

EXPIRIMENT-4:

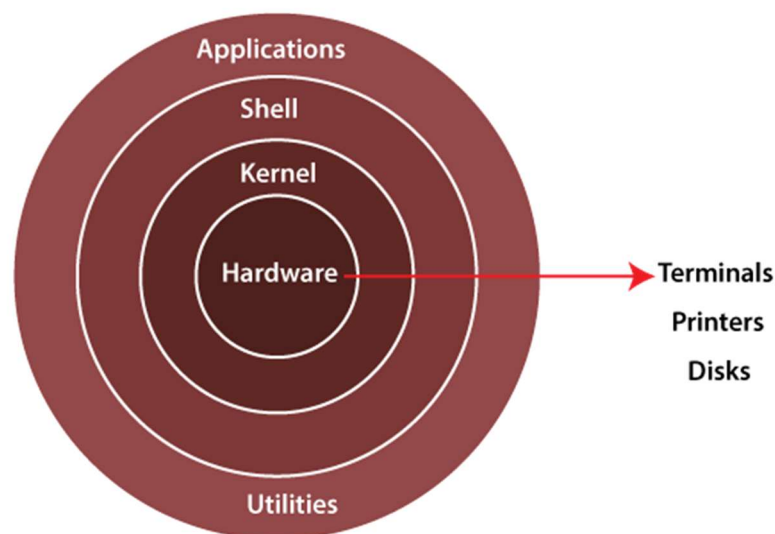
Aim: Shell scripting: study bash syntax, environment variables, variables, control constructs such as if, for and while, aliases and functions, accessing command line arguments passed to shells. Study of startup scripts, login and logout scripts, familiarity with systemd and system 5 init scripts is expected.

Solution :-

SHELL SCRIPTING

Shell Scripting is an open-source computer program designed to be run by the Unix/Linux shell. Shell Scripting is a program to write a series of commands for the shell to execute. It can combine lengthy and repetitive sequences of commands into a single and simple script that can be stored and executed anytime which, reduces programming efforts.

Shell is a UNIX term for an interface between a user and an operating system service. Shell provides users with an interface and accepts human-readable commands into the system and executes those commands which can run automatically and give the program's output in a shell script.



An Operating is made of many components, but its two prime components are:

- Kernel
- Shell

A Kernel is at the nucleus of a computer. It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one.

A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.

When you run the terminal, the Shell issues a command prompt (usually \$), where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.

The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name Shell.

Bash

BASH is an acronym for Bourne Again Shell, a punning name, which is a tribute to Bourne Shell (i.e., invented by Steven Bourne).

Bash is a shell program written by Brian Fox as an upgraded version of Bourne Shell program 'sh'. It is an open-source GNU project. It was released in 1989 as one of the most popular shell distribution of GNU/Linux operating systems. It provides functional improvements over Bourne Shell for both programming and interactive uses. It includes command line editing, key bindings, command history with unlimited size, etc.

In basic terms, Bash is a command line interpreter that typically runs in a text window where user can interpret commands to carry out various actions. The combination of these commands as a series within a file is known as a Shell Script. Bash can read and execute the commands from a Shell Script.

Bash is the default login shell for most Linux distributions and Apple's mac OS. It is also accessible for Windows 10 with a version and default user shell in Solaris 11.

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by # sign, describing the steps.

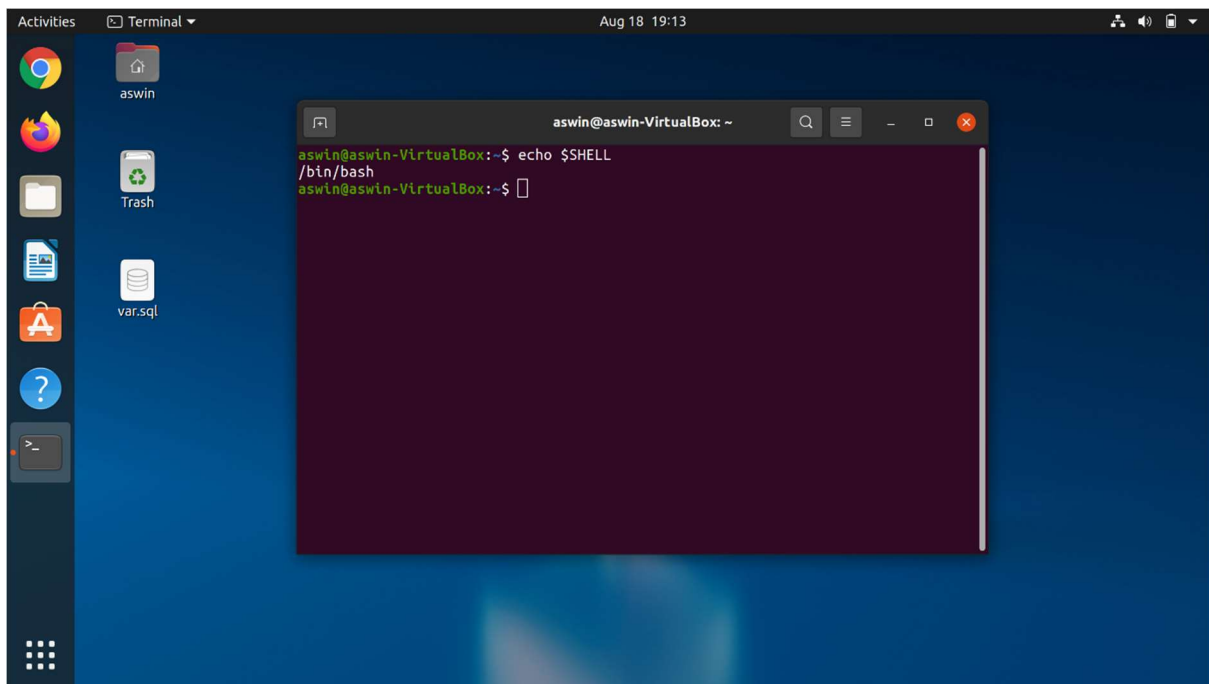
There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

We are going to write many scripts in the next sections. It would be a simple text file in which we would put all our commands and several other required constructs that tell the shell environment what to do and when to do it.

Shell scripts and functions are both interpreted. This means they are not compiled.

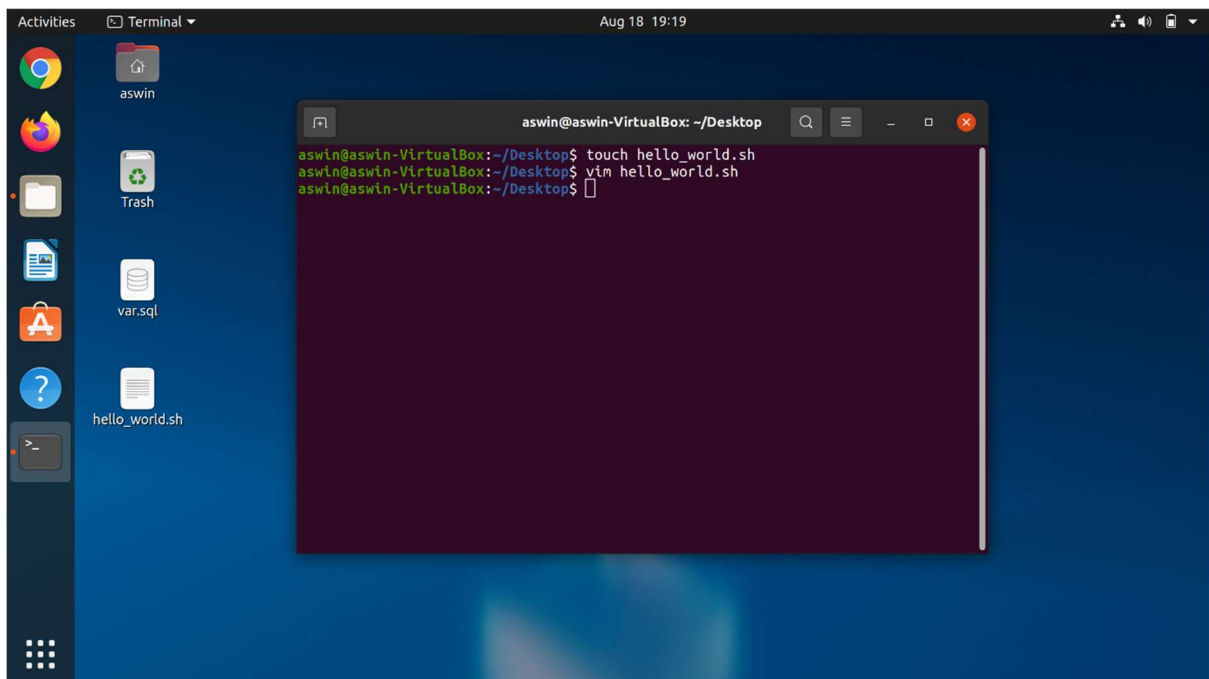
In a Bash Shell Script First you need to find out where is your bash interpreter located. Enter the following into your command line:

```
echo $shell
```

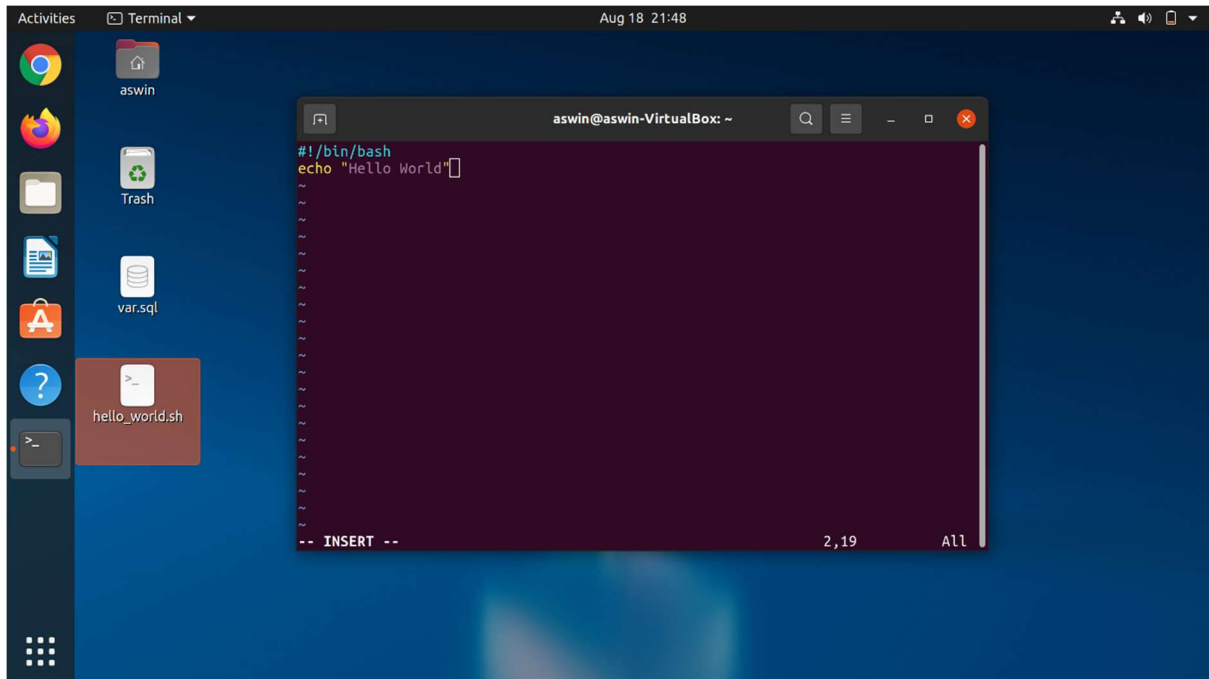


Example:

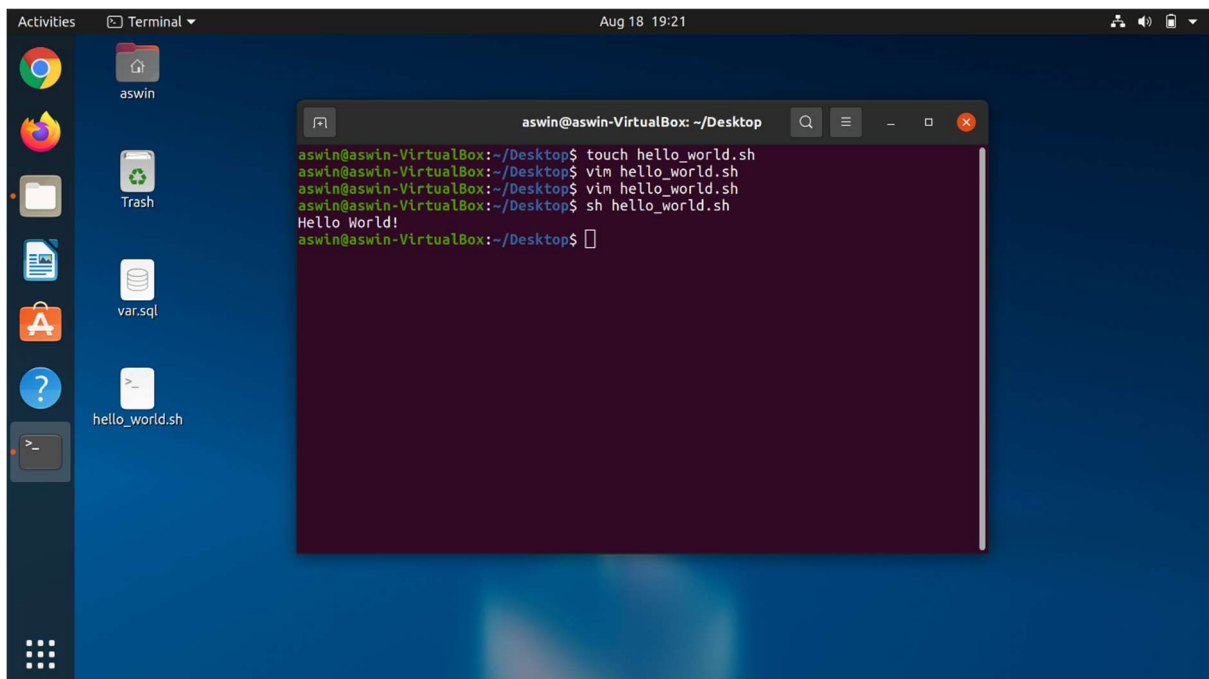
1. Open up vim text editor and create file called hello_world.sh:



2. Insert the following lines to a file:



3. Execute the bash script:



Environment Variables in Linux/Unix

Environment variables or **ENVs** basically define the behaviour of the environment. They can affect the processes ongoing or the programs that are executed in the environment.

Scope of an environment variable

Scope of any variable is the region from which it can be accessed or over which it is defined. An environment variable in Linux can have **global** or **local** scope.

Global

A globally scoped ENV that is defined in a terminal can be accessed from anywhere in that particular environment which exists in the terminal. That means it can be used in all kind of scripts, programs or processes running in the environment bound by that terminal.

Local

A locally scoped ENV that is defined in a terminal cannot be accessed by any program or process running in the terminal. It can only be accessed by the terminal (in which it was defined) itself.

Accessing ENVs

Syntax:

```
$NAME
```

NOTE: Both local and global environment variables are accessed in the same way.

Displaying ENVs

To display any ENV

Syntax:

```
$ echo $NAME
```

To display all the Linux ENVs

Syntax:

```
$ printenv //displays all the global ENVs
```

or

```
$ set //display all the ENVs(global as well as local)
```

or

```
$ env //display all the global ENVs
```

Setting environment variables

To set a global ENV

```
$ export NAME=Value
```

or

```
$ set NAME=Value
```

To set a local ENV

Syntax:

```
$ NAME=Value
```

Some commonly used ENVs in Linux:

\$USER	: Gives current user's name.
\$PATH	: Gives search path for commands.
\$PWD	: Gives the path of present working directory.
\$HOME	: Gives path of home directory.
\$HOSTNAME	: Gives name of the host.
\$LANG	: Gives the default system language.
\$EDITOR	: Gives default file editor.
\$UID	: Gives user ID of current user.
\$SHELL	: Gives location of current user's shell program.

variables

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.

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

Variable Names

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

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names –

```
_ALI  
TOKEN_A  
VAR_1  
VAR_2
```

Defining Variables

Variables are defined as follows:

```
variable_name=variable_value
```

For example:

```
NAME="Zara Ali"
```

The above example defines the variable NAME and assigns the value "Zara Ali" 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, prefix its name with the dollar sign (\$).

For example, the following script will access the value of defined variable NAME and print it on STDOUT

```
#!/bin/sh  
  
NAME="Zara Ali"  
  
echo $NAME
```

The above script will produce the following value:

```
Zara Ali
```

Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME:

```
#!/bin/sh  
  
NAME="Zara Ali"  
  
readonly NAME  
  
NAME="Qadiri"
```

The above script will generate the following result:

```
/bin/sh: NAME: This variable is read only.
```

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command –

```
unset variable_name
```

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works:

```
#!/bin/sh

NAME="Zara Ali"

unset NAME

echo $NAME
```

The above example does not print anything. You cannot use the unset command to **unset** variables that are marked **readonly**.

Variable Types

When a shell is running, three main types of variables are present:

- **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 needed by the programs that it runs.
- **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.

Control constructs

You can control the execution of Linux commands in a shell script with control structures. Control structures allow you to repeat commands and to select certain commands over others. A control structure consists of two major components: a test and commands. If the test is successful, then the commands are executed. In this way, you can use control structures to make decisions as to whether commands should be executed.

There are two different kinds of control structures: loops and conditions. A loop repeats commands, whereas a condition executes a command when certain conditions are met. The BASH shell has three loop control structures: while, for, and for-in. There are two condition structures: if and case. The control structures have as their test the execution of a Linux command. All Linux commands return an exit status after they have finished executing. If a command is successful, its exit status will be 0. If the command fails for any reason, its exit status will be a positive value referencing the type of failure that occurred. The control structures check to see if the exit status of a Linux command is 0 or some other value. In the case of the if and while structures, if the exit status is a zero value, then the command was successful and the structure continues.

Test Operations

With the test command, you can compare integers, compare strings, and even perform logical operations. The command consists of the keyword test followed by the values being compared, separated by an option that specifies what kind of comparison is taking place. The option can be thought of as the operator, but it is written, like other options, with a minus sign and letter codes. For example, -eq is the option that represents the equality comparison. However, there are two string operations that actually use an operator instead of an option. When you compare two strings for equality you use the equal sign, =. For inequality you use !=. Here is some of the commonly used options and operators used by test.

The syntax for the test command is shown here:

```
test value -option value
```

```
test string = string
```

Integer Comparisons	Function
-gt	Greater-than
-lt	Less-than
-ge	Greater-than-or-equal-to
-le	Less-than-or-equal-to
-eq	Equal
-ne	Not-equal

String Comparisons	Function
-z	Tests for empty string
=	Equal strings
!=	Not-equal strings

Logical Operations	Function
-a	Logical AND
-o	Logical OR
!	Logical NOT

File Tests	Function
-f	File exists and is a regular file
-s	File is not empty

-r	File is readable
-w	File can be written to, modified
-x	File is executable
-d	Filename is a directory name

Conditional Control Structures

The BASH shell has a set of conditional control structures that allow you to choose what Linux commands to execute. Many of these are similar to conditional control structures found in programming languages, but there are some differences. The if condition tests the success of a Linux command, not an expression. Furthermore, the end of an if-then command must be indicated with the keyword `fi`, and the end of a case command is indicated with the keyword `esac`.

The if structure places a condition on commands. That condition is the exit status of a specific Linux command. If a command is successful, returning an exit status of 0, then the commands within the if structure are executed. If the exit status is anything other than 0, then the command has failed and the commands within the if structure are not executed. The if command begins with the keyword `if` and is followed by a Linux command whose exit condition will be evaluated. The keyword `fi` ends the command. The `elsels` script in the next example executes the `ls` command to list files with two different possible options, either by size or with all file information. If the user enters an `s`, files are listed by size; otherwise, all file information is listed.

if

`if` executes an action if its test command is true.

Syntax:

```
if command then
    command
fi
```

if then else

`if-else` executes an action if the exit status of its test command is true; if false, then the else action is executed.

Syntax:

```
if command then
    command
else
    command
fi
```

elif

elif allows you to nest if structures, enabling selection among several alternatives; at the first true if structure, its commands are executed and control leaves the entire elif structure.

Syntax:

```
if command then
    command
elif command then
    command
else
    command
fi
```

case

case matches the string value to any of several patterns; if a pattern is matched, its associated commands are executed.

Syntax:

```
case string in
pattern)
    command;;
esac
```

Logical AND

The logical AND condition returns a true 0 value if both commands return a true 0 value; if one returns a non-zero value, then the AND condition is false and also returns a non-zero value.

Syntax:

```
command && command
```

Logical OR

The logical OR condition returns a true 0 value if one or the other command returns a true 0 value; if both commands return a non-zero value, then the OR condition is false and also returns a non-zero value.

Syntax:

```
command || command
```

Logical NOT

The logical NOT condition inverts the return value of the command.

Syntax:

```
! command
```

Loop Control Structures

The while loop repeats commands. A while loop begins with the keyword while and is followed by a Linux command. The keyword do follows on the next line. The end of the loop is specified by the keyword done. The Linux command used in while structures is often a test command indicated by enclosing brackets.

The for-in structure is designed to reference a list of values sequentially. It takes two operands—a variable and a list of values. The values in the list are assigned one by one to the variable in the for-in structure. Like the while command, the for-in structure is a loop. Each time through the loop, the next value in the list is assigned to the variable. When the end of the list is reached, the loop stops. Like the while loop, the body of a for-in loop begins with the keyword do and ends with the keyword done.

while

while executes an action as long as its test command is true.

Syntax:

```
while command  
  
do  
  
    command  
  
done
```

until

until executes an action as long as its test command is false.

Syntax:

```
until command  
  
do  
  
    command  
  
done
```

for-in

for-in is designed for use with lists of values; the variable operand is consecutively assigned the values in the list.

Syntax:

```
for variable in list-values  
  
do  
  
    command  
  
done
```

for

for is designed for reference script arguments; the variable operand is consecutively assigned each argument value.

Syntax:

```
for variable
do
    command
done
```

select

select creates a menu based on the items in the item-list; then it executes the command; the command is usually a case.

Syntax:

```
select string in item-list
do
    command
done
```

Shell functions

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual tasks when needed.

Using functions to perform repetitive tasks is an excellent way to create **code reuse**. This is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions

To declare a function, simply use the following syntax:

```
function_name () {
    list of commands
}
```

The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, followed by a list of commands enclosed within braces.

Example

Following example shows the use of function:

```
#!/bin/sh

# Define your function here

Hello () {
    echo "Hello World"
}

# Invoke your function

Hello
```

Upon execution, you will receive the following output:

```
$/test.sh
Hello World
```

Pass Parameters to a Function

You can define a function that will accept parameters while calling the function. These parameters would be represented by **\$1**, **\$2** and so on.

Following is an example where we pass two parameters *Zara* and *Ali* and then we capture and print these parameters in the function.

```
#!/bin/sh

# Define your function here

Hello () {
    echo "Hello World $1 $2"
}

# Invoke your function

Hello Zara Ali
```

Upon execution, you will receive the following result:

```
$/test.sh
Hello World Zara Ali
```

Returning Values from Functions

If you execute an **exit** command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the **return** command whose syntax is as follows:

return code

Here **code** can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example

Following function returns a value 10:

```
#!/bin/sh

# Define your function here
Hello () {
    echo "Hello World $1 $2"
    return 10
}

# Invoke your function
Hello Zara Ali

# Capture value returned by last command
ret=$?

echo "Return value is $ret"
```

Upon execution, you will receive the following result:

```
$/test.sh
Hello World Zara Ali
Return value is 10
```

Nested Functions

One of the more interesting features of functions is that they can call themselves and also other functions. A function that calls itself is known as a recursive function.

Following example demonstrates nesting of two functions:

```
#!/bin/sh

# Calling one function from another
number_one () {
    echo "This is the first function speaking..."
    number_two
}

number_two () {
    echo "This is now the second function speaking..."
}

# Calling function one.
```

```
number_one
```

Upon execution, you will receive the following result:

```
This is the first function speaking...
This is now the second function speaking...
```

Function Call from Prompt

You can put definitions for commonly used functions inside your .profile. These definitions will be available whenever you log in and you can use them at the command prompt.

Alternatively, you can group the definitions in a file, say *test.sh*, and then execute the file in the current shell by typing:

```
$. test.sh
```

This has the effect of causing functions defined inside *test.sh* to be read and defined to the current shell as follows:

```
$ number_one
This is the first function speaking...
This is now the second function speaking...
$
```

To remove the definition of a function from the shell, use the unset command with the *.f* option. This command is also used to remove the definition of a variable to the shell.

```
$ unset -f function_name
```

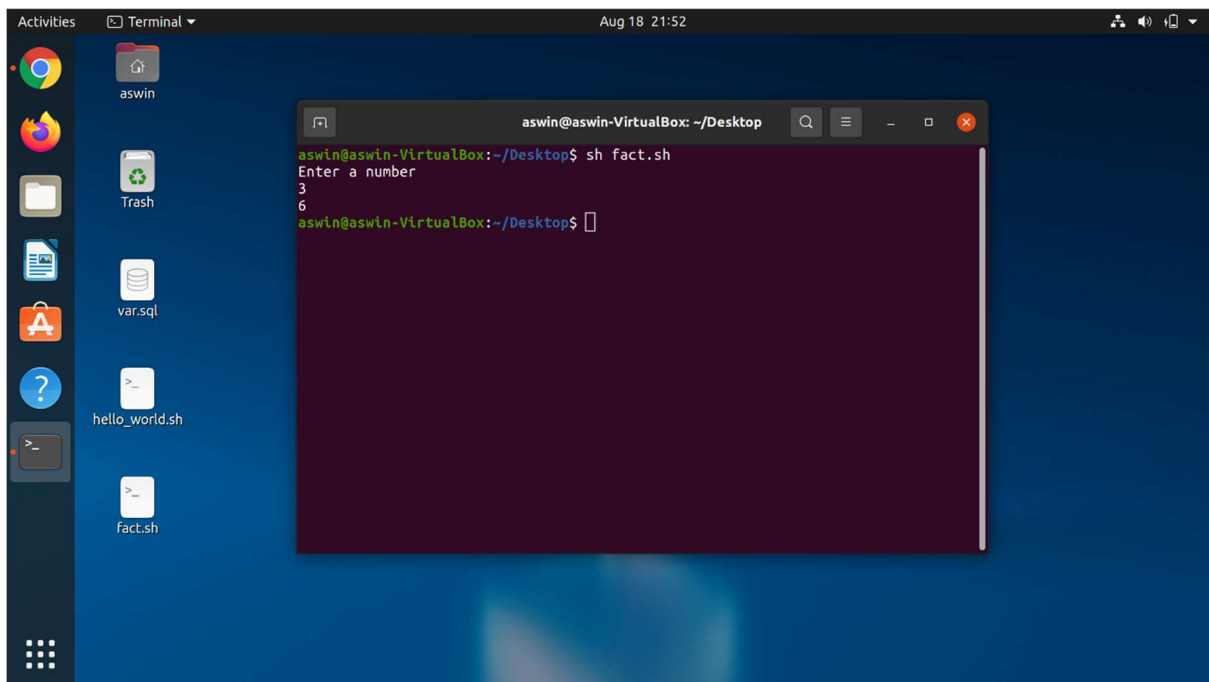
Simple Shell Program

Shell Program to Calculate the Factorial of a Number

Source code:

```
echo "Enter a number"
read num
fact=1
while [ $num -gt 1 ]
do
    fact=$((fact * num)) #fact = fact * num
    num=$((num - 1))     #num = num - 1
done
echo $fact
```


Output:



The screenshot shows a Linux desktop environment with a dark blue background. On the left is a vertical dock containing icons for Google Chrome, Firefox, a file manager, a trash can, a document, a database icon labeled 'var.sql', an application icon, a question mark, and a terminal icon. The desktop has several icons: 'aswin' (home), 'Trash', 'var.sql', 'hello_world.sh', and 'fact.sh'. A terminal window is open in the center, titled 'aswin@aswin-VirtualBox: ~/Desktop'. The terminal shows the following text:

```
aswin@aswin-VirtualBox:~/Desktop$ sh fact.sh
Enter a number
3
6
aswin@aswin-VirtualBox:~/Desktop$
```