useful information.

System Monitoring Script: Monitor system resources like CPU, memory, and disk usage.

Password Generator: Generate random passwords of specified length.

File Renamer: Rename files in bulk based on certain criteria.

Website Uptime Checker: Check the uptime of a list of websites and send alerts if they're down.

Data Encryption/Decryption: Encrypt and decrypt sensitive data using shell scripting.

CSV File Processor: Parse and manipulate CSV files using shell scripts.

System Cleanup Script: Automate tasks like removing temporary files, cleaning up logs, etc.

Automated Deployment Script: Automate the deployment process for applications.

These topics and programs should give you a solid foundation in shell scripting, from basic to advanced levels. Practice is key to mastering shell scripting, so try writing scripts for various tasks and challenges you encounter.