Shell scripting

What is a Shell Script?

A shell script is a file containing commands to be executed by the shell (like , zsh). Scripts automate repetitive tasks in DevOps like backups, system monitoring, and application deployment.

How do you create and run a shell script?

To create: Use touch script.sh to create and nano script.sh to edit.
To run: Use script.sh or give execute permission (chmod +x script.sh) and run with ./ script.sh.

What is #! /bin/ at the start of a script?

Known as the "shebang," it tells the system to use /bin/ to interpret the script.

What is Shebang?

Shebang is a special character sequence #! that appears at the very beginning of a script file. It tells the operating system which interpreter to use to run the script.

How to pass arguments to a shell script?

Use $1, $2, etc., to access arguments in a script. For example, $0 is the script name, $1 is the first argument, and so on.

What are environment variables, and how do you set them in a script?

Environment variables store data like paths or settings used by the system or applications. Set with export VAR_NAME=value, and access with $VAR_NAME.

How do you handle & Debugging errors in a shell script?

Option	Simple Explanation
`-e`	Stop the script if any command fails.
`-o errexit`	Same as `-e`; exit if a command fails.
`-x`	Print each command before running it for debugging.
`-n`	Check for syntax errors without executing the script.

Explain piping and redirection.

Piping (|) passes output of one command to another.

Redirection (>, >>, <) saves command output/input to files.

Usage: When you use >, it creates a new file if the specified file does not exist, or it replaces the content of an existing file.
Usage: When you use >>, it creates a new file if it does not exist, but it does not overwrite existing content.
Usage: When you use <, it takes the content of the specified file and uses it as input for a command. (EX: sort < unsorted.txt)

How can you automate backups using shell scripting?

Use tar to compress files and cron for scheduling.

```
tar -czf backup.tar.gz /path/to/dir
```

How to read user input in a shell script?

```
echo "Enter your name:”
read name
echo "Hello, $name"
```

What is grep and how is it used in shell scripting?

grep searches text. For instance, grep "pattern" filename finds lines containing “pattern".

How to write a function in a shell script?

```
my_function() {
   echo “Hello"
}
my_function
```

How to schedule a shell script using cron jobs?

Use crontab -e to edit and add jobs in this format: * * * * * /path/to/script.sh. Here, * fields represent minute, hour, day, month, and weekday.

What is sed, and how do you use it in a script?

sed edits text in a stream.

```
sed 's/old/new/g' filename  # Replace all 'old' with 'new'
```

Explain the difference between $@ and $*.

$@ treats arguments as separate quoted strings, while $* treats them as a single string.

How to manage log files in shell scripting?

Use >> to append to logs : `echo "Script started" >> script.log`

How do you use arrays in shell scripts?

Define with arr=("item1" "item2"), access with ${arr[0]}, and loop with:

```
for i in "${arr[@]}"; do echo $i; done
```

Explain how to use traps in shell scripting.

trap captures signals (like SIGINT) to handle events. For example, to clean up on exit:

```
trap "echo 'Cleaning up'; exit" SIGINT
```

How to send an email from a shell script?

```
echo "Body" | mail -s "Subject" recipient@example.com
```

How to perform string manipulation in shell scripting?

Examples: `${#str}` for length, `${str:0:3}` for substring, `${str/old/new}` for replacement.

Explain how to work with JSON in shell scripting.

jq command parses JSON : `echo '{"name": "Hema"}' | jq '.name'`

How do you connect to a remote server in shell scripting?

Use ssh for remote connections and scp for file transfers:

```
ssh user@hostname "command"
```

How to check memory usage in shell scripting?

Use `free -h` or `vmstat` to monitor system memory.

How to write a script to monitor disk usage?

Using df -h and checking for usage:

```
df -h | awk '$5 > 80 {print "Disk usage alert"}'
```

How to use file descriptors and redirection?

File descriptors 0 (input), 1 (output), 2 (error) can be redirected:

```
command > output.txt 2>&1
```

How to implement a retry mechanism in a script?

Use a loop to retry failed commands:

```
for i in {1..5}; do
command && break || sleep 2
done
```

How to handle complex date manipulations in shell scripts?

date command supports custom formatting:
`date +%Y-%m-%d`

Explain process monitoring and managing scripts.

Use ps, top, kill, nohup, and & to manage processes. Example: ps aux | grep process_name to find processes.

How to manage service restarts in a script?

Use systemctl or service.
`systemctl restart nginx`

1. Conditional Statements

if Statement :

The if statement is used to execute a block of code only if a specified condition is true.

Syntax:

```
if [ condition ]; then
   # Commands to execute if condition is true
fi
```

Example:

```
#!/bin/
number=5
if [ $number -gt 0 ]; then
   echo "The number is positive."
fi
```

if-else Statement :

The if-else statement executes one block of code if a condition is true and another block if it is false.

Syntax:

```
if [ condition ]; then
   # Commands if condition is true
else
   # Commands if condition is false
fi
```

Example:

```
#!/bin/
number=-3
if [ $number -gt 0 ]; then
   echo "The number is positive."
else
   echo "The number is not positive."
fi
```

if-elif-else Statement :

The if-elif-else structure allows checking multiple conditions in sequence.

Syntax:

```
if [ condition1 ]; then
   # Commands if condition1 is true
elif [ condition2 ]; then
   # Commands if condition2 is true
else
   # Commands if none of the above conditions is true
fi
```

Example:

```
#!/bin/
number=0
if [ $number -gt 0 ]; then
   echo "The number is positive."
elif [ $number -lt 0 ]; then
   echo "The number is negative."
else
   echo "The number is zero."
fi
```

Example2:

```
#!/bin/
num1=10
num2=20

if [ $num1 -gt $num2 ]; then
   echo "$num1 is greater than $num2"
elif [ $num1 -lt $num2 ]; then
   echo "$num1 is less than $num2"
else
   echo "Both numbers are equal"
fi
```

Loops

for Loop :

The for loop is used to iterate over a list of items.

Syntax:

```
for variable in list; do
    # Commands to execute in each iteration
done
```

Example:

```
#!/bin/
for i in 1 2 3 4 5; do
    echo "Number: $i"
done
```

for Loop with Conditional Statements :

You can combine for loops with conditions for more complex logic.

Example:

```
#!/bin/
for i in {1..10}; do
    if [ $((i % 2)) -eq 0 ]; then
        echo "$i is even."
    else
        echo "$i is odd."
    fi
done
```

while Loop :

The while loop repeatedly executes a block of commands as long as the specified condition evaluates to true.

Syntax:

```
while [ condition ]; do
    # Commands to execute while condition is true
done
```

Example: This example counts from 1 to 5.

```
count=1
while [ $count -le 5 ]; do
    echo "Count is: $count"
    count=$((count + 1))  # Increment count
done
```

until Loop :

The until loop is the opposite of the while loop. It executes a block of commands as long as the specified condition is false.

Syntax:

```
until [ condition ]; do
    # Commands to execute until condition is true
done
```

Example: This example counts from 1 to 5 using an until loop.

```
count=1
until [ $count -gt 5 ]; do
    echo "Count is: $count"
    count=$((count + 1))  # Increment count
done
```

Feature	**`while` Loop**	**`until` Loop**
Condition	Executes while the condition is true	Executes until the condition is true
Usage	Use when you want to continue until a condition becomes false	Use when you want to stop until a condition becomes true
Flow Control	Starts if the condition is true	Starts if the condition is false

Example: Using while with a File Check - This example checks if a file exists and waits for it to appear.

```
file="/path/to/file"
while [ ! -e "$file" ]; do
    echo "Waiting for the file to be created..."
    sleep 2  # Wait for 2 seconds
done
echo "File found!”
```

Example: Using until with a User Input

This example keeps asking the user for input until they provide a valid response.

```
response=""
until [ "$response" = "yes" ]; do
   read -p "Please type 'yes' to proceed: " response
done
echo "Thank you for confirming!”
```

Nested Loops

You can also nest while and until loops inside each other to perform more complex operations.

Ex: Nested while Loop This example generates a multiplication table for the number 2.

```
number=2
count=1
while [ $count -le 10 ]; do
   echo "$number x $count = $((number * count))"
   count=$((count + 1))
don
```

Example: Nested until Loop

This example waits until a number is less than a threshold while decrementing.

```
threshold=0
number=5
until [ $number -lt $threshold ]; do
   echo "Current number: $number"
   number=$((number - 1))  # Decrement number
done
```

break: Exit the loop entirely.
continue: Skip the rest of the current iteration and proceed to the next iteration.

Example with break and continue

```
count=0
while [ $count -lt 10 ]; do
   count=$((count + 1))

   # Skip even numbers
   if [ $((count % 2)) -eq 0 ]; then
```

```
        continue
    fi

    # Break when count reaches 7
    if [ $count -eq 7 ]; then
        echo "Breaking at 7"
        break
    fi

    echo "Count is: $count"
done
```

2. Condition Syntax

String Comparisons:

["$a" = "$b"] → True if strings are equal.

["$a" != "$b"] → True if strings are not equal.

[-z "$a"] → True if string is empty.

[-n "$a"] → True if string is not empty.

Numeric Comparisons:

[$a -eq $b] → True if numbers are equal.

[$a -ne $b] → True if numbers are not equal.

[$a -gt $b] → True if $a is greater than $b.

[$a -lt $b] → True if $a is less than $b.

[$a -ge $b] → True if $a is greater than or equal to $b.

[$a -le $b] → True if $a is less than or equal to $b.

File Conditions:

[-e file] → True if file exists.

[-f file] → True if file exists and is a regular file.

[-d dir] → True if directory exists.

[-r file] → True if file has read permission.

[-w file] → True if file has write permission.

[-x file] → True if file is executable.

3. Logical Operators in Conditions

AND (&& or -a): Both conditions must be true.

```
if [ condition1 ] && [ condition2 ]; then

    # Commands if both are true
fi
```

OR (|| or -o): At least one condition must be true.

```
if [ condition1 ] || [ condition2 ]; then

    # Commands if either is true
fi
```

NOT (!): Negates a condition.

```
if [ ! condition ]; then

    # Commands if condition is false
fi
```

Example1: Check if a File Exists

```
if [ -e "/path/to/file" ]; then
    echo "File exists."
else
    echo "File does not exist."
fi
```

Example2: Check for Empty or Non-Empty String

```
name=""
if [ -z "$name" ]; then
    echo "Name is empty."
else
    echo "Name is not empty."
fi
```

Example 3: Multiple Conditions with elif

```
num=20
if [ $num -lt 10 ]; then
   echo "Number is less than 10."
elif [ $num -ge 10 ] && [ $num -le 20 ]; then
   echo "Number is between 10 and 20."
else
   echo "Number is greater than 20."
fi
```

Example 4: Check Read/Write Permissions

```
file="/path/to/file"
if [ -r "$file" ] && [ -w "$file" ]; then
   echo "File has read and write permissions."
else
   echo "File does not have read and/or write permissions."
fi
```

Example 5: Nested Conditions

```
if [ "$USER" = "hemalatha" ]; then
   if [ -d "/home/hemalatha" ]; then
      echo "Directory exists and is user hemalatha."
   fi
else
   echo "User is not hemalatha or directory does not exist."
fi
```

Advanced Conditional Use Cases

Checking a Service Status:

```
if systemctl is-active --quiet nginx; then

   echo "Nginx is running"
else
   echo "Nginx is not running"
fi
```

Number Comparison with Logical Operators:

```
num=15

if [ $num -gt 10 ] && [ $num -lt 20 ]; then
```

```
    echo "Number is between 10 and 20."
else
    echo "Number is out of range."
fi
```

Using Conditions in Loops:

```
for i in {1..5}; do

    if [ $i -eq 3 ]; then
        echo "Reached 3, stopping the loop."
        break
    fi
    echo "Number: $i"
done
```

Multiline Comments

Shell scripts do not have a direct syntax for multiline comments, but you can use a trick by using a : <<'COMMENT' ... COMMENT block or simply prefix lines with #.

Example:

```
#!/bin/
: << 'COMMENT'
This is a multiline comment.
You can add as many lines as you want.
COMMENT
```

Or by using # on each line:

```
#!/bin/
# This is a multiline
# comment using the #
# symbol at the start of each line.
```

EOF (End of File)

EOF is used in shell scripting to denote the end of a block of code, typically with here documents. Here documents allow multiline input or output.

Syntax:

```
cat << EOF
This is a
multiline text
block.
EOF
```