Ghulam Ishaq Khan Institute of Engineering Sciences and Technology

Operating System Lab - 03

Lab Manual

Contents

Objective

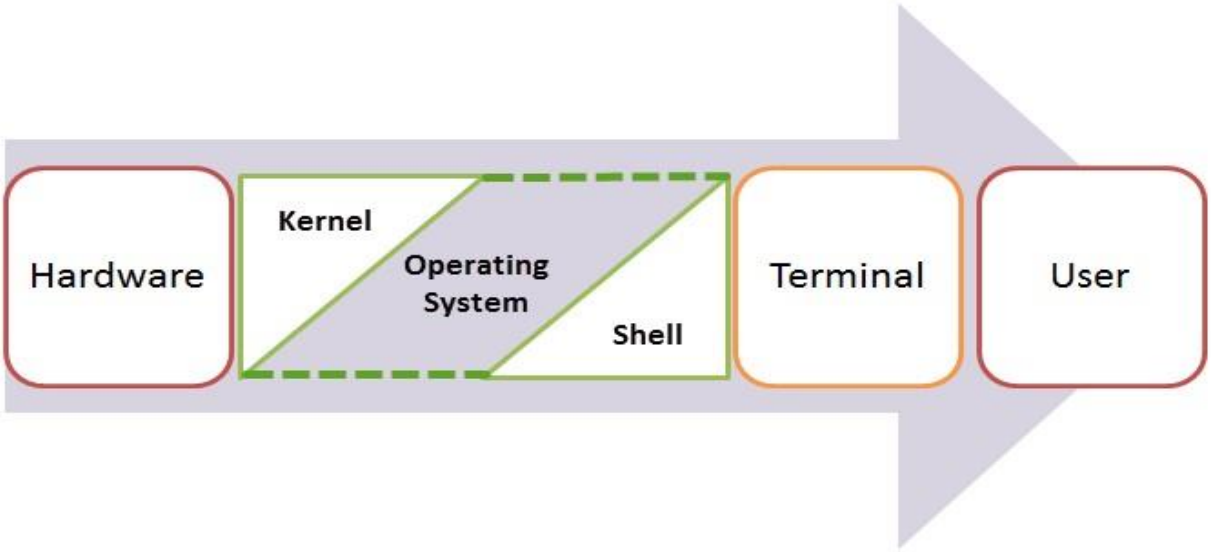
The lab examines the use of shell scripts as a way to write programs that accomplish various forms of processing in a Linux environment.

Shell Scripting

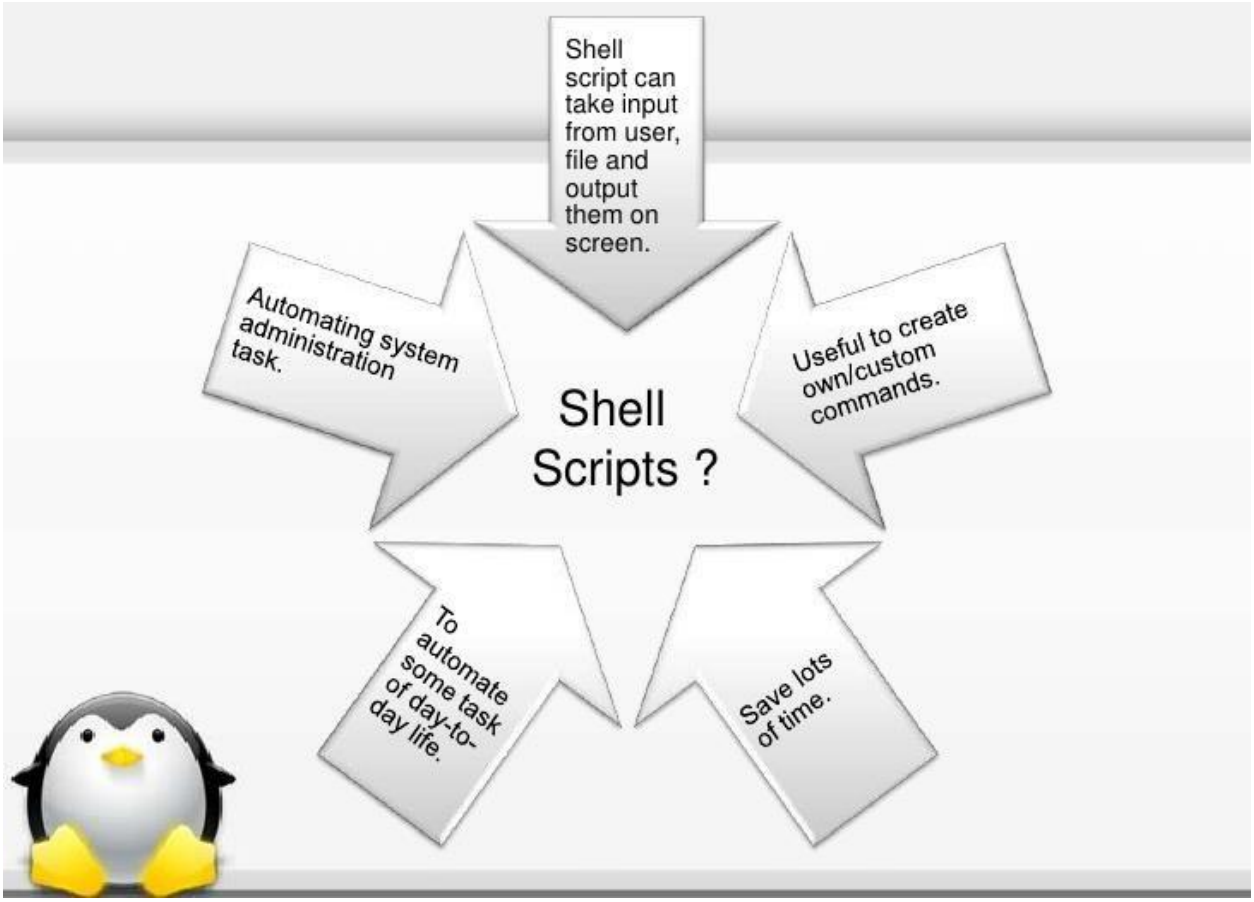
We have already talk about shell programming/scripting in *Lab Manual 02 – Introduction to Shell Scripting*. A shell script can be viewed as a high level program that is created with a simple text editor. Once created, a user may execute a shell script by simply invoking the filename of the shell script**. It is unnecessary to compile the shell script first. Rather, each time the shell script is invoked, a shell interprets or compiles the shell script as it executes it.**

To Sum up we can say in other words,

"Shell programming is a group of commands grouped together under single filename. The shell can be used either interactively - enter commands at the command prompt, or as an interpreter to execute a shell script. Shell scripts are dynamically interpreted, NOT compiled."



A shell script invokes commands that you can issue through a command-line. In addition, a shell script also allows for more sophisticated processing, such as decision making and repetition for the work it invokes. In this manner, a shell script can be written to accomplish a series of tasks more easily and quickly than if the user had to type the commands for the tasks every time. Shell scripts are also a common way to schedule jobs to be automatically invoked by the system at specific times.



To create a shell script file, create a new file with any name and with extension '.sh'. Note you can also create the shell script file without the extension specified but then the file editor e.g. gedit's content will no longer be content-aware. The demonstration of creating the file is shown below:

```
student@oslab-vm:~/Desktop$ nano shelly.sh
student@oslab-vm:~/Desktop$ cat shelly.sh
echo "hellow from shelly shell script";
student@oslab-vm:~/Desktop$
```

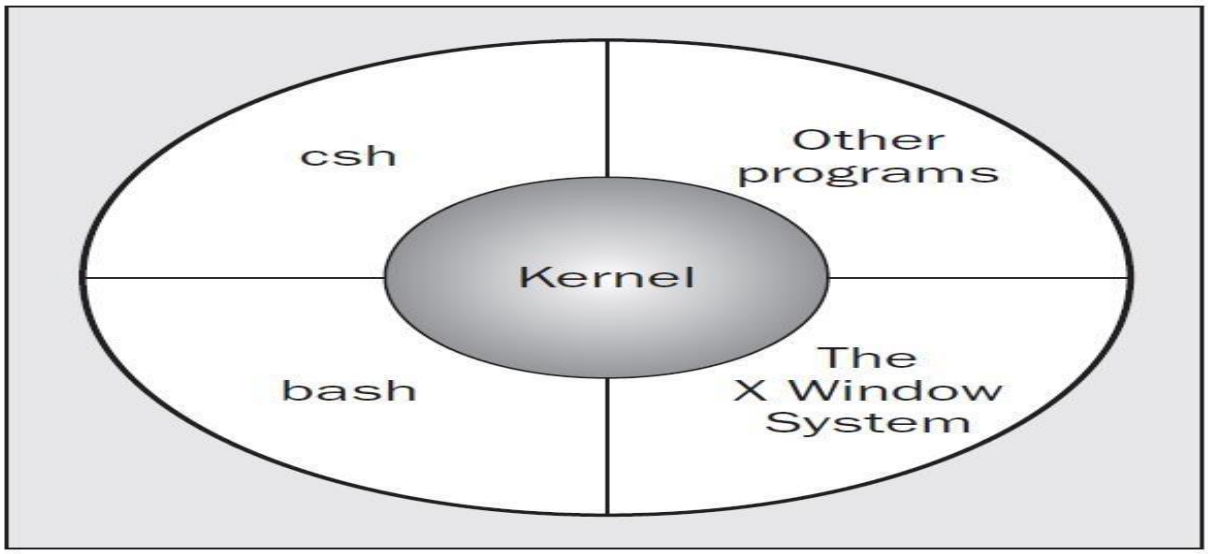
To invoke a shell script simply type in your terminal: './' followed by the filename with the extension '.sh'. For example, if I want to execute shell script having filename: 'shelly.sh' I would execute it as shown in the screenshot below.

Make sure the file i.e. in my case 'shelly.sh' is executable. In case the file is not executable you need to add access permission using 'chmod' command which we discussed in *lab manual 02*.

```
student@oslab-vm:~$ chmod +x Desktop/shelly.sh
student@oslab-vm:~$ ls -l Desktop/shelly.sh
-rwxrwxr-x 1 student student 40 Aug 27 01:33 Desktop/shelly.sh
student@oslab-vm:~$ ./Desktop/shelly.sh
hellow from shelly shell script
student@oslab-vm:~$ ./shelly.sh
bash: ./shelly.sh: No such file or directory
student@oslab-vm:~$ cd Desktop
student@oslab-vm:~/Desktop$ ./shelly.sh
hellow from shelly shell script
student@oslab-vm:~/Desktop$
```

Notice the path when I invoke shell script. The first one is: ‘./Desktop/shelly.sh’ which specified that in folder Desktop there is an executable filename ‘shelly.sh’ which it simply executed. Now when I tried ‘./shelly.sh’ it gave me the error of no such file or directory because the parent directory of Desktop does not contain that file. After changing directory to Desktop, I retried the command and it executed. Hence in invoking a shell script you need to specify the path where the filename is. Otherwise the system won’t be able to find it and/or execute it.

Shell Name	A Bit of History
sh (Bourne)	The original shell from early versions of UNIX.
csh, tcsh, zsh	The C shell, and its derivatives, originally created by Bill Joy of Berkeley UNIX fame. The C shell is probably the third most popular type of shell after bash and the Korn shell.
ksh, pdksh	The Korn shell and its public domain cousin. Written by David Korn, this is the default shell on many commercial UNIX versions.
bash	The Linux staple shell from the GNU project. bash, or Bourne Again SHell, has the advantage that the source code is freely available, and even if it’s not currently running on your UNIX system, it has probably been ported to it. bash has many similarities to the Korn shell.



```
student@oslab-vm:~/Desktop$ /bin/sh
$ date
Sun Aug 27 01:59:54 EDT 2017
$ date; pwd
Sun Aug 27 01:59:59 EDT 2017
/home/student/Desktop
$ echo 'hellow from alishah'; ./shelly.sh
hellow from alishah
hellow from shelly shell script
$ exit
student@oslab-vm:~/Desktop$ /bin/sh
$ message="This is operating system lab 03";
$ echo $message;
This is operating system lab 03
$ pwd; date; echo $message; ./shelly.sh;
/home/student/Desktop
Sun Aug 27 02:11:30 EDT 2017
This is operating system lab 03
hellow from shelly shell script
$
```

Variables

Like most other programming languages, a shell script can use variables to store data for future retrieval. The names for shell script variable may consist of alphabetic letters, underscores, or digits (but not in the beginning). Letters are case sensitive. There are two types of variables used in shell programming.

1. System Variables
2. User Defined Variables

System variables are created and maintained by Linux. These types of variables will be denoted in upper case letters. To get the details of available system variables, issue the command $set. Users variables are defined by users. They are usually defined in lower case letters.

To store a value in a variable within a shell script, specify the assignment as shown below.

```
X=1
y=100.25
message='hellow world'
```

In the figure above, the value 1 is stored in x, the value 100.25 in y, and the string value 'hello world' in message. When a variable appears initially in a script in an assignment statement, the script automatically declares the variable with the appropriate data type to hold the assigned value.

To obtain the value stored in the variable, specify the $ symbol followed by the variable name. As shown below:

```
echo $message
newMessage=$message
echo “The message is: $newMessage”
```

In the example above, the first line shows how to write the value of the variable ‘message’ to the screen, the second line, assigns the value of the variable ‘message’ to the variable ‘newMessage’ and lastly, the third line shows how to write the value of a variable along with other text. Notice that the example uses double quotes to enclose the text and variable name. You may omit the double quotes, but you cannot use single quotes. The use of single quotes in this example will actually instruct echo to write the enclosed contents exactly as they appear.

Comments

A comment is a line or part of the line that not executed in any programming language. In shell script, a comment starts with the # symbol and continues to the end of that line. This implies that a comment may either take an entire line if # appears as the first character on any line in a shell script. Alternatively, it may appear on the same line with some other action, after the action specification. The example below will make this paragraph more meaningful.

```
# I am a comment, I will not be executed echo ‘hellow from comment’;
date; pwd #this line will be executed till the # symbol appears
```

#!/bin/sh is a special form of comment;
The #! characters tell the system that the argument that follows on the line is the program to be used to execute this file. In this case, /bin/sh is the default shell program.

‘echo’ & ‘read’ Statements

To send text, number or other information to the screen. You can use ‘echo’ command. It is the simplest form, you provide information you want as output as an argument to echo. You can also specify multiple arguments to output multiple arguments to output multiple pieces of information, one after the other, as shown below

```
echo “Hello, how are you today”
echo “this is first piece” “this is second piece” 123.45
echo “this is string” $variable
```

To take input from the keyboard into a shell script variable, you can use the read command followed by the variable. Examples below shows how a shell script can prompt the user for a value and read that value into a variable.

```
#printing a prompt and reading input

echo “Enter a number: ”

read num

echo “You have entered the number: $num”

#alternative of read to print a prompt and read input

read -p “Enter you first name: ” FirstName

echo “You have entered the first name: ” $FirstName

#utilizing read to print a prompt and read input into two variables

read -p “Enter your first name and last name: “ FirstName LastName

echo -n “First Name: $FirstName” “LastName: ” echo -n $LastName
```

Accessing Arguments

You can send arguments when you invoke a shell script, the arguments can be in any number. The shell store those arguments passed in variable $n where n = {1,2,3…9} and you pass the arguments just like in any other command. The below shell script and terminal shows the execution of a shell script which is using command line arguments.

```
echo “Argument 1: $1”
echo “Argument 2: $2”
echo “Argument 3: $3”
```

The screenshot below shows the execution of the above shell script.

Other arguments available in shell script and their usage/description is provided in the table below.

S.NO	Variable	Description
1	$0	The filename of the current script.
2	$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).

3	$#	The number of arguments supplied to a 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. If a script receives two arguments, $@ is equivalent to $1 $2.
6	$?	The exit status of the last command executed.
7	$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
8	$!	The process number of the last background command.

Arithmetic Operations

In shell script, you can perform arithmetic operations, in calculations or within the formula. You must enclose the formula or calculation with a set of double parentheses (( )). There should be no space between the two right and left parentheses. The example below shows how to use arithmetic operations.

```
Num=10
(( Double = 2 * $Num ))
echo “Twice the number is $Double”
(( DoublePlus1 = 2 * $Num + 1 ))
echo “Twice the Number Plus 1 is $DoublePlus1”
(( DoubleQPlus1 = 2 * ($Num +1) ))
echo “Twice the number plus 1 is $DoubleQPlus1”
(( half = $Num/2 ))
echo “Half the number is $half”
```

Some arithmetic operators which are commonly used is given below in the table:

S.NO	Name	Operator	Example
1	Assignment	‘=’	X=100, y=2.45
2	Addition	‘+’	((x=$x + 100))
3	Subtraction	‘-‘	((z=1+1+100))
4	Multiplication	‘*’	((abc=2*200))
5	Division	‘/’	((abc=10/5))
6	Increment	‘++’	((x++)), ((++x))
7	Decrement	‘--'	((x--)), ((--x))

Conditional Statements

Shell script can use conditions and branches to specify under what circumstances one or more statements should execute. One form of conditions with branches involves if statement structure. A decision structure requires one or more conditions to indicate when a branch should be executed. A condition in aa shell script can involve numeric values or text values. A condition is typically enclosed within a set of double brackets [[ ]]. There must be a space before and after each double bracket. Otherwise the script may not function properly.

When comparing numeric values or text values in a condition, you can use the comparison operators listed in the table below. Be careful to include space before and after the operator to

separate the operator from the variables and/or values on its left and right side. Otherwise, the shell may incorrectly evaluate the condition

S.NO	Operator	Equivalent	Description	Example
1	-eq	=	Equal to	[[$count -eq 10]]
2	-ne	!=	Not equal to	[[$total -ne 1000]]
3	-lt	<	Less than	[[$balance -lt 0]]
4	-le	<=	Less than or equal to	[[$C -le $B]]
5	-gt	>	Greater than	[[5 -gt 6]]
6	-ge	>=	Greater than or equal to	[[$total -ge $subtotal]]

Shell script supports logical operators such as OR and AND to specify a compound condition such as condition that contain multiple comparisons. You can also use logical not to negate a condition. Table below shows the logical operator with example of their use.

S.NO	Operator	Description	Example
1	||	OR	[[$x = 0 || $y = 0]]
2	&&	AND	[[$A > $B && $B > $C]]
3	!	NOT	[[! $sum = 0]]

Taking a look at a simple one-branch decision shown below, shows how to specify a simple branch statement with if-then-fi. The example below takes two numbers and reports if the first one is smaller than the other one.

```
read -p “Enter First Number: ” num1
read -p “Enter Second Number: ” num2
if [[ $num1 < $num2 ]]; then
    echo “$num1 is smaller than $num2”
fi
```

On the other hand, you can use two-branch statement in a manner where one branch is executed if the condition is true but another branch is executed when the condition is false. You can accomplish this by adding a two-branch statement. One can you if-then-else-fi statements. By adding else statement, the first branch states whether the first number is smaller than the other. But second branch reports when the first number is not smaller than the second. Example below will make the above statement much clearer.

```
read -p “Enter First Number: ” num1
read -p “Enter Second Number: ” num2
if [[ $num1 < $num2 ]]; then
        echo “$num1 is smaller than $num2”
else
        echo $num1 “is not smaller than” $num2
fi
```

For a situation that requires more than one condition and/or more than two branches. You can incorporate elif statement into one of the previous decision structure. The example below shows the use of if-then-elif-then-else statement. The condition for elif statement is same as if statement.

```
read -p “Enter First Number: ” num1
read -p “Enter Second Number: ” num2
if [[ $num1 < $num2 ]]; then
   echo “$num1 is smaller than $num2”
elif [[ $num1 > $num2 ]]; then
   echo $num1 “is greater than” $num2
else
   echo $num1 “is equal to” $num2
fi
```

While all the previous examples with if statement and conditions involved numeric variables and values, you can also compare text variables and text values with the operator listed previously. Such text comparisons refer to the ASCII value of the individual text characters on left and right sides. Here the first character is compared, followed by the second then the third and so on. Until it can determine whether the given comparison is true or false. Additionally, keep in mind that text is considered case sensitive and capital letters have lower ASCII values than the lowercase counterparts.

Consequently, a comparison of two string is true when both sides have exactly same text and case. The example below shows how to use text variable in string comparison however, the example below may exist potential error.

```
if [[ $YN = ‘YES’ ]]; then
   echo “Yes is specified”

else

   echo “You did not specify yes”
fi
```

However, imagine that $YN has no value then the condition becomes [[= ‘YES’]], resulting in a runtime error where the shell script does not execute correctly. To fix this problem you need to treat both sides of the comparisons as a string with an added letter. To ensure that each side contains at least one letter. The fix is demonstrated in an example below.

```
if [[ “x$YN” = ‘xYES’]]; then
        echo “Yes is specified”

else
        echo “You did not specify yes”
fi
```

In the script above we have added the letter ‘x’ on both sides however, it could have been any other letter. The reason of adding an extra letter on both sides is because how a shell script interprets the script contents as it executes the script content. In this case, when executing if statement using any variable the shell script uses its value rather than the variable. If the variable is empty, the side of that comparison is considered empty hence resulting in a runtime error generated by the shell executing that script.

In a situation where you wish to compare the text variable’s value without distinguishing of being upper case or lower case. One can use logical operator to compare against both lowercase and uppercase forms of the text. The example of this demonstration is shown below.

```
 read -p “Enter y (yes) or n (no): ” YN
if [[ “x$YN” = “xy” || “x$YN” = “xY” ]]; then
        echo “you specified yes”
else
        echo “you specified no”
fi
```

Another form of a condition containing numeric variables or values involves the test operator. It allows you to specify a numeric condition by omitting enclosing brackets entirely and preceding the word test. The example below demonstrates the test command to determine whether one numeric variable contain smaller value than the other one.

```
if test $num1 -lt $num ; then
        echo $num1 “is smaller than” $num2
else
        echo $num2 “is smaller than” $num1
fi
```

‘case’ Structure

Alternative form of shell script branching involves case structure. It allows the specification of a controlling value by a case statement. Following the case statement is a series of values or patterns, each with a branch of one or more statements. Like switch-case structure in C/C++, the

shell compares the controlling value against the values to find a match, starting with the first and continuing till the last. When a match is found, the shell executes the corresponding branch. You can also include a default branch that is executed if the controlling value does not mtch any of the values or pattern.

```
read -p “Enter the starting balance: ” balance
echo “Menu options”
echo “     a) Deposit”
echo “     b) Withdraw”; echo
read -p “Enter your selection: ” selection
case $selection in
          a|A) read -p “Enter the deposit amount: ” deposit
             ((balance = $balance + $deposit))
             ;;

          b|B) read -p “Enter the withdraw amount: ” withdraw
             ((balance = $balance - $withdraw))
             ;;
          *) echo “ERROR! Invalid input was provided”
             ;;
esac
echo “The final balance is: $balance”
```

Iterative Structure

To accomplish iteration in a shell script, you have a number of mechanisms available for that purpose. These involves while loops, for loops and a form of while loop called until. The syntax of these stated iterative loops is shown below.

Example of utilizing while loop, the below examples takes then number of directories to be created then using while loop it takes the names of those directories from user and tries to create them.

```
read -p “Enter the number of directories to be created: ” numDir
while [[ $numDir > 0 ]]
do
          read -p “Enter the name of the directory: ” dirName
          mkdir $dirName
          if [[ $? = 1 ]] ; then
                    echo “Directory creation failed”
          fi
          ((numDir--))
done
```

The utilization of for loop of the same script requirement is demonstrated in an example below.

```
for i in {1..4}
   do
      read -p “Enter the name of the directory: ” dirName
      mkdir $dirName
      if [[ $? = 1 ]] ; then
         echo “Directory creation failed”
      fi
   done
```

```
for i in hello 1 * 2 goodbye
do
  echo "Looping ... i is set to $i"
done

END=5
for i in $(seq 1 $END)
do
  echo "Looping ... i is set to $i"
done
```

The demonstration of the form of while loop aka until is shown below

```
read -p “Enter the number of directories to be created: ” numDir
until [[ $numDir -eq 0 ]]
do
        read -p “Enter the name of the directory: ” dirName
        mkdir $dirName
        if [[ $? = 1 ]] ; then
                echo “Directory creation failed”
        fi
        (($numDir--)
done
```

Functions

To help modularize and reuse shell script code, you can use functions within a shell script like you can in other programming languages. As a short example shows a shell script that defines a function named 'Min' that prints the smaller of two arguments. The script asks the user to input two numbers and calls the Min function to report the smaller number from the two provided.

```
Min() {
        if [[ $1 -lt $2 ]]; then
                Smallest=$1

else
                Smallest=$2
        fi
}
read -p "Enter two whole numbers, Separated by space: " N1 N2
Min $N1 $N2
echo "The smallest is: " $Smallest
```

At first line of a function definition, it specifies the function name and an empty set of parentheses. Unlike many other programming languages, the parentheses at the first line of a function definition are always empty, whether or not you supply arguments to the function. The remainder of the function definition consists of the commands in the function body enclosed with { and } braces. To access any arguments with a function, use the shell script variable explained in 'Accessing Arguments' section.

Special Symbols

The following are the meaning of symbols used in shell scripting:

S.NO	Symbol	Description
1	#	Usually denotes the beginning of a comment
2	;	Separates two statements appearing in the same line
3	\	An escape sequence
4	$	Variable e.g. $PATH will give the value of the variable PATH
5	"" or ''	Strings
6	{}	Block of code
7	:	A null (no operator) statement
8	()	Group of commands to be executed by a sub shell
9	[[]]	Condition e.g. [[$count -eq 0]]
10	$(())	Evaluation of an arithmetic operations

Lab Activity

- Write scripts /commands / syntax to print a message “Welcome to the World of Shell Scripting” 10 times.
- Write scripts /commands / syntax to take input and print your name, degree program information, batch No and course title.
- Write scripts /commands / syntax that take input and check the number is positive or negative.
- Write scripts /commands / syntax that take input and check the number is prime or not.
- Write scripts /commands / syntax that take input and check the maximum and minimum of 5 input numbers.
- Write scripts /commands / syntax to generate the Fibonacci series.
- Write scripts /commands / syntax to find out the factorial of given input number.
- Write scripts /commands / syntax for moving files into three subdirectories directories: shelldir, cdir, jpgdir according to their extensions.
- **Write a single script of all necessary software installation in your system. (Important)**

Coursera

- The Unix Workbench (cover Week 3 for Lab 3)

https://www.coursera.org/learn/unix

https://seankross.com/the-unix-workbench/

Reference Tutorial

https://www.shellscript.sh