LP ASSIGNMENT-7

ENG24CY0159

SHADIK KHAN

1. What is a bash shell script? Give one example.

A **bash shell script** is simply a plain text file containing a list of commands. Instead of you typing the commands one by one into the terminal, the shell reads this file and executes the commands in order. It's like a recipe or a to-do list for your computer, used to automate repetitive tasks.

Example: Here is a simple script that creates a new project directory, moves into it, and creates two empty files.

Bash

```
#!/bin/bash


# A simple script to set up a new project folder
echo "Creating project directory..."
mkdir my_new_project
cd my_new_project
touch index.html styles.css
echo "Project setup complete!"
ls
```

---

2. Write a simple shell script to print “Hello World”.

This is the classic "first program" for any language!

1. Create a file named hello.sh.
2. Put the following text inside it:

Bash

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

```
# This script prints "Hello World" to the terminal.

echo "Hello World"
```

3. Save the file. Before you can run it, you need to make it executable:

Bash

```
chmod +x hello.sh
```

4. Now, run it:

Bash

```
./hello.sh
```

The line #!/bin/bash at the top is called a **shebang**. It's important because it tells your system which interpreter (in this case, bash) to use to run the script.

---

3. What is the purpose of comments (#) in a shell script?

Think of comments as **sticky notes for humans**. Any line in a script that begins with a hash symbol (#) is completely ignored by the shell when it runs.

Their purpose is to:

- **Explain your code:** Leave notes for your future self or for other people on your team to explain *why* you wrote a complex line of code.
- **Improve readability:** Add titles or section breaks to make your script easier to understand.
- **Temporarily disable code:** You can "comment out" a line of code to stop it from running without deleting it, which is very useful for debugging.

---

4. How do you declare variables (int, float, double, string, Boolean, and char) in a shell script?

This is actually a bit of a trick question! Unlike many other programming languages, **bash does not have typed variables**.

By default, every variable in bash is just a **string**. You don't declare a type; you just assign a value.

- **String:** my_name="Alice"

- **Number-like String:** my_age=30

Bash will treat my_age as a number if you use it in a math context (e.g., echo $((my_age + 5))), but it's still fundamentally a string.

Important limitations:

- **Floats/Decimals:** Bash's built-in math **cannot handle decimals**. You have to use external tools like bc or awk to work with floating-point numbers.
- **Booleans:** There's no true/false type. Programmers typically simulate it using 1 for true and 0 for false, or with the strings "true" and "false".

---

5. Write a shell script to display the current date and time of the system.

This script simply runs the standard date command that you would use in your terminal.

Bash

```
#!/bin/bash


# This script displays the system's current date and time.
echo "The current date and time is:"
date
echo "--------------------------"
echo "Have a great day!"
```

---

6. Explain the difference between a constant and a variable in bash script.

A **variable** is like a name written on a whiteboard. You can assign it a value, and then you can erase it and change that value as many times as you want later in the script.

Bash

```
# A variable
GREETING="Hello"
```

```
echo $GREETING  # Outputs: Hello

GREETING="Hi"

echo $GREETING  # Outputs: Hi
```

A **constant**, on the other hand, is like a name written in permanent marker. Once you set it, you're not supposed to change it. Bash creates this behavior using the readonly command.

Bash

```
# A constant

readonly API_KEY="xyz123abc"

echo $API_KEY    # Outputs: xyz123abc

# The next line will cause the script to exit with an error

API_KEY="newkey" # Error: API_KEY: readonly variable
```

---

7. Write a shell script to read two integer numbers from the user and compute the sum of both the numbers.

This script uses the read command to get input from the user and bash's arithmetic expansion ((...)) to perform the calculation.

Bash

```
#!/bin/bash


# This script reads two numbers and calculates their sum.


echo "--- Simple Sum Calculator ---"


echo "Please enter the first number:"

read num1
```

```
echo "Please enter the second number:"
read num2

# Calculate the sum
sum=$((num1 + num2))

# Display the result
echo "The sum of $num1 and $num2 is: $sum"
```

---

8. What is the use of the source command in shell scripting?

The source command (which can be shortened to a single dot .) executes a script's commands **within your current shell session**, instead of starting a new, separate one.

Think of it this way:

- **Running a script normally (./script.sh)** is like hiring a temporary assistant who works in their own separate office. When they're done, they leave and take all their notes (variables, functions) with them.
- **Sourcing a script (source script.sh)** is like having that assistant come into *your* office and put all their notes and tools on *your* desk. After they leave, all their definitions are still there for you to use.

This is most commonly used for loading configuration files (like your .bashrc) or libraries of shared functions into your active terminal.

---

9. How can you debug a shell script? Give two methods.

Debugging is a crucial skill! Here are two of the most common methods.

Method 1: The Built-in Debugger (bash -x) This is like turning on a "narrator mode" for your script. The shell will print every command it's about to execute, right to the terminal, prefixed with a + sign. This lets you see the exact flow and

how variables are being interpreted. You can activate it by changing your script's shebang from #!/bin/bash to #!/bin/bash -x.

Method 2: Strategic echo Statements This is the classic, simple approach. If you're not sure what value a variable holds at a certain point in your script, just add a temporary echo statement to print it out.

Bash

```
# ... some code ...

echo "DEBUG: The value of the variable 'FILENAME' is now: $FILENAME"

# ... more code ...
```

This helps you trace the state of your script and pinpoint exactly where things are going wrong.

---

10. Write a bash script to create and delete a file.

This script will create a file, prove it exists, and then delete it, narrating its actions as it goes.

Bash

```
#!/bin/bash


# A script to demonstrate creating and deleting a file.


FILENAME="temporary_file_for_testing.txt"


echo "Step 1: Creating the file named '$FILENAME'..."

touch "$FILENAME"

echo "-----------------------------------"


echo "Step 2: Verifying the file exists..."

ls -l "$FILENAME"
```

```
echo "-----------------------------------"

echo "Step 3: Now, deleting the file '$FILENAME'..."
rm "$FILENAME"
echo "-----------------------------------"

echo "Step 4: Verifying the file is gone..."
# This ls command will show an error, proving the file is deleted.
ls "$FILENAME"
echo "-----------------------------------"
echo "Script finished."
```
