Assignment 5

NAME- Shreya Rajni

SECTION-3A

ROLL NO.-59

USN- ENG24CY0161

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

Think of a **Bash shell script** as your personal to-do list for the computer . It’s just a simple text file packed with Linux commands that the Bash interpreter runs one after the other. It's how you automate those boring, repetitive tasks!

A script always starts with the **shebang** line, `#!/bin/bash`, which tells the system, "Hey, use the Bash program to run this file!"

Example: A simple script to back up a user's Documents folder.

```
#!/bin/bash
# Backup the Documents folder to a timestamped tar archive
DATE=$(date +%Y%m%d)
tar -czf ~/backup-$DATE.tar.gz ~/Documents/
echo "Backup successful: backup-$DATE.tar.gz created."
```

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

This is the classic, simplest script! All you need is the shebang and the trusty echo command:

```
#!/bin/bash:Hello World Script:hello_world.sh
echo "Hello World"
```

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

Comments (#) are your way of leaving notes for yourself (and others!) inside the script. They’re super important for two main reasons:

1. **Explanation:** They help explain *what* your complicated code blocks or functions are doing, making the script readable and maintainable.
2. **Exclusion:** During testing or debugging, you can use them to temporarily "hide" lines of code without deleting them.

The great thing is, the shell interpreter ignores anything that starts with a # (unless it's inside quotes).

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

This is where Bash is pretty chill! It's **dynamically and weakly typed**, meaning you don't have to stress about declaring specific types like `int` or `float`. Variables are like temporary sticky notes—you just name them and assign a value, and Bash treats it all as text (strings) by default.

Declaration/Assignment Syntax:

```
# General Syntax for any type (stored as a string)
VARIABLE_NAME="Value"

# Example String
GREETING="Hello"

# Example Integer (used in arithmetic context)
# The 'declare -i' helps Bash treat it strictly as an
integer for math.
declare -i AGE=30

# Note: For decimal math, you have to use external tools
like 'bc'.
RESULT=$(echo "5.5 * 2.1" | bc)
```

Friendly Reminder: Explicit types like `float`, `double`, `Boolean`, and `char` don't really exist in standard Bash; you manage complex values using string manipulation or external utilities.

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

The `date` command is your friend here! You can use it alone for the full timestamp, or use formatting options to get specific views:

```
#!/bin/bash:Display Date and Time:system_time.sh
echo "Current System Date and Time:"
date
echo "Formatted time (Year-Month-Day Hour:Min:Sec):"
date +"%Y-%m-%d %H:%M:%S"
```

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

This is a key security and stability concept:

Feature	Variable	Constant
Flexibility	Its value **can be changed** throughout the script's execution.	Its value, once set, **cannot be changed** or unset.
Syntax	`MY_VAR="initial_value"`	`readonly MY_CONSTANT="fixed_value"`
Use Case	Storing temporary results, counters, or user input.	Storing fixed values like application paths or configuration limits.

In Bash, a constant is simply a variable that you lock in place using the built-in **`readonly`** command.

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

We use the `read` command to ask the user for input and the powerful arithmetic expansion `$(( ))` to handle the math:

```
#!/bin/bash:Sum Two Numbers:sum_numbers.sh
echo "Enter the first integer:"
read NUM1

echo "Enter the second integer:"
read NUM2

# Perform arithmetic calculation using $(( ))
SUM=$((NUM1 + NUM2))

echo "The sum of $NUM1 and $NUM2 is: $SUM"
```

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

When you usually run a script (`./script.sh`), it opens its own little sandbox (a sub-shell). But if you **`source`** a script (or use the dot: `.`), you run it **inside your current shell environment**.

Why this matters: This is vital when the script sets new environment variables (like updating your $PATH) or defines functions. If you just ran it normally, those changes would disappear when the script finished, but sourcing makes them stick around in your current session!

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

Debugging is when you put on your detective hat to figure out why your script isn't doing what you told it to do!

1. **Method 1: The `-x` option (Execution Trace):**

   a. This is like watching a step-by-step movie of your script running. You run it like this: `bash -x ./my_script.sh`
   b. **Result:** Bash will print every single command and its substituted variables, preceded by a + sign. This shows you *exactly* what the shell is trying to execute.
2. **Method 2: The `-n` option (Syntax Check):**
   a. This is your quick grammar check. You run it like this: `bash -n ./my_script.sh`
   b. **Result:** Bash reads the commands but **does not execute them**. It's totally safe, and it just checks for fundamental syntax errors (like missing quotes or an unclosed `if` statement).

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

This script uses the fundamental Linux commands for file management: `touch` to create the file and `rm` to safely remove it, with an `if/then` check for confirmation:

```
#!/bin/bash:Create and Delete File:file_operations.sh
FILE_NAME="temp_test_file.txt"

echo "Attempting to create file: $FILE_NAME"
# Command to create an empty file
touch $FILE_NAME

# Check if the file was created successfully
if [ -f "$FILE_NAME" ]; then
    echo "$FILE_NAME created successfully."
else
    echo "Error creating $FILE_NAME."
    exit 1
fi

# Command to delete the file
echo "Deleting file: $FILE_NAME"
rm $FILE_NAME

# Verify deletion
if [ ! -f "$FILE_NAME" ]; then
```

```
    echo "$FILE_NAME deleted successfully."
else
    echo "Error deleting $FILE_NAME."
fi
```