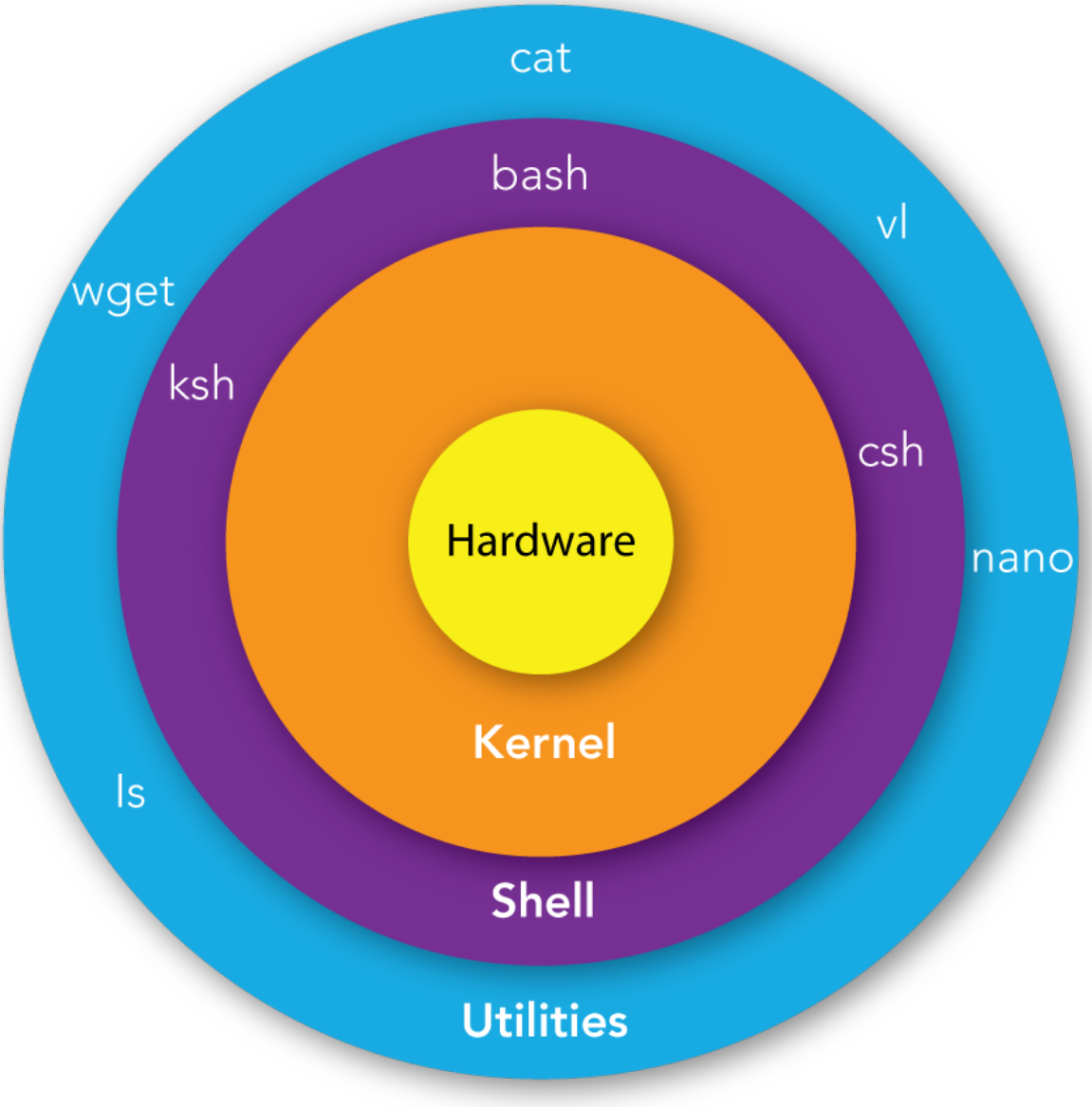
Shell scripting



What is Shell Script

A shell script is a list of commands in a computer program that is run by the Unix shell which is a command-line interpreter. A shell script usually has comments that describe the steps.

- **Allow you to run if-else statements and loops.**
- **A file that contains a series of commands.**
- **A plain text file.**
- **It executes the command on each line, one line at a time.**
- **The terminal usually allows just one command at a time.**
- **Shell script allows you to combine and run multiple commands together**

When do we need a shell script?

- **You need to enter multiple shell commands and you will need to do it again in the future.**
- **If you know how a piece of work need to perform.. you can put that knowledge in a shell script**
- **ShellScript eliminates repetitive tasks through automation.**
- **Using a good script can reduce the change of error.**
- **A shell script can perform a task faster than a human can.**
- **You have to do a task more than once, but it's something that you rarely do.**

Why use Shell

- **Speed of deployment**
- **No worrying about low-level programming objects.**
- **Ease and speed of learning.**
- **Performance and efficiency.**
- **Anything you can do on the command line can be automated by writing a shell script.**
- **Can automate tedious or repetitive tasks.**
- **allow you to hand off work to others.**

- **Act as a form of documentation.**
- **Fairly quick and easy to write.**

let's create the first script with the first.sh name.

If we run the bunch of commands it will take more time to executive one by one instead of doing this, we will do shell scripting

First we need to create a file instead that file we will put all our command together and run at a time

Example

$ touch shell.txt

Vi shell.txt

```
echo "Hello World"
echo "this is our first shellscript."
```

To run the above shell script by using the command

$ ./shell.txt

o/p:

```
 hello world
this is our
```

Now well run many command at a time for example

```
➜  ~ cat shell.txt
echo "hello world"
echo "this is our "

ls
date
pwd
```

Then we will get output by listing out whole data, and date and where we are

Shell Script Shebang

The #! syntax is used in scripts to indicate an interpreter for execution under UNIX / Linux operating systems. The directive must be the first line in the Linux shell script and must start with shebang #!.

The sharp sign (#) and the bang sign (!) thats why called is the shebang. (sharpbang)

Shebang starts with #! characters and the path to the bash or other interpreter of your choice. Make sure the interpreter is the full path to a binary file. For example: /bin/bash.

Shell Built-in and Commands

Shell Built-in

Shell Built-in does not request another program to run a process because it is a shell built-in. so use shell built-in if it is available.

A built-in command is simply a command that the shell carries out itself, instead of interpreting it as a request to load and run some other program. This has two main effects. First, it's usually faster, because loading and running a program takes time. Of course, the longer the command takes to run, the less significant the load time is compared to the overall run time (because the load time is fairly constant).

Keywords

Keywords are the words whose meaning has already been explained to the shell. the keywords cannot be used as variable names because it is a reserved word containing reserved meanings.

Sequence

when you run a command bash will search a function with the same name if the function with the same name is not present then bash will search it in builtins. if builtin is also not available then it will search the command at PATH locations.

~ uptime —--it will show us our system how many hours our system is running , how many user are connected

13:53:15 up 1:34, 1 user, load average: 0.86, 0.84, 0.91

➜ ~ type -a uptime ——————

The `type -a` command in a Unix-like operating system is used to display all the locations where a particular command is present in the system's executable path.

uptime is /usr/bin/uptime

uptime is /bin/uptime

➜ ~ type -a echo —--- In this output, the first line indicates that `echo` is a shell built-in command,

echo is a shell builtin

echo is /usr/bin/echo

echo is /bin/echo

➜ ~ type -a pwd —---------- it is an build in command in our shell script, it is present in user bin

pwd is a shell builtin

pwd is /usr/bin/pwd

pwd is /bin/pwd

➜ ~ type -a if

if is a reserved word

➜ ~ echo $PATH —----------it will check totally path of our commands

/home/mohammed/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin

Print Hello World In Different Color

In a shell script

, you can change the color of text in the terminal by using escape sequences. These sequences are special combinations of characters that are interpreted by the terminal to change the color or other properties of the text.

Here is an example of how to change the color of text in a shell script:

```
#!/bin/bash

# Set text color to red
echo -e "\e[31mHello, world!\e[0m"

# Set text color to green
echo -e "\e[32mHello, world!\e[0m"

# Set text color to yellow
echo -e "\e[33mHello, world!\e[0m"

# Set text color to blue
echo -e "\e[34mHello, world!\e[0m"

# Set text color to magenta
echo -e "\e[35mHello, world!\e[0m"

# Set text color to cyan
echo -e "\e[36mHello, world!\e[0m"
```

To change the color you can use this things

Create a file

Echo.sh

Vi echo.sh

```
#!/bin/bash

echo "this is mohammed"

echo "this is our shell script"

echo -e "\033[0;31m fail message"

echo -e "\033[0;32m success message"

echo -e "\033[0;33m warning message here"

echo -e "\e[31mHello, world!\e[0m"

echo -e "\e[32mHello, world!\e[0m"

echo -e "\e[33mHello, world!\e[0m"
```

```
echo -e "\e[34mHello, world!\e[0m"
```

Code . by using this command we can open the visual code,

Then

Give the executive permissions

Chmod +x echo.sh

The use the executive command

./echo,sh

After using this every thing will be exxecutite

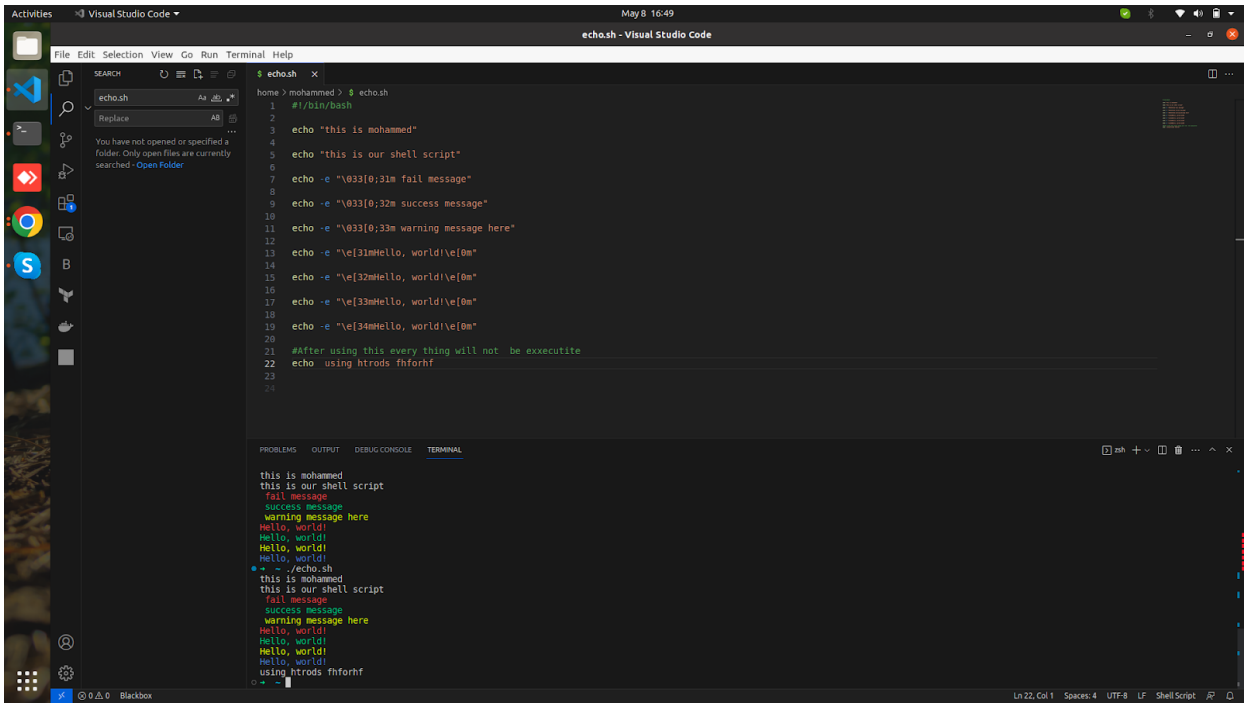
Comments and Escape

Comments are the useful information that the developers provide to make the reader understand the source code.

There are two types of comments:

1. **Single-line comment**
2. **Multi-line comment**

By using the # the comment will not be executed, just understanding purpose.



User Define Variables

A variable is a character string to which we assign a value.

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

Rules:

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

Variable names cannot be reserved words

Variable names cannot have whitespace in between

The variable name cannot have special characters.

The first character of the variable name cannot be a number.

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

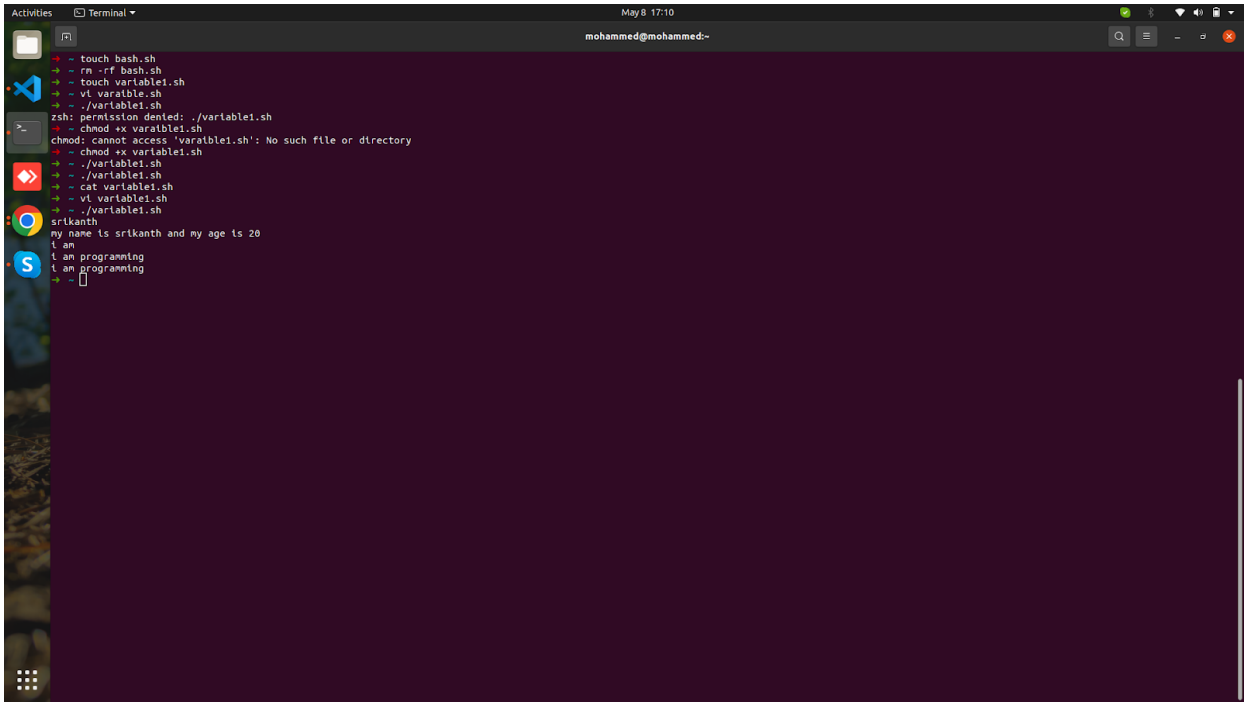
The following examples are valid variable names −

First create a file, then

Touch variable.sh

Vi variable.sh

```
#!/bin/bash
# variable.sh.
# user Define Variables.
name="Saurav"
age="20"
echo  ${name}
echo  "my name is ${name} and my age is ${age}"
# echo 'my name is ${name} and my age is ${age}'
work="programm"
var="ing"
echo  "i am $working"
echo  "i am ${work}ing"
echo  "i am ${work}${var}"
```



System Variable in ShellScript

Created and maintained by Linux bash shell itself. This type of variable (with the exception of auto_resume and histchars) is defined in CAPITAL LETTERS. You can configure aspects of the shell by modifying system variables such as PS1, PATH, LANG, HISTSIZE, and DISPLAY, etc.

There are several commands available that allow you to list and set environment variables in Linux:

- **env – The command allows you to run another program in a custom environment without modifying the current one. When used without an argument it will print a list of the current environment variables.**
- **printenv – The command prints all or the specified environment variables.**
- **set – The command sets or unsets shell variables. When used without an argument it will print a list of all variables including environment and shell variables, and shell functions.**
- **unset – The command deletes shell and environment variables.**
- **export – The command sets environment variables.**

Below are some of the most common environment variables:

- **USER - The current logged in user.**
- **HOME - The home directory of the current user.**
- **EDITOR - The default file editor to be used. This is the editor that will be used when you type edit in your terminal.**

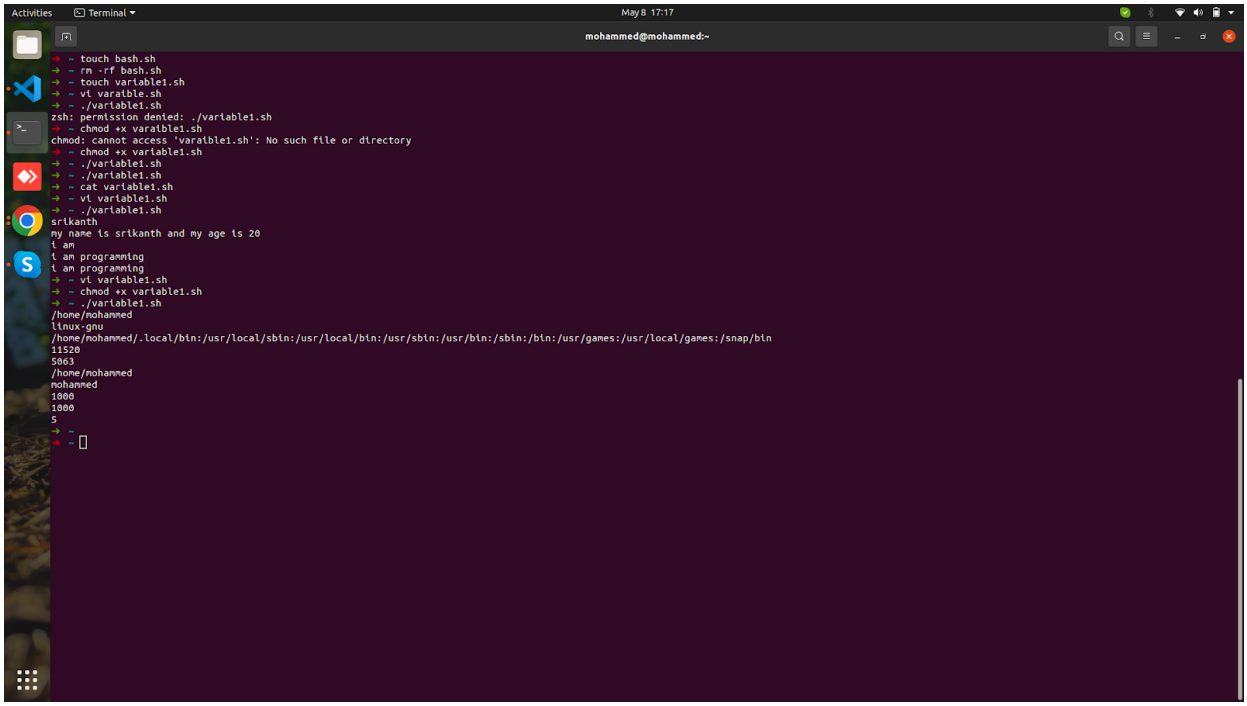
- **SHELL - The path of the current user's shell, such as bash or zsh.**
- **LOGNAME - The name of the current user.**
- **PATH - A list of directories to be searched when executing commands. When you run a command the system will search those directories in this order and use the first found executable.**
- **LANG - The current locales settings.**
- **TERM - The current terminal emulation.**

Create a file by using the command touch

Touch varaibale1.sh
touch variable1.sh
➜ ~ vi varaible.sh

```
#!/bin/bash
# System define variables.
# echo ${SHELL}
echo ${HOME} # will show the home directory of current user
echo ${OSTYPE} # type type of operating system.
echo $PATH # A list of directories to be searched when executing commands.
echo ${$} # process id
echo ${PPID} # parent process id

echo $PWD # present working directory
echo $HOSTNAME # hostname of machine.
echo $UID # user id
# UID="5000"
echo $UID
sleep 5
echo ${SECONDS}
```

➜ ~ chmod +x varaible1.sh
./varaible1.sh



Read Input From User In Shell Script

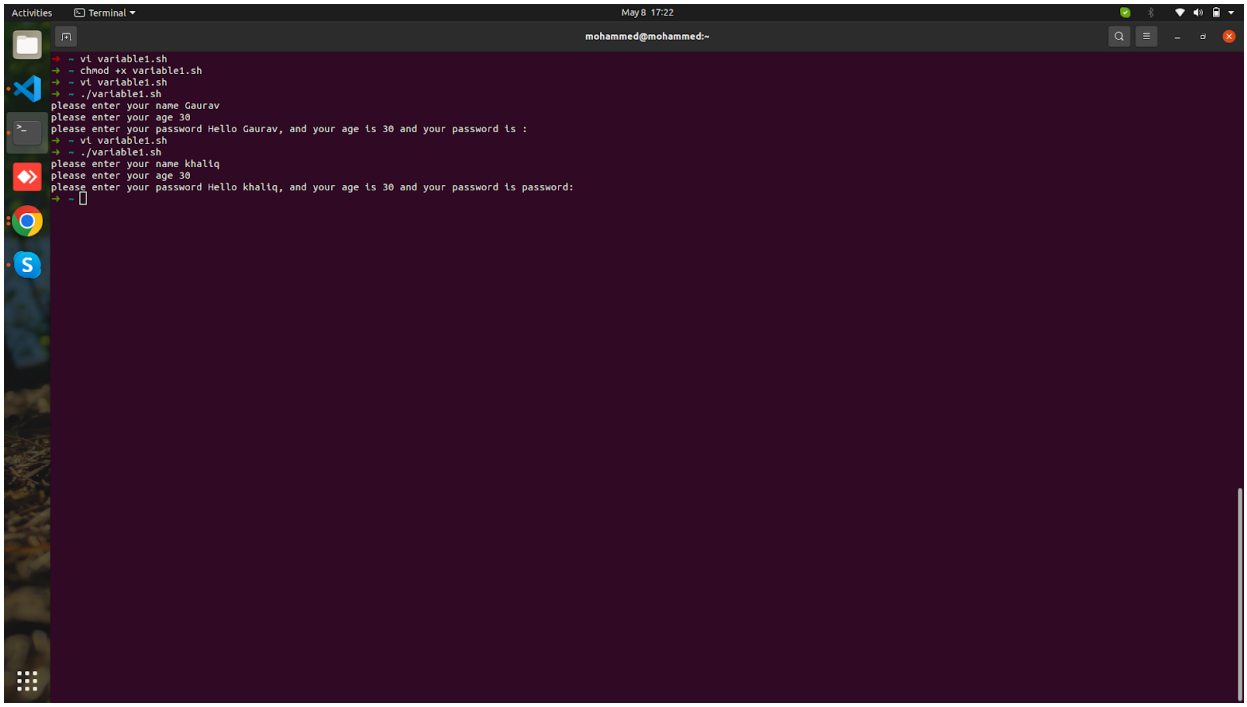
We can simply get user input from the read command in BASH. It provides a lot of options and arguments along with it more flexible usage

Create a file by using the touch command

Touch variable1.sh
Vi varaible1.sh

```
#!/bin/bash
read -p "please enter your name " name
read -p "please enter your age " age
read -p "please enter your password " -s password
echo "Hello ${name}, and your age is ${age} and your password is ${password}"
```

chmod +x variable1.sh
➜ ~ ./variable1.sh



Command Line Arguments in Shell Script

Arguments are inputs that are necessary to process the flow. Instead of getting input from a shell program or assigning it to the program, the arguments are passed in the execution part.

Positional Parameters

Command-line arguments are passed in the positional way i.e. in the same way how they are given in the program execution.

Create a file by using the touch command

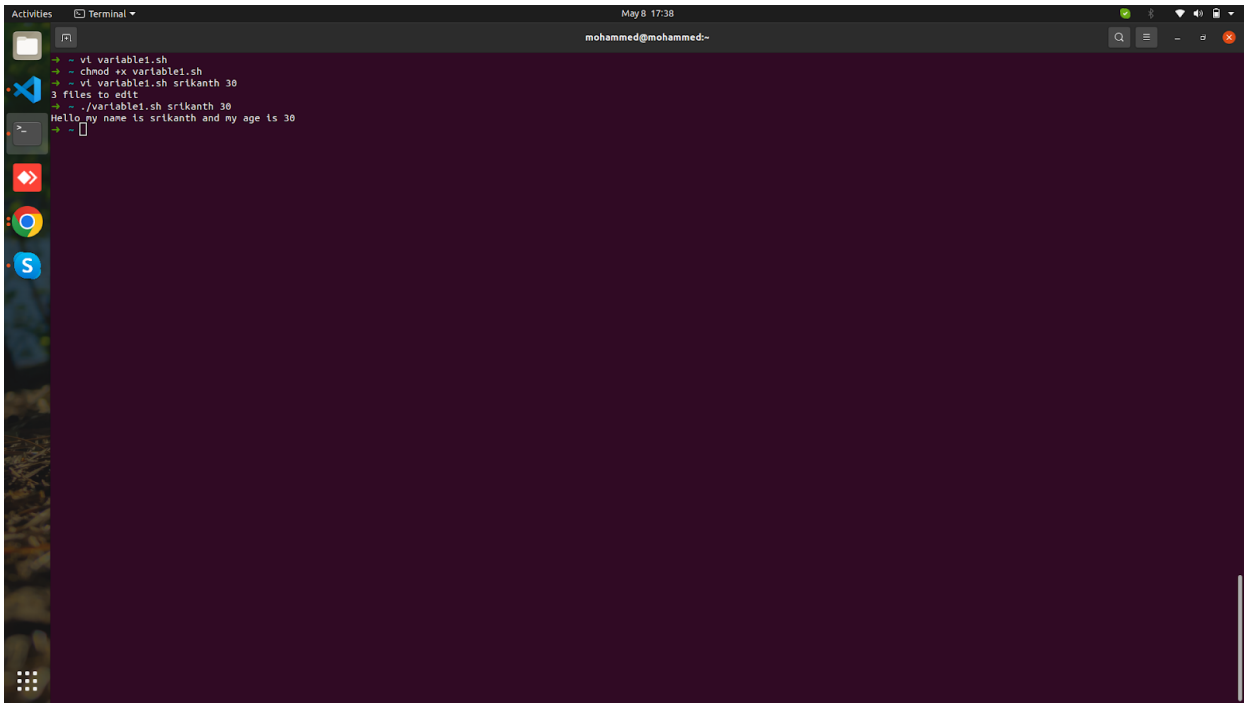
Touch variable1.sh
Vi varaible1.sh

```
#!/bin/bash

name=${1}
age=${2}
```

```
echo  "Hello my name is ${name} and my age is ${age}"
```

Chmod +x varaible1.sh

./varaible1.sh srikanth 30



Assign a Command's Output to a Variable in Shell Script

Suppose we want to store a command output to a variable in a serval way.

- **we can use back tick `**

```
VARIABLE_NAME=`command_here`
```

- **we can use the brackets (recommend way)**

```
VARIABLE_NAME=$(command_here)
```

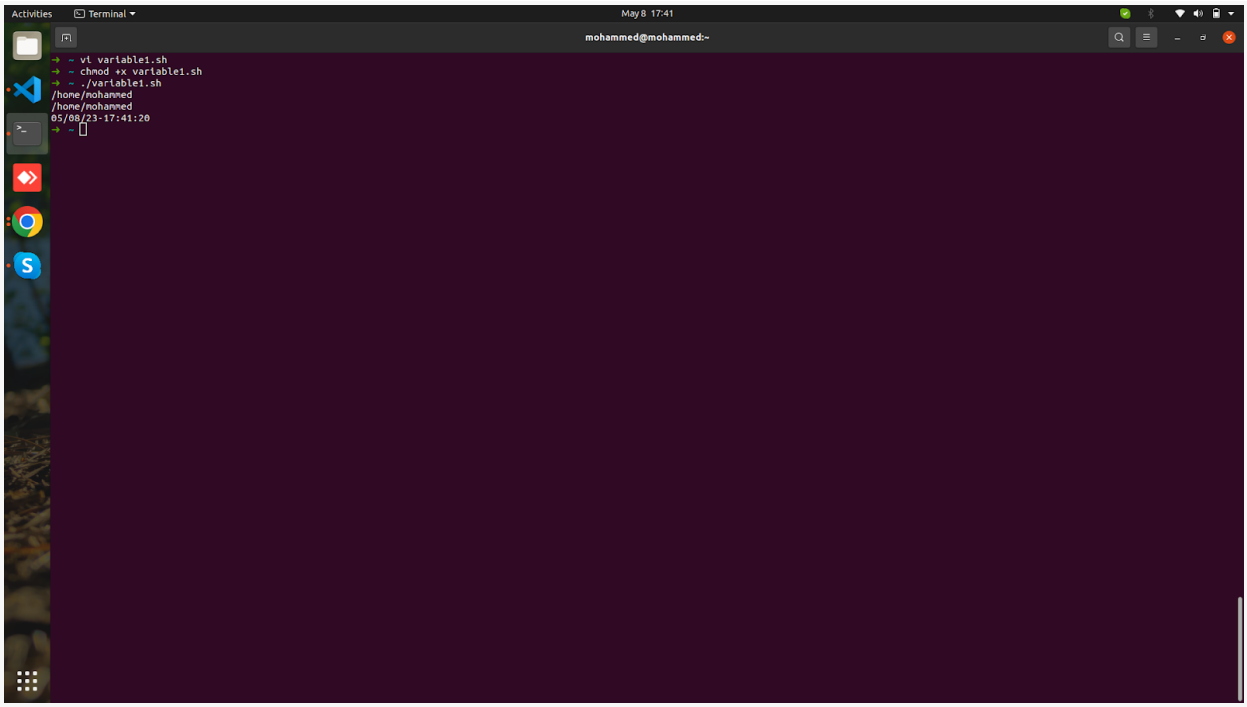
Touch varaible1.sh

Vi variable.sh

```
#!/bin/bash
CURRENT_WORKING_DIR=$(pwd)
VARIABLE_SECOND_METHOD=`pwd`
echo "${CURRENT_WORKING_DIR}"
echo "${VARIABLE_SECOND_METHOD}"
date_time=$(date +"%D-%T")
echo "${date_time}"
```

Chmod +x variable.sh

./varaible.sh



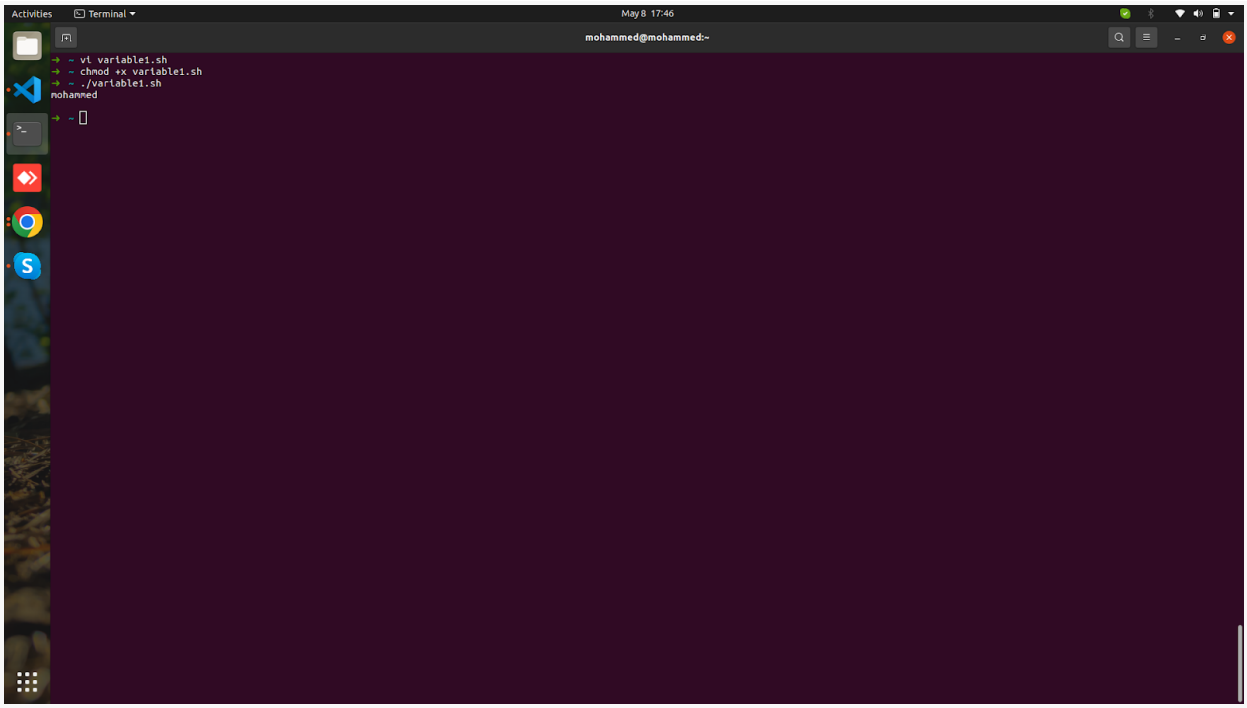
How to create a read-only variable in Shell Script

We can create read-only variable in shell script using readonly.

syntax:

```
$ readonly VARIABLE_NAME
```

```
#!/bin/bash
name="mohammed"
# readonly name
echo "${name}"
unset name
# name="mohammed"
echo "${name}"
```



Convert String in Shell Script

Convert First Character to Upper Case in Shell Script

```
#!/bin/bash
string="my name is khaliq"
echo "${string^}"    # My name is khaliq
```

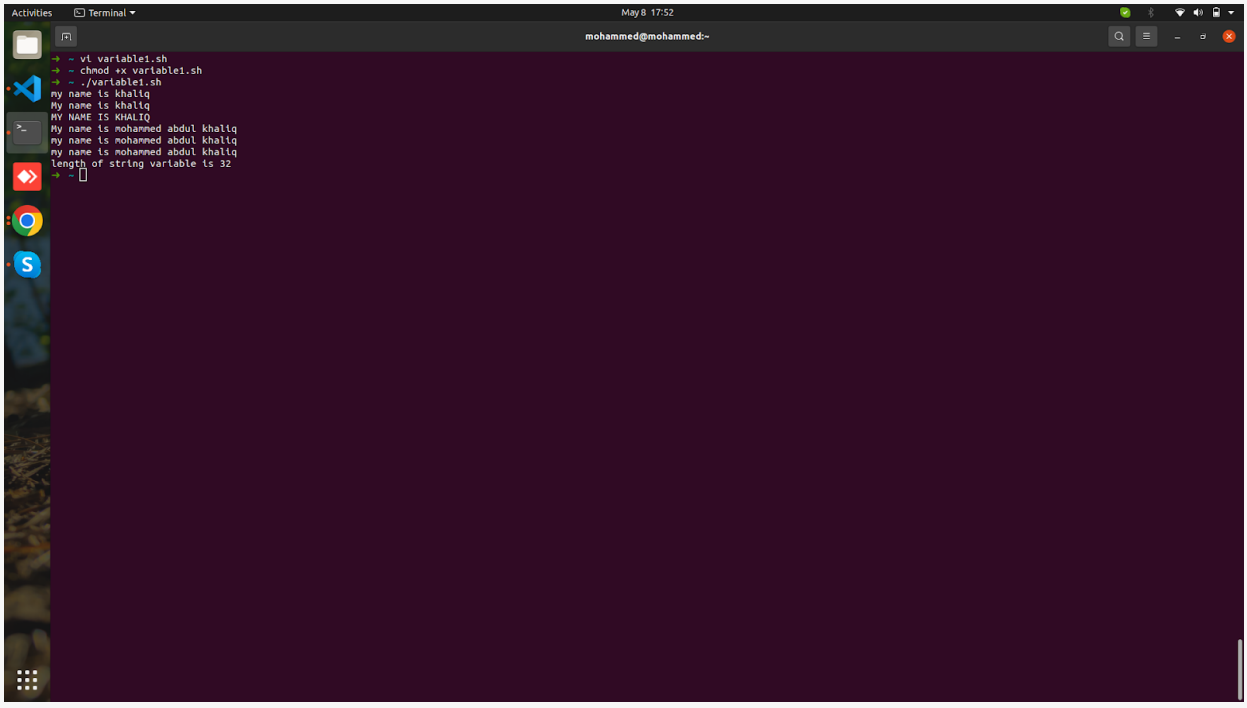
Convert a String to Upper Case in Shell Script

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

```
string="my name is Khaliq"
echo  "${string^^}"  # MY NAME IS KHALIQ
```

Convert First Character to Lower Case in Shell Script

```
#!/bin/bash
string="My name is Khaliq"
echo  "${string,}"  # my name is Khaliq

Touch variable1.sh
Vi varaible1.sh

#!/bin/bash
string="my name is khaliq"
echo "${string}"   # my name is Khaliq
echo "${string^}"  # My name is Khaliq
echo "${string^^}" # MY NAME IS KHALIQ
string="My name is mohammed abdul khaliq"
echo "${string}"  # My name is Abdul
echo "${string,}" # my name is mohammed
echo "${string,,}" # my name is khaliq
echo "length of string variable is ${#string}"

Chmod +x variable.sh
./varaible.sh
```



Substring in Shell Script

Get Substring from a String

syntax

```
${string:position}
```

example

```
#!/bin/bash
string="abckhaliqabcxyz"
echo  "${string:0}"    # output -> abckhaliqabcxyz
echo  "${string:1}"    # bckhaliqbcxyz
echo  "${string:4}"    # auravabcxyz
```

Get Last n Character from a String

```
#!/bin/bash
string="abckhaliqabcxyz"
```

```
echo "${string: -3}" # xyz
```

Get Substring With Specific Length From String

```
#!/bin/bash
string="abckhaliqbcxyz"
echo "${string:0:3}"
echo "${string:3:3}"
```

Get Shortest Match from Starting in A String

```
#!/bin/bash
string="abckhaliqbcxyz"
echo "${string#a*c}" # from starting, shortest match
```

Get Longest Match from Starting in A String

```
#!/bin/bash
string="abckhaliqbcxyz"
echo "${string##a*c}" # from starting, longest match
```

Get Shortest Match from the End

```
#!/bin/bash
string="abckhaliqbcxyz"
echo ${string%b*z} # from ending, shortest match
```

Get Longest Match from the End

```
#!/bin/bash
string="abckhaliqbcxyz"
echo "${string%%b*z}" # from ending, longest match
```

Replace First Occurrence of Character in String

```
#!/bin/bash
string="abckhaliqbcxyz"
echo "${string/abc/xyz}"
```

Replace All Occurrence of Character in String

```
#!/bin/bash
string="abckhaliqbcxyz"
```

```
echo  "${string//abc/xyz}"
```

Remove First Occurrence of Character in String

```
#!/bin/bash
string="abckhaliqbcxyz"
echo  "${string/abc}"
```

Remove All Occurrence of Character in String

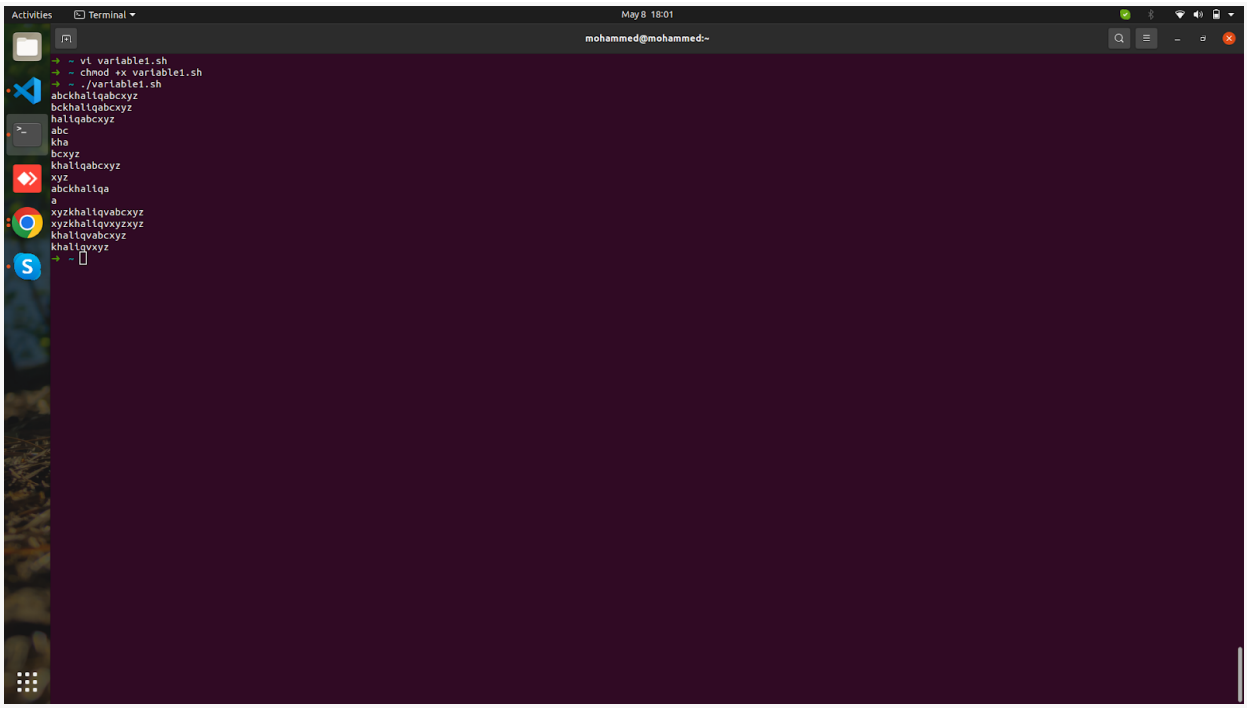
```
#!/bin/bash
string="abckhaliqcxyz"
echo  "${string//abc}"
```

Crate file by using touch variable1.sh
Vi variable1.sh

```
#!/bin/bash

string="abckhaliqabcxyz"

echo "${string:0}"
echo "${string:1}"
echo "${string:4}"
echo "${string:0:3}"
echo "${string:3:3}"
echo "${string: -5}"

echo "${string#a*c}"  # from starting, shortest match
echo "${string##a*c}" # from starting, longest match

echo ${string%b*z}  # from ending, shortest match
echo "${string%%b*z}" # from ending, longest match

string="abckhaliqvabcxyz"

echo "${string/abc/xyz}"
echo "${string//abc/xyz}"

echo "${string/abc}"
echo "${string//abc}"
```

Chmod +X varaible1.sh

./varaible1.sh



Set Default Value to Shell Script Variable

If parameter is unset or null Set Default Value

${parameter:-word}

If parameter is unset or null, the expansion of word is substituted. Otherwise, the value of parameter is substituted.

If parameter is unset then Set Default Value

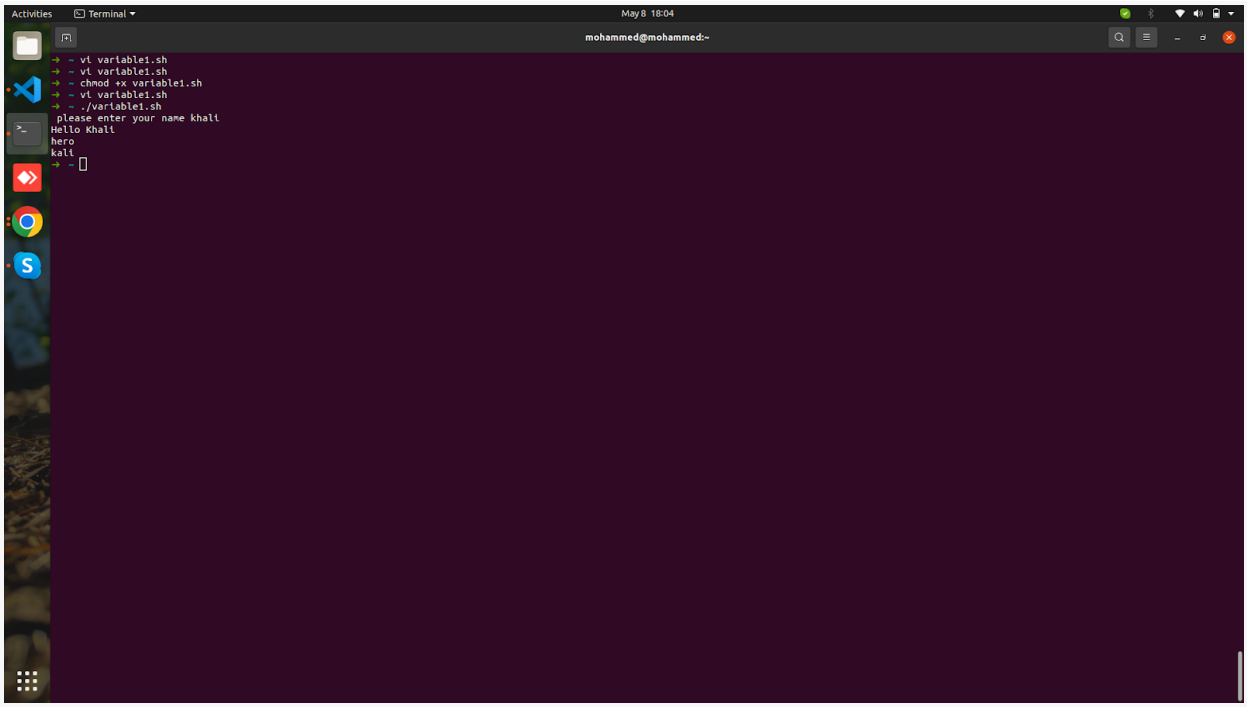
${parameter-word}

If parameter is unset, the expansion of word is substituted. Otherwise, the value of parameter is substituted.

```
 Create the folder touch variable1.sh
Vi variable.sh


#!/bin/bash
```

```
read -p " please enter your name " name

name=${name:-World}

echo "Hello ${name^}"

yourname=${unsetvariable-hero}
echo $yourname

myname=""
mytestname=${myname:-kali}
echo ${mytestname}
~
```

```
Chmode +x variable1.sh

./varaible1.sh
```



Arithmetic Operations in Shell Script

We can do arithmetic operations in shell script in a serval way (using let command, using expr command) but we will recommend using brackets for that.

Different ways to compute Arithmetic Operations in Bash

1. **Using Double Parenthesis**
2. **Using let command**
3. **using expr command**

Using Double Parenthesis

Addition

```
Sum=$((20+2))
echo  "Sum = ${Sum}"  # output will be 22
```

Subtraction

```
sub=$((29-2))
echo "sub = ${sub}" # output will be 27
```

Multiplication

```
mul=$((20*4))
echo  "Multiplication = ${mul}"    # output will be 80
```

Division

```
div=$((10/3))
echo  "Division = ${div}"  # output will be 3
```

Modulo

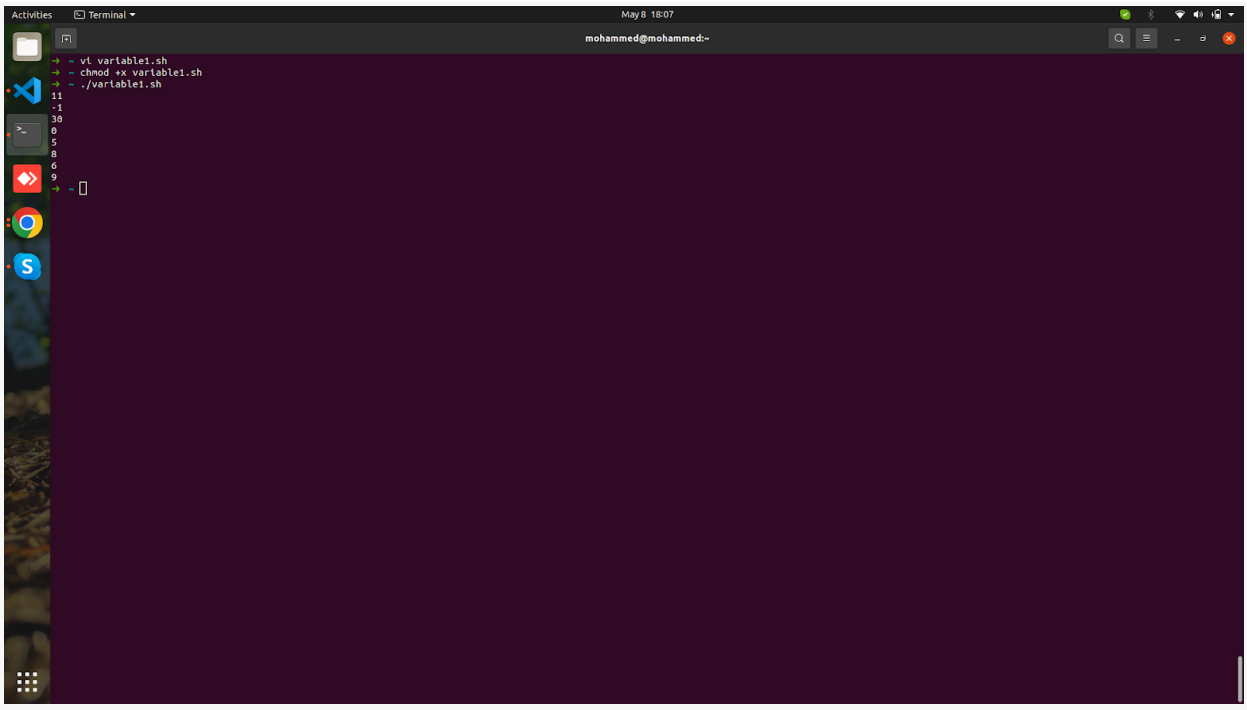
```
mod=$((10%3))
echo  "Modulo = ${mod}"    # output will be one.
```

Exponentiation

```
exp=$((10**2))
echo  "Exponent = ${exp}"  # output will be 100.
```

```
Create a touch file

Touch varaible1.sh


Vi Varaible1.sh
```

```
#!/bin/bash
a=5
b=6
echo  "$((a+b))"
echo  "$((a-b))"
echo  "$((a*b))"
echo  "$((a/b))"  # 5/6
echo  "$((a%b))"
echo  "$((2**3))"  # 2*2*2
((a++)) # a=a+1
echo  $a
((a+=3)) # a=a+3
echo $a

Chmod +x variable1.sh

./varaible1.sh
```



Functions In Shell Script

Shell Functions are used to specify the blocks of commands that may be repeatedly invoked at different stages of execution.

The main advantages of using unix Shell Functions are to reuse the code and to test the code in a modular way.

The purpose of the function

- **Don't repeat yourself.**
- **Write once, use many times.**
- **Reduce the script length.**
- **A single place to edit and troubleshoot.**
- **Easier to maintain.**
- **If you are repeating yourself, use a function.**
- **Reusable code.**
- **Must define before use.**
- **Has parameters supports.**
- **The best practice is to put all the functions on top of the script.**

How to Create Function in Shell Script

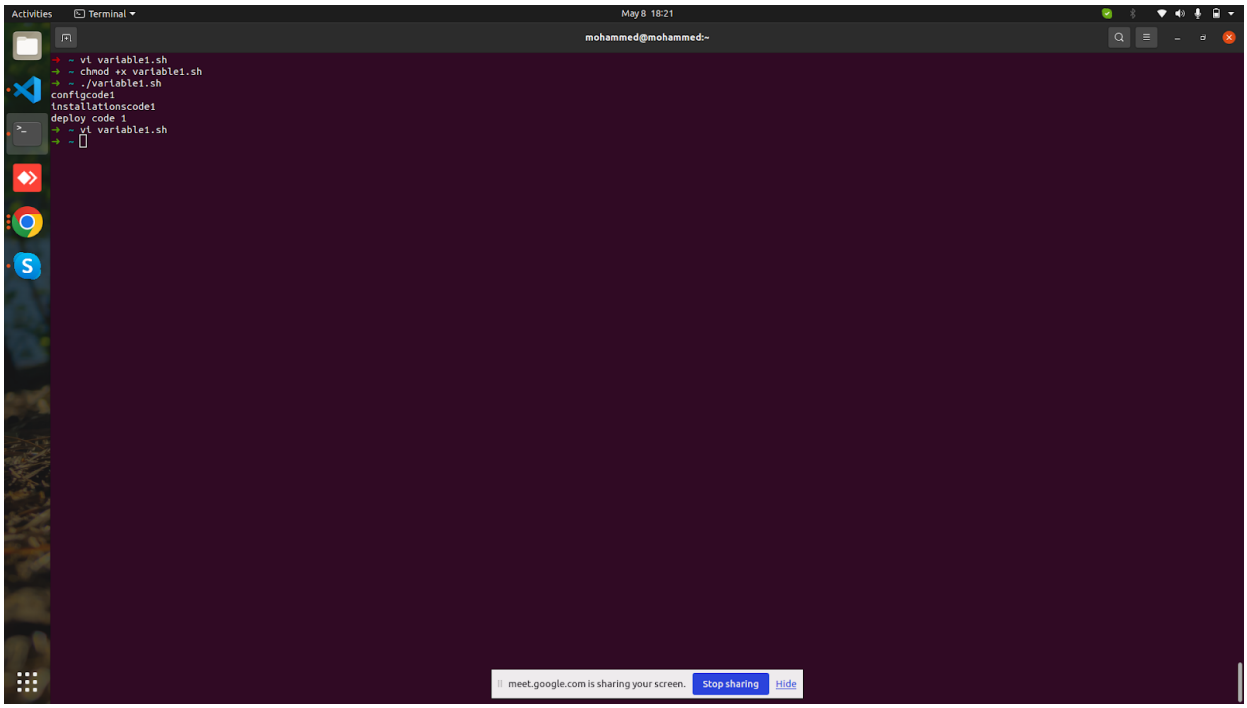
In Shell script, we can write functions in a variety of different ways.

```
# type one.
function function_name(){
        # code goes here
}


# type two
function_name(){
        # code goes here
}
# method three
function function_name {
     # function code here.
}
```

Simply use the function name as a command to run a function.

```
# invode the function
function_name
```

```
#!/bin/bash

# funtions
function install(){
    #### installations code.
    echo "installationscode1"

}
configuration(){
    # configurations code
    echo "configcode1"
}


function deploy {
    # deploy code.
    echo "deploy code 1"
}
configuration
install
deploy

Chmod +x variable1.sh
./varaible1.sh
```



Pass Parameters to a Function

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

Functions can be Recursive.

A function may return a value in one of four different ways:

- **Change the state of a variable or variables**
- **Use the return command to end the function, and return the supplied value to the calling section of the shell script**
- **echo output to stdout, which will be caught by the caller just as c=`expr $a + $b` is caught**
- **we can get the function name using FUNCNAME variable.**

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 −

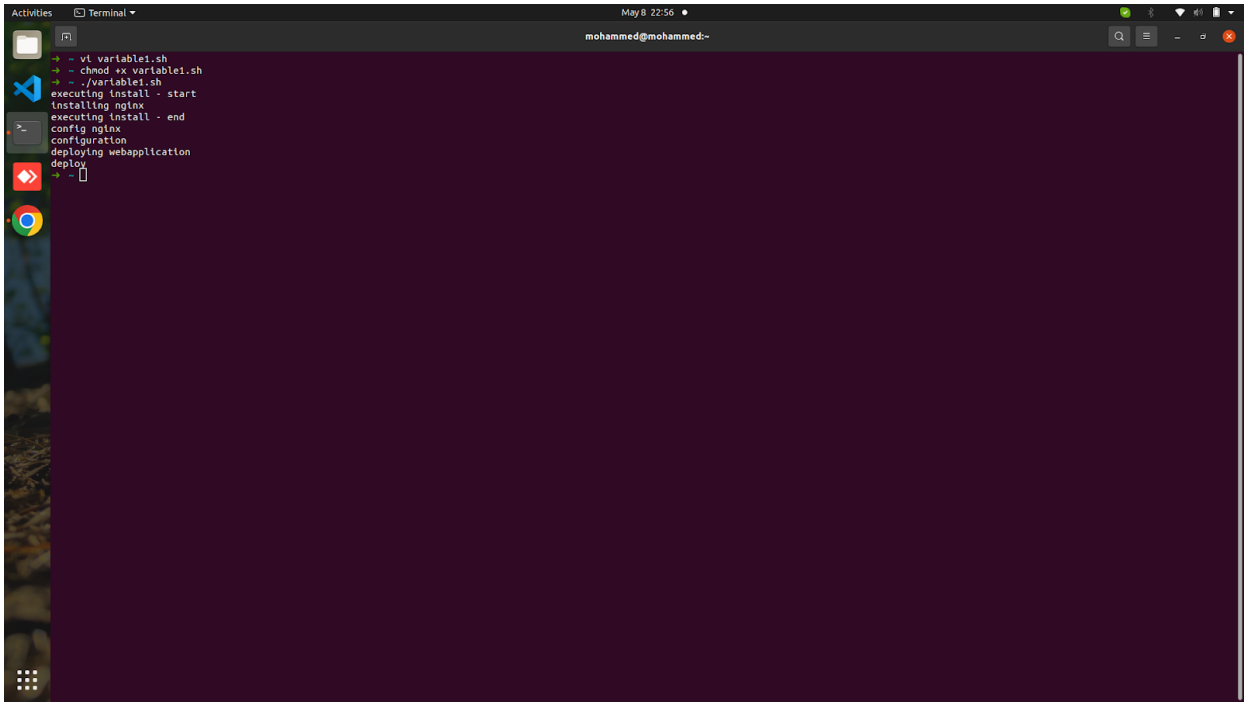
Touch varaible1.txt
Vi varaible1.txt

```
#!/bin/bash

function install(){
        echo "executing ${FUNCNAME} - start"
        echo "installing ${1}"
        echo "executing ${FUNCNAME} - end"
}
function configuration(){
        echo "config ${1}"
        echo "${FUNCNAME}"


}

function deploy() {
        echo "deploying ${1}"
        echo "${FUNCNAME}"
```

```
}
install "nginx"
configuration "nginx"
deploy "webapplication"
```

Chmod +x variable1.txt

./variable1.txt



Create Local Variable In Shell Script

All variables are global by default.

Modifying a variable in a function changes it in the whole script.

This could lead to issues.

local command can only be used within a function.

It makes the variable name have a visible scope restricted to that function and its children only.

All function variables are local. This is a good programming practice.

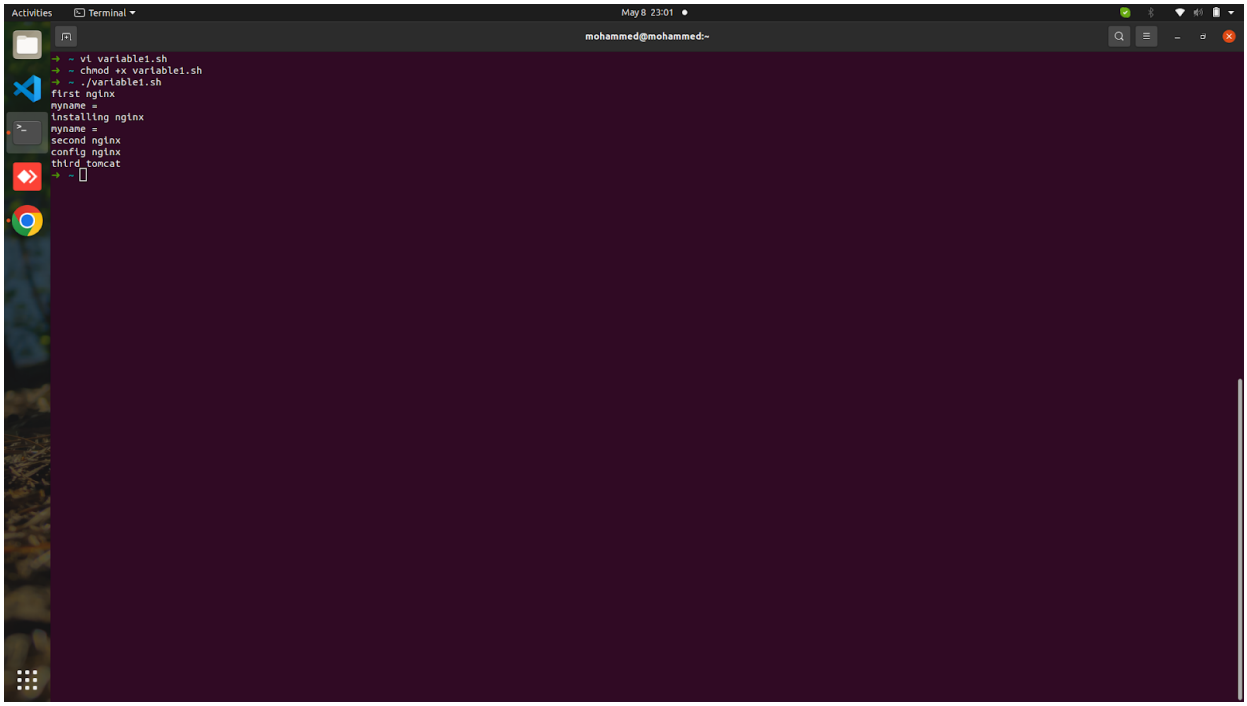
Touch varaible1.sh

Vi variable.sh

```
#!/bin/bash
packageName="nginx"
function install(){
        local myname="khaliq"
        echo "installing ${1}"
}

function configuration(){
        packageName="tomcat"
        echo "config ${1}"
}

echo "first ${packageName}"
echo "myname = ${myname}"
install "${packageName}"
echo "myname = ${myname}"
echo "second ${packageName}"
configuration "${packageName}"
echo "third ${packageName}"
```

Chmod +x varaible1.sh

./varaible.sh



If With Command

This block will process if the exit status of COMMAND is zero(or a command executed successfully).

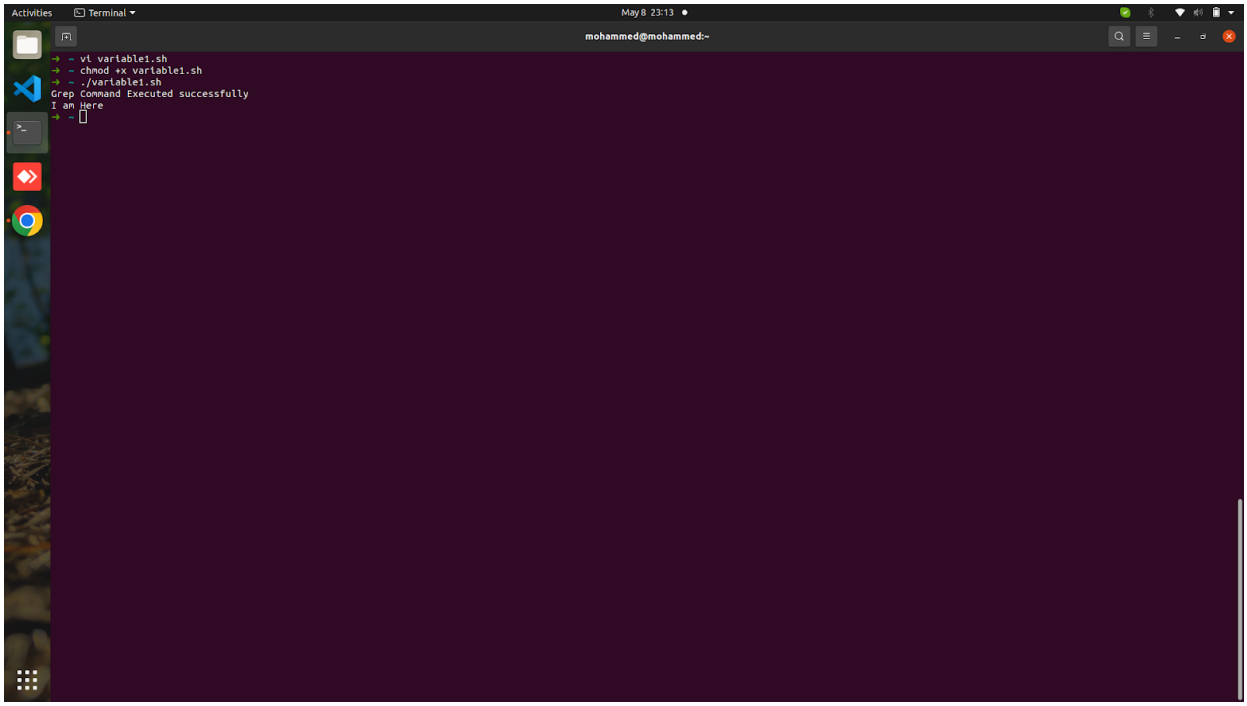
Touch varaible1.sh

Vi varaible1.sh

```
#!/bin/bash
if grep -i localhost /etc/hosts>/dev/null
then
        echo  "Grep Command Executed successfully"
fi
echo  "I am Here"
```

Chmod +x varaible1.sh

./varaible1.sh



Not Operator in ShellScript

The NOT logical operator reverses the true/false outcome of the expression that immediately follows.

For example, if a file does not exist, then display an error on the screen.

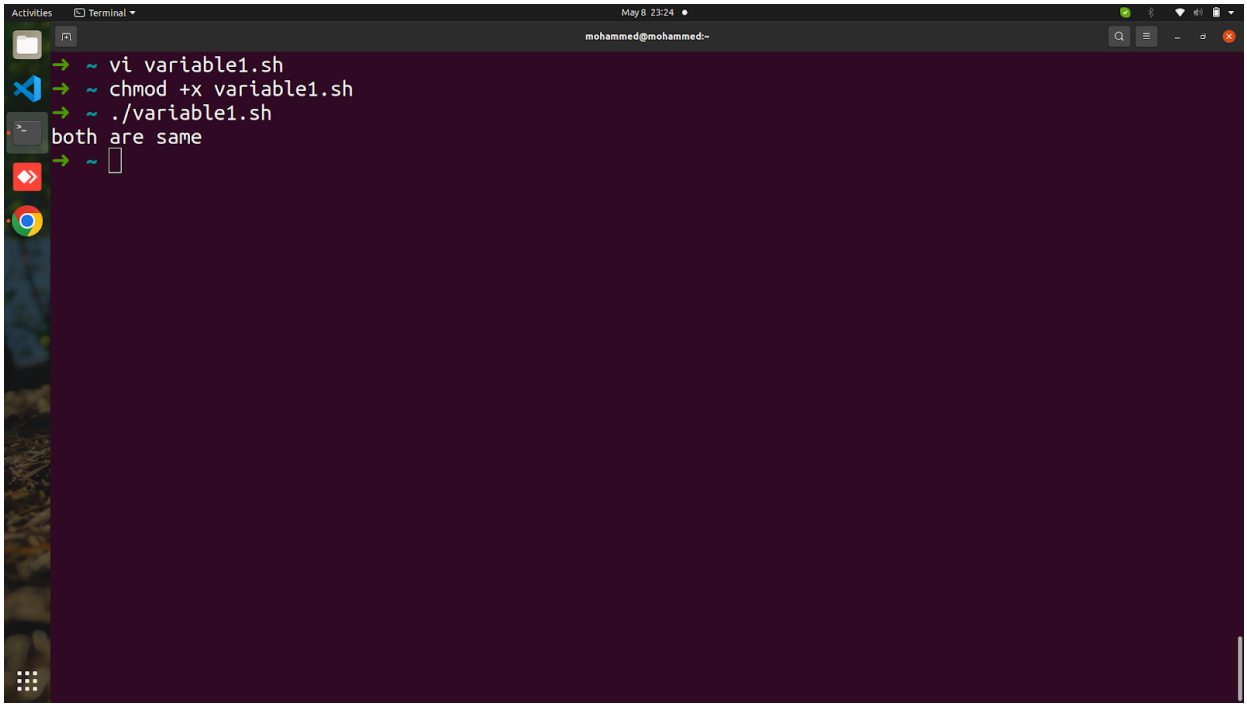
Touch varaible1.sh

Vi variable.sh

```
#!/bin/bash
name="smohammed abdulkhaliq"
othername="mohammed khaliq"
if [[ ! ${othername} == ${name} ]]
then
        echo  "both are same"
fi
```

Chmod +x variable.sh

./varaiable.sh



OR (||) Operator in ShellScript

This is logical OR. If one of the operands is true, then the condition becomes true.

Syntax:

```
command1 && command2
```

Touch varaible1.sh

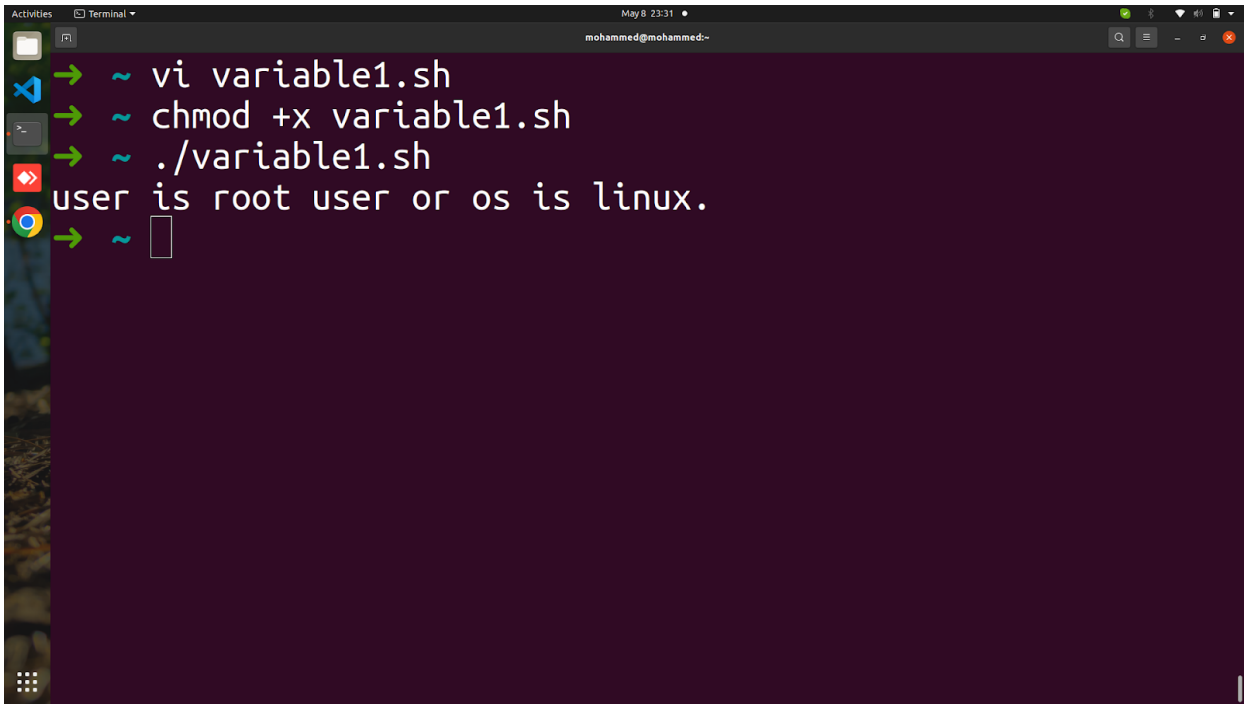
Vi varaible1.sh

```
#!/bin/bash
# os == linux && user == root
OS_TYPE=$(uname)
if [[ ${OS_TYPE} == "Linux" ||  ${UID} -eq 0 ]]
then
```

```
    echo  "user is root user or os is linux."
fi
```

Chmod +x variable1.sh

./varaible1.sh



If-Else in ShellScript

If the specified condition is not true in the if part then the else part will be executed.

Syntex:

```
if [ expression ]
then
   statement1
else
   statement2
fi
```

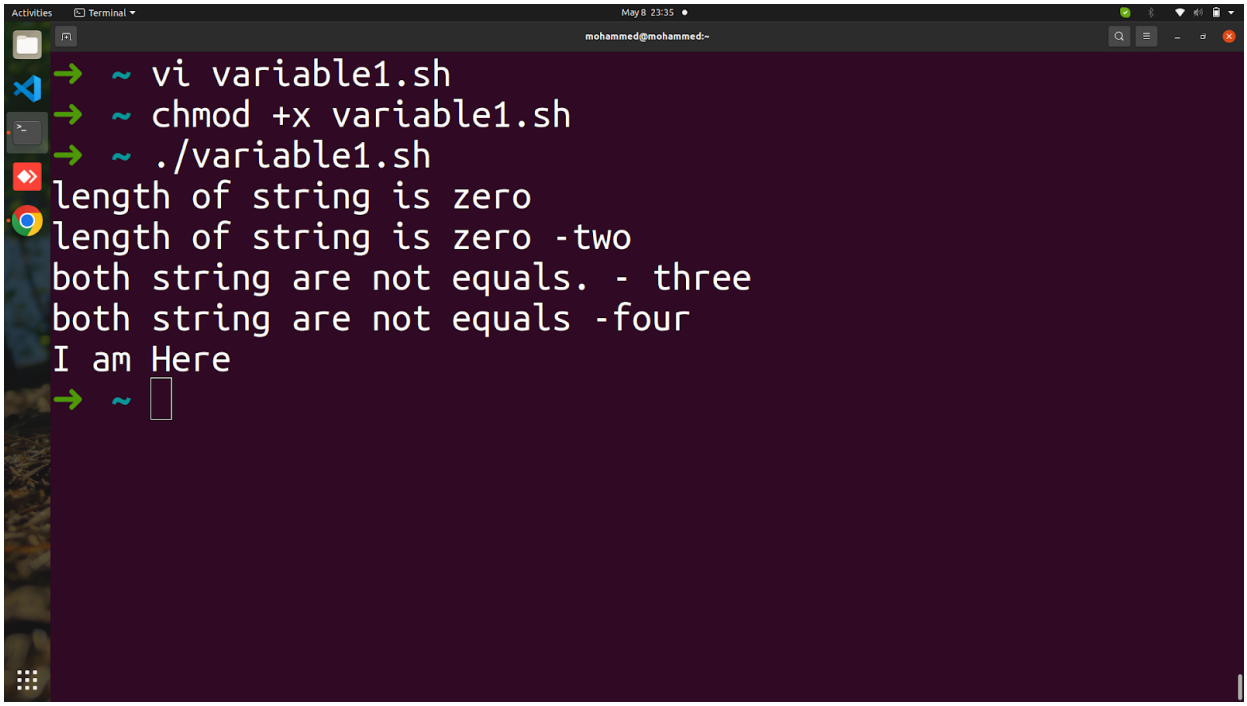
Touch variable1.sh

Vi variable1.sh

```
#!/bin/bash
name=""
othername="mohammed abdulkhaliq"

if [[ -n ${name} ]]
then
        echo  "length of string is non zero"
else
        echo  "length of string is zero"
fi

if [[ -z ${name} ]]
then
        echo  "length of string is zero -two"
else
        echo  "length of string is non zero. = two"
fi

if [[ ${name} == ${othername} ]]
then
        echo  "both string are equals - three"
else
        echo  "both string are not equals. - three"
fi

if [[ ${name} != ${othername} ]]
then
        echo  "both string are not equals -four"
else
        echo  "both strings are equals -four"
fi

echo  "I am Here"
```

Chmod +x varaible1.sh

./varaible1.sh



Nested If-Else

we can define if-else inside if-else.

A nested if-else block can be used when one condition is satisfied then it again checks another condition.

Touch variable1.sh

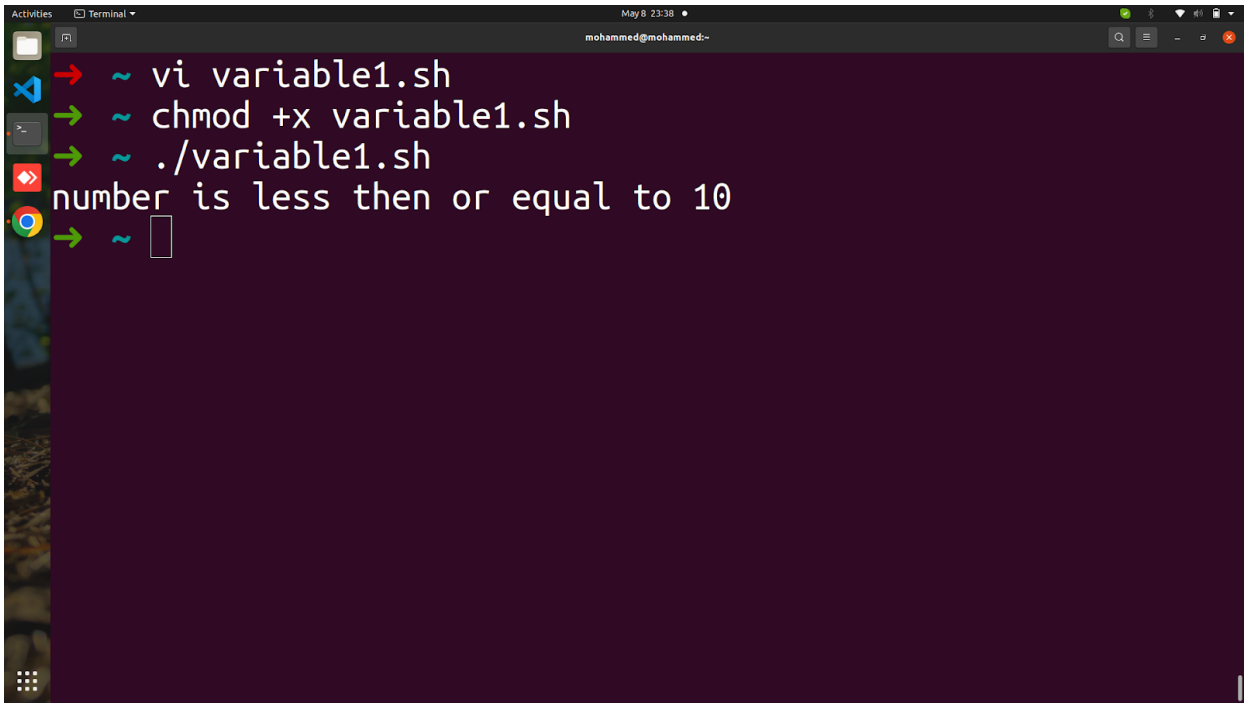
Vi varaible1.sh

```
#!/bin/bash
number=9
if [[ ${number} -gt 10 ]]
then
    if [[ $number -gt 50 ]]
    then
        if [[ ${number} -gt 100 ]]
```

```
            then
                  echo  "number is grater then 100"
            fi
      else
            echo  "number is in between 11 to 50"
      fi
else
      echo  "number is less then or equal to 10"
fi
```

Chmod +x varaible1.sh

./varaiable1.sh



Elif in ShellScript

if..elif..else..fi statement (Else If ladder)

To use multiple conditions in one if-else block, then elif keyword is used in shell. If the expression1 is true then it executes statements 1 and 2, and this process continues. If none of the conditions is true then it processes else part.

Syntax

```
if [ expression1 ]
then
   statement1
   statement2
   .
   .
elif [ expression2 ]
then
   statement3
   statement4
   .
   .
else
   statement5
fi
```

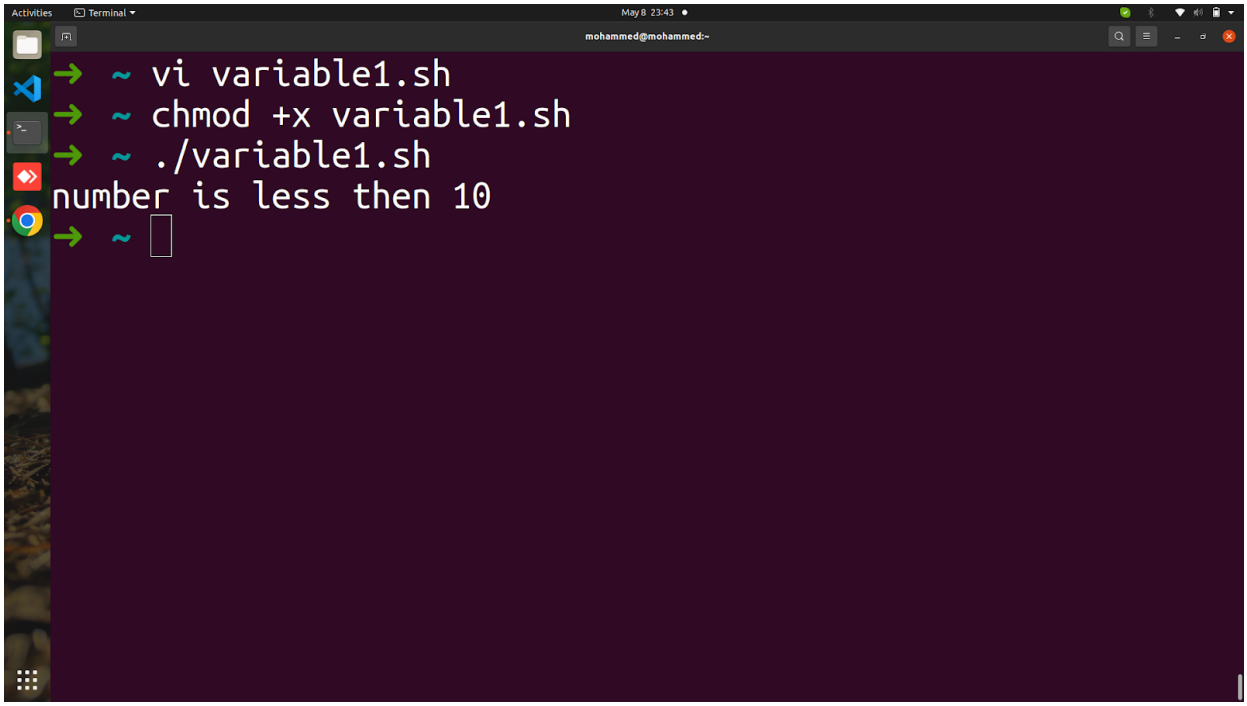
Touch varaible1.sh

Vi varaibale1.sh

```
#!/bin/bash
number=4

if [[ ${number} -eq 10 ]]
then
      echo "number is 10"
elif [[ ${number} -lt 10 ]]
then
      echo "number is less then 10"
elif [[ ${number} -lt 5 ]]
then
      echo "number is less then 5"
else
      echo "number is grater then 10"
fi
```

Chmod +x variable.sh

./varaible1.sh



While Loop in ShellScript

A while loop is a statement that iterates over a block of code till the condition specified is evaluated to true. We can use this statement or loop in our program when do not know how many times the condition is going to evaluate to false before evaluating to true.

This repeats until the condition becomes false.

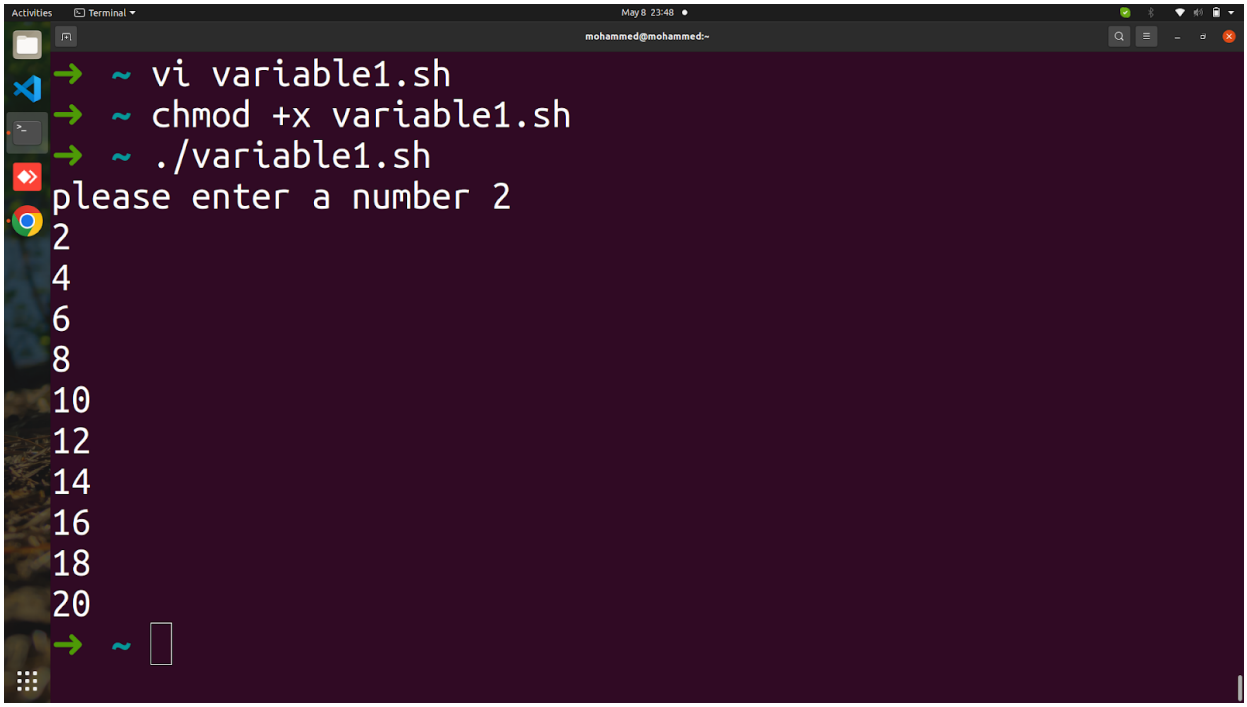
Touch varaiba,e1.sh

Vi varaible1.sh

```
#!/bin/bash
read -p "please enter a number " number
initNumber=1
while [[ ${initNumber} -le 10 ]]
do
    echo $((initNumber*number))
    ((initNumber++))
done
```

Chmod +xvaraible1.sh

./varaible1.sh



Read File in ShellScript

we can read a file with the help of read and while.

The return code of read command is zero, unless the end-of-file is encountered.

Touch variable1.sh

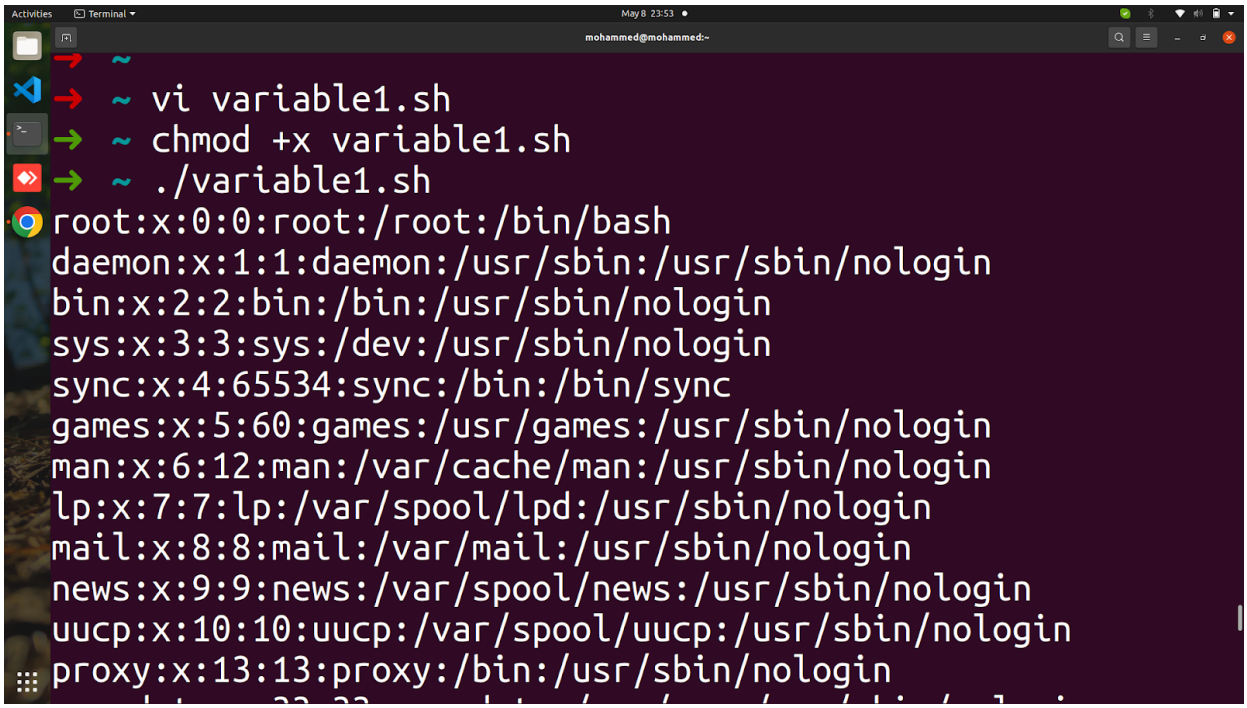
Vi varaible1.sh

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

```
file_path="/etc/passwd"
while read line
do
      echo "$line"
      sleep 0.20
done < $file_path
```

Chmod +x varaible1.sh

./varaible1.sh



Until Loop

The Until loop is used to iterate over a block of commands until the required condition is false.

Syntax:

```
until [ condition ];
do
 block-of-statements
done
```

Here, the flow of the above syntax will be –

- Checks the condition.
- if the condition is false, then executes the statements and goes back to step 1.
- If the condition is true, then the program control moves to the next command in the script.

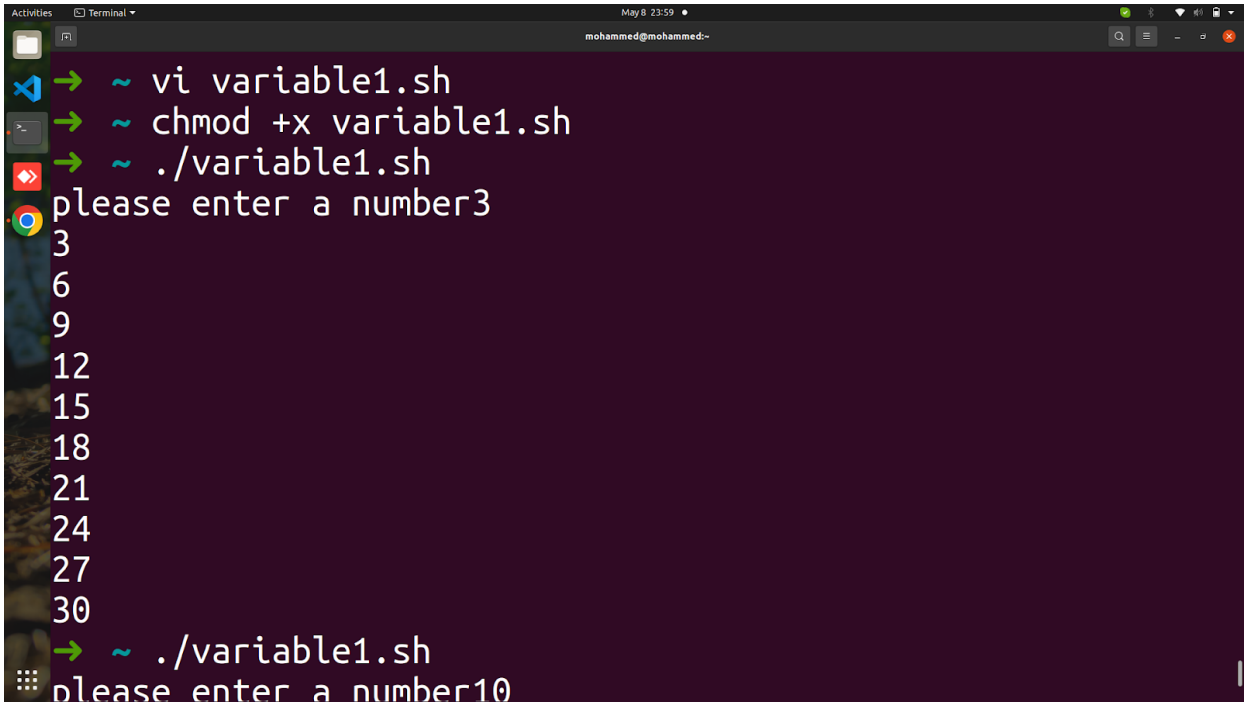
Touch varaible1.sh

Vi varaible1.sh

```
#!/bin/bash
read -p "please enter a number" number
initNumber=1

until [[ initNumber -eq 11 ]]
do
    echo $((initNumber*number))
    ((initNumber++))
done
```

Chmod +x variable.sh

./varaible1.sh



For Loop

The for loop moves through a specified list of values until the list is exhausted.

- Keywords are for, in, do, done
- List is a list of variables which are separated by spaces. If list is not mentioned in the for statement, then it takes the positional parameter value that were passed into the shell.
- Varname is any variable assumed by the user.

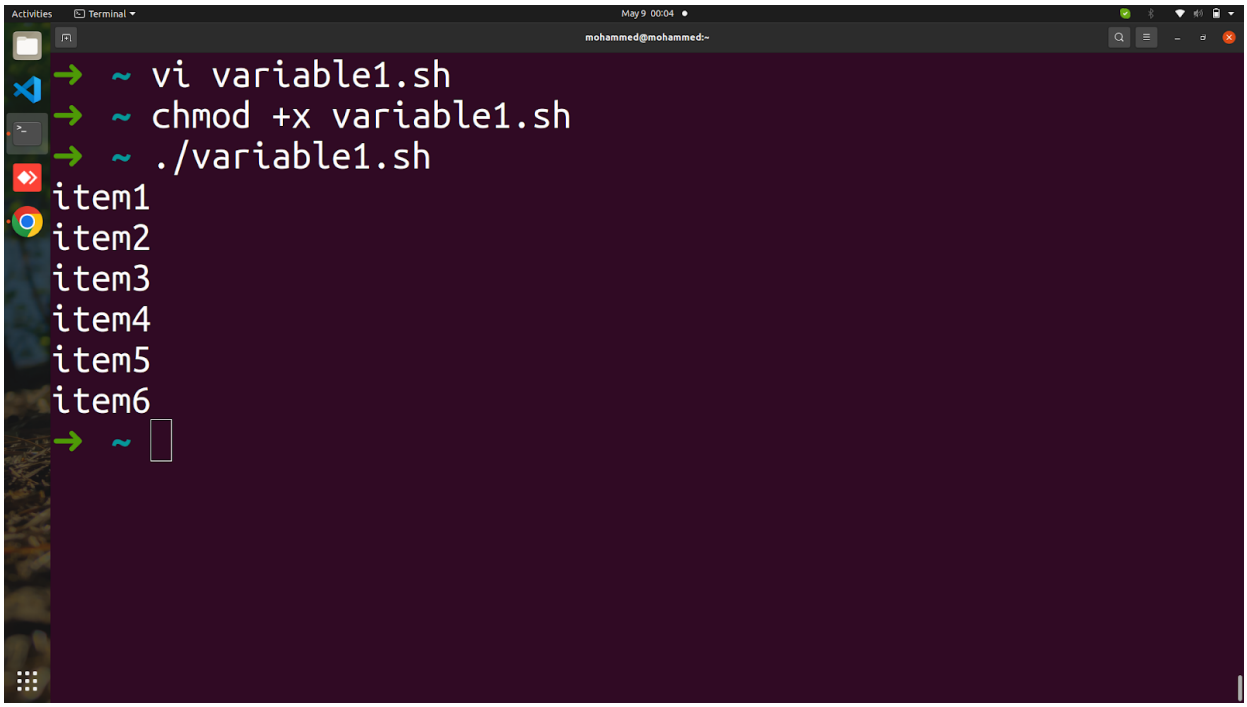
Touch varaiable1.sh

Vi varaible1.sh

```
#!/bin/bash

for variableName in item1 item2 item3 item4 item5 item6
do
    echo "${variableName}"
done
```

Chmod +x variable1.sh

./varaible1.sh



$@ and $* in ShellScript

$@ behaves like $* except that when quoted the arguments are broken up properly if there are spaces in them.

Take this script for example (taken from the linked answer):

Touch varaible1.sh

Vi varaible1.sh

```
#!/bin/bash

echo "========== loop one ========"
for i in "$*"
do
        echo $i
```

```
done

echo  "=========== loop two ========"
for  i  in  "$@"
do
      echo  $i
done
```

Chmod +x variable1.sh

./varaible1.sh mohammed abdul khaliq

Select Loop in ShellScript

The select loop is an infinite loop that only ends when there's a keyboard interrupt or a break statement is encountered. But that's not what makes it unique or interesting. The select statement allows users to choose from multiple options by default and it will prompt the user for an input. You do not have to write any code to accept user input as the select loop is pre-built to handle it.

This loop can be used to make menus within your script while keeping the script looping infinitely. Another benefit of the select loop in shell scripts is that it can be combined with the switch case statements to create really interactive menus or script pivots. Let's learn how to make use of this loop and work with it.

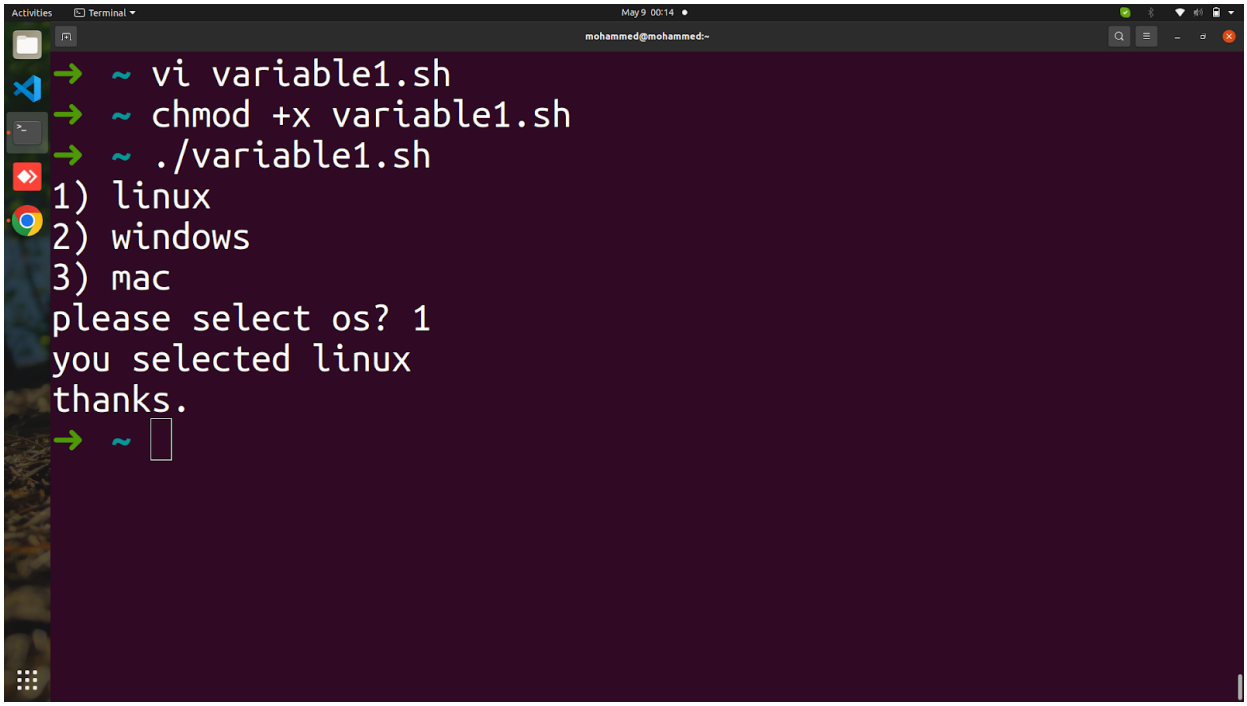
Touch varaible1.sh

Vi varaible1.sh

```
#!/bin/bash
PS3="please select os? "
select os in linux windows mac
do
    case ${os} in
        linux)
            echo "you selected linux"
            echo "thanks."
            break
            ;;
        windows)
            echo "you selected windows"
            echo "thanks."
            break
            ;;
        mac)
            echo "you selected mac"
            echo "thanks."
            break
            ;;
        *)
            echo "Invalid"
    esac
done
```

Chmod +x varaible1.sh

./varaible1.sh



Continue statement in ShellScript

The continue statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

Syntax

```
continue
```

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

```
continue n
```

Here n specifies the nth enclosing loop to continue from.

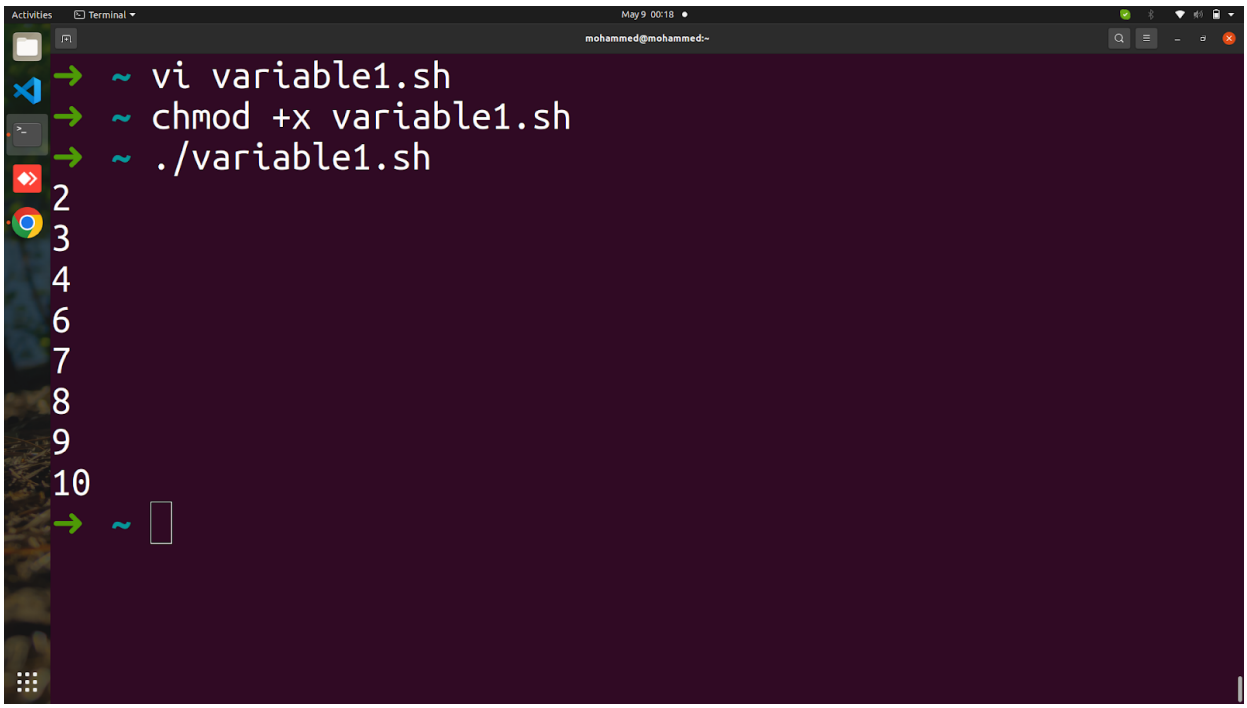
Touch varaible1.sh

Vi varaible1.sh

```
#!/bin/bash
initNumber=1
while [[ ${initNumber} -lt 10 ]]
do
    ((initNumber++))
    if [[ ${initNumber} -eq 5 ]]
    then
        continue
    fi
    echo ${initNumber}
done
```

Chmod +x varaible1.sh

./varaible1.sh



Break Statement in Shellscript

The break statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Syntax

The following break statement is used to come out of a loop −

```
break
```

The break command can also be used to exit from a nested loop using this format −

```
break n
```

Here n specifies the nth enclosing loop to the exit from.

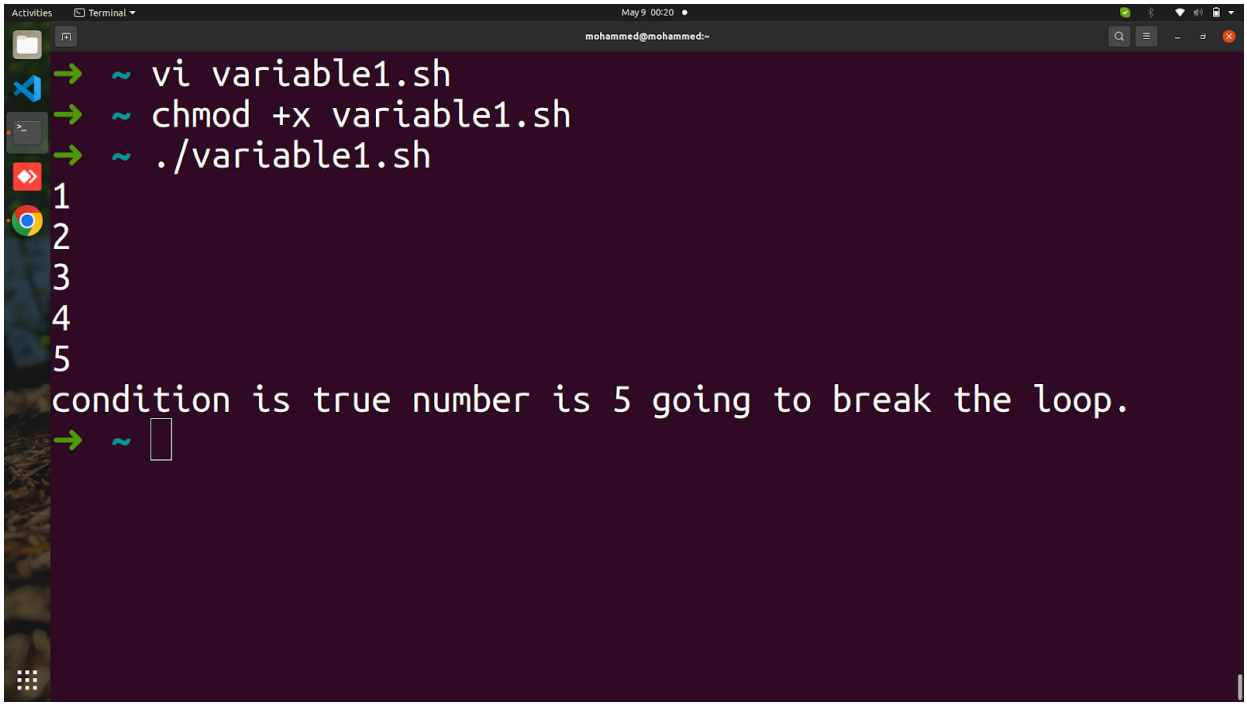
Touch varaible1.sh

Vi variable1.sh

```
#!/bin/bash
initNumber=1
while [[ ${initNumber} -lt 10 ]]
do
    echo ${initNumber}
    if [[ ${initNumber} -eq 5 ]]
    then
        echo "condition is true number is ${initNumber} going to break the loop."
        break;
    fi
    ((initNumber++))
done
```

Chmod +x variable.sh

./varaible1.sh



Nested Loop In ShellScript

Nested for loops means loop within loop. They are useful for when you want to repeat something several times for several things.

Touch varaible1.sh

Vi varaible1.sh

```
#!/bin/bash
initNumber=1
while [[ ${initNumber} -lt 3 ]]
do
    for i in item1 item2 item3
    do
        echo "${initNumber} - ${i}"
    done
    ((initNumber++))
done
```

Chmod +x variable1.sh

./varaible1.sh