Lab 2 - Bash Scripting and Automations

What is Bash and Bash Scripting

Bash (Bourne Again Shell) is a command language interpreter that we use in order to interact with the linux kernel. It is a command line interpreter that we use in order to interact with the linux kernel.

Bash scripting is the process of writing a set of commands in a file and then executing them in order to automate a task.

Introduction to Text Editors

In Linux, we have multiple commands that act as the `notepad` of linux. Such softwares include (but are not limited to):

- nano
- vim
- emacs

In this course, we'll be studying nano.

Nano

Nano is a command line text editor that we can use to write, read and delete data from within a file.

In order to open a file in nano, we type the following command:

```
nano <file-name>
```

Now, in order to save data into the file, we will firstly press `CTRL+S` and then, in order to exit, we need to press `CTRL+X`.

Writing your first bash script

The first line in a bash script must be `#!/bin/bash` and is called as `SHEBANG` line.

> A she-bang is set of sequence that begins with `#!` and then the interpreter is specified.
> In our case, we'll be using `/bin/bash` as the interpreter.

Then, we will use `echo` command in order to print data into the stdout.

```
#!/bin/bash

echo "Hello World"
```

Now, in order to execute this file, we need to give it executable permissions. We can do that by using the `chmod` command.

```
chmod +x <file-name>
```

Now, we can execute the file by using the following command:

```

./<file-name>
```

Variables

Variables are used to store data in a program. In bash, we can declare a variable by using the following syntax:

```
# NOTE: There should be no space between the variable name and the equal sign
variable_name=value
```

Now, in order to access the value of the variable, we need to use the `$` sign before the variable name.

```
echo $variable_name
```

Variable Types

There are two types of variables in bash:

- System Variables
- User Defined Variables

System Variables System variables are the variables that are defined by the system and are used to store system related information.

Some of the system variables are:

- `$HOME`: Stores the path to the home directory of the user
- `$PWD`: Stores the path to the current working directory
- `$BASH`: Stores the path to the bash shell
- `$BASH_VERSION`: Stores the version of the bash shell
- `$LOGNAME`: Stores the name of the user

User Defined Variables User defined variables are the variables that are defined by the user and are used to store user related information.

Unsetting a Variable

In order to unset a variable, we can use the `unset` command.

```
unset <variable-name>
```

Variables Expansion

Variable expansions refers to expanding a variable inside another variable or in a command. Variables can be expanded by prepending the name with `$`. Also, in order for the variable to be expanded, it must be wrap in `"`.

Example:

```
# Suppose, we have a variable called
name="Ali"

# if we want to expand it, we can call it like this:
echo "My name is $name"

## Also, if we want to you it with another command, we can use it like this:
ip=192.168.0.0
octet=24
nmap -sn "$ip/$octet"
```

Another case, in which we want tsotre the output of a command inside a variable, we will use “‘ symbol.

Example:

```
ping_output=`ping 192.168.0.1`
echo "Output: $ping"
```

Read Input from the User

In order to read input from the user, we can use the `read` command.

```
read <variable-name>

## If we want a message to be displayed before the user enters the value, we can use the following
read -p "Enter your name: " <variable-name>
```

Now, the value that the user enters will be stored in the variable.

Pipes

Pipes in linux; simply work by getting the output of command and pass as input to another command. These are represented by `|` symbol. These are useful when we want to chain multiple commands and then work on the output of their commands equally.

Example

Suppose a scenario, where want to only get the ip address of a single interface from `ip addr show` command.

```
ip addr show <interface> | grep <ip-to-search>

## Look at what the command `grep` does.

## These can further be chained
ip addr show <interface> | grep <ip-to-search> | cut -d '/' -f 1

## Look at what cut command does.
```

Redirectors

Redirectors; simply redirect the output to a file or from a file into the command. These can be useful in many scenarios.

```
#  > represents redirecting the output of stdout into a file or anything
# >> represents redirecting the output whilst mainting the existing output
#  < represnts taking input from a file and pass it into a specific command.
```

Example:

```
# Suppose, we want to redirect the output of `ip addr show` into a file called `ip.txt`
ip addr show > ip.txt

# Suppose, we want to redirect the output of `ip addr show` into a file called `ip.txt` whilst ma
ip addr show >> ip.txt

# Now, suppose we want to take the input from a file and pass it into a command

less < ip.txt

> If anyone wants to read more about redirectors, they can read [this](https://www.gnu.org/softwar

## If-Else Statements

In order to use if-else statements in bash, we can use the following syntax:
```

~~~
```bash
if [ <condition> ]
then
    <statements>
else
    <statements>
fi

## These can also be oneliners as well
if [ <condition> ]; then <statements>; else <statements>; fi
~~~

For Loop

In order to use for loop in bash, we can use the following syntax:

```
for <variable-name> in <list>
do
    <statements>
done
```

These can also be written on one line as:

```
for <variable-name> in <list>; do <statements(seperated-by-a-comma)>; done
```

Consider a simple example that will print 1 to 10

```
## Multiple lines:
for i in $(seq 1 10) # this can also be written as `seq 1 10`
do
    echo "Current number is $i"
    echo "======"
done

## Single line:
for i in `seq 1 10`; do echo "Current number is $i"; echo "======"; done
```

While Loop

In order to use while loop in bash, we can use the following syntax:

```
while [ <condition> ]

do
    <statements>
done
```

Arguments

Arguments are the values that are passed to the script when it is executed.

In order to access the arguments, we can use the following syntax:

```
$0 # Stores the name of the script
$1 # Stores the first argument
$2 # Stores the second argument
$n # Stores the nth argument

# In order to find the total number of arguments, we can use the following syntax:
$# # Stores the total number of arguments
```

Exit Status

Exit status is the status that is returned by the script when it is executed.

In order to access the exit status, we can use the following syntax:

```
$? # Stores the exit status
```

Functions

Functions are the set of statements that are executed when they are called.

In order to define a function, we can use the following syntax:

```
function_name() {
    <statements>
}
```

In order to call a function, we can use the following syntax:

```
function_name
```

Functions can be passed arguments as well.

Example: We'll write a function that will take name as an argument and print it

```
function print_name() {
    echo "My name is: $1"
}
print_name "Ali Taqi"
```

Class Tasks

Task 1

Write a bash script that will ask user for input; name and age. Then, print the name and age to the stdout. Example:

> Enter your name: Ali Enter your age: 20 Your name is Ali and you are 20 years old.

Task 2

Using `ip addr show <interface>`, extract the ip address of a specific interface.
