Shell Scripting – Interview Questions & Answers

1. What are some commonly used shell commands?

Command	Purpose	Explanation
ls	List files	Lists all files/directories
file filename	Identify file type	Checks if file is text/binary/etc.
ps -ef	View running processes	Shows all running processes
top	Monitor real-time usage	Displays CPU and memory usage
sar	System activity history	Shows system activity reports
df -h	Check disk usage	Human-readable disk space usage
tail -f logfile.log	View logs in real time	Live log monitoring
grep pattern file	Search text	Finds matching text
wc -w, wc -l	Count words/lines	Word and line count
echo "text"	Print text	Display output
cd path	Change directory	Navigate folders
free -m	Memory info	Shows RAM usage

2. Write a shell script to list all running processes.

```
#!/bin/bash
ps -ef
```

Explanation:

ps -ef lists all running processes with PID, UID, PPID, and command.

To print only Process IDs:

ps -ef | awk '{print $2}'

3. How do you fetch only ERROR logs from a remote server?

```
        curl <remote-link> | grep ERROR
```

Explanation:
curl retrieves logs from the remote server, and **grep ERROR** filters lines containing the word "ERROR".

4. Write a script to print numbers divisible by 3 and 5 but not by 15 (within 1–100).

```
    #!/bin/bash
    for i in {1..100}; do
        if ( [ $(expr $i % 3) == 0 ] || [ $(expr $i % 5) == 0 ] ) && [ $
(expr $i % 15) != 0 ]; then
    echo $i
        fi
    done
```

Other variations:

Requirement	Condition
Even numbers	i % 2 == 0
Odd numbers	i % 2 != 0
Divisible by 3	i % 3 == 0
Prime numbers	Check divisibility inside loop

5. Write a script to count the number of “S” in the word Mississippi.

```
        #!/bin/bash
        x=mississippi
        grep -o "s" <<< "$x" | wc -l
```

Explanation:
grep -o prints each match on a new line, and **wc -l** counts the lines.

6. How do you debug a shell script?

Add this at the beginning of the script:

```
set -x
```

This displays each command before execution, helping with debugging.

7. What is crontab in Linux? Give an example.

Concept:
crontab is used to **schedule commands or scripts to run automatically at specific times** (like a built-in task scheduler).

Key points to say in an interview:
- It's time-based job scheduling.
- Good for repeated tasks: backups, reports, cleanup, health checks, etc.
- Uses a special syntax: **minute hour day-of-month month day-of-week command**.

Example scenario:
"I want a server health report to be generated every day at 6 PM automatically."

crontab entry:

```
0 18 * * * /home/admin/health_report.sh
```

Explanation (very short):
- 0 → minute (0th minute)
- 18 → hour (6 PM, 24-hour format)
- * * * → every day, every month, every day of week
- Command: /home/admin/health_report.sh

8. How do you open a file as read-only?

If you want to **view a file without accidentally modifying it**, you can open it in read-only mode.

Command:

```
vim -R test.txt
```

Explanation:
- **vim** → editor
- **-R** → read-only mode
  You can still quit normally (**:q**), but it will **not let you save changes** to overwrite the original file (unless you force it).

9. What is the difference between a soft link and a hard link?

Think of:

- **Hard link** → another *name* for the same file content.
- **Soft link (symbolic link)** → a *shortcut* pointing to another path.

Hard Link

- Points directly to the data on disk (same inode).
- If the original file is deleted, the hard link still works.
- Cannot span across different filesystems.

```
Command:
ln original.txt copy_hard.txt
```

Soft Link (Symbolic Link)

- Points to the **path** of the file, not directly to the data.
- If the original file is deleted or moved, the soft link becomes **broken**.
- Can point to directories or files across different filesystems.

```
Command:
ln -s original.txt link_soft.txt
```

One-line explanation for interview:

> A hard link is another reference to the same data, while a soft link is like a shortcut to the file path.

10. What is the difference between break and continue statements?

These are used inside loops (**for**, **while**, **until**) to control the flow.

break

- **Exits** the loop immediately.
- No further iterations are executed.

```
for i in {1..5}; do
  if [ "$i" -eq 3 ]; then
    break    # stops the loop when i=3
  fi
  echo "$i"
done
```

Output: 1 2

continue

- **Skips the current iteration** and moves to the next one.

```
for i in {1..5}; do
  if [ "$i" -eq 3 ]; then
    continue   # skip printing 3
  fi
  echo "$i"
done
```

Output: 1 2 4 5

One-liner:

> **break** stops the loop, **continue** skips one turn and continues.

11. What are some disadvantages of shell scripting?

1. **Not ideal for large, complex applications**
   - Shell scripts become hard to read and maintain when logic grows big.
1. **Performance can be slower**
   - Many shell commands spawn new processes each time, which is heavier than in languages like C, Go, or Java.
1. **Limited data structures**
   - No rich built-in types like dictionaries, complex objects, etc. Usually just strings, arrays, and simple constructs.
1. **Error handling and debugging are tricky**
   - Small syntax mistakes (missing quotes, spaces, brackets) can break the script.
   - Need to rely on **set -x**, **set -e**, and manual echo debugging.
1. **Portability issues**
   - Scripts written for one shell or OS might not behave exactly the same in others (bash vs sh, Linux vs macOS).

summarize in an interview as:

> Shell scripting is great for automation and glue tasks, but for big, complex, performance-critical applications, other languages are more suitable.

12. What are the different types of loops in shell scripting and when do you use them?

In bash, the main loops are:

1. for loop
Used when you **know the range or list** in advance.

```
for i in {1..5}; do
  echo "$i"
done
```

Or through items:

```
for file in *.log; do
  echo "$file"
done
```

2. while loop
Used when you want to **keep looping as long as a condition is true**.

```
count=1
while [ $count -le 5 ]; do
  echo "$count"
  count=$((count + 1))
done
```

3. until loop
Opposite of **while**. It **runs until the condition becomes true**.

```
count=1
until [ $count -gt 5 ]; do
  echo "$count"
  count=$((count + 1))
done
```

Short answer to remember:

> Use **for** known sets, **while** when you keep checking a condition, and **until** when you loop until a condition is satisfied.

13. Is bash dynamically or statically typed? Why?

Answer:
Bash is **dynamically typed**.

Reason:
You can assign different types of values to the same variable without declaring its type.

```
x=5
echo "$x"   # number

x="hello"
echo "$x"   # string
```

No type is declared like:

```
var x int    # in Go (static)
```

In bash:

- No **int**, **string**, etc. declaration.
- Everything is generally treated as a string, and interpreted as number when used in arithmetic.

One-line interview answer:

> Bash is dynamically typed because variables don't have fixed types and can hold different kinds of values at different times.

14. Explain a network troubleshooting utility you use in Linux.

can talk about **traceroute** (or **tracepath**).

Command:

```
traceroute google.com
```

or

```
tracepath google.com
```

Explanation:

- These commands show the **path (hops)** your packets take from your machine to the destination.
- Helpful to find:
  - Where network delay is happening.
  - If some router hop is dropping packets.
  - If there is a routing issue between your system and the target.

Short explanation for interview:

> I use **traceroute** or **tracepath** to see each hop between my system and a target server, which helps locate where the latency or failure occurs in the network path.

15. How will you sort a list of names in a file?

Assume the file is **names.txt** with one name per line.

Command:

```
sort names.txt
```

If you want to **remove duplicates** as well:

```
sort names.txt | uniq
```

For **reverse order:**

```
sort -r names.txt
```

Short explanation:

> **sort** reads the file line by line and prints the lines in sorted order.

16. How will you manage logs of a system that generates huge log files every day?

In real environments, we use a tool like **logrotate**.

What logrotate does:

- Rotates logs (creates new logfile and renames old ones).
- Compresses old logs (e.g., to **.gz**) to save space.
- Deletes very old logs after a certain number of days/rotations.
- Can be configured to run daily via cron.

Example concept:

For an application writing to **/var/log/app.log**, I configure **logrotate** to:

- Rotate the log daily,
- Keep last 7 days,
- Compress older files,
- And optionally restart the service after rotation.

A sample **logrotate** config (just for understanding, not to memorize):

```
/var/log/app.log {
    daily
    rotate 7
    compress
    missingok
    notifempty
}
```

One-line interview answer:

> I would configure **logrotate** to rotate, compress, and clean old logs periodically so that disk usage stays under control.