SHELL SCRIPTING

1. What is Shell Scripting?

Shell scripting is the process of writing a series of commands in a text file, which is then interpreted by the **shell** (command-line interface) to perform various operations. The **shell** acts as a bridge between the user and the operating system, executing the commands from the script.

- **Shell**: A command-line interpreter that runs commands.
- **Shell Script**: A text file with a list of commands for the shell to execute.

Common Shell Types:

- **Bash**: Bourne Again Shell (the most commonly used).
- **sh**: Bourne Shell (basic Unix shell).
- **zsh**: Z Shell.
- **csh**: C Shell (C-like syntax).

2. Creating and Running a Shell Script

Steps to Write a Shell Script:

1. **Create a new file**:

```bash
touch script.sh
```

2. **Add the Shebang (Interpreter Directive)**: The **shebang** is the first line in your script and tells the system which interpreter to use to execute the script.

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

ANUPAM SHUKLA

3. **Write your script**: Add the commands you want to run inside the script file.

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

4. **Make the script executable**: You need to give execution permissions to your script.

```bash
chmod +x script.sh
```

5. **Run the script**: You can execute your shell script using the following command:

```bash
./script.sh
```

3. Variables in Shell Scripting

Defining Variables:

- Variables in shell scripts are created by assigning a value without spaces.

```bash
NAME="Anupam"
AGE=22
```

Accessing Variables:

- Use the `$` sign before the variable name to access its value.

```bash
echo "My name is $NAME and I am $AGE years old."
```

Reading Input Dynamically:

- You can prompt the user to input values using the `read` command.

bash Example

```bash
read -p "Enter your name: " USER_NAME
echo "Hello, $USER_NAME"
```

4. Command-Line Arguments

Shell scripts can take arguments from the command line when executed. These arguments are accessed using special positional parameters:

- **$0**: The name of the script.
- **$1**, **$2**, ..., **$n**: The first, second, ..., nth argument.

Example:

bash Example

```bash
#!/bin/bash
echo "Script Name: $0"
echo "First Argument: $1"
echo "Second Argument: $2"
```

$?	exit status of the most recent process
$0	Bash Script name
$n	These variables correspond to the arguments with which a script was invoked
$@	All the arguments on the command line
$#	Count of all command line arguments
$*	All arguments as one String
$$	Process id of the current shell
$!	Process id of the background command

Running the script:

```bash
./script.sh arg1 arg2
```

Output:

```yaml
Script Name: ./script.sh
First Argument: arg1
Second Argument: arg2
```

5. Conditional Statements

Conditional statements allow you to execute commands based on conditions.

If-Else Syntax:

```bash
if [ condition ]
then
  # Commands to run if the condition is true
else
  # Commands to run if the condition is false
fi
```

Example:

bash Example

```bash
#!/bin/bash
if [ $1 -gt 10 ]
then
  echo "The argument is greater than 10"
else
  echo "The argument is less than or equal to 10"
fi
```

Common Conditions:

- **String Comparison**:
  - `=` : Equals.
  - `!=` : Not equals.
  - `-z` : String is empty.
  - `-n` : String is not empty.
- **Integer Comparison**:
  - `-eq` : Equals.
  - `-ne` : Not equals.
  - `-gt` : Greater than.
  - `-lt` : Less than.
  - `-ge` : Greater than or equal to.
  - `-le` : Less than or equal to.

6. Loops in Shell Scripting

Loops allow you to iterate over a set of data or commands.

For Loop:

bash Example

```bash
for i in {1..5}
do
  echo "Number: $i"
done
```

While Loop:

bash Example

```bash
counter=1
while [ $counter -le 5 ]
do
  echo "Counter: $counter"
  counter=$((counter + 1))
done
```

Until Loop:

The loop runs until the condition becomes true.

bash Example

```bash
counter=1
until [ $counter -gt 5 ]
do
  echo "Counter: $counter"
  counter=$((counter + 1))
done
```

7. Functions in Shell Scripting

Functions allow you to organize code into reusable blocks. You can define a function and call it multiple times.

Defining a Function:

bash Example

```bash
function greet() {
  echo "Hello, $1"
}
```

Calling a Function:

bash Example

```bash
greet "Anupam"
```

Example of a Function:

bash Example

```bash
#!/bin/bash
function add_numbers() {
  echo "Sum: $(($1 + $2))"
}

add_numbers 10 20
```

8. File Operations

Shell scripts can be used to manipulate files and directories.

Creating Files:

bash Example

```bash
touch filename.txt
```

Writing to a File:

- Overwrite the file:

bash Example

```bash
echo "This is a line" > file.txt
```

- Append to the file:

bash Example

```bash
echo "This is another line" >> file.txt
```

Reading from a File:

bash Example

```bash
while read line
do
  echo $line
done < file.txt
```

9. Redirecting Output and Errors

- `>` : Redirects standard output to a file (overwrites if the file exists).
- `>>` : Appends standard output to a file.
- `2>` : Redirects standard error to a file.

Example:

bash Example

```bash
ls non_existing_file 2> error.log   # Redirects errors to error.log
```

10. Pipes and Filters

- **Pipes (`|`)**: Pass the output of one command as input to another.

bash Example

```bash
ls | grep ".txt"
```

- **Filters**: Commands that process data:
  - **grep**: Search for patterns.

bash Example

```bash
grep "pattern" file.txt
```

  - **sort**: Sort lines in a file.
  - **awk**: Pattern scanning and processing language.

11. Exit Status and Error Handling

Every command returns an exit status that can be used for error handling. The exit status of the last executed command is stored in `$?`.

- **0**: Command was successful.
- **Non-zero**: Command failed.

Example:

bash Example

```bash
mkdir newdir
if [ $? -eq 0 ]; then
  echo "Directory created successfully"
else
  echo "Failed to create directory"
fi
```

12. Advanced Topics

- **Case Statement**: Switch-like control structure.

bash Example

```bash
case $1 in
  start)
    echo "Starting..."
    ;;
  stop)
    echo "Stopping..."
    ;;
  *)
    echo "Unknown command"
    ;;
esac
```

- **Trap Command**: Used to handle signals (like Ctrl+C) and run cleanup tasks.

bash Example

```bash
trap "echo 'Script interrupted!'; exit" SIGINT
```

13. Best Practices

- **Always include** `#!/bin/bash` at the top of your script.
- **Use comments** to explain sections of your script:

bash Example

```bash
# This script displays user info
```

- **Test small sections** of your script to avoid errors.
- **Check for command failures** using `$?`.
- **Use functions** to organize and reuse code.