Practical 1

- Aim: Study of basic commands of Linux/UNIX.

❖ ls

Description: This commands lists all the contents in the current working directory.

Syntax: ls

Example: $ls-l file1

❖ cd

Description: It changes into the given directory or into the home directory when no parameter is provided.

Syntax: cd[directory]

Example: $cd dir1 dir2

❖ du

Description: “du” command is used to show disk space usage of files present in a directory as well as its sub directories as below.

Syntax: du

Example: $du/home/aronilik

- ❖ **cp**

Description: “cp” command is used for copying files and directories from on location to another.

Syntax: cp file1 file2

Example: cp/home/techmint/file1/home/techmint/personal/

- ❖ **rmdir**

Description: “rmdir” command helps to delete/ rename empty directories as follows.

Syntax: rmdir directory

Example: $rmdir /backup/all

- ❖ **pwd**

Description: “pwd” command displays the name of current working directory. {pwd – present working directory}

Syntax: pwd

Example: $pwd

- ❖ **mv**

Description: “mv” command is used to rename files or directories. It also moves a file/directory to another location in the directory structure.

Syntax: mv file1 file2

Example: $mv test.sh sysinfo.sh

❖ <u>mkdir</u>

Description: “mkdir” command is used to create single or more directories, if they do not already exists.

Syntax: mkdir directory

Example: $mkdir techmint-files

❖ <u>rm</u>

Description: “rm” command helps to delete/remove empty directories as shown below.

Syntax: rm directory

Example: $rm file1

❖ <u>touch</u>

Description: “touch” command changes file stamps, it can also be used to create a file.

Syntax: touch file1 file2

Example: $touch file.txt

Practical 2

- Aim: Study of advance commands and filters of Linux/UNIX.

❖ cat

Description: “cat” command is used to view contents of a file or concatenate files or data provided on standard input, and display it on the standard output.

Syntax: cat file1

Example: $cat file.txt

❖ more

Description: “more” command enables you to view through relatively lengthy text files one screen at a time.

Syntax: more file 1

Example: $ more file.txt

❖ head

Description: “head” command is used to show first lines (10 lines by default) of the specified file or stdin to the screen….

Syntax: head[OPTION]…[file]…

Example: $ head state.txt

Assam

Gujrat

Maharashtra

❖ tail

Description: "tail" command is used to display the last lines (10 lines by default) of each files to standard output.

Syntax: head[OPTION]…[file]…

Example: $tail longfile.

❖ find

Description: "find" command lets you search for files in a directory as well as its sub-directories. It searches for files by attributes such as permissions, users, groups, file type , data, size, and other possible criteria.

Syntax: find[where to start searching from].[what to find]

Example: $find/home/techmint/-name.techmint.txt

❖ diff

Description: "diff" command is used to compare two files line by line. It can also be used to find the difference between two directories.

Syntax: diff dir1 dir 2

Example:$ diff file1 file2

❖ chgrp

Description: “chgrp” command is used to change the group ownership of a file. Provide the new group name as its first argument and the name of file as the second argument.

Syntax: chgrp[OPTION]…GROUP FILE…..

Example: $ chgrp techmint users.txt

❖ chmod

Description: “chmod” command is used to change or update file permissions like given below:

Syntax: chmod[reference][operator][mode]file…

Example: $chmod +x sysinfo.sh

❖ chown

Description: “chown” command changes or updates the user and group ownership of a file or directory.

Syntax: chown[OPTION]…[OWNER][:[Group]]FILE…

Example: $chmod –R www-data:www-data/var/www/html

❖ wc

Description: “wc” command is used to display new line,word and byte counts for each file specified, and a total for many files

Syntax: wc[OPTION]…[FILE]…

Example: $ wc filename

❖ split

Description: “split” command is used to split a large file into small parts.

Syntax: wc[OPTION]…[INPUT[PREFIX]]

Example: $tar-cvjf backup.tar.bz2/home/techmint/Document*

❖ sort

Description: “sort” command is used to sort lines of text in the specified files or from stdin,

Syntax: sort file

Example: $cat words.txt or $sort file.txt

❖ grep

Description: “grep” command searches for a specified pattern in a file and displays in output lines containing that pattern.

Syntax: grep[options] pattern[files]

Example:$grep ‘techmint’ domain-list.txt

❖ tr

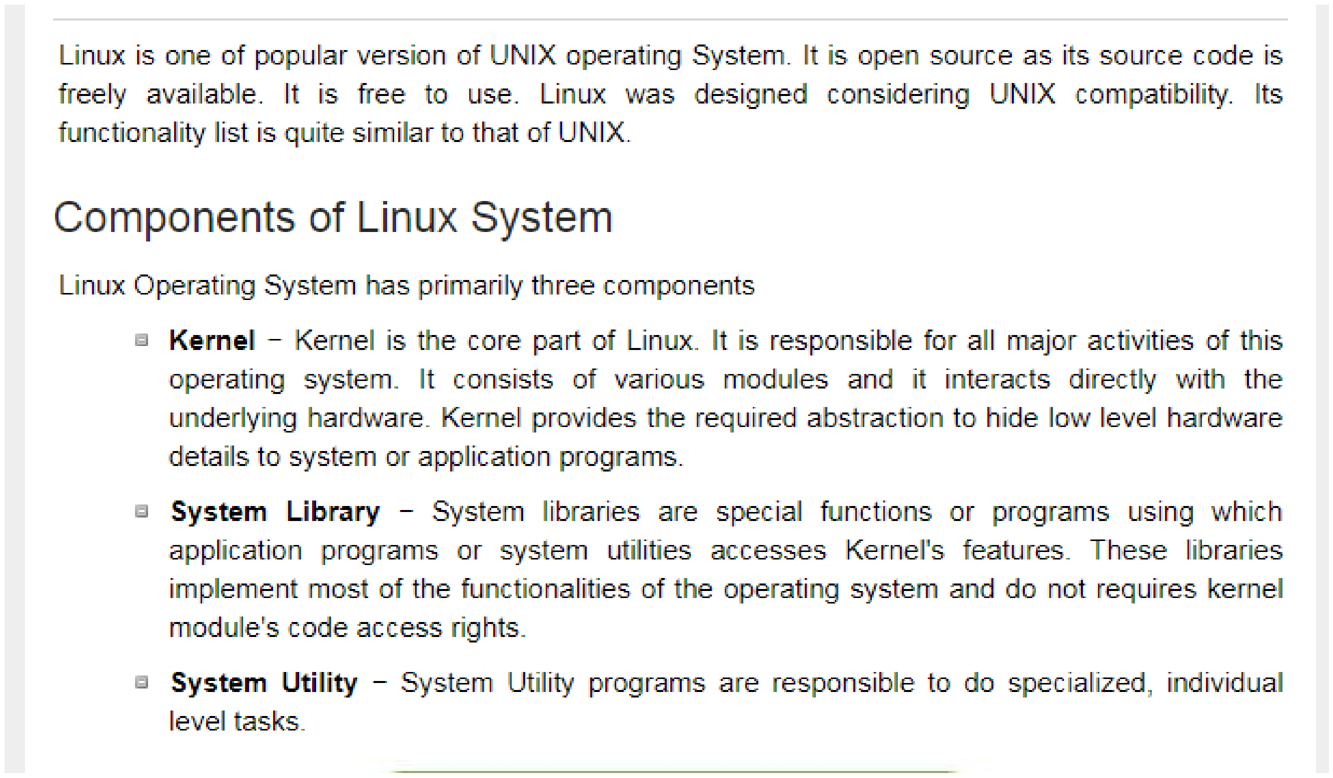
Description: “tr” command is used for transating or deleting or squeezing repeated characters. It will read from stdin and write to stdout.

Syntax: tr[option] set1[set2]

Example: tr[pooja] Priya

Practical 3

- Aim: Study of Linux architecture with commands.

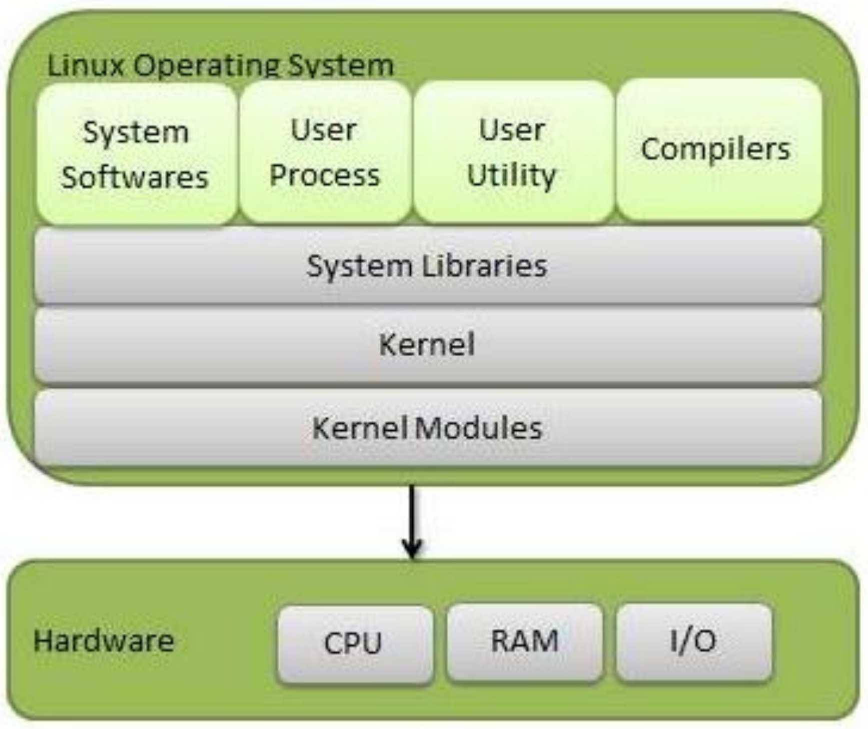


Linux is one of popular version of UNIX operating System. It is open source as its source code is freely available. It is free to use. Linux was designed considering UNIX compatibility. Its functionality list is quite similar to that of UNIX.

Components of Linux System

Linux Operating System has primarily three components

- **Kernel** − Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.
- **System Library** − System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of the functionalities of the operating system and do not requires kernel module's code access rights.
- **System Utility** − System Utility programs are responsible to do specialized, individual level tasks.



Kernel Mode vs User Mode

Kernel component code executes in a special privileged mode called **kernel mode** with full access to all resources of the computer. This code represents a single process, executes in single address space and do not require any context switch and hence is very efficient and fast. Kernel runs each processes and provides system services to processes, provides protected access to hardware to processes.

Support code which is not required to run in kernel mode is in System Library. User programs and other system programs works in **User Mode** which has no access to system hardware and kernel code. User programs/ utilities use System libraries to access Kernel functions to get system's low level tasks.

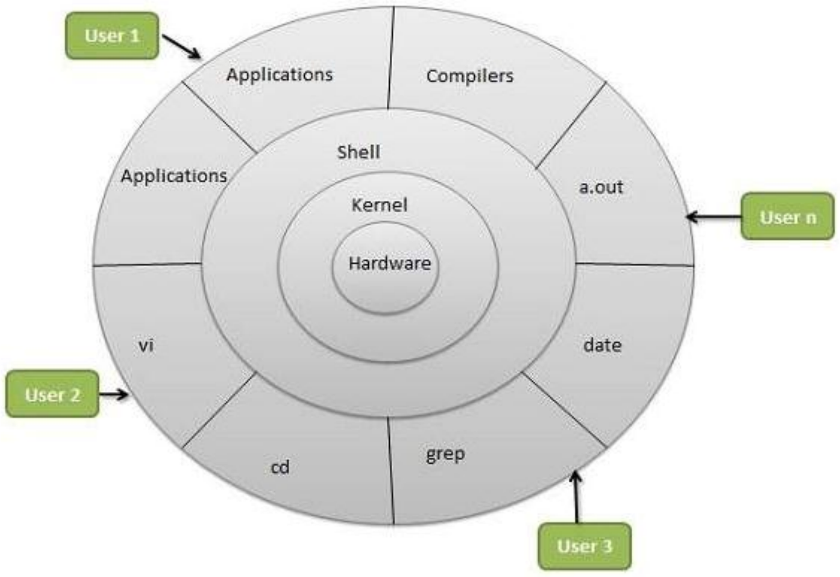
Basic Features

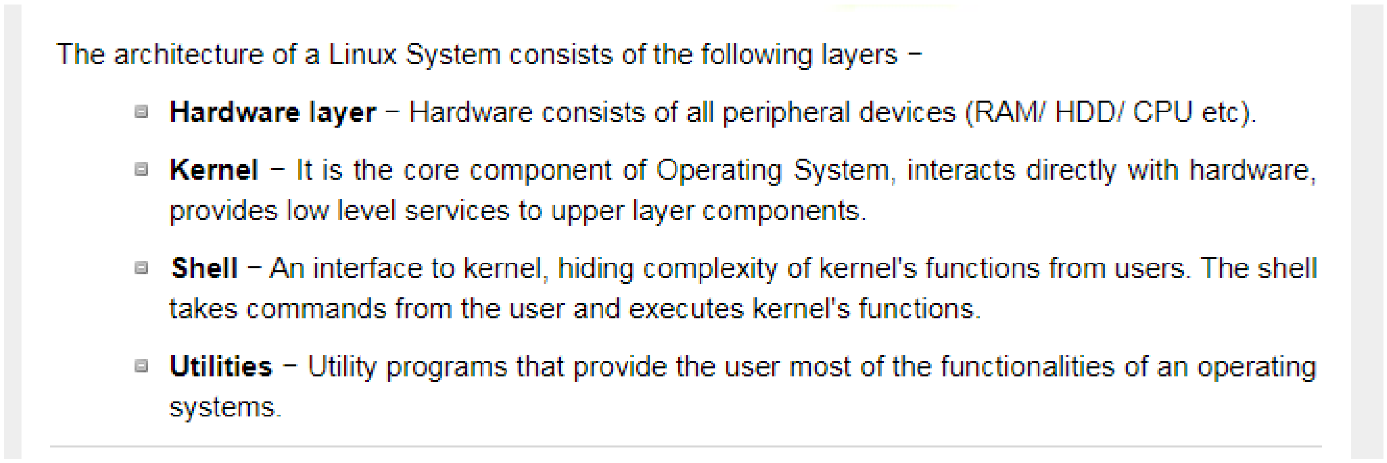
Following are some of the important features of Linux Operating System.

- **Portable** − Portability means software can works on different types of hardware in same way. Linux kernel and application programs supports their installation on any kind of hardware platform.
- **Open Source** − Linux source code is freely available and it is community based development project. Multiple teams work in collaboration to enhance the capability of Linux operating system and it is continuously evolving.
- **Multi-User** − Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at same time.
- **Multiprogramming** − Linux is a multiprogramming system means multiple applications can run at same time.
- **Hierarchical File System** − Linux provides a standard file structure in which system files/ user files are arranged.
- **Shell** − Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.
- **Security** − Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

Architecture

The following illustration shows the architecture of a Linux system −



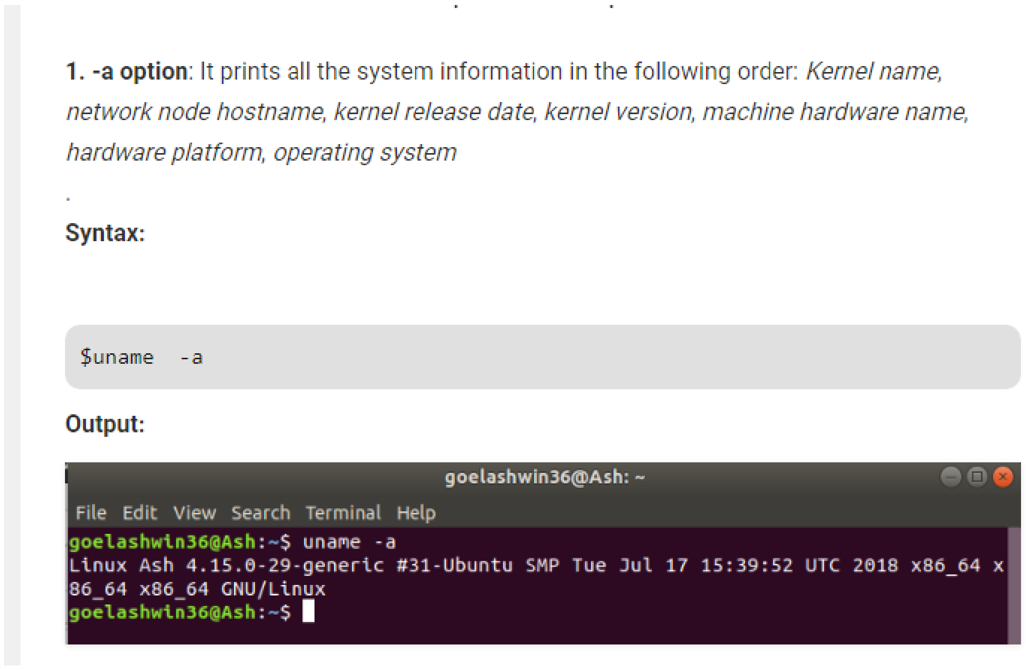


The architecture of a Linux System consists of the following layers −

- **Hardware layer** − Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).
- **Kernel** − It is the core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.
- **Shell** − An interface to kernel, hiding complexity of kernel's functions from users. The shell takes commands from the user and executes kernel's functions.
- **Utilities** − Utility programs that provide the user most of the functionalities of an operating systems.

- ➢ Here are some five simple methods to verify your Linux system's OS type
- ➢ The following command will work on almost aa operating system such as RHEL, CentOS, Fedora, Scientific Linux, Ubuntu, Linux Mint, open SUSE etc.

❖ <u>uname command</u>



1. -a option: It prints all the system information in the following order: *Kernel name, network node hostname, kernel release date, kernel version, machine hardware name, hardware platform, operating system*

.

Syntax:

```
$uname  -a
```

Output:

```
goelashwin36@Ash:~$ uname -a
Linux Ash 4.15.0-29-generic #31-Ubuntu SMP Tue Jul 17 15:39:52 UTC 2018 x86_64 x
86_64 x86_64 GNU/Linux
goelashwin36@Ash:~$ 
```

❖ dpkg command

Description

The primary and more user-friendly front-end for **dpkg** is **aptitude**. **dpkg** itself is controlled entirely via command line parameters, which consist of exactly one action and zero or more options. The parameter tells **dpkg** what to do and options control the behavior of the action in some way.

dpkg can also be used as a front-end to **dpkg-deb** and **dpkg-query**. The list of supported actions is below (in the "Actions" section). If any such action is encountered dpkg just runs **dpkg-deb** or **dpkg-query** with the parameters given to it, but no specific options are currently passed to them, to use any such option the back-ends need to be called directly.

Syntax

```
dpkg [option...] action
```

Examples

```
dpkg -l '*vi*'
```

List installed packages related to the editor **vi**.

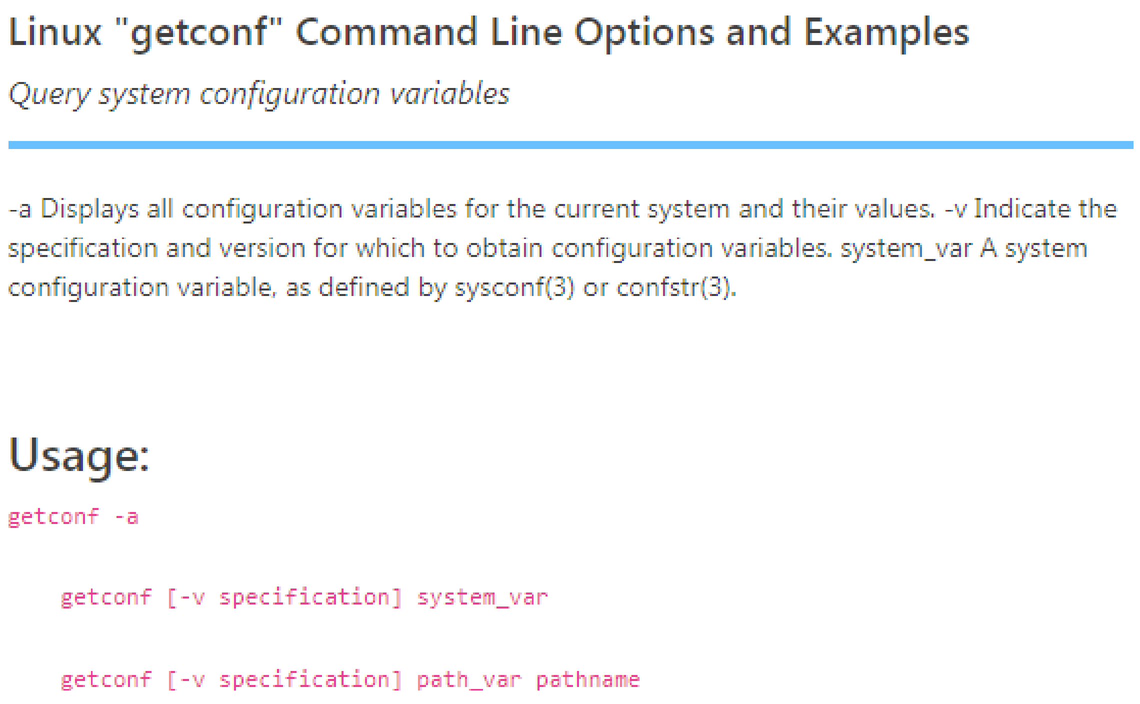
```
dpkg --print-avail elvis vim | less
```

View the entries of the packages **elvis** and **vim** as listed in **/var/lib/dpkg/available**.

```
less /var/lib/dpkg/available
```

Manually view the list of available software packages.

❖ getconf command



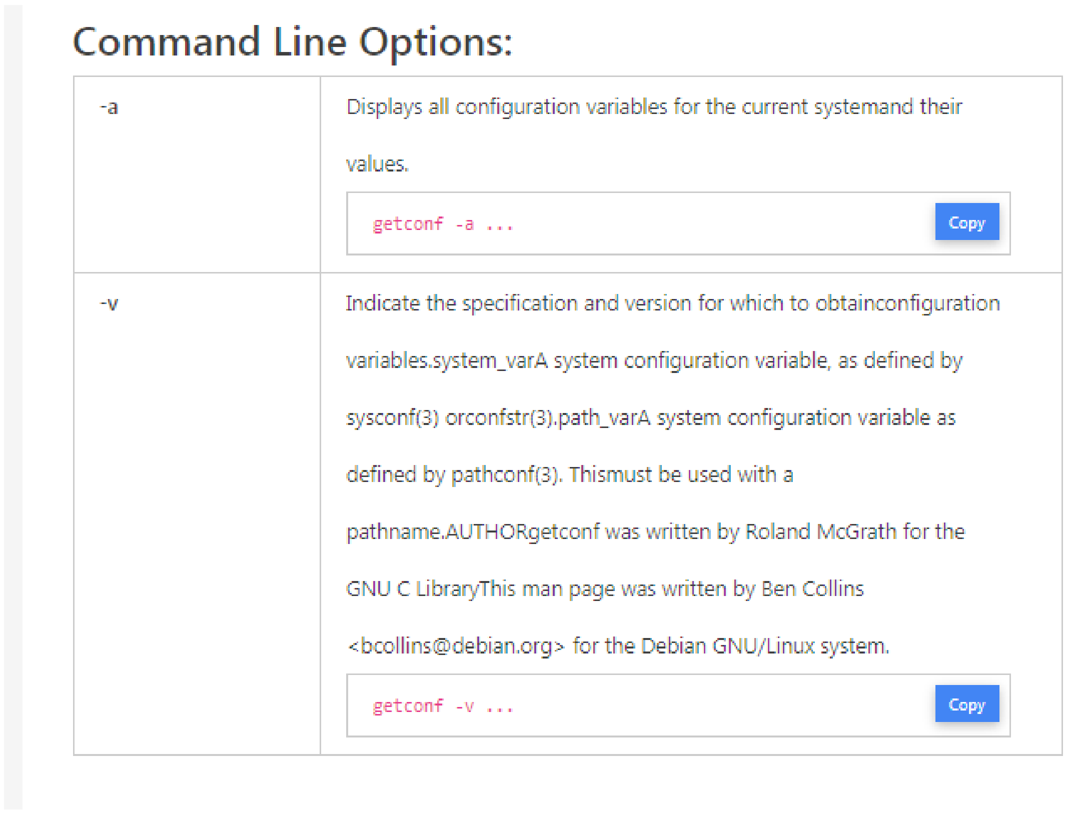
Linux "getconf" Command Line Options and Examples

Query system configuration variables

-a Displays all configuration variables for the current system and their values. -v Indicate the specification and version for which to obtain configuration variables. system_var A system configuration variable, as defined by sysconf(3) or confstr(3).

Usage:

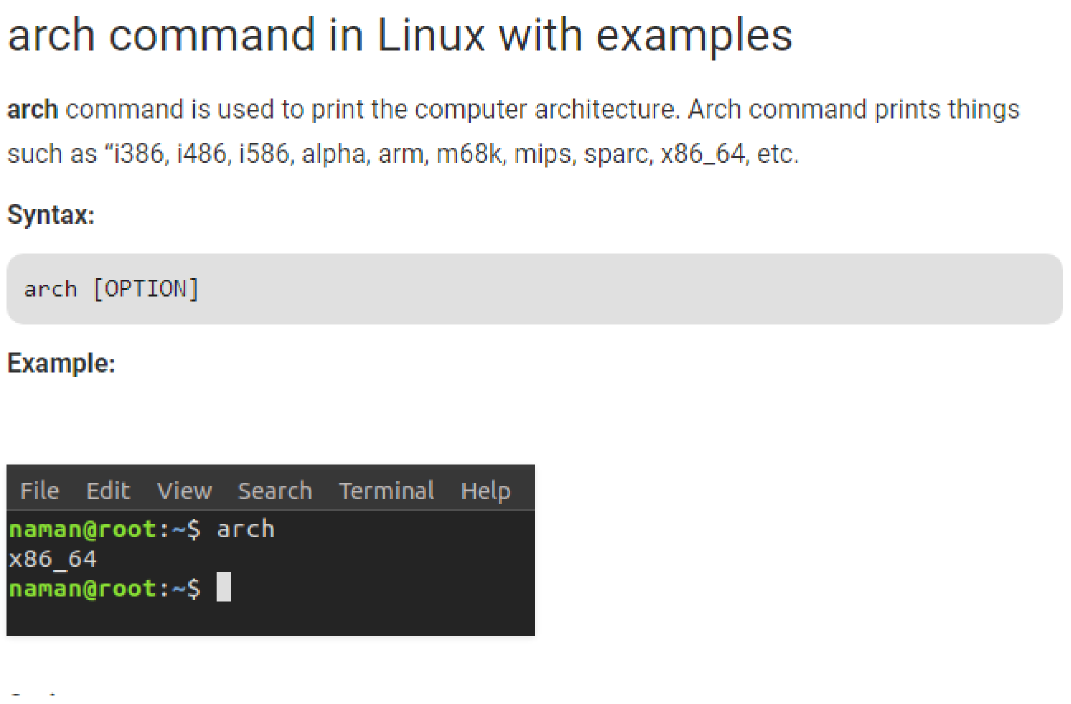
```
getconf -a

    getconf [-v specification] system_var

    getconf [-v specification] path_var pathname
```



Command Line Options:

-a	Displays all configuration variables for the current systemand their values. `getconf -a ...` Copy
-v	Indicate the specification and version for which to obtainconfiguration variables.system_varA system configuration variable, as defined by sysconf(3) orconfstr(3).path_varA system configuration variable as defined by pathconf(3). Thismust be used with a pathname.AUTHORgetconf was written by Roland McGrath for the GNU C LibraryThis man page was written by Ben Collins <bcollins@debian.org> for the Debian GNU/Linux system. `getconf -v ...` Copy

❖ arch command



arch command in Linux with examples

arch command is used to print the computer architecture. Arch command prints things such as "i386, i486, i586, alpha, arm, m68k, mips, sparc, x86_64, etc.

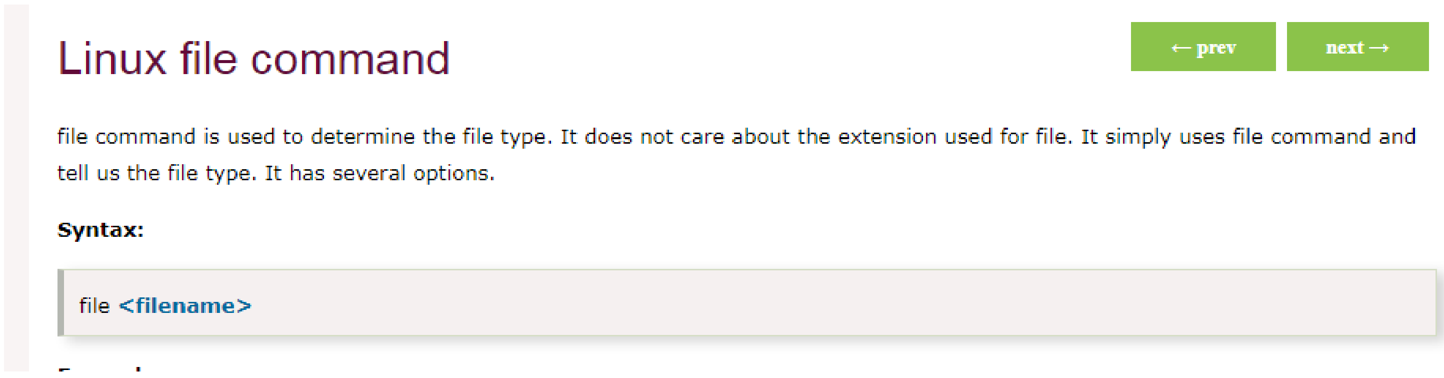
Syntax:

```
arch [OPTION]
```

Example:

```
naman@root:~$ arch
x86_64
naman@root:~$ 
```

❖ file command



Linux file command

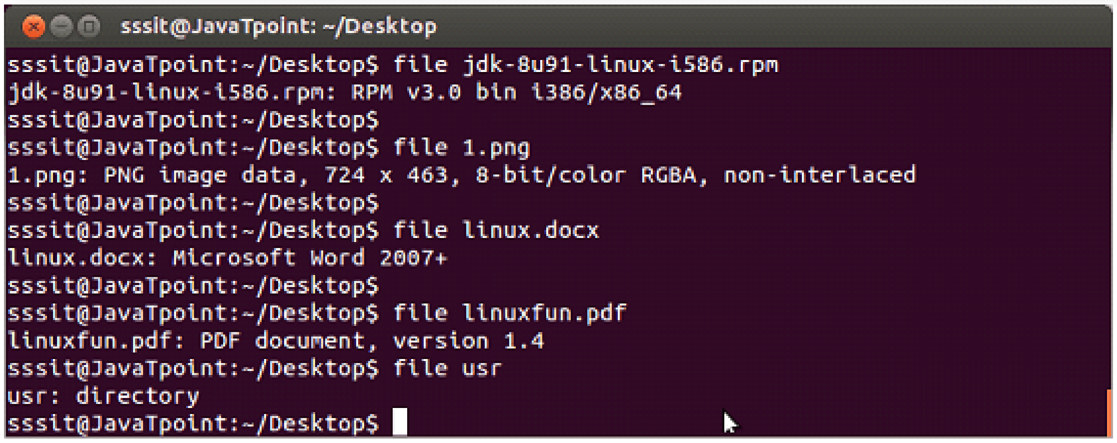
file command is used to determine the file type. It does not care about the extension used for file. It simply uses file command and tell us the file type. It has several options.

Syntax:

```
file <filename>
```

Example:

```
file 1.png
```

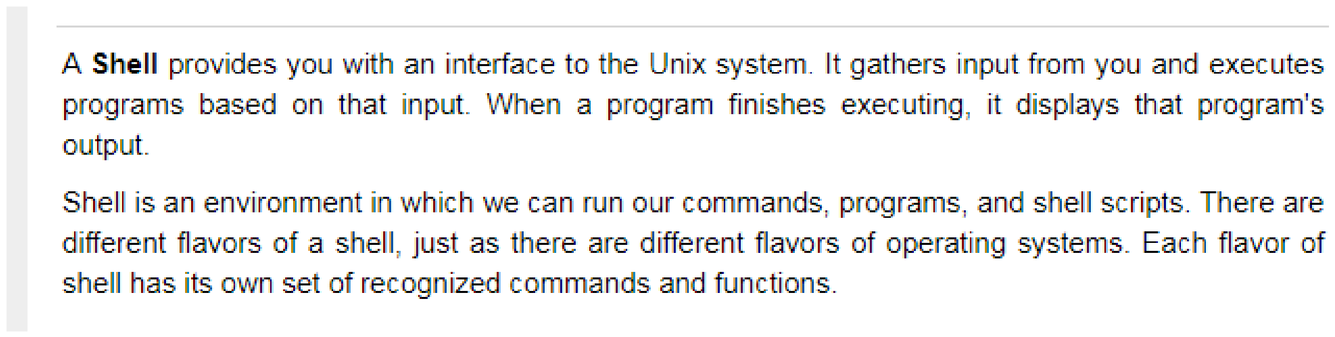


```
sssit@JavaTpoint:~/Desktop$ file jdk-8u91-linux-i586.rpm
jdk-8u91-linux-i586.rpm: RPM v3.0 bin i386/x86_64
sssit@JavaTpoint:~/Desktop$
sssit@JavaTpoint:~/Desktop$ file 1.png
1.png: PNG image data, 724 x 463, 8-bit/color RGBA, non-interlaced
sssit@JavaTpoint:~/Desktop$
sssit@JavaTpoint:~/Desktop$ file linux.docx
linux.docx: Microsoft Word 2007+
sssit@JavaTpoint:~/Desktop$
sssit@JavaTpoint:~/Desktop$ file linuxfun.pdf
linuxfun.pdf: PDF document, version 1.4
sssit@JavaTpoint:~/Desktop$ file usr
usr: directory
sssit@JavaTpoint:~/Desktop$
```

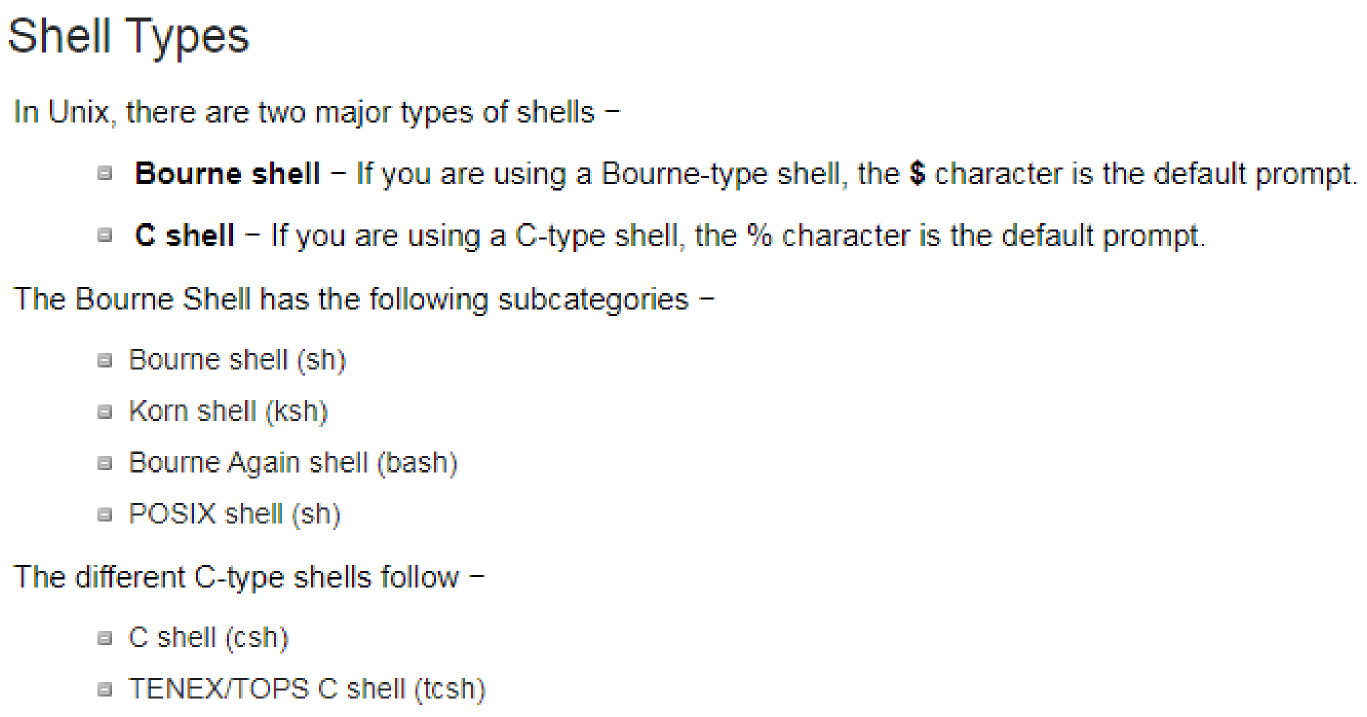
Practical 4

- Aim: Study of UNIX shell and environment variables.

❖ <u>Shell</u>



> A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.
>
> Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.



Shell Types

In Unix, there are two major types of shells −

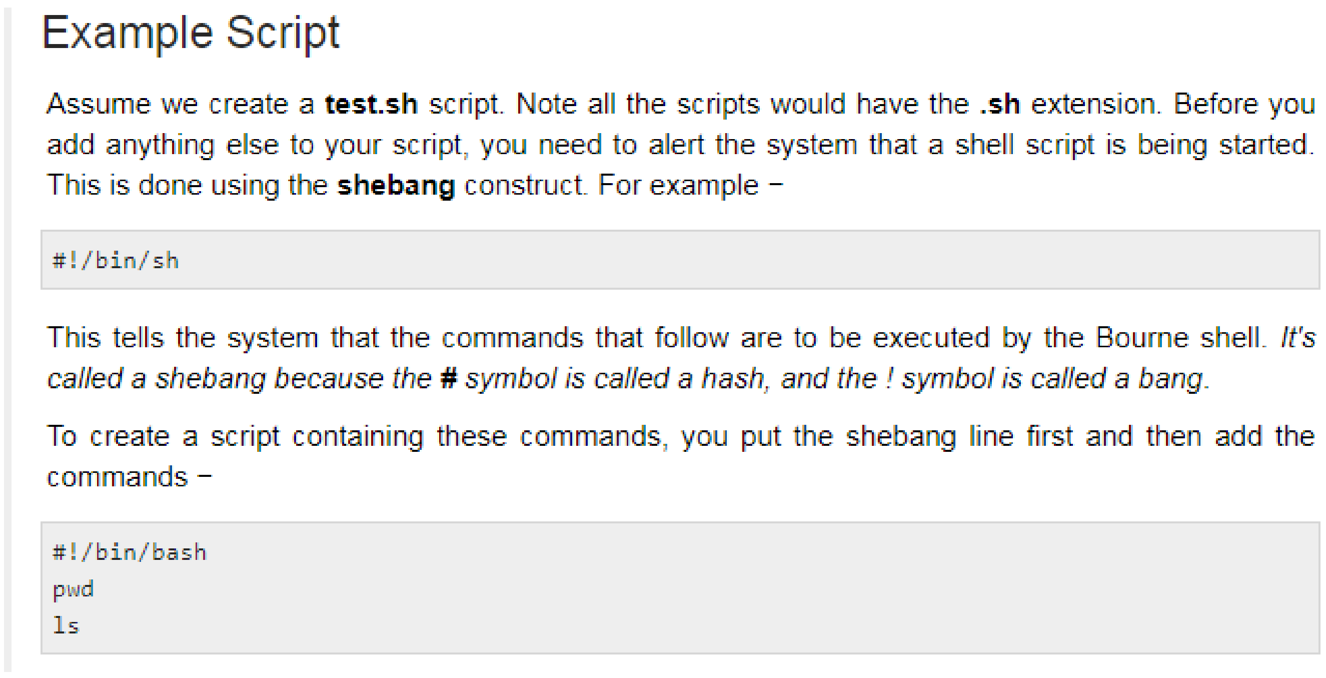
- **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow −

- C shell (csh)
- TENEX/TOPS C shell (tcsh)



Example Script

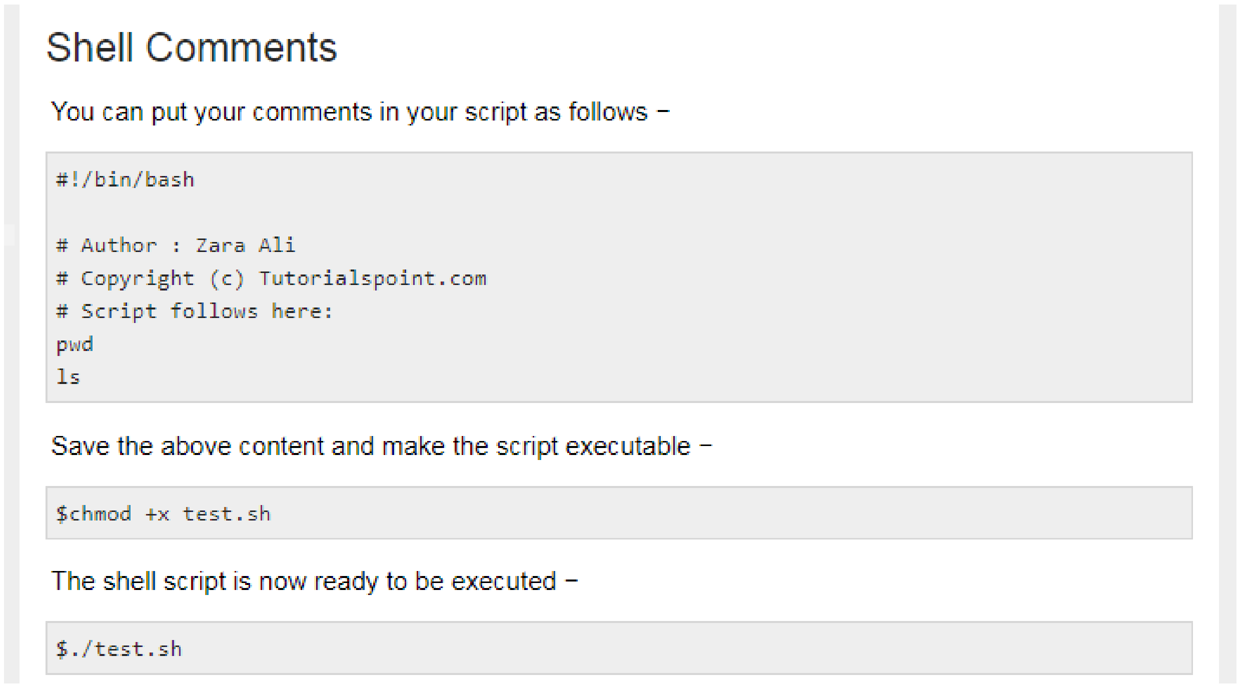
Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct. For example −

```
#!/bin/sh
```

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the **#** symbol is called a hash, and the ! symbol is called a bang.*

To create a script containing these commands, you put the shebang line first and then add the commands −

```
#!/bin/bash
pwd
ls
```



Shell Comments

You can put your comments in your script as follows −

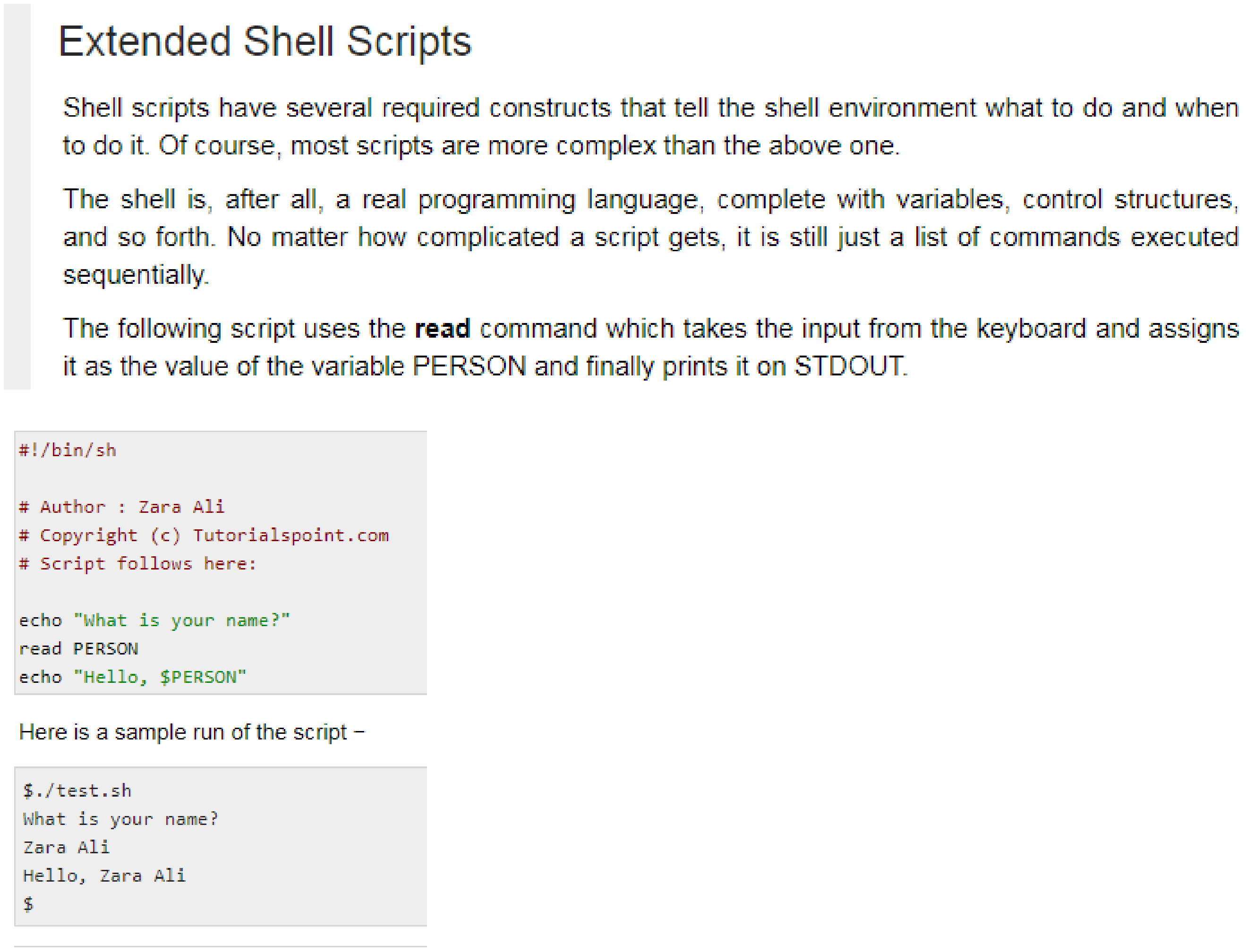
```
#!/bin/bash

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:
pwd
ls
```

Save the above content and make the script executable −

```
$chmod +x test.sh
```

The shell script is now ready to be executed −

```
$./test.sh
```



Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

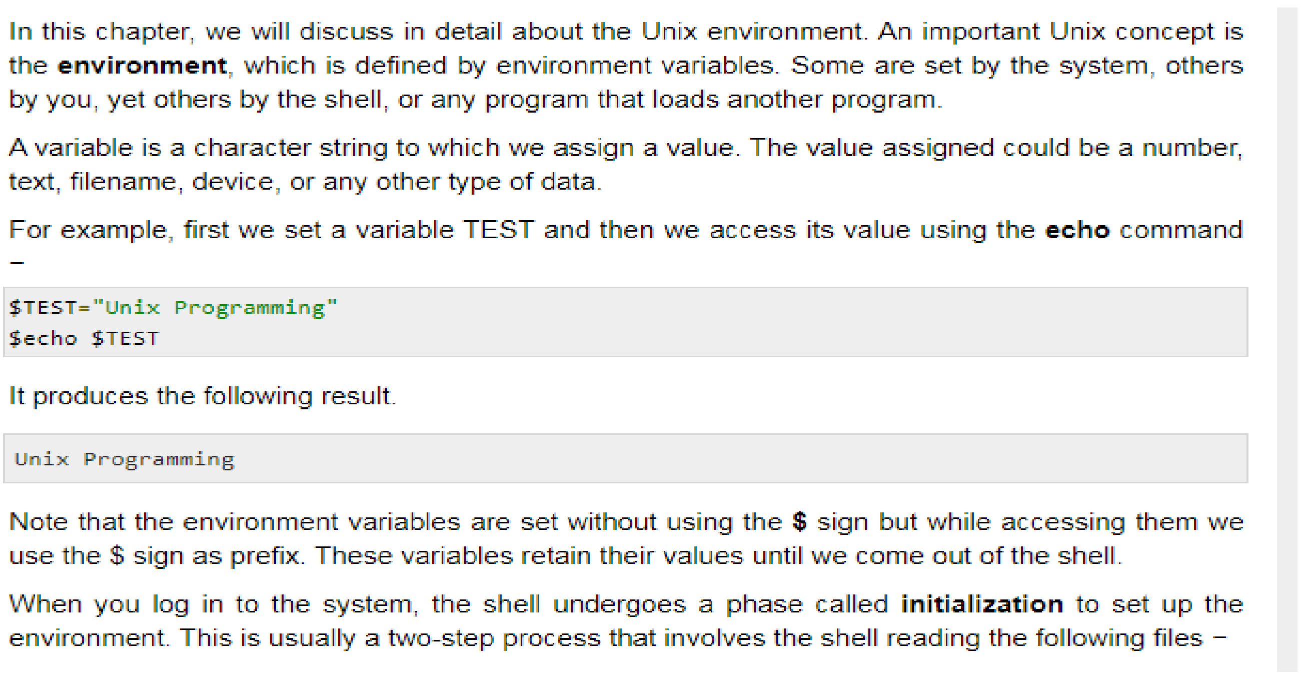
The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is a sample run of the script −

```
$./test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

❖ **Environmental variable**



In this chapter, we will discuss in detail about the Unix environment. An important Unix concept is the **environment**, which is defined by environment variables. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

For example, first we set a variable TEST and then we access its value using the **echo** command −

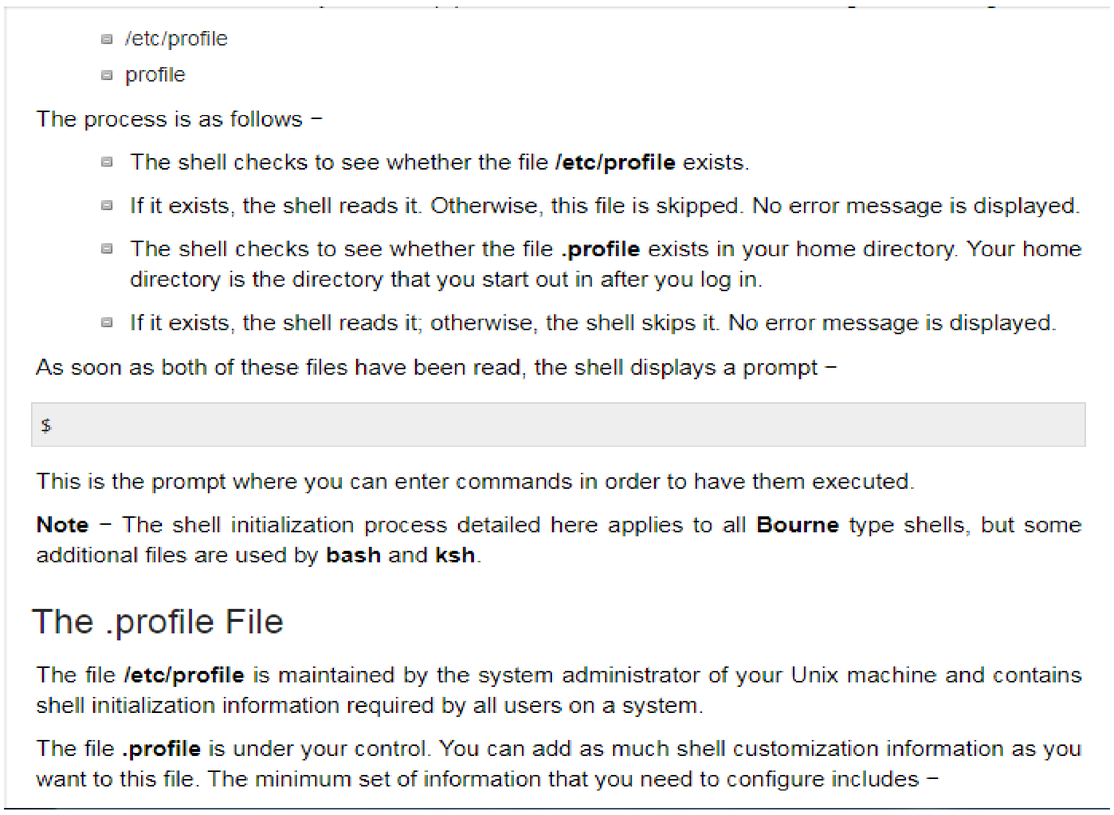
```
$TEST="Unix Programming"
$echo $TEST
```

It produces the following result.

```
Unix Programming
```

Note that the environment variables are set without using the **$** sign but while accessing them we use the $ sign as prefix. These variables retain their values until we come out of the shell.

When you log in to the system, the shell undergoes a phase called **initialization** to set up the environment. This is usually a two-step process that involves the shell reading the following files −



- /etc/profile
- profile

The process is as follows −

- The shell checks to see whether the file **/etc/profile** exists.
- If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.
- The shell checks to see whether the file **.profile** exists in your home directory. Your home directory is the directory that you start out in after you log in.
- If it exists, the shell reads it; otherwise, the shell skips it. No error message is displayed.

As soon as both of these files have been read, the shell displays a prompt −

```
$
```

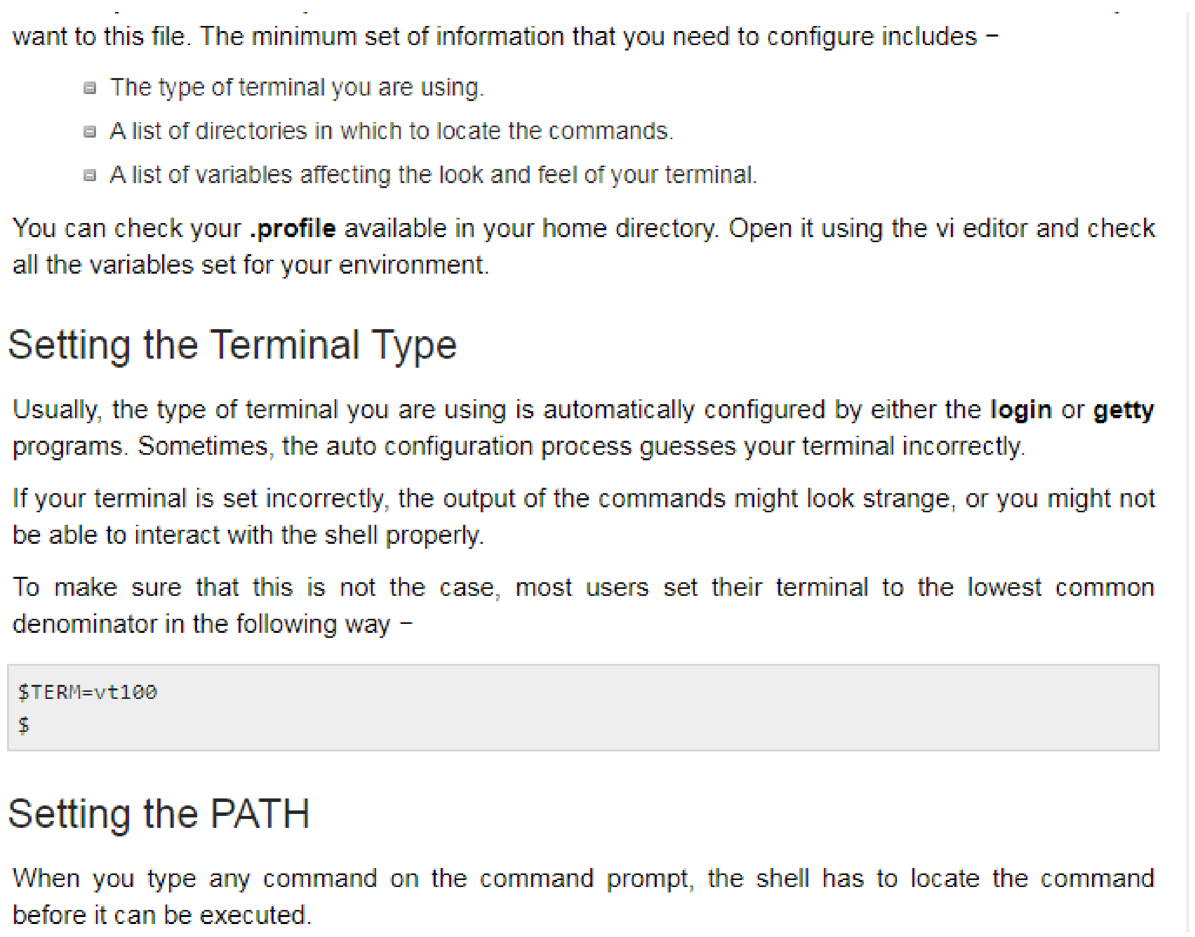
This is the prompt where you can enter commands in order to have them executed.

Note − The shell initialization process detailed here applies to all **Bourne** type shells, but some additional files are used by **bash** and **ksh**.

The .profile File

The file **/etc/profile** is maintained by the system administrator of your Unix machine and contains shell initialization information required by all users on a system.

The file **.profile** is under your control. You can add as much shell customization information as you want to this file. The minimum set of information that you need to configure includes −



want to this file. The minimum set of information that you need to configure includes −

- The type of terminal you are using.
- A list of directories in which to locate the commands.
- A list of variables affecting the look and feel of your terminal.

You can check your **.profile** available in your home directory. Open it using the vi editor and check all the variables set for your environment.

Setting the Terminal Type

Usually, the type of terminal you are using is automatically configured by either the **login** or **getty** programs. Sometimes, the auto configuration process guesses your terminal incorrectly.

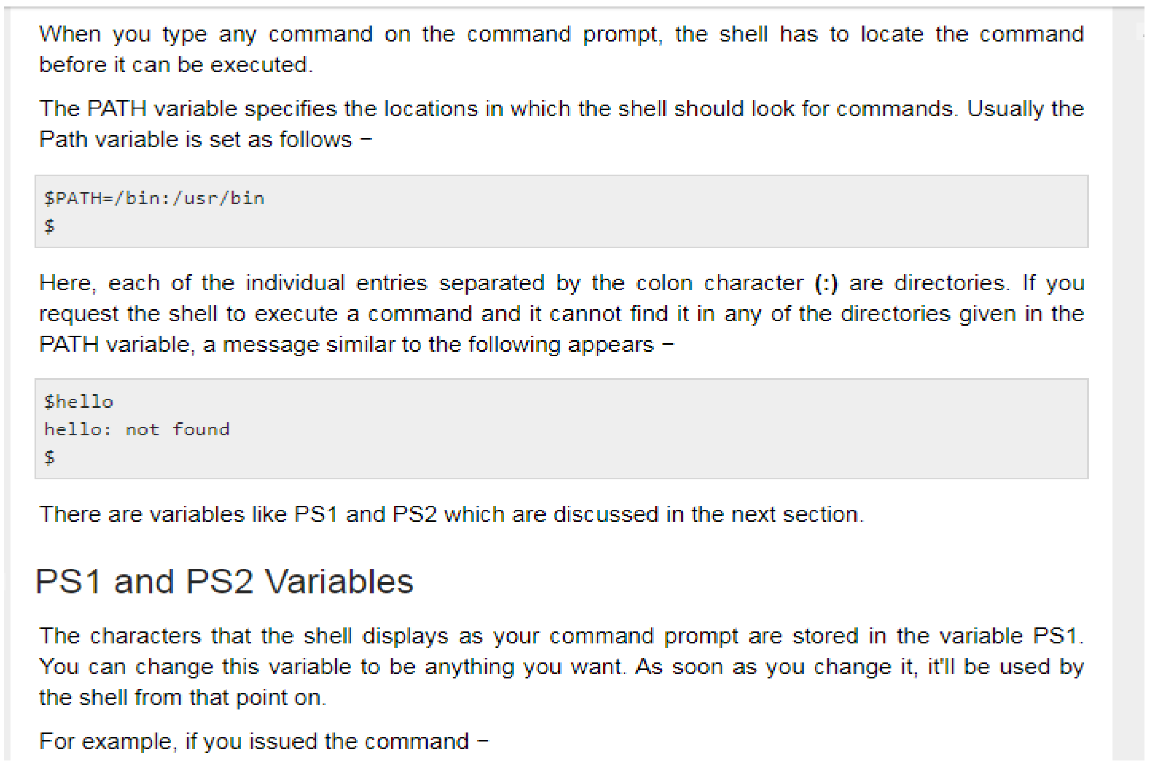
If your terminal is set incorrectly, the output of the commands might look strange, or you might not be able to interact with the shell properly.

To make sure that this is not the case, most users set their terminal to the lowest common denominator in the following way −

```
$TERM=vt100
$
```

Setting the PATH

When you type any command on the command prompt, the shell has to locate the command before it can be executed.



When you type any command on the command prompt, the shell has to locate the command before it can be executed.

The PATH variable specifies the locations in which the shell should look for commands. Usually the Path variable is set as follows −

```
$PATH=/bin:/usr/bin
$
```

Here, each of the individual entries separated by the colon character **(:)** are directories. If you request the shell to execute a command and it cannot find it in any of the directories given in the PATH variable, a message similar to the following appears −

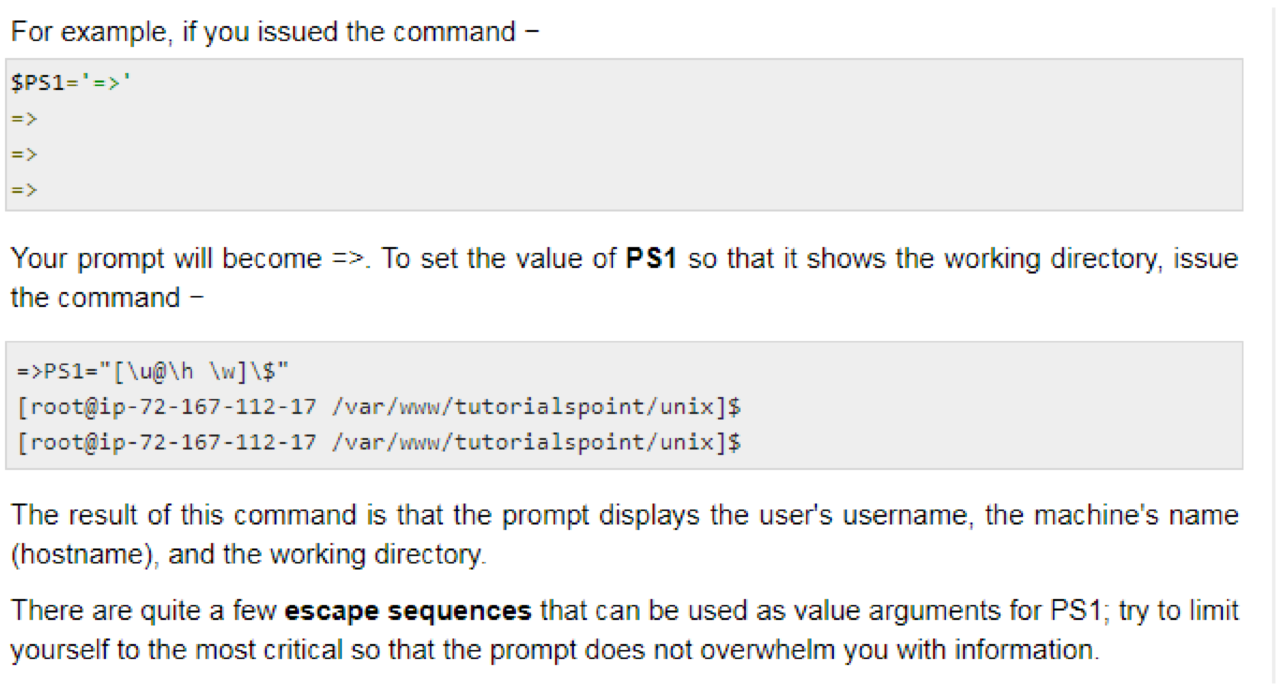
```
$hello
hello: not found
$
```

There are variables like PS1 and PS2 which are discussed in the next section.

PS1 and PS2 Variables

The characters that the shell displays as your command prompt are stored in the variable PS1. You can change this variable to be anything you want. As soon as you change it, it'll be used by the shell from that point on.

For example, if you issued the command −



For example, if you issued the command −

```
$PS1='=>'
=>
=>
=>
```

Your prompt will become =>. To set the value of **PS1** so that it shows the working directory, issue the command −

```
=>PS1="[\u@\h \w]\$"
[root@ip-72-167-112-17 /var/www/tutorialspoint/unix]$
[root@ip-72-167-112-17 /var/www/tutorialspoint/unix]$
```

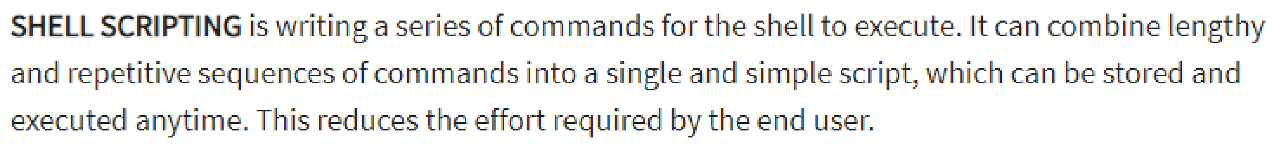
The result of this command is that the prompt displays the user's username, the machine's name (hostname), and the working directory.

There are quite a few **escape sequences** that can be used as value arguments for PS1; try to limit yourself to the most critical so that the prompt does not overwhelm you with information.

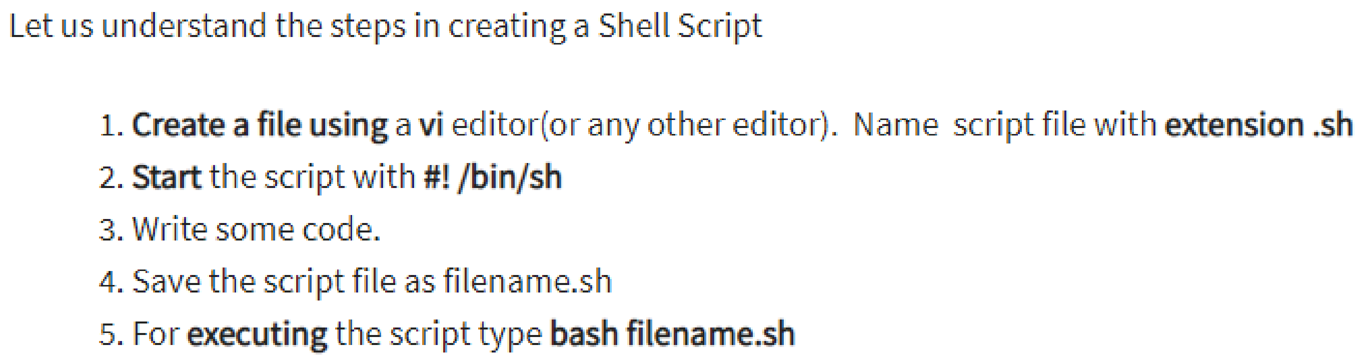
Practical 5

- Aim: What is shell script programming? Write down various steps for run programming in shell script.

❖ **Introduction to shell script steps of shell scripting:**

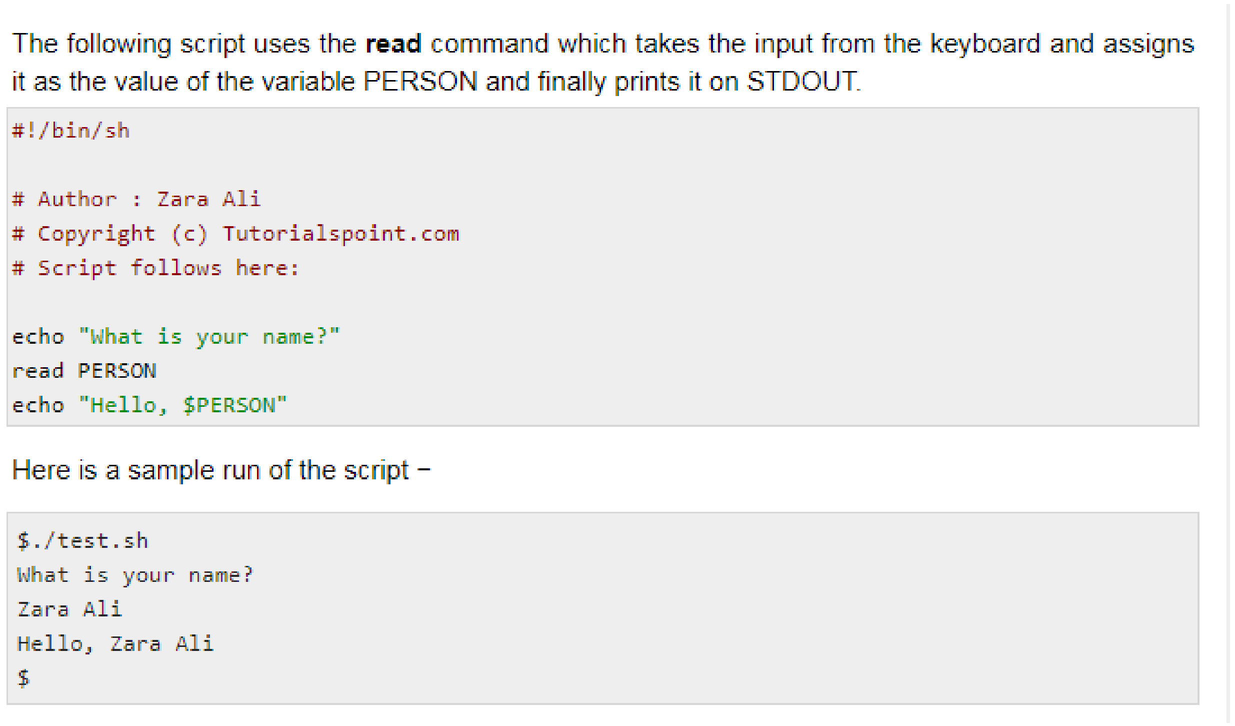


SHELL SCRIPTING is writing a series of commands for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script, which can be stored and executed anytime. This reduces the effort required by the end user.



Let us understand the steps in creating a Shell Script

1. **Create a file using** a **vi** editor(or any other editor). Name script file with **extension .sh**
2. **Start** the script with **#! /bin/sh**
3. Write some code.
4. Save the script file as filename.sh
5. For **executing** the script type **bash filename.sh**



The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is a sample run of the script −

```
$./test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

Practical 6

- Aim: Write a shell script to find factorial of given number n.

Step 1: Create an empty ".c" document.

Input:

```
        #include<stdio.h>
        int main()
{
                int c;
                int n;
                int f=1;
                printf("Enter a number to calculate its factorial\n");
                scanf("%d",&n);
        for(c=1; c<=n; c++)
        f = f*c;
        printf("Factorial of %d = %d\n",n,f);
        return 0;
        getch();
}
```

Step 2: save the document and open terminal.

Step 3: Then compile and run the program, by following steps:

- a: $ cd desktop
- b: $ cd Pooja
- c: $ gedit 1.c //open 1.c document
- d: $ gcc 1.c – 0 1 // compiling
- e: $ ./1   //to run the program

<u>Output:</u>

```
Enter a number to calculate its factorial.

5

Factorial of 5 = 120
```

Practical 7

- Aim: Write a shell script which will generate first n fibonnaci numbers like 1,1,2,3,5,8,13….

Step 1: Create an empty ".c" document.

Input:

```
#include<stdio.h>
int main()
    {
        int n1=0;
        int n2=1;

        int n3;
        int i;
        int number;
    printf("Enter a number to calculate its factorial\n");
    scanf("\n%d %d",&number);
    printf("n%d %d",n1,n2);    //printing 0 and 1

    for(i=2; i<number; ++i)    // loop starts from 2 because
    0 and 1 are already printed.
```

```
        {
                n3 = n1 + n2;
                 printf(“%d”,n3);
                 n1 = n2;
                 n2=n3;
                }
        return 0;


    }
```

Step 2: save the document and open terminal.

Step 3: Then compile and run the program, by following steps:

- a: $ cd Desktop
- b: $ cd Pooja
- c: $ gedit 2.c //open 2.c document
- d: $ gcc 2.c – 0 2 // compiling
- e: $ ./2   //to run the program

<u>Output:</u>

```
Enter a number of elements: 6

0 1 1 2 3 5

Enter a number of elements: 16

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Practical 8

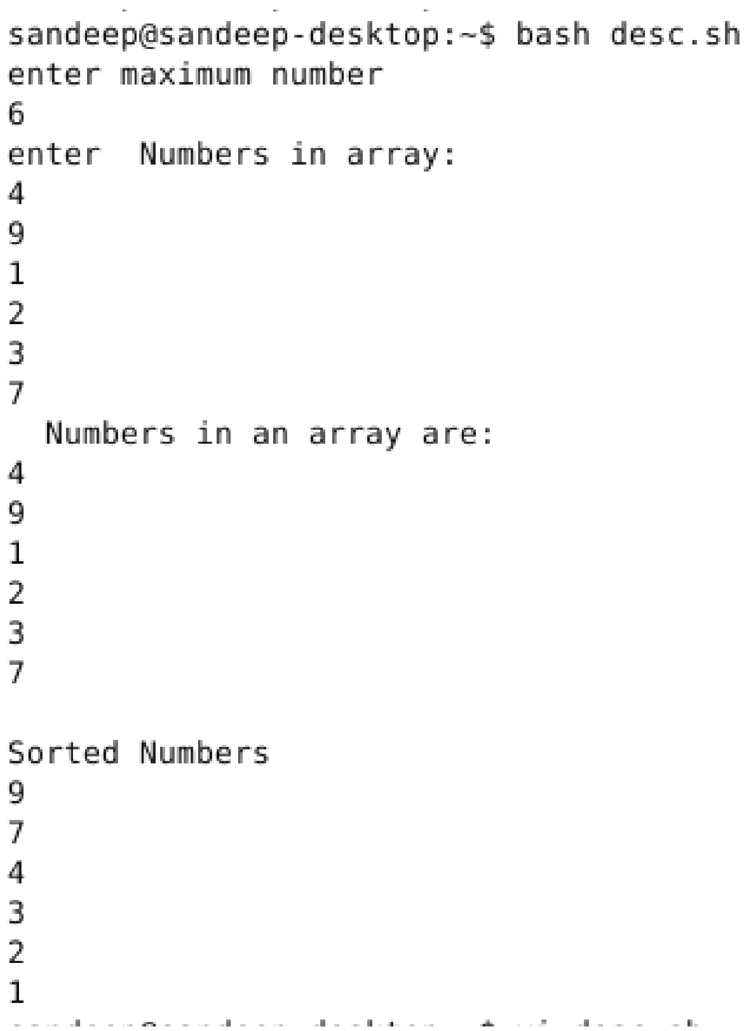
- Aim: Write a shell script to read n numbers as command arguments and sort them in descending order.

Input:

```
#!/bin/bash

echo “enter maximum number”

read n

#taking input from user

echo “enter Numbers in array:”

for((i=0; i<$n; i++))

do

read nos[$i]

done

#Printing the number before sorting

echo “Numbers in array are:”

for((i=0; i<$n; i++))

do

echo${nos[$i]}
```

```
done
#Now do the sorting of numbers
for((i=0; i<$n; i++))
do
for((j=0; i<$n; i++))
do
if [${nos[$i]}-lt${nos[$j]}];
then
t=${nos[$i]}
nos[$i]=${nos[$j]}
nos[$j]=$t
fi
done
done
#Printing the sorted number in descending order
Echo –e "\nSorted Numbers"
for((i=0; i<$n; i++))
do
echo${nos[$i]}
done
```

Output:



```
sandeep@sandeep-desktop:~$ bash desc.sh
enter maximum number
6
enter  Numbers in array:
4
9
1
2
3
7
  Numbers in an array are:
4
9
1
2
3
7

Sorted Numbers
9
7
4
3
2
1
```

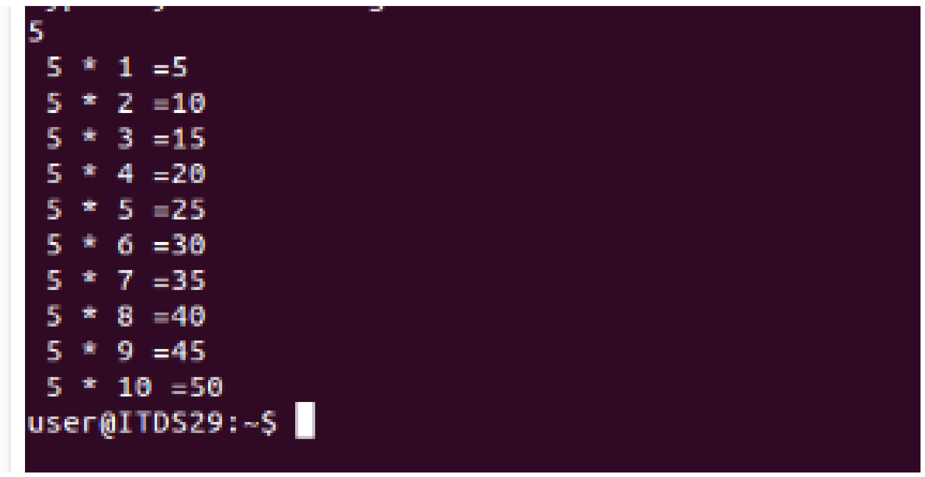
Practical 9

- Aim: Write a shell script to display multiplication table of given number.

Input:

```
echo “Enter a Number”
read n
i=0
while[$i –le 10]
do
echo “ $n x Si = ‘expr $n\*$i’ ”
i = ‘expr $i + 1’
```

<u>Output:</u>



```
5
 5 * 1 =5
 5 * 2 =10
 5 * 3 =15
 5 * 4 =20
 5 * 5 =25
 5 * 6 =30
 5 * 7 =35
 5 * 8 =40
 5 * 9 =45
 5 * 10 =50
user@ITDS29:~$
```

Practical 10

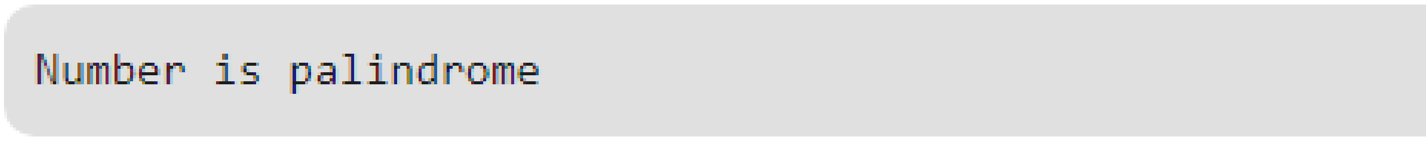
- Aim: Write a shell script to check entered string is palindrome or not.

Input:

```
Num =545
#Storing the reminder
s=0
#Store number in reverse
#order
 rev = “”
#Store original number
#in another variable
temp=$num
while[$num –gt 0]
do
#Get Reminder
 s=$(($num%10))
#Get next digit
```

```
num=$(($num/10))
#Store previous number and
#current digit in reverse
rev =$(echo${rev}${S})
done
if[$temp –eq$rev];
then
echo “Number is palindrome”
else
echo “Number is not palindrome”
fi
```

<u>Output:</u>



```
Number is palindrome
```