Introduction to Scripting “Bash”

https://github.com/BilalMaz/DevOps-Architect-BootCamp

About me

Hi there, my name is Bilal and I will Welcome you to DevOps boot camp! I am thrilled to have you join us for this exciting journey of learning and discovery.

In this boot camp, we will be exploring the principles and practices of DevOps, which is a set of methodologies and tools that aims to bridge the gap between software development and operations. DevOps is an increasingly important area in the field of software engineering, as it helps organizations to streamline their processes, improve their agility, and deliver better value to their customers.

By the end of this boot camp, you will have gained a comprehensive understanding of DevOps and its key concepts, as well as practical skills in areas such as infrastructure automation, continuous integration and delivery, monitoring and logging, and more. You will be equipped with the knowledge and tools to apply DevOps principles in your own work and contribute to the success of your organization.

I am always looking to connect with other professionals in the field, share ideas and insights, and stay up to date on the latest trends and developments. I welcome the opportunity to connect with you and explore ways in which we can collaborate and support each other.

Please find my Linkedin profile

https://www.linkedin.com/in/bilalmazhar-cyber-security-consultant/

What is scripting in Linux ?

A script is essentially a program written in a scripting language, such as Bash, Python, Perl, or Ruby, which can automate tasks or perform complex operations. Scripts are generally used for tasks such as system administration, file management, software installation, and automation of repetitive tasks.

The most common scripting language used in Linux is Bash (Bourne Again SHell), which is a command-line interpreter and scripting language that comes pre-installed with most Linux distributions. Bash scripts can be created using a simple text editor, such as Vim or Nano, and can be run using the terminal or command line interface.

Bash

Bash is a shell or a command-line interpreter that is commonly used in Linux and other Unix-like operating systems. It is the default shell for most Linux distributions and is used to execute commands, run scripts, and interact with the operating system. Bash stands for "Bourne-Again SHell," which is a reference to the Bourne shell, an earlier Unix shell that Bash was based on.

Bash is the default shell in Linux, other shells like Zsh, Ksh, and Tcsh are also available and can be installed and used. However, Bash is the most commonly used shell, and most Linux tutorials and scripts assume the use of Bash.

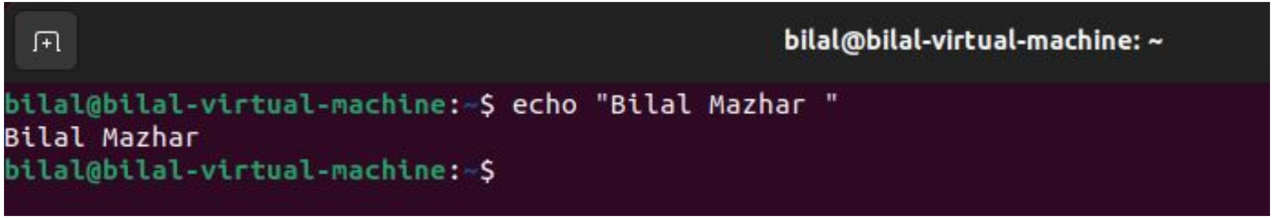
Type of Shell ?

There are several shells available in Linux, including:

- Bash (Bourne-Again SHell) - This is the default shell for most Linux distributions and is the most widely used shell in Linux. It is an enhanced version of the original Bourne shell and provides features such as command line editing, history, and job control.
- Csh (C SHell) - This is a shell that provides a C-like syntax and is popular among programmers who are used to the C programming language.
- Tcsh (TENEX C Shell) - This is an enhanced version of the C shell and provides features such as command-line editing, history, and job control.
- Zsh (Z SHell) - This is an interactive shell that is designed to be more user-friendly and customizable than Bash. It provides advanced features such as spelling correction, path expansion, and file globbing.Dash (Debian Almquist Shell) - This is a minimal shell that is designed to be fast and efficient. It is commonly used as the default system shell in many Linux distributions.
- Ksh (Korn SHell) - This is a shell that provides a combination of features from the Bourne shell and the C shell. It is widely used in commercial Unix environments.

Hello World

1. Interactive Shell: An interactive shell in Linux refers to a shell that allows a user to enter commands and receive immediate feedback in real-time. The interactive shell is typically used in a terminal window or console where a user types in commands to interact with the operating system.

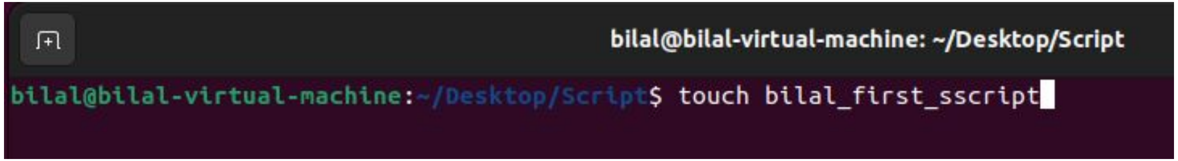


```
bilal@bilal-virtual-machine:~$ echo "Bilal Mazhar "
Bilal Mazhar
bilal@bilal-virtual-machine:~$
```

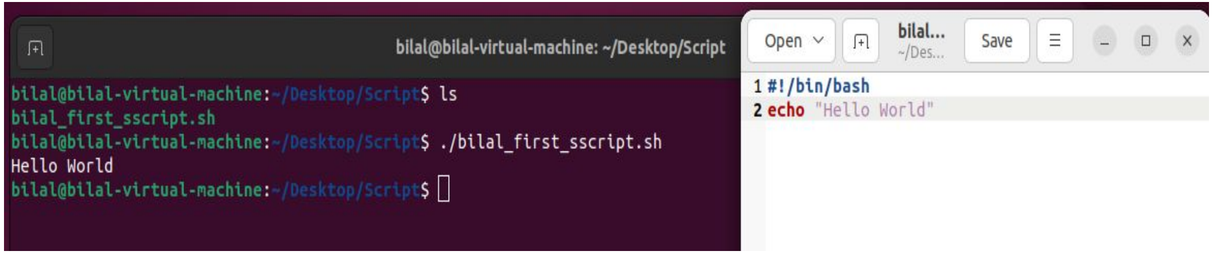
- echo is a Bash builtin command that writes the arguments it receives to the standard output. It appends a newline to the output, by default

Non-Interactive Shell

1. The Bash shell can also be run non-interactively from a script, making the shell require no human interaction.Interactive behavior and scripted behavior should be identical – an important design consideration of Unix V7 Bourne shell and transitively Bash. Therefore anything that can be done at the command line can be put in a script file for reuse.



```
bilal@bilal-virtual-machine:~/Desktop/Script$ touch bilal_first_sscript
```

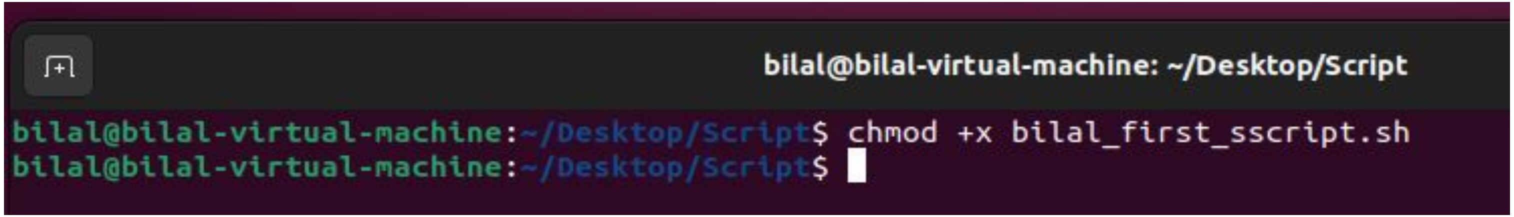


```
bilal@bilal-virtual-machine:~/Desktop/Script$ ls
bilal_first_sscript.sh
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_first_sscript.sh
Hello World
bilal@bilal-virtual-machine:~/Desktop/Script$
```

```
#!/bin/bash
echo "Hello World"
```

Change permission

- chmod +x is a command used in Unix-based operating systems to change the permissions of a file, making it executable.
- The chmod command is used to modify the permissions of files or directories. The +x option is used to add the execute permission to a file, allowing the user to run the file as a program.



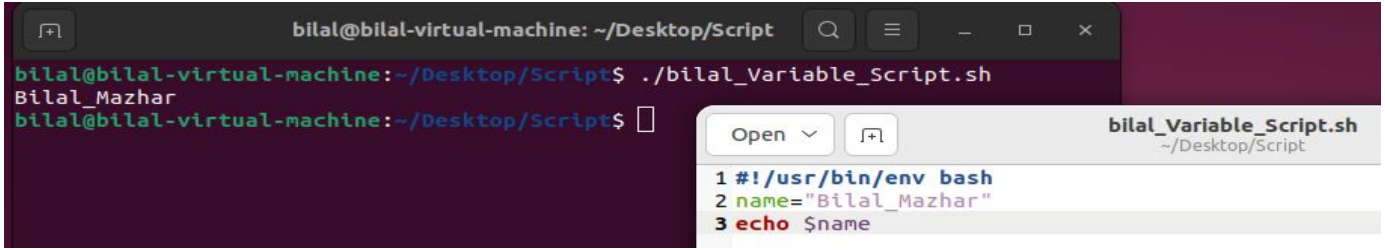
```
bilal@bilal-virtual-machine:~/Desktop/Script$ chmod +x bilal_first_sscript.sh
bilal@bilal-virtual-machine:~/Desktop/Script$
```

Variable

In Bash, a variable is a name that represents a value. A variable can hold a variety of data types, such as numbers, strings, arrays, and more.

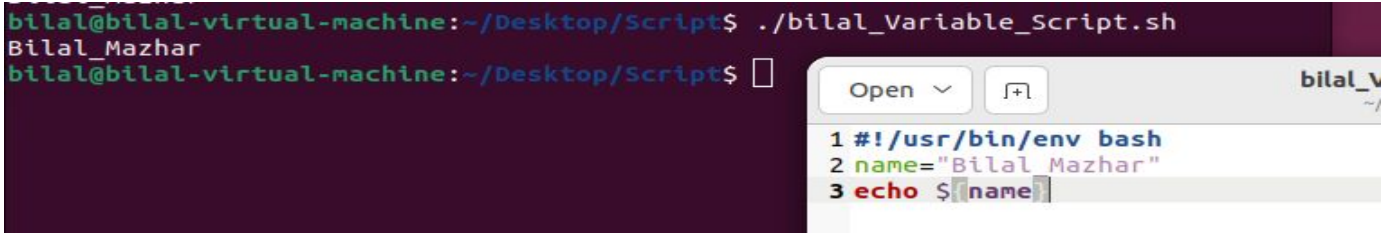
Example	Command
To define a variable in Bash, you simply give it a name and assign a value to it. Here's the basic syntax	variable_name=value
To define a variable called "**name**" with a value of "John", you would use the following command	echo $name
To access the value of a variable, you use the variable name with a dollar sign prefix. For example, to print the value of the "name" variable, you would use the following command	echo ${name}

Example



```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_Variable_Script.sh
Bilal_Mazhar
bilal@bilal-virtual-machine:~/Desktop/Script$
```

```
#!/usr/bin/env bash
name="Bilal_Mazhar"
echo $name
```



```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_Variable_Script.sh
Bilal_Mazhar
bilal@bilal-virtual-machine:~/Desktop/Script$
```

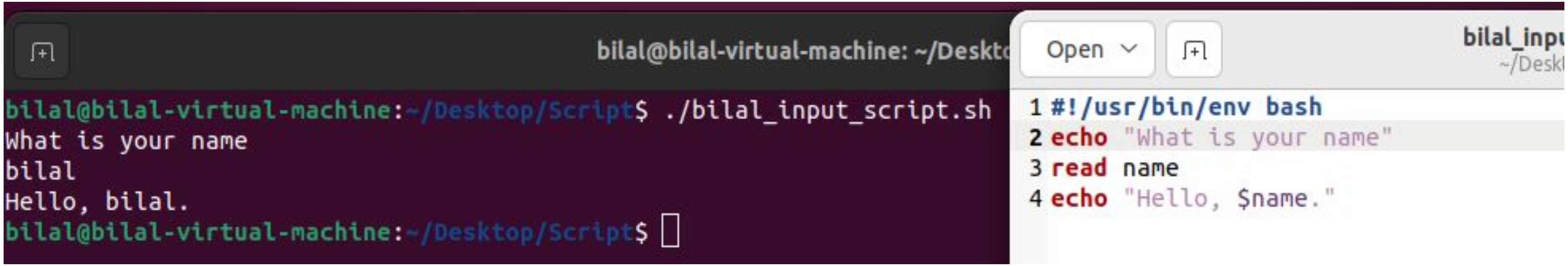
```
#!/usr/bin/env bash
name="Bilal_Mazhar"
echo ${name}
```

User Input

In Bash, you can get user input using the read command. The read command reads a line of input from the user and stores it in a variable.

Example	Command
Basic Syntax for read user inputs	read variable_name
to prompt the user for their name and store it in a variable called "name", you would use the following command	read name
You can also provide a prompt for the user by including a message before the read command, like this	read -p "Enter your name: " name

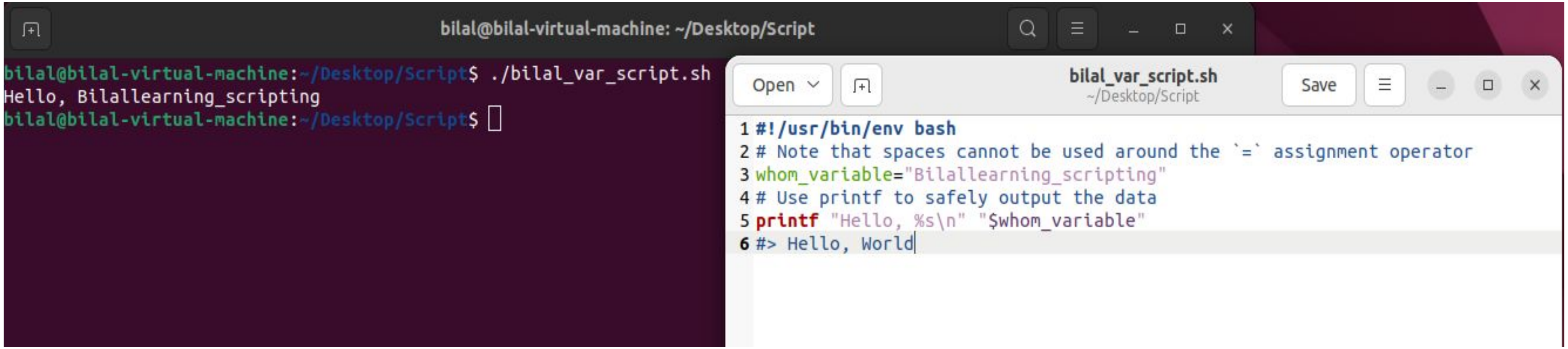
Hello World with User Input



```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_input_script.sh
What is your name
bilal
Hello, bilal.
bilal@bilal-virtual-machine:~/Desktop/Script$
```

```
#!/usr/bin/env bash
echo "What is your name"
read name
echo "Hello, $name."
```

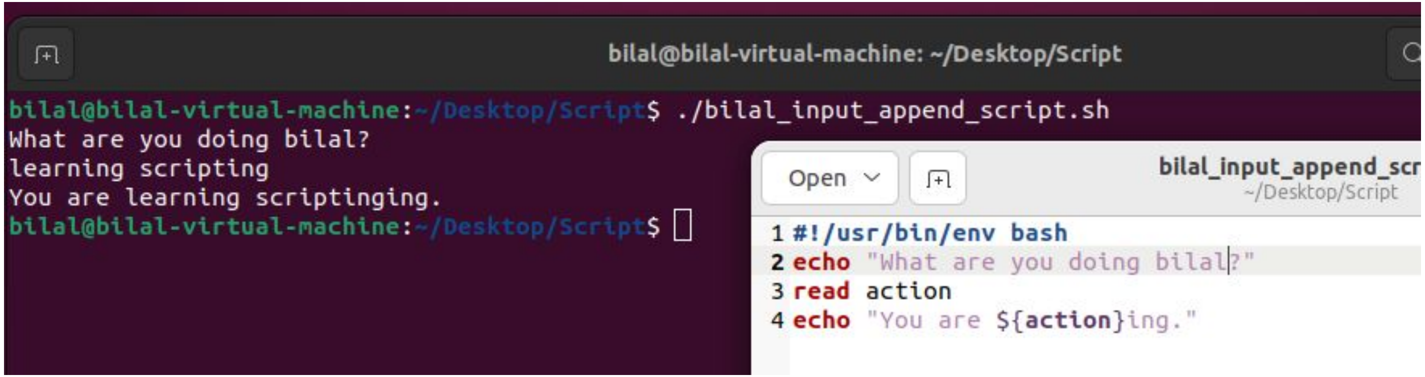
Hello World Using Variables



```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_var_script.sh
Hello, Bilallearning_scripting
bilal@bilal-virtual-machine:~/Desktop/Script$ 
```

```
#!/usr/bin/env bash
# Note that spaces cannot be used around the `=` assignment operator
whom_variable="Bilallearning_scripting"
# Use printf to safely output the data
printf "Hello, %s\n" "$whom_variable"
#> Hello, World
```

Append something to the variable value



```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_input_append_script.sh
What are you doing bilal?
learning scripting
You are learning scriptinging.
bilal@bilal-virtual-machine:~/Desktop/Script$ 
```

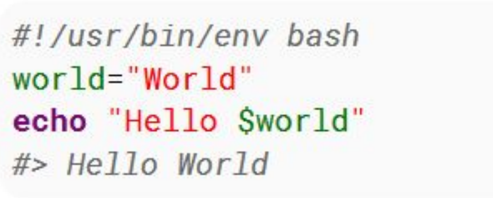
```
#!/usr/bin/env bash
echo "What are you doing bilal?"
read action
echo "You are ${action}ing."
```

Importance of Quoting in Strings

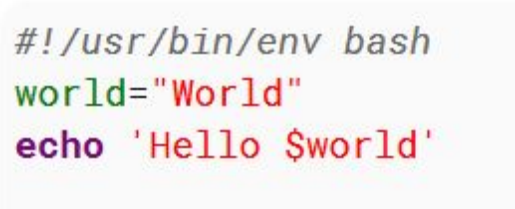
Quoting is important for string expansion in bash. With these, you can control how the bash parses and expands your strings.

There are two types of quoting:

- **Weak:** uses double quotes: **"**
- **Strong:** uses single quotes: **'**

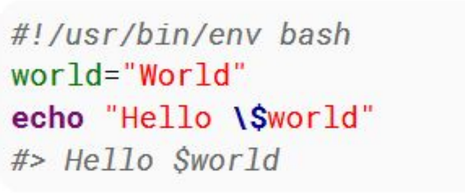


```
#!/usr/bin/env bash
world="World"
echo "Hello $world"
#> Hello World
```



```
#!/usr/bin/env bash
world="World"
echo 'Hello $world'
```

You can also use escape to prevent expansion:



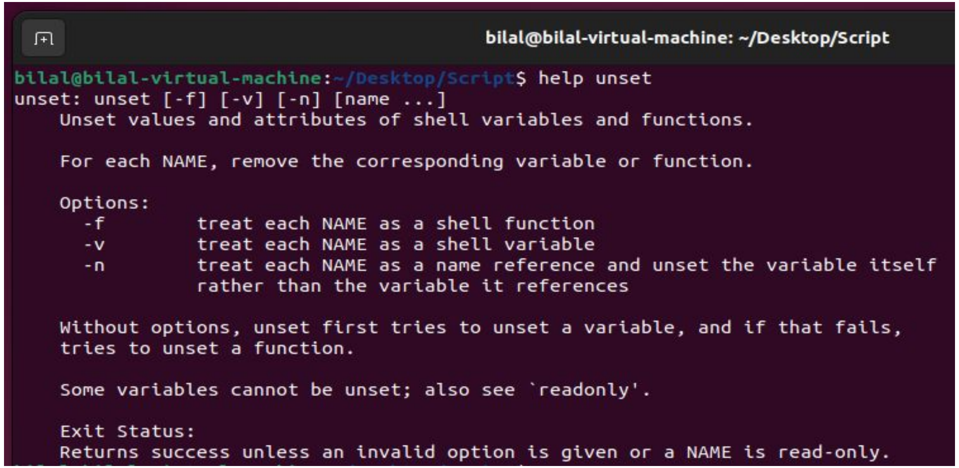
```
#!/usr/bin/env bash
world="World"
echo "Hello \$world"
#> Hello $world
```

Viewing information for Bash built-ins

help <command>

This will display the Bash help (manual) page for the specified built-in.

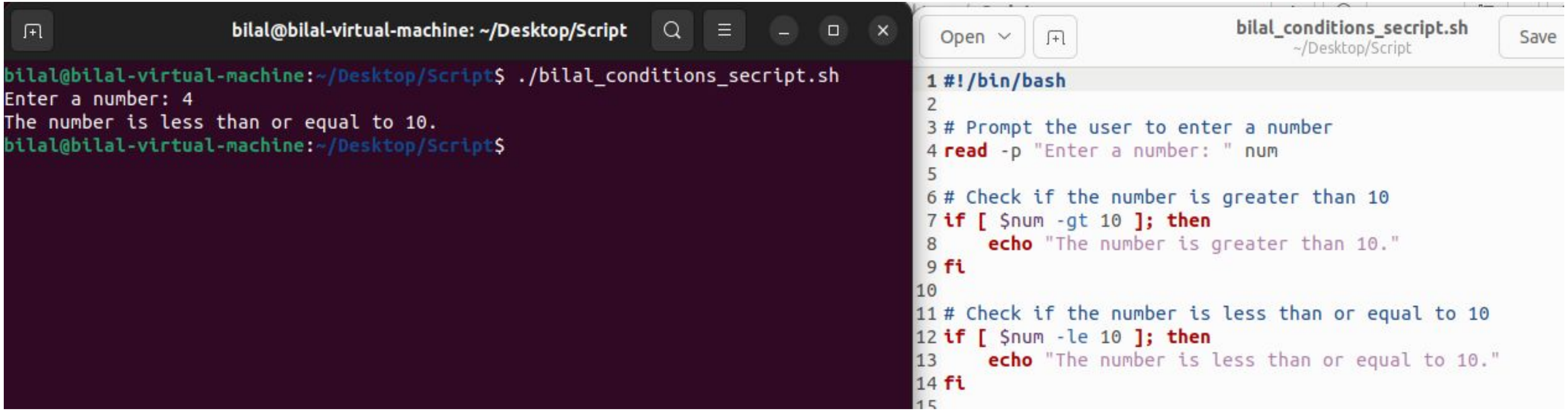
For example, help unset will show:



```
bilal@bilal-virtual-machine:~/Desktop/Script$ help unset
unset: unset [-f] [-v] [-n] [name ...]
    Unset values and attributes of shell variables and functions.

    For each NAME, remove the corresponding variable or function.

    Options:
      -f        treat each NAME as a shell function
      -v        treat each NAME as a shell variable
      -n        treat each NAME as a name reference and unset the variable itself
                rather than the variable it references

    Without options, unset first tries to unset a variable, and if that fails,
    tries to unset a function.

    Some variables cannot be unset; also see `readonly'.

    Exit Status:
    Returns success unless an invalid option is given or a NAME is read-only.
```

Conditionals : if

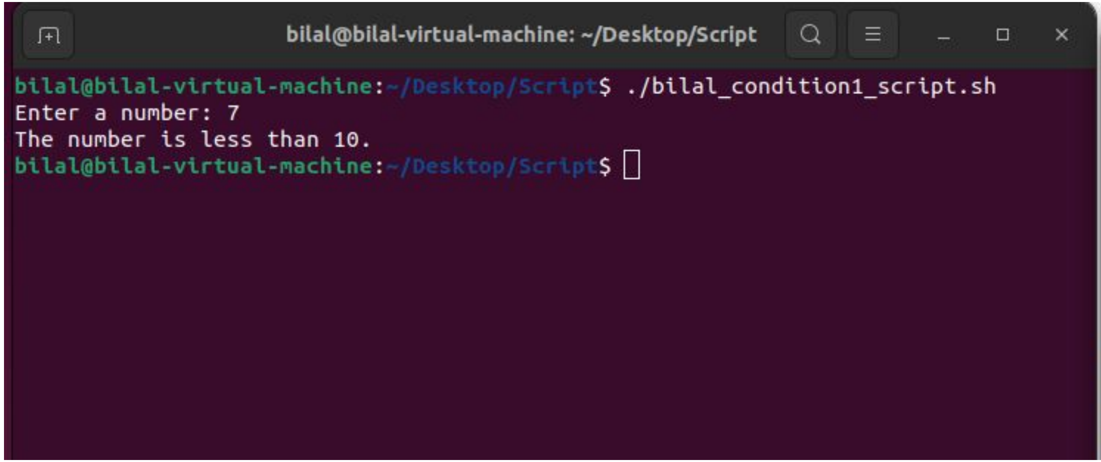
In Bash, conditional statements are used to control the flow of execution in a script based on certain conditions. The two main types of conditional statements are if statements and case statements.


bilal@bilal-virtual-machine: ~/Desktop/Script
Open
bilal_conditions_secript.sh
~/Desktop/Script
Save

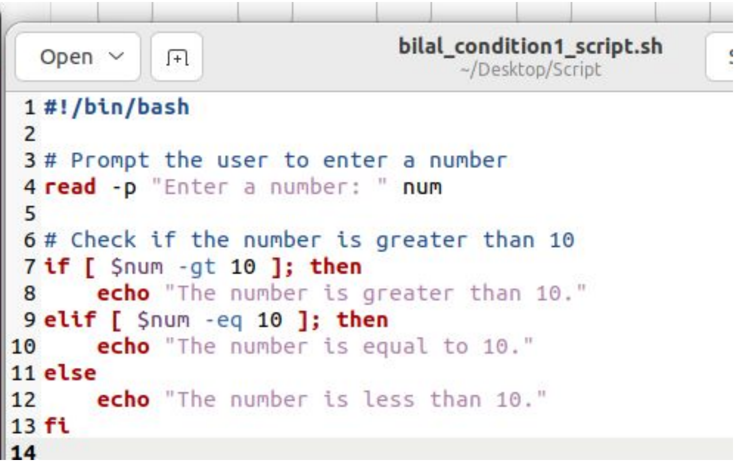
```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_conditions_secript.sh
Enter a number: 4
The number is less than or equal to 10.
bilal@bilal-virtual-machine:~/Desktop/Script$
```

```
#!/bin/bash

# Prompt the user to enter a number
read -p "Enter a number: " num

# Check if the number is greater than 10
if [ $num -gt 10 ]; then
    echo "The number is greater than 10."
fi

# Check if the number is less than or equal to 10
if [ $num -le 10 ]; then
    echo "The number is less than or equal to 10."
fi
```

Conditionals : elif



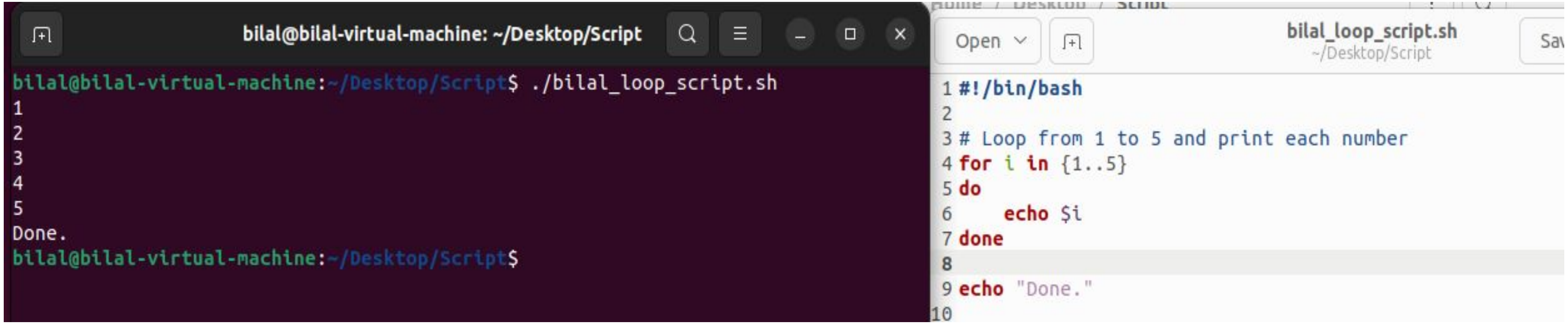
```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_condition1_script.sh
Enter a number: 7
The number is less than 10.
bilal@bilal-virtual-machine:~/Desktop/Script$ 
```



```bash
#!/bin/bash

# Prompt the user to enter a number
read -p "Enter a number: " num

# Check if the number is greater than 10
if [ $num -gt 10 ]; then
    echo "The number is greater than 10."
elif [ $num -eq 10 ]; then
    echo "The number is equal to 10."
else
    echo "The number is less than 10."
fi

```

Loop

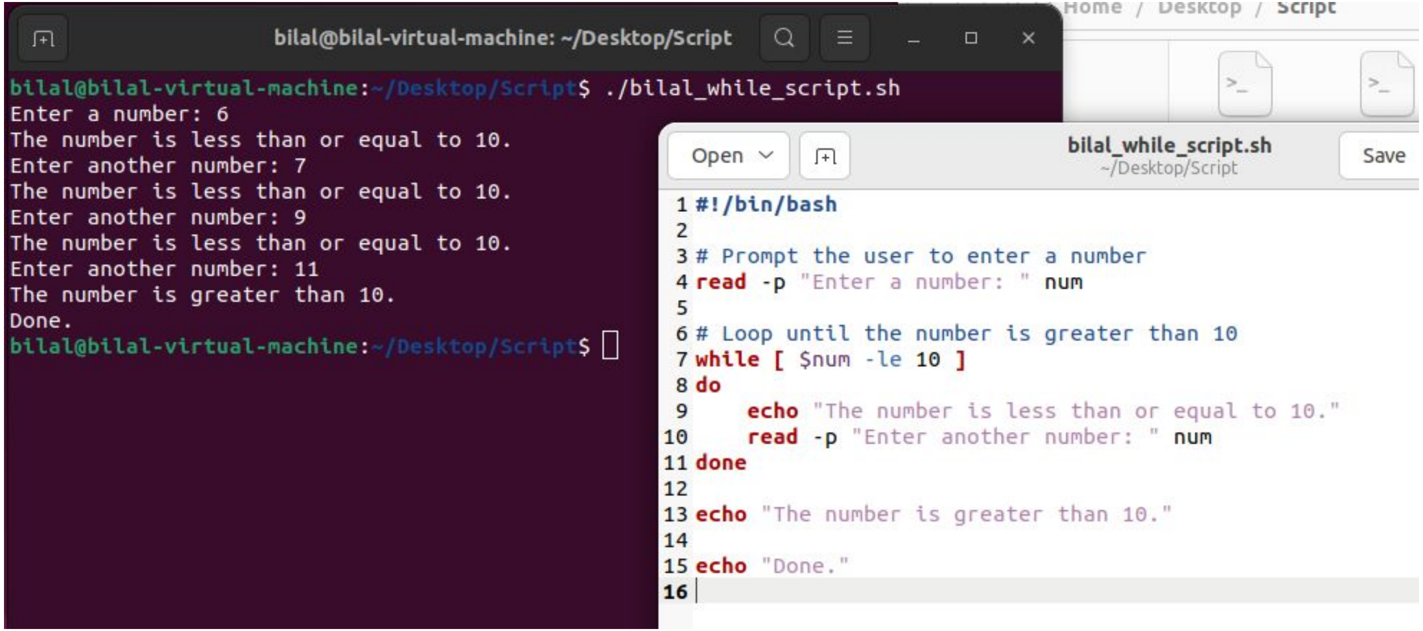
In Bash scripting, loops are used to repeat a block of code a certain number of times, or until a certain condition is met. There are two main types of loops in Bash: for loops and while loops.



```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_loop_script.sh
1
2
3
4
5
Done.
bilal@bilal-virtual-machine:~/Desktop/Script$
```

```
#!/bin/bash

# Loop from 1 to 5 and print each number
for i in {1..5}
do
    echo $i
done

echo "Done."

```

While

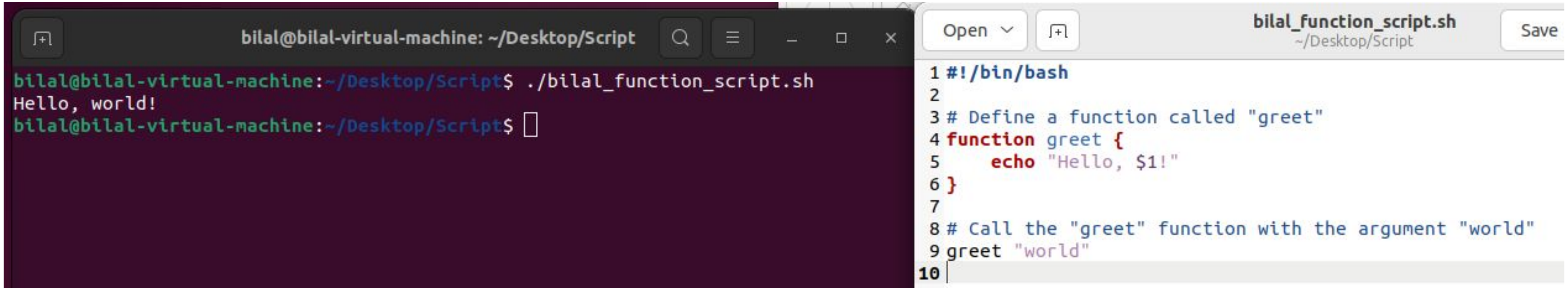


```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_while_script.sh
Enter a number: 6
The number is less than or equal to 10.
Enter another number: 7
The number is less than or equal to 10.
Enter another number: 9
The number is less than or equal to 10.
Enter another number: 11
The number is greater than 10.
Done.
bilal@bilal-virtual-machine:~/Desktop/Script$ 
```

```
#!/bin/bash

# Prompt the user to enter a number
read -p "Enter a number: " num

# Loop until the number is greater than 10
while [ $num -le 10 ]
do
    echo "The number is less than or equal to 10."
    read -p "Enter another number: " num
done

echo "The number is greater than 10."

echo "Done."

```

Functions

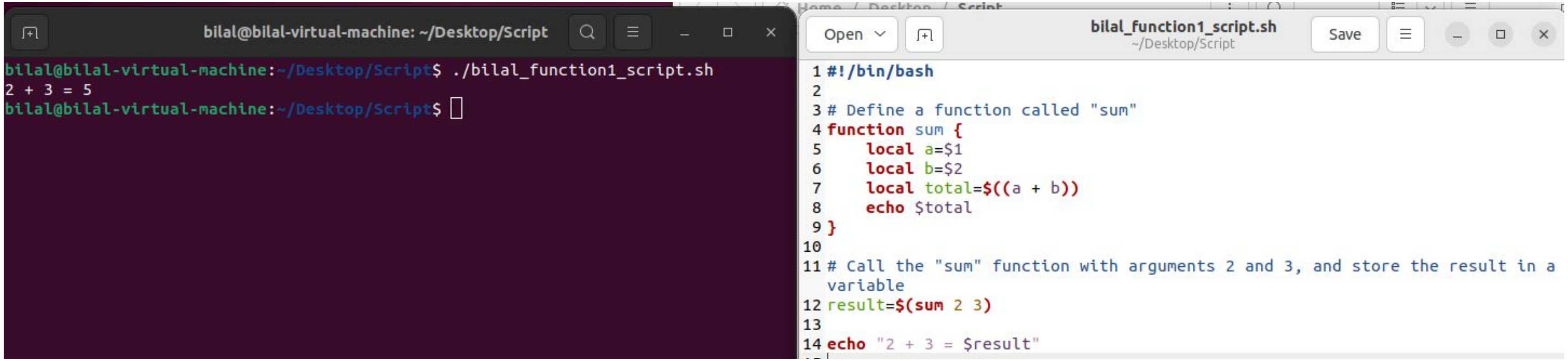
In Bash scripting, functions are blocks of code that can be called by name, and can optionally accept arguments and return values. Functions can be useful for encapsulating complex or repetitive tasks, and for breaking up large scripts into smaller, more manageable pieces.



```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_function_script.sh
Hello, world!
bilal@bilal-virtual-machine:~/Desktop/Script$ 
```

```bash
#!/bin/bash

# Define a function called "greet"
function greet {
    echo "Hello, $1!"
}

# Call the "greet" function with the argument "world"
greet "world"
```

Example


bilal@bilal-virtual-machine: ~/Desktop/Script
Open
bilal_function1_script.sh
~/Desktop/Script
Save

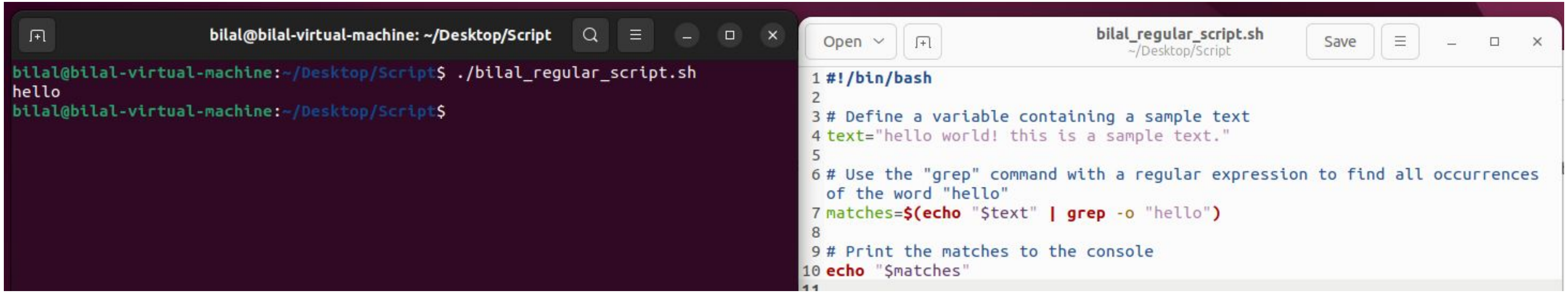
```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_function1_script.sh
2 + 3 = 5
bilal@bilal-virtual-machine:~/Desktop/Script$ 
```

```bash
#!/bin/bash

# Define a function called "sum"
function sum {
    local a=$1
    local b=$2
    local total=$((a + b))
    echo $total
}

# Call the "sum" function with arguments 2 and 3, and store the result in a variable
result=$(sum 2 3)

echo "2 + 3 = $result"
```

Regular expressions

Regular expressions, or "regex" for short, are patterns of characters that are used to search, match, and manipulate text. In Bash scripting, regular expressions are commonly used with the grep, sed, and awk commands to process text files and streams.

Here's an example of a simple regular expression that matches any string that contains the word "hello"



```
bilal@bilal-virtual-machine:~/Desktop/Script$ ./bilal_regular_script.sh
hello
bilal@bilal-virtual-machine:~/Desktop/Script$
```

```
#!/bin/bash

# Define a variable containing a sample text
text="hello world! this is a sample text."

# Use the "grep" command with a regular expression to find all occurrences of the word "hello"
matches=$(echo "$text" | grep -o "hello")

# Print the matches to the console
echo "$matches"
```