UNIT 9: LINUX SHELLS

UNIT STRUCTURE

9.1 Learning Objectives

9.2 Introduction

9.3 Linux Shells

9.4 Shell Scripting

- 9.4.1 Shell Scripts
- 9.4.2 Shell Variables
- 9.4.3 Array in Shell Script
- 9.4.4 Operators used in Shell
- 9.4.5 Conditional and Loop Control Statements in Shell
- 9.4.6 Loop in Shell
- 9.4.7 Linux Shell Substitution
- 9.4.8 Input/Output Redirections in Shell
- 9.4.9 Functions in Shell

9.5 Let Us Sum Up

9.6 Further Readings

9.7 Answer to Check Your Progress

9.8 Model Questions

9.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- understand about shell
- learn different shell in Linux
- understand shell script writing
- describe array in shell script
- learn to use conditional statement in shell script
- learn to use function in shell script

9.2 INTRODUCTION

A shell can be named as a special user program which provides an

interface to facilitate the users to use different operating system services. Shell accepts commands from user and converts them into kernel understandable format. This is why shell can also be termed as a command interpreter in Linux operating system. Shell is used to run other programs and commands. The shell enables the users with an interface to run the programs on Unix/GNU Linux environment. By using shell, the users can execute different tools/utilities or commands with a few input data. After executing the programs/codes, the shell sends the result to the common output device. This is why it is known as "command interpreter".

In addition to act as a command interpreter, the shell is also called a programming language with all the features of the programming language such as functions, variables, loops, conditional statements etc. Layered structure of Linux operating system is presented in figure 1 which includes Kernel, Shell, and Utilities along with Hardware in Innermost layer.

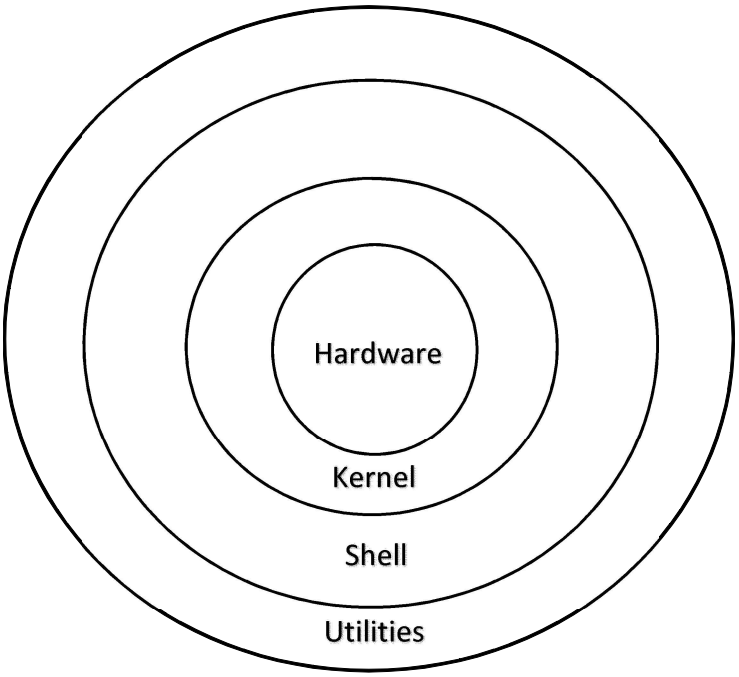


Figure 1. Different layers in Linux

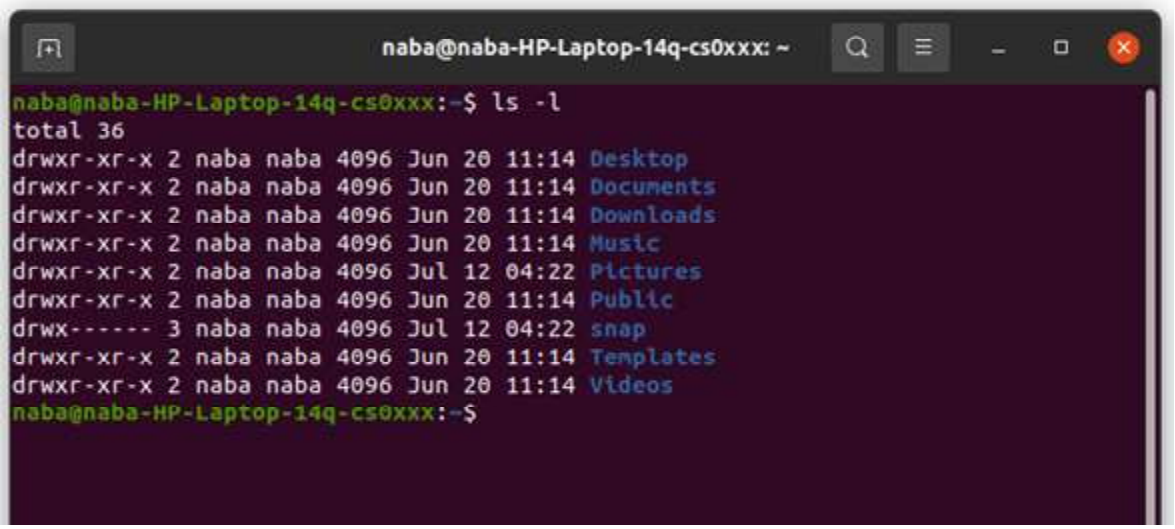
The shell can be categorized into two main classes depending on the user interface:

- Command-line Shell
- Graphical Shell

This categorization basically specifies the manipulation of programs, whether they are based on the graphical user interface (GUI) by permitting for operations like moving, closing, resizing, and opening windows etc. or command line where user has to input command for every task performed. Ubuntu OS or Windows OS could be examined as a good example that offers a graphical user interface to the user to interact with the program. Various users don't need for typing in any command for all the actions.

Command-line Shell

Shell can be accessed by user using a command line interface. Terminal in Linux / macOS and Command Prompt in Windows OS seems to be similar, providing an interface to the users to type Linux commands such as "cat", "ls" etc. which are to be executed by the kernel. The result is then displayed on the terminal for the user. An example of the Linux command "ls - l" in ubuntu version is given below.



"ls - l" list all the files in current working directory in long listing format.

Working in command line shell is difficult for the beginners because one has to memorize all the commands before using command line shell. Command line shell is very powerful, it also allows user to write commands in a file and execute them together. This way any repetitive task can be easily automated. These files are usually called Shell Scripts in Linux/mac OS.

Graphical Shells

Graphical shells provide users a graphical interface by which different operations such as opening, closing, moving and resizing windows etc. are performed. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User is not required to type in command for every actions. A typical GUI in Ubuntu system is shown in figure 2.



Figure 2. Typical GUI in Ubuntu system



EXERCISE

Q 1: Run different Linux commands in the shell and check the output.

9.3 LINUX SHELLS

Shells are one of the most powerful tools available for Linux user. Without shells, it is practically impossible for a user to utilise all the features and functionality offered by the kernel in a Linux system. Different types of Linux shells are discussed below. Each of these shells has some properties that make them highly efficient for a specific type of use over other shells and thereby it makes them different from each other.

- **The Bourne Shell (sh):** Bourne Shell is developed at AT&T Bell Labs by Steve Bourne. This shell is known as the first UNIX shell ever. It is

denoted as sh. It gained popularity due to its compact nature and high speeds of operation. Bourne Shell is used as the default shell for all Solaris system administration scripts.

Some major drawbacks in Bourne shell are observed. It doesn't have in-built functionality to handle logical and arithmetic operations. Also, unlike most different types of shells in Linux, it cannot recall previously used commands. It also does not have comprehensive features to offer a proper interactive use.

The complete path-name for the Bourne shell is /bin/sh and /sbin/sh. By default, it displays the command prompt # for the root user and $ for the non-root users.

- **The GNU Bourne-Again Shell (bash):** GNU Bourne-Again Shell is popularly known as the Bash shell. This bash shell was designed to be compatible with the Bourne shell. It includes useful features from different types of shells in Linux such as Korn shell and C shell. It allows us to automatically recall previously used commands and edit them with the help of arrow keys, unlike the Bourne shell. The complete path-name for the bash shell is /bin/bash. By default, it uses the prompt bash-VersionNumber# for the root user and bash-VersionNumber$ for the non-root users. Bash shell has following functionalities:
  - Job Control
  - Command-line editing
  - Shell Aliases and Functions
  - Unlimited size command history
  - Integer arithmetic in a base from 2-64
- **The C Shell (csh)** : The C shell was introduced by Bill Joy at the University of California. It is denoted as csh, also known as Tcsh. It was developed to include useful programming features like in-built support for arithmetic operations and the syntax similar to the C programming language. Further, it integrates command history which was missing in different types of shells in Linux like the Bourne shell. Another prominent feature of C shell is "aliases". The complete path-name for the C shell is /bin/csh. By default, it uses the prompt hostname# for the root user

and hostname% for the non-root users. Csh shell contains the following characteristics:

- C like syntax
- Filename completion and programmable word
- Command-line editor
- Job control
- Spelling correction

- **The Korn Shell (ksh):** The Korn shell was developed at AT&T Bell Labs by David Korn, to extend the features which are not available in the Bourne shell. It is denoted as ksh. The Korn shell is essentially a superset of the Bourne shell. Besides the functionalities of the Bourne shell, Korn shell supports additional functionalities to the user. It allows in-built support for arithmetic operations while offering interactive features which are similar to the C shell. The Korn shell runs scripts made for the Bourne shell, while offering string, array and function manipulation similar to the C programming language. It also supports scripts which were written for the C shell. Further, it is faster than other shells in Linux, including the C shell. The complete path-name for the Korn shell is /bin/ksh. By default, it uses the prompt # for the root user and $ for the non-root users.
- **The Z Shell (zsh):** The Z Shell or zsh is an extention of sh shell with various improvements and customization. If someone looking for a modern shell that has all the features and much more, the zsh shell can be considered. Some notable features of the z shell are:
  - It can generate filenames based on given conditions.
  - It has plugins and theme support.
  - It has index of built-in functions.
  - It has feature like auto completion of command.

A summary of different shells available in Linux is given below in a tabular format.

Shell	**Complete path-name**	**Prompt for root user**	**Prompt for non-root user**
Bourne shell (sh)	/bin/sh and /sbin/sh	#	$
GNU Bourne-Again shell (bash)	/bin/bash	bash-VersionNumber#	bash-VersionNumber$
C shell (csh)	/bin/csh	#	%
Korn shell (ksh)	/bin/ksh	#	$
Z Shell (zsh)	/bin/zsh	<hostname>#	<hostname>%



CHECK YOUR PROGRESS-1

Q 1: Which of the following is true?

a) Shell is a process and can be started by super user only

b) Shell is a built-in Kernel functionality

c) Shell is a wrapper for all the commands and utilities

d) None of the above

Q 2: BASH shell stands for?

Q 3: Shell is a Command Interpreter. (TRUE/ FALSE)

Q 4: Korn Shell is developed at AT & T Bell lab (TRUE/ FALSE)

9.4 SHELL SCRIPTING

Shell provides the interface to the user to work on Unix/Linux system. It acts as an environment in which we can run our commands, programs, and shell scripts.

Shell Prompt: The prompt, $, also called the command prompt, is provided by the shell to user. User can type commands in this prompt. After entering the command, shell reads the input, delivers it to the kernel for execution and finally displays the output.

For example, the date command displays the current date and time?

$ date

Thu Jun 25 08:30:19 MST 2009

We can customize the command prompt using the environment

variable.

9.4.1 Shell Scripts

Basically, a shell script is nothing but a program having a list of commands, which are listed in order. Following are some important points about shell scripts.

- Comments in the shell scripts are written, preceded by # sign.
- Shell functions and scripts are interpreted by the shell, not compiled.
- All the shell scripts have .sh as file extension.
- The shells are usually interactive where users can run their commands.
- The shells have conditional statements and looping statements, files to store and read data and variables etc.
- Syntax of shell script is similarly to other programming languages.

The shell script basically has following components:

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

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.
- Writing shell scripts are much quicker.
- Interactive debugging etc.

Disadvantages of shell scripts

- Prone to 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.

Shell Script Example 1:

Let us create a test.sh script file. First, we need to tell the system that we are going to use shell, 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. This command is called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.

Next, we need to add the commands in the script by using an editor like vi, vim, as follows -

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

In the above shell script #!/bin/sh is used to tell the system that we are going to use bash shell and pwd command prints the present working directory followed by ls command listing the files in the present working directory.

Shell Comments

We can put comments in the script as follows -

```
#!/bin/sh
#Author: Rahul Das
#Script follows here:
pwd
ls -l
```

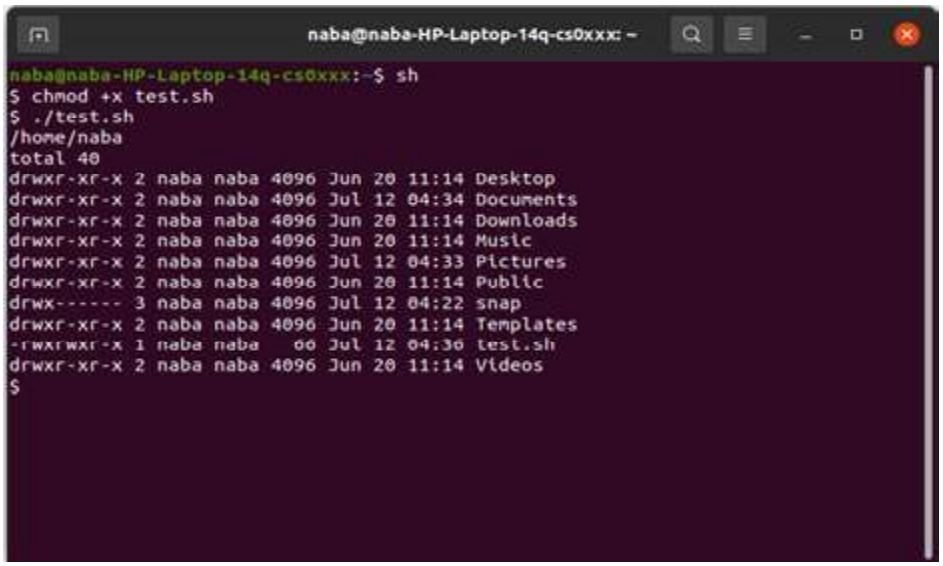
Then, we need to save the shell script and make the script executable by using chmod ?

$ chmod +x test.sh

Finally, the shell script is executed as follows ?

$ ./ test.sh

After execution, we will receive the following result -



Note ? To execute a program in the current directory, we have to run as ./ program_name

Shell Scripts Example 2:

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. The shell script is a real programming language, having with variables, control structures, and so forth. Apart from command execution shell scripts are used for more task to perform. The following script uses the read command which takes the input from the keyboard and assigns it as the value of the variable and finally prints it on the standard output device.

```
#!/bin/sh
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

output of the script ?

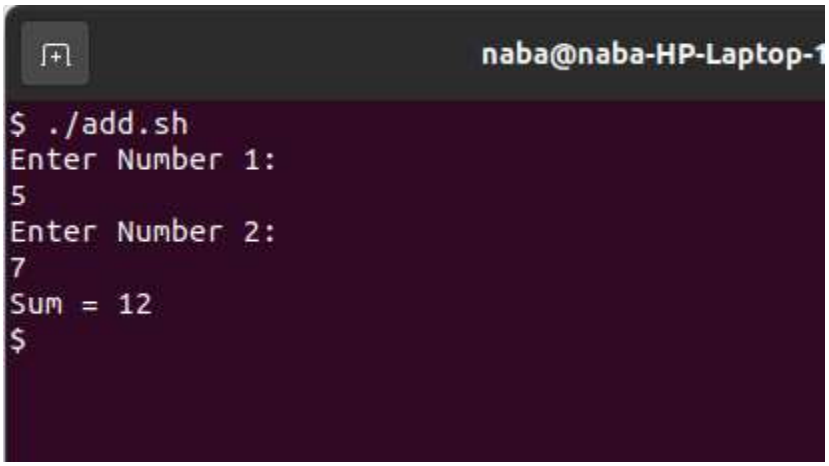


Shell Scripts Example 3:

The following script uses the read command to read two values from the keyboard and print the sum of two numbers on standard output devices.

```
#!/bin/sh
echo "Enter Number 1: "
read A
echo "Enter Number 2: "
read B
sum=$(expr $A + $B)
echo "Sum = $sum"
```

Output of the script is ?





EXERCISE

Q 1: Write a shell script including different Linux commands and check the output.

9.4.2 Shell Variables

A variable is a character string where we can store value. The value assigned in the variable could be number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables us to create, store values to the variable, and delete variables.

Variable Names

Name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_). The variable names must start with letters.

The following examples are valid variable names ?

TOKEN_A

VAR_1

VAR_2

Defining Variables

Variables are defined as follows ?

variable_name = value

For example ?

NAME="Rahul Roy"

The above example defines the variable NAME and assigns the value "Rahul Roy" to it. Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

Accessing Values

To access the value stored in a variable, dollar sign ($) is added as prefix to the variable name.

For example

```
#!/bin/sh
NAME="Rahul Roy"
echo $NAME
```

Output ?

Rahul Roy

Read-only Variables

In shell script we can define a variable as read-only by using the read-only command as follows-

```
#!/bin/sh
NAME="Rahul Roy"
readonly NAME
NAME="Partha Das"
```

The above script will generate error result as bellow ?



Unsetting or Deleting Variables

Unsetting or deleting a variable will remove the variable from the list of variables. Once the variable is deleted, we cannot access the stored value in the variable.

Following is the syntax to unset a variable ?

```
unset variable_name
```

Note- We cannot use the unset command to unset variables that are marked readonly.

Variable Types

Variables may be of three types which are ?

- Local Variables - A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- Environment Variables - An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are required for the programs during runtime.

- 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. Some of these variables are environment variables whereas others are local variables.

Special Variables

In shells some variables are reserved for specific functions, which are known as special or environment variables.

For example, the $ character represents the process ID number, or PID, of the current shell ?

$echo $$

The above command tells the PID of the current shell ?

29949

The table 1 lists the number of special variables that are used in shell scripts.

Table 1. Special variables with their purposes

Sl No	Variable	Description
1	$0	The filename of the current script.
2	$n	The variables correspond to the arguments with which a script was invoked.
3	$#	The number of arguments supplied to the script.
4	$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
5	$@	All the arguments are individually double quoted.
6	$?	Tells the exit status of the last command executed.
7	$$	The process number of the current shell. That is the process ID under which shell is executing.
8	$!	The process number of the last background command.

Command-Line Arguments

We can deliver command-line arguments when we run a shell script. Following example shows the use of command line arguments.

```
#!/bin/sh
Sum=$(expr $1 + $2 + $3)
```

```
Avg=$(expr $Sum / $#)
echo "SUM = $Sum"
echo "Average= $Avg"
echo "Values are: $@"
echo "File Name: $0"
```

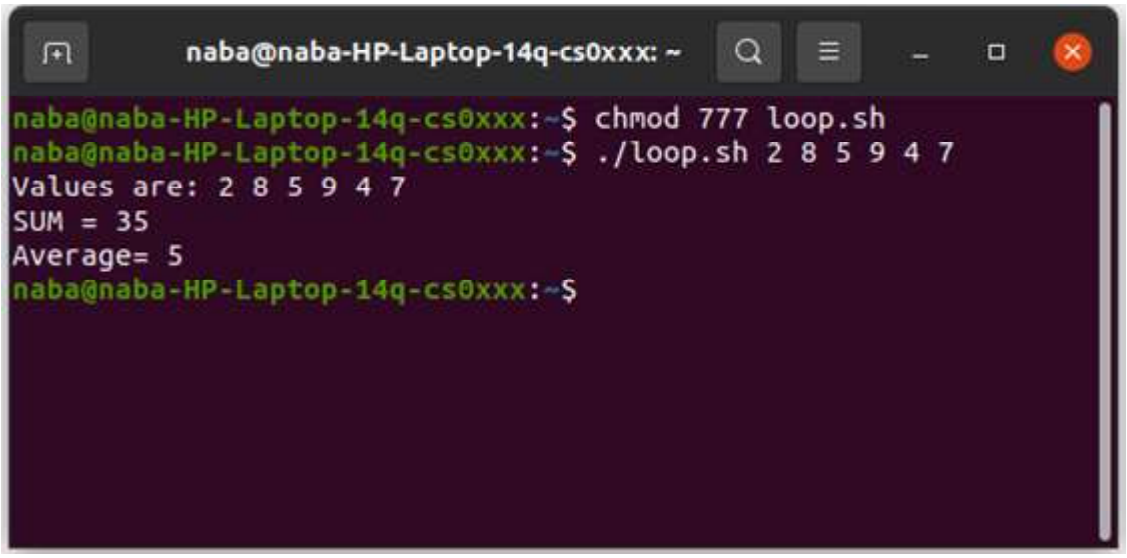
Output of the above script ?



Following example is to find out the average of n numbers using $*

```
#!/bin/sh
SUM=0
for VALUE in $*
do
SUM=$(expr $SUM + $VALUE)
done
AVG=$(expr $SUM / $#)
echo "Values are: $@"
echo "SUM = $SUM"
echo "Average= $AVG"
```

Output of the above script -



Note - In the above example, for loop is used.



EXERCISE

Q 1: Write a shell script to input 3 numbers as the command line arguments and find the maximum value of these numbers.

9.4.3 Array in Shell Script

A shell variable which is capable of holding only single value are called scalar variables. Shell supports a different type of variable called an array, which can hold multiple values (may be of same type or different type) in consecutive memory locations at the same time. Arrays provide a method of grouping a set of variables.

Defining Array Values

The syntax of declaring an array variable is-

array_name=(value1 value2 …. valuen)

Here array_name is the name of the array, index is the index of the item. Index generally starts at 0 (zero) and go on. Example of array?

Arr=(10 20 30 40 50)

In ksh shell, the syntax of array initialization is ?

set -A array_name value1 value2 ... valuen

In bash shell, the syntax of array initialization is ?

array_name=(value1 ... valuen)

Accessing Array Values

After initialization of the array variable, we can access the array as follows ?

${array_name[index]}

Here array_name is the name of the array, and index is the index of the value to be accessed.

We can access all the items in an array as follows ?

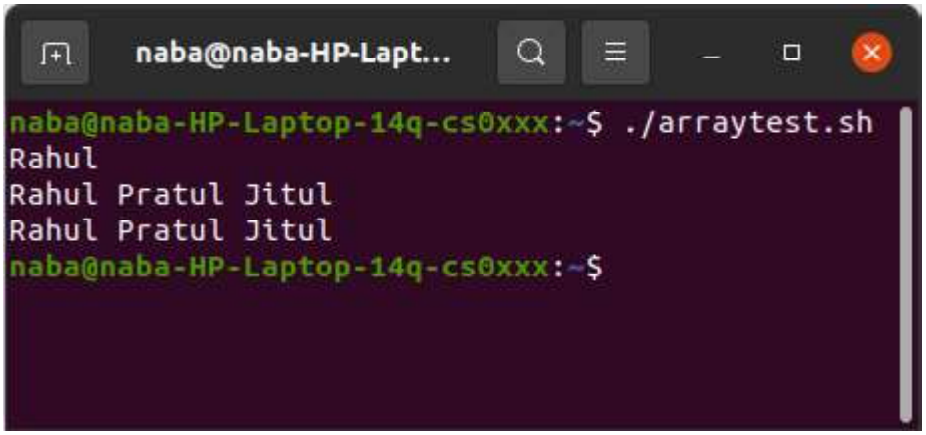
${array_name[*]}

${array_name[@]}

Example

```
#!/bin/sh
arr=(Rahul Pratul Jitul)
echo "${arr[0]}"
echo "${arr[*]}"
echo "${arr[@]}"
```

The above shell script once executed will generate the following result -





EXERCISE

Q 1: Write a shell script to find the average of 10 integers stored in an array.

Q 2: Write a shell script to find the maximum of n numbers in an array.

9.4.4 Operators used in Shell

There are various operators supported by shell. In this unit bash shell is used to discuss the different operators supported by shell. The operators in bash shell are broadly categorized as follows-

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

Arithmetic Operators

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

Note: All the conditional expressions should be inside square braces with spaces around them, for example [$a == $b] is correct whereas, [$a==$b] is incorrect. All the arithmetical calculations are

done using long integers.

Relational Operators

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b]
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b]
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b]
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b]
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b]
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b]

Boolean Operators

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true if a=45 and b=120

-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false if a=45 and b=120

String Operators

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b]
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b]
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a]
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a]
Str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a]

File Test Operators

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file]
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file]
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file]
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file]
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file]
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file]

-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file]
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file]
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file]
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file]
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file]
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file]
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file]
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file]

Note:- Apart from the above operators, different shell supports different set of operators. Some shells like C Shell and K Shell have few more additional operators and few operators are different from bash shell operators.



EXERCISE

Q 1: List different operators used in shell programming. Find the difference between C programming operators and shell programming operators.

9.4.5 Conditional and loop control Statements in Shell

Unix Shell supports conditional statements which are used to perform different tasks based on the condition values. There are mainly two different conditional statements supported by shell ?

- if...else statement
- case...esac statement

if...else statements

If else statements are useful in decision-making where condition like either or is present.

Linux Shell supports following types of if…else statement ?

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

The if statements check the condition by using relational operators and depending on the value of the conditional statement, the if or else block is executed.

- if...fi statement

Syntax of if…fi statement

```
if [ expression ]
then
Statement(s) to be executed if expression is true
fi
```

- if...else…fi statement

Syntax of if…else…fi statement

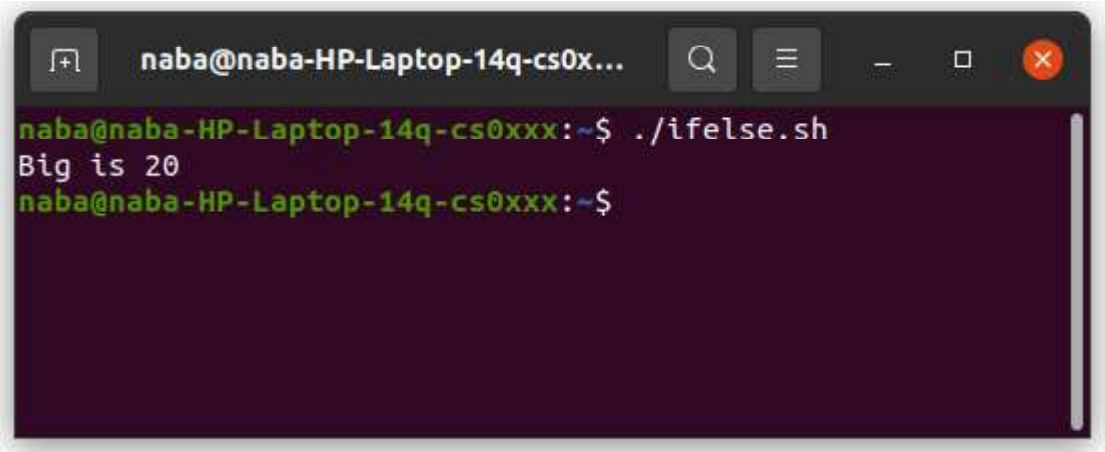
```
if [ expression ]
then
Statement(s) to be executed if expression is true
else
Statement(s) to be executed if the expression is false
fi
```

Example

```
#!/bin/sh
a=10
b=20
if [ $a -gt $b ]
then
  echo "Big is $a"
else
    echo "Big is $b"
```

fi

OUTPUT



- if...elif…else…fi statement

```
Syntax of if…else…fi statement
if [ expression 1 ]
then
Statement(s) to be executed if expression 1 is
   true
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is
   true
   elif [ expression 3 ]
   then
      Statement(s) to be executed if expression 3
   is true
   else
      Statement(s) to be executed if no expression
   is true
fi
```

Example

```
#!/bin/sh
a=$1
b=$2
```

```
c=$3
if [ $a -gt $b ]
then
if [ $a -gt $c ]
then
   echo "Big is $a"
fi
elif [ $b -gt $c ]
then
   echo "Big is $b"
else
   echo "Big is $c"
fi
```

OUTPUT



The case...esac Statement

We can use multiple if...elif statements to perform a multiway branch. However, it is better to use case...esac statement which handles exactly the same situation, and it does so more efficiently than repeated if...elif statements.

The case...esac statement in the Linux shell is very similar to the switch...case statement in C/C++ programming language.



CHECK YOUR PROGRESS-2

Q 5: The shell script is a file containing a series of commands. (TRUE/ FALSE)

Q 6: To read data from user, which command is used in Linux?

Q 7: The command line arguments $0 represents_____.

Q 8: C=A+B is written in shell script as_____.

Q 9: A>B is written in shell script as_____.

Q 10: All the elements of an array can be displayed by _____.

Q 11: != is a legal operator in shell script. (TRUE / FALSE)

Q 12: >= is a legal operator in shell script. (TRUE / FALSE)

9.4.6 Loop in Shell

A loop is a powerful programming construct which enables us to execute a set of commands repeatedly. Following types of loops available in shell ?

- The while loop
- The for loop
- The until loop
- The select loop
- The while loop

Syntax

while command

do

Statement(s) to be executed if command is true

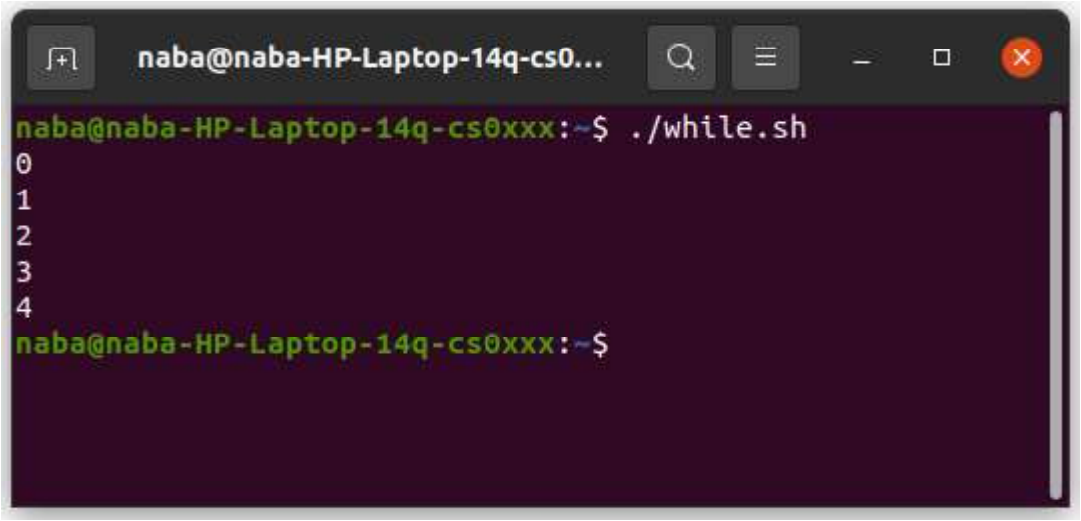
done

Example

```
#!/bin/sh
a=0
while [ $a -lt 5 ]
do
   echo $a
```

```
    a=`expr $a + 1`
done
```

OUTPUT



- The for loop

Syntax

for var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

Example

```
#!/bin/sh
for var in 7 8 9 6 3
do
   echo $var
done
```

OUTPUT



- The until loop

Syntax

until condition

do

Statement(s) to be executed until command is true

done

- The select loop

Syntax

select var in word1 word2 ... wordN

do

Statement(s) to be executed for every word.

done

Nesting Loops

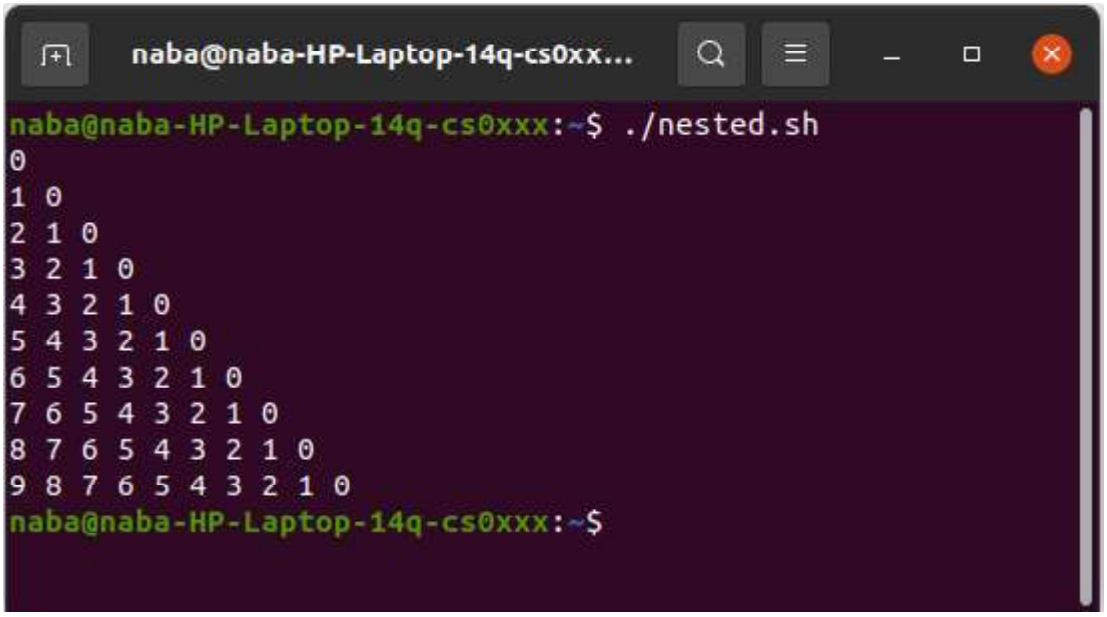
All the loops support nesting concept which means that we can put one loop inside another similar one or different loops. This nesting can go up to unlimited number of times based on our requirement.

Example

```
Here is a simple example of nested loop?
#!/bin/sh
a=0
while [ $a -lt 10 ]    # this is loop1
do
b=$a
```

```
        while [ $b -ge 0 ]  # this is loop2
           do
              echo -n $b
              b=`expr $b - 1`
     done
       echo
       a=`expr $a + 1`
done
```

Output of the above shell script is as follows.



Note: In the above shell script, -n option lets echo avoid printing a new line character.

Loop Control statements

There are two loop control statements supported by shell ?

- break statement
- continue statement

The infinite Loop

A loop continues forever if the required condition is not met. A loop that executes forever without terminating infinite number of times is called an infinite loop.

The break Statement

The break statement is used to terminate the execution of loop

statements. After execution of break statement, the control came out of the loop and executes the next statement after loop.

Syntax

break

break n

Here n specifies the nth enclosing loop to break from.

The continue statement

The continue statement is similar to the break, except that it causes the current iteration of the loop to exit, rather than the entire loop and again start from beginning of the loop.

Syntax

continue

continue n

Here n specifies the nth enclosing loop to continue from.



EXERCISE

Q 1: List different loops used in shell programming. Find the difference between loops used in C programming with loops used in shell programming.

9.4.7 Linux Shell Substitution

The following escape sequences are used with echo command on

Sl. No.	Escape sequence	Description
1	\\	backslash
2	\a	alert (BEL)
3	\b	Backspace
4	\c	suppress trailing newline
5	\f	form feed
6	\n	new line
7	\r	carriage return
8	\t	horizontal tab
9	\v	vertical tab

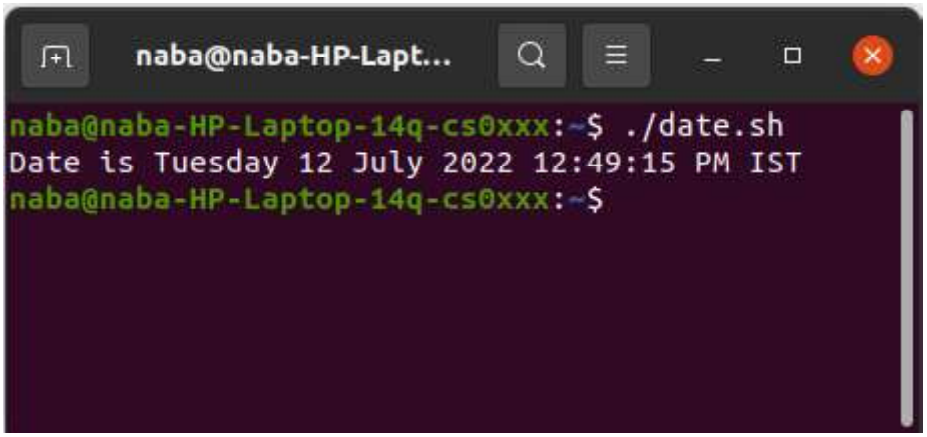
Command Substitution

By command substitution shell performs a given set of commands and then substitutes by the output in place of commands.

Example

```
#!/bin/sh
DATE=`date`
echo "Date is $DATE"
```

Output of the above shell is ?



Variable Substitution

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

Sl. No.	Command	Form & Description
1	**${var}**	Substitute the value of *var*.
2	**${var:-word}**	If *var* is null or unset, *word* is substituted for **var**. The value of *var* does not change.
3	**${var:=word}**	If *var* is null or unset, *var* is set to the value of **word**.
4	**${var:?message}**	If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly.
5	**${var:+word}**	If *var* is set, *word* is substituted for var. The value of *var* does not change.

Metacharacters in Shell

Unix Shell provides various metacharacters which have special meaning in the Shell Script and causes termination of a

word unless quoted.

For example, ? matches with a single character while listing files in a directory and * matches more than one character. List of popular metacharacters in shell is given bellow?

```
* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab
```

Sl. No.	Quoting & Description
1	**Single quote** All special characters between these quotes lose their special meaning.
2	**Double quote** Most special characters between these quotes lose their special meaning with these exceptions − • $ • ` • \$ • \' • \" • \\
3	**Backslash** Any character immediately following the backslash loses its special meaning.
4	**Back quote** Anything in between back quotes would be treated as a command and would be executed.

9.4.8 Input/Output Redirections in Shell

Most Unix system commands take input from the terminal and show the result in the terminal. A command normally reads the input from the standard input devices and displays the output in the standard output devices. But in some situations, input/output is not performed from standard I/O devices instead they are redirected from other place like file etc. This is called input/output redirection

Output Redirection

If the output of a shell script or a command is diverted from standard output device (STDOUT) to some other location then it is called as

output redirection.

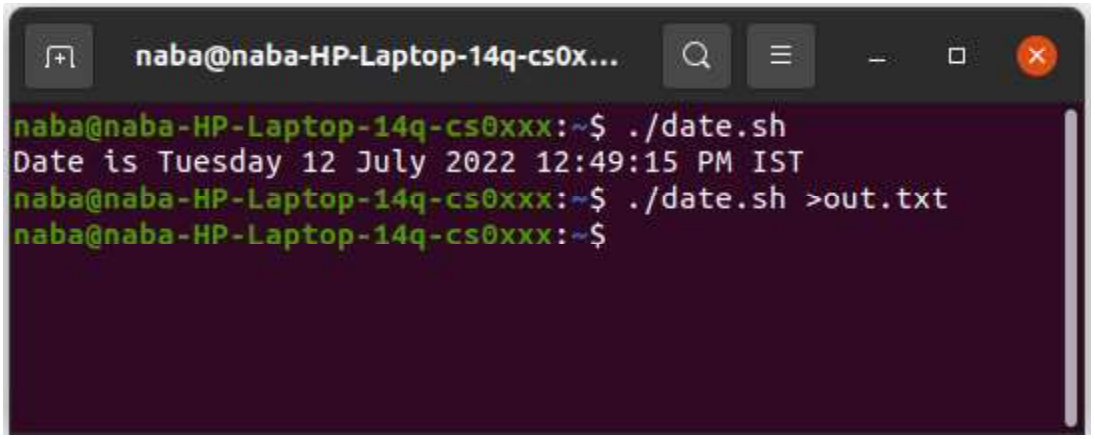
The notation '>' is used for output redirection, for example

$ who > users

Runs the who command and write the output of the command in the file 'users'.

Input Redirection

Similar to output redirection, input to a shell script or a command can also be redirected from some other location instead of standard input device (STDIN). less-than character < is used for input redirection. For example





Here output of the previous program date is redirected to out.txt file.

Here Document

Here document is used to redirect input into an interactive

shell script or program.

We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script. This is done by using '<<' symbol.

Discard the output

Sometimes we need to execute command or shell script, but we don't want the output displayed on the monitor. In such cases, we can discard the output by redirecting it to the file /dev/null ?

$ command > /dev/null

Redirection Commands

Following is a complete list of commands which you can use for redirection ?

Sl. No.	Command	Description
1	**pgm > file**	Output of pgm is redirected to file
2	**pgm < file**	Program pgm reads its input from file
3	**pgm >> file**	Output of pgm is appended to file
4	**n > file**	Output from stream with descriptor **n** redirected to file
5	**n >> file**	Output from stream with descriptor **n** appended to file
6	**n >& m**	Merges output from stream **n** with stream **m**
7	**n <& m**	Merges input from stream **n** with stream **m**
8	**<< tag**	Standard input comes from here through next tag at the start of line
9	**|**	Takes output from one program, or process, and sends it to another

Note: The file descriptor 0 is used for standard input (STDIN), 1 for standard output (STDOUT) and 2 for standard error output (STDERR).

9.4.9 Functions in shell

Functions enable us to break down the shell script into smaller, logical subsections, which can perform some tasks individually.

Using functions, we can perform repetitive tasks very easily. Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Function Creation

```
Syntax for function declaration in shell is ?
function_name ()
  {
      list of commands
  }
Example:
#!/bin/sh
# Function Definition
Hello () {
   echo "Hello World"
}
# Function call
Hello
```

Output of the shell script -



Parameter in Function

We can define function which accept parameters while calling the function. These parameters would be represented by the variable $1, $2 and so on.

Example:

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

```
# Function Definition
Add () {
   SUM=$(expr $1 + $2)
   echo "Sum is= $SUM "
}
# Function Call
Add 5 8
```

Output of the script ?



Returning Values from Function

Function can return a value by using 'return' command.

Syntax of 'return' ?

return code

Here code can be anything that we want to return.

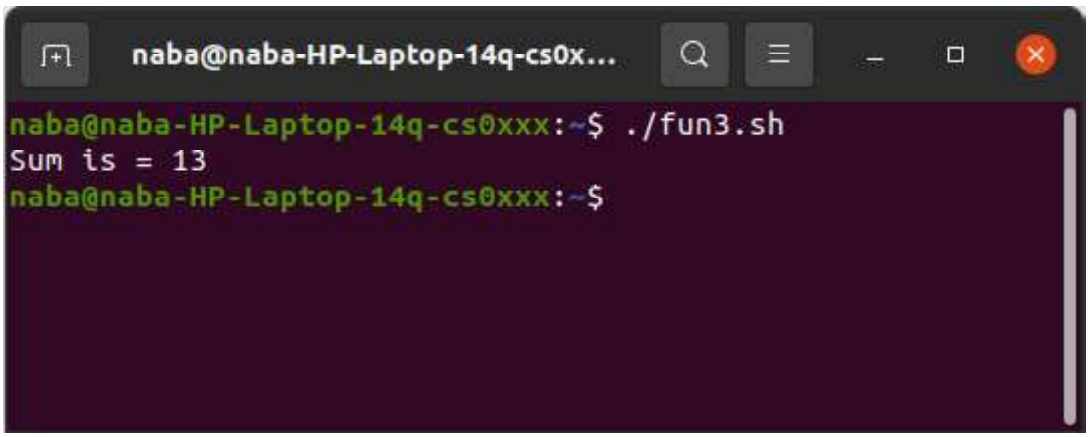
Example

Following function returns a value ?

```
#!/bin/sh
# Function Definition
Add () {
   SUM=$(expr $1 + $2)
   return $SUM
}
# Function Call
Add 5 8
# Return value of the function
```

```
ret=$?
echo "Sum is = $ret"
```

Output of the shell script -



Nested Functions

A function can call another function. This feature of function calling is called nested function. A function that calls itself is known as a recursive function.



CHECK YOUR PROGRESS-3

Q 13: In shell script a function can return a value. (TRUE/ FALSE)

Q 14: break and continue statement are supported in shell programming. (TRUE/ FALSE)

Q 15: We cannot redirect output from shell script. (TRUE/ FALSE)

Q 16: Syntax of for loop is _____



9.5 LET US SUM UP

- A shell is a special user program which provide an interface to user to use operating system services.
- Shell accepts commands from user and converts them into kernel understandable version.
- Shell is also known as "command interpreter".
- Broadly, the shell is categorized into two main classes- Command-line Shell, Graphical Shell

- There are five different shells in Linux- The Bourne Shell (sh), The GNU Bourne-Again Shell (bash), The C Shell (csh), The Korn Shell (ksh), The Z Shell (zsh).
- Bourne shell is the first UNIX shell. It is compact and high speeds.
- GNU Bourne-Again Shell is popularly known as the Bash shell, the GNU Bourne-Again shell was designed to be compatible with the Bourne shell. It incorporates useful features from different types of shells in Linux such as Korn shell and C shell.
- The C shell support for arithmetic operations and a syntax similar to the C programming language.
- The Korn shell is an improvement over the Bourne shell.
- The Z Shell is a sh shell extension with various improvements and customization.
- Shells are one of the most powerful tools available for Linux user. Without shells, it is practically impossible for a user to utilise all the features and functionality offered by the kernel in a Linux system.
- A shell script is a list of commands, which are listed in order they need to be executed.
- Shell scripts have several required constructs that tell the shell environment what to do and when to do it. The shell script is a real programming language, complete with variables, control structures, and so forth.
- In shell script, a variable is nothing more than a pointer to the actual data. The shell enables us to create, store values to the variable, and delete variables.
- There are three main types of variables- Local Variables, Environment Variables, Shell Variables.
- Shell Programming supports Command-Line Arguments. Via command line arguments we can provide input to the shell script.
- Shell supports array, which can hold multiple values at the same time. Arrays provide a method of grouping a set of variables.
- Shell programming supports the different types of operators as supported by any high-level languages.

- Shell programming also supports conditional statements- if, if-else loop statements while, for, etc .
- Shell script support Input/Output Redirections, via which user can redirect the output to some other file or can provide input from file.
- Like any other high level programming languages shell scripts also support functions.



9.6 FURTHER READINGS

1) Shell Scripting by Jason Cannon
2) Linux Command Line and Shell Scripting Bible by Richard Blum & Christine Bresnahan.
3) Mastering Linux Shell Scripting by Andrew Mallett & Mokhtar Ebrahim



9.7 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: c) Shell is a wrapper for all the commands and utilities

Ans to Q No 2: Bourne-again Shell

Ans to Q No 3: TRUE

Ans to Q No 4: TRUE

Ans to Q No 5: TRUE

Ans to Q No 6: Read

Ans to Q No 7: Name of the file

Ans to Q No 8: C=$(expr $A + $B)

Ans to Q No 9: $A -gt $B

Ans to Q No 10: ${arr[*]} or ${arr[@]}

Ans to Q No 11: TRUE

Ans to Q No 12: FALSE

Ans to Q No 13: TRUE

Ans to Q No 14: TRUE

Ans to Q No 15: FALSE

Ans to Q No 16: for var in word1 word2 ... wordN

```
do
Statement(s) to be executed for every word.
```

```
done
```



9.8 MODEL QUESTIONS

Q 1: What is shell? What are the different shells available in Linux Operating System? Explain briefly.

Q 2: What is shell scripting? What are the advantages of it?

Q 3: What are the different operators supported by shell? Explain briefly.

Q 4: Write a shell script to find the average of n numbers.

Q 5: Write a shell script to find the minimum of n numbers.

Q 6: Write a shell script to print the Fibonacci sequence up to n.

Q 7: Write a shell script to find the average of array elements.

*** ***** ***