Shell Scripting Advanced

Day 1 — Advanced Shell Concepts & Debugging

1. Creating an Advanced Script
Example: nodeHealth.sh

```
vim nodeHealth.sh
```

Insert the script:

```
#!/bin/bash

########################
# Author: Karthik
# Date: 10/11/2025
#
# This script outputs the node health
#
# Version: v1
########################

df -h
free -g
nproc
```

Explanation:

- **df -h** → shows disk usage in human readable format
- **free -g** → prints memory usage in GB
- **nproc** → displays CPU core count

Save using **:wq!**

Give execution permission:

```
chmod 777 nodeHealth.sh
```

Run the script:

```
./nodeHealth.sh
```

Output appears, but raw — not user-friendly.

2. Improving Readability — Adding Meaningful Output

Modify script:

```
#!/bin/bash

########################
# Author: Karthik
# Version: v2
########################

echo "=== Disk Space ==="
df -h

echo "=== Memory Info ==="
free -g

echo "=== CPU Count ==="
nproc
```

Result

Outputs become descriptive and easier to read.

3. Debug Mode (set -x)

Debugging helps trace execution flow.

Updated Script:

```
#!/bin/bash
########################
# Version: v3
########################

set -x

df -h
free -g
nproc
```

- **set -x** prints commands before executing them
- Used to validate logic and troubleshoot scripts

Example debug output:

```
+ df -h
+ free -g
+ nproc
```

4. Switching Users and Privileges
To become root:

```
sudo su -
```

To login as another user:

```
su <username>
```

Used in automation scripts where administrative or user-specific access is needed.

Day 2 — Pipes, Filters, Text Processing & Error Handling

1. Process Monitoring
Full process listing:

```
ps -ef
```

- **e** → all processes
- **f** → full format tree view

Searching a specific process:

```
ps -ef | grep amazon
```

| (pipe) sends output of the first command to second.

2. Using Pipes for Output Filtering

Example Script:

```
vim test.sh
```

Contents:

```
echo 1
echo 11
echo 12
echo 55
echo 99
```

Run:

```
./test.sh | grep 1
```

Prints lines containing 1.

Key Behaviour Notes

```
date
```

Output:

```
Thu Dec 1 12:08:41 UTC 2023
```

Now run:

```
date | echo "today is"
```

Output:

```
today is
```

Because **echo** overrides pipe input — **date** output goes into stdin but isn't printed.

3. Combining ps, grep, awk

Sometimes we need a specific field (like PID).
Command:

```
ps –ef | grep amazon | awk –F" " '{print $2}'
```

- **grep amazon** → filter lines
- **awk** extracts column 2 (PID)

Example: Text Extraction

File:

```
vim test
```

Contents:

```
My name is Karthik
My colleague is Engineer
```

Run:

```
grep name test | awk –F" " '{print $4}'
```

Output:

```
Karthik
```

awk '{print $4}' prints 4th word from matching lines.

4. Script Safety Controls — Error Handling

Scripts can silently fail when commands pipe outputs.
To prevent this, add:

```
set –x         # Trace debug
set –e         # Stop script if any command fails
set –o pipefail  # Detect failure inside piped commands
```

Why Important?

In large automation pipelines:

- A command may fail silently.
- **set -e** ensures script stops immediately.
- **set -o pipefail** ensures pipe failures are detected.

Helps debugging and reliability.

5. curl & wget — Retrieving Remote Data

Command:

```
curl <URL>
```

Fetches data from REST APIs / web services / logs.
Example:

```
curl –X GET api.foo.com
```

Filtering Errors in Logs:

```
curl <log-file-url> | grep ERROR
```

To learn options:

```
man curl
```

wget

```
wget <URL>
```

Downloads files/content directly.

6. Searching Files Efficiently — find Command

Useful when file path is unknown.
Example:

```
find / -name pam.d
```

Searches whole filesystem for file matching "pam.d".

7. Conditional Statements — Decision Making

If-Else Example:

```
vim ifelse.sh
```

Content:

```
#!/bin/bash

a=4
b=10

if [ $a -gt $b ]
then
    echo "a is greater than b"
else
    echo "b is greater than a"
fi
```

Run:

```
./ifelse.sh
```

-gt means greater than.

8. Looping — Repetitive Execution
For Loop Example

```
vim forloop.sh
```

Correct version:

```
#!/bin/bash

for i in {1..100}
do
    echo $i
done
```

Notes:

• **{1..100}** → sequence 1 through 100
• Loop executes 100 times
Loops are used to automate repeated actions — file creation, monitoring, iteration, etc.

Additional Best Practices

Always start scripts with **#!/bin/bash**
Include header comments (author, purpose, version)
Use **set -e -o pipefail** for production scripts
Use **echo** statements for readable output
Use debug (**set -x**) only when testing
Validate user permissions when handling system components

Suggested Practice Tasks

1. Write a script to monitor top 5 memory consuming processes.
2. Write script to ping multiple hosts in loop and print alive/dead status.
3. Write script to extract only ERROR logs from remote file using curl + grep.
4. Write script using awk to extract usernames from **/etc/passwd**