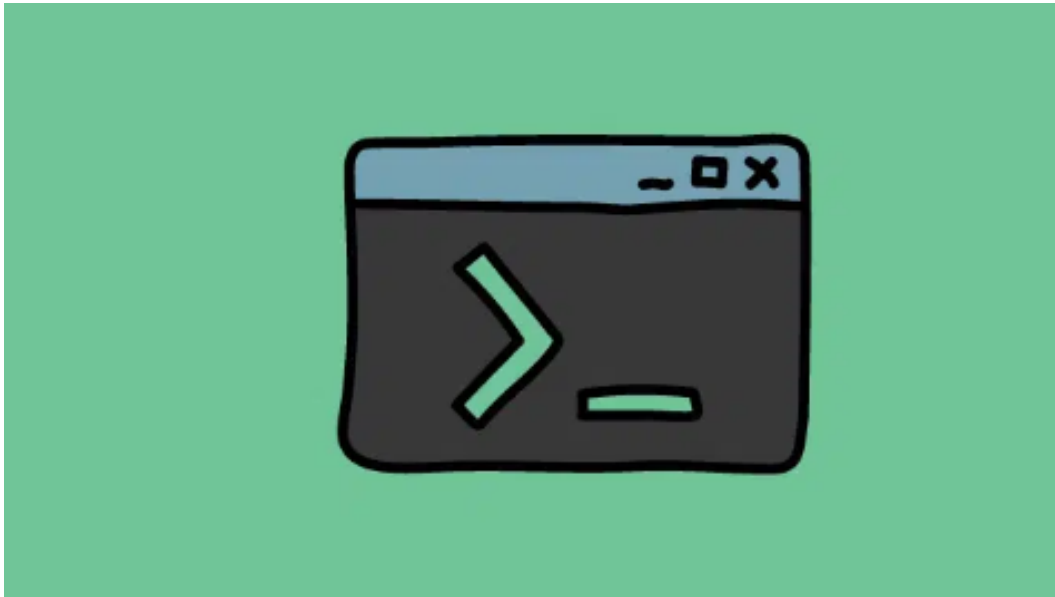


What is Shell scripting?

Shell scripting is a way to **automate tasks** in Linux and other Unix-based operating systems. (or) Shell scripting refers to the process of writing scripts or sequences of commands in a shell (command-line interpreter) to automate tasks, perform system operations, and manage various processes. A shell script is essentially a text file containing a series of commands that the shell interprets and executes sequentially.



- A shell script is a plain text file containing a sequence of shell commands. These commands are read and executed by the shell interpreter (bash, sh, zsh etc).
- Shell scripts have a .sh extension and need to start with a shebang like `#!/bin/bash` to indicate the shell interpreter.
- Scripts can do anything that you can do at the command line - execute commands, take input, print output, conditional logic, loops, functions etc.
- Common operations done via shell scripts include file manipulation, program execution, printing text, variable assignment, user input, and calling system commands.
- Scripts allow automation of repetitive tasks, complex sequential steps, scheduled jobs etc. saving effort and time.
- Scripts can take arguments and have access to environment variables and shell variables.
- Control flow statements like if-else, for loops, while loops, case statements provide logic and flow.
- Useful commands include echo, read, expr, test, seq, grep, sed, awk etc. for printing, math, text processing, conditional tests.
- Scripts can be made into executable files with `chmod +x` and executed directly.

- Overall, shell scripting provides a way to programmatically control the shell and automate admin tasks, reducing manual effort. Knowledge of basic scripting is very useful for Linux/Unix admins and power users.

What is shebang?

The shebang (also called hashbang) refers to the special two characters '#' that occur as the first two characters in scripts for many languages including shell scripts.



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

Some key points about the shebang:

- It is written as `#!` at the start of scripts.
- It is followed by the full path to the interpreter that should execute the script.
- For example, `#!/bin/bash` indicates that the script should be executed by the bash interpreter.
- `#!/usr/bin/python` would indicate the python interpreter should be used.
- The shebang allows scripts to be executed directly on the command line rather than needing the interpreter specified explicitly.
- When a script starting with `#!` is executed directly, the program loader parses the shebang line to find the interpreter to run.
- This makes the scripts portable and executable on any system where the specified interpreter is installed.
- Shebang is also called hashbang due to the leading `#` character used in Unix based systems for comments.
- It is common to see `#!/bin/sh` used for portable bourne shell compatibility.
- Scripts without a shebang may fail to execute properly or require the interpreter path to be explicitly specified.
- Shebang needs to be on the very first line of the script file.

- It allows `chmod +x` to be used on scripts to make them executable.

What is the purpose of `#!/bin/bash` or `#!/bin/sh`?

The purpose of `#!/bin/bash` or `#!/bin/sh` at the beginning of a script is to specify the interpreter that should be used to execute the script. These shebang lines serve as directives to the operating system on how to run the script.

- **`#!/bin/bash`:** This shebang indicates that the script should be executed using the Bash shell interpreter. Bash (Bourne-Again SHell) is a popular Unix shell with extended capabilities beyond the basic Bourne shell.
- **`#!/bin/sh`:** This shebang specifies that the default system shell should be used to execute the script. In many Unix-like systems, `/bin/sh` is a symbolic link to the system's default shell, which is typically a POSIX-compatible shell like `dash`, `bash`, or `zsh`.

What is the difference between `ksh`, `bash` and `dash`?

The Bourne shell (`sh`), Korn shell (`ksh`), and `bash` are all command interpreters or shells that are used to interact with the Linux operating system. They are all POSIX-compliant, meaning that they adhere to the POSIX standard for shell scripting. However, there are some key differences between these shells.



- **`sh`** is the original Bourne shell, and it is the simplest of the three shells. It is not as feature-rich as `ksh` or `bash`, but it is still a popular choice for simple shell scripts.
- **`ksh`** is a more advanced shell than `sh`. It includes features such as command history, job control, and shell functions. `ksh` is also more efficient than `sh`, making it a good choice for interactive use.
- **`bash`** is a newer shell that is based on `ksh`. It includes all of the features of `ksh`, plus a number of additional features, such as programmable completion, syntax highlighting, and a

variety of user-configurable options. bash is the most popular shell on Linux distributions, and it is the default shell for many distributions.

Here is a table that summarizes the key differences between ksh, bash, and dash:

Feature	ksh	bash	dash
POSIX compliance	Yes	Yes	Yes
Features	Command history, job control, shell functions	Command history, job control, shell functions, programmable completion, syntax highlighting, user-configurable options	Command history, job control, shell functions
Efficiency	More efficient than sh	Less efficient than ksh	Most efficient
Popularity	Less popular than bash	Most popular	Less popular than bash

In general, ksh is a good choice for interactive use, while bash is a good choice for scripting. Dash is a good choice for systems where performance is critical, such as embedded systems.

Here are some additional details about the differences between ksh, bash, and dash:

- **ksh** has better support for loop handling than bash.
- **bash** has a more powerful scripting language than ksh.
- **dash** is a smaller and faster shell than ksh or bash.

Ultimately, the best shell for you will depend on your specific needs. If you are looking for a simple shell for interactive use, then sh or dash may be a good choice. If you are looking for a more powerful shell with more features, then ksh or bash may be a better choice. If performance is critical, then dash is the best choice.

How to write a simple shell script?

Writing a simple shell script involves creating a text file containing shell commands and saving it with a `.sh` extension. Here's a step-by-step guide to creating a basic shell script:

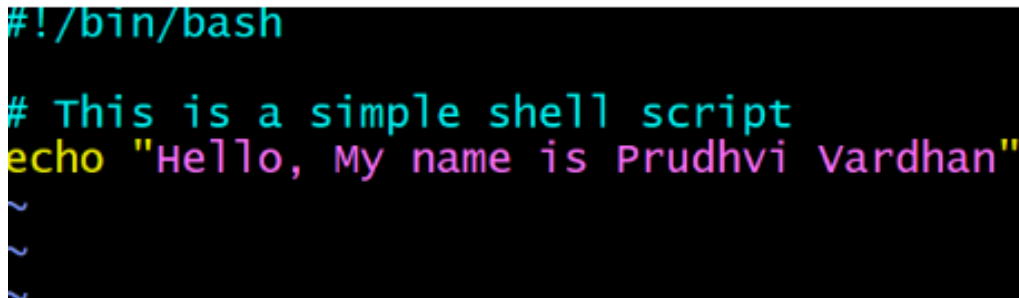
1. **Choose a Text Editor:** You can use any text editor to write your shell script. Popular options include Notepad (Windows), nano (Linux/macOS terminal), Sublime Text, Visual Studio Code, and others.
2. **Open a New File:** Open your chosen text editor and create a new blank file.
3. **Add Shebang Line:** Start with a shebang line that specifies the interpreter. For example, to use Bash:

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

4. **Write Shell Commands:** Add your shell commands to the script file. You can start with something simple, like printing a message:

```
#!/bin/bash

# This is a simple shell script
echo "Hello. My name is Prudhvi Vardhan"
```



```
#!/bin/bash

# This is a simple shell script
echo "Hello, My name is Prudhvi Vardhan"
```

5. **Save the File:** Save the file with a `.sh` extension. For example, `myscript.sh`.
6. **Make the Script Executable:** Before running the script, you need to make it executable. Open your terminal and navigate to the directory containing the script. Use the `chmod` command to give the script execute permissions:

```
chmod +x myscript.sh
```

7. **Run the Script:** Now you can run the script using the `./` prefix and the script's filename:
- ```
./myscript.sh
```

If you encounter a permissions error, ensure that you've made the script executable (step 6).

That's it! You've created and executed a simple shell script. As you become more comfortable with shell scripting, you can gradually add more complex commands, logic, variables, functions, and interactions to your scripts. Experimentation and practice will help you learn and refine your shell scripting skills.

```
user@Prudhvi MINGW64 ~ (master)
$ chmod 700 myscripts.sh

user@Prudhvi MINGW64 ~ (master)
$./myscripts.sh
Hello, My name is Prudhvi Vardhan

user@Prudhvi MINGW64 ~ (master)
$
```

## How to check CPU and RAM of a Linux Machine?

### To check CPU usage:

- **top**: Displays real-time process activity and CPU usage. Press Shift+P to sort processes by CPU usage.
- **mpstat**: Reports processor-related statistics. Use `mpstat -P ALL` to see usage per CPU core.
- **lscpu**: Provides information about the CPU architecture.
- **uptime**: Shows the uptime and average CPU load over 1, 5, and 15-minute intervals.

### To check RAM usage:

- **free -m**: Shows used and free memory in megabytes.
- **htop**: Interactive process viewer where you can sort by memory usage.
- **cat /proc/meminfo**: Gives a detailed output of memory usage statistics.
- **vmstat**: Reports virtual memory statistics, including used, free, buffered, and cached memory.

### To check both CPU and RAM usage:

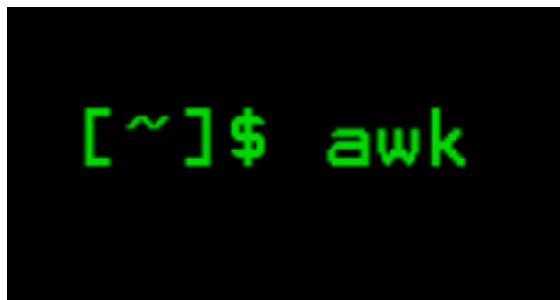
- **top**: Interactive process viewer that shows CPU and memory usage per process.
- **glances**: Multi-subsystem overview with CPU, RAM, disk, network usage in a terminal UI.
- **nmon**: Analyzes system performance with interactive CPU, memory, processes, disks, network, etc.

You can also use graphical applications like GNOME System Monitor or KSysGuard for a friendlier interface.

In summary, `top`, `htop`, `free`, and `glances` are easy-to-use interactive commands to quickly check both CPU and RAM usage on a Linux system. These commands provide valuable insights into the performance of your machine.

## What is AWK command?

AWK is a versatile text processing and data extraction tool available on Unix-like systems, including Linux. It's primarily used for scanning and processing text files, extracting data, and performing various operations based on patterns. AWK is often used in shell scripts and command-line pipelines for tasks such as data manipulation, reporting, and analysis.



### Basic Syntax:

```
awk 'pattern { action }' input_file
```

- `pattern` : Specifies the condition to match.
- `action` : Defines the action to perform if the pattern matches.
- `input_file` : The file you want to process. If not specified, AWK reads from standard input.

### Common Uses:

- **Print Specific Columns:**

```
awk '{ print $1, $3 }' input.txt # Prints the 1st and 3rd columns
```

- **Conditional Statements:**

```
awk '{ if ($3 > 50) print $1, "Pass"; else print $1, "Fail" }' input.txt
```

- **Calculations:**

```
awk '{ total += $2 } END { print "Total:", total }' input.txt
```

- **Pattern Matching:**

```
awk '/error/ { print $0 }' logfile.txt # Prints lines containing
"error"
```

- **FNR and NR:** FNR represents the record number in the current file, while NR represents the overall record number.

```
awk '{ print FNR, NR, $0 }' file1.txt file2.txt
```

- **Using Field Separators:** AWK splits records into fields based on the delimiter (default is whitespace).

```
awk -F',' '{ print $2 }' data.csv # Prints the 2nd field using co
mma as delimiter
```

- **Awk Built-in Functions:** AWK provides various built-in functions like `length()` , `substr()` , `tolower()` , etc.

```
awk '{ print length($0) }' input.txt # Prints the length of each
line
```

- **BEGIN and END Blocks:** Code in the `BEGIN` block executes before processing, and code in the `END` block executes after processing.

```
awk 'BEGIN { print "Start" } { print $0 } END { print "End" }' inpu
t.txt
```

## What is set -x command?

The `set -x` command is used in shell scripting to enable debugging mode. When you include `set -x` in your script, the shell will print each command before it's executed. This is useful for troubleshooting scripts and understanding how they behave at each step.

### Example:

Let's say you have the following simple shell script named `myscript.sh` :

```
#!/bin/bash

set -x # Enable debug mode

echo "Starting script..."
x=10
echo "Value of x is $x"
result=$((x + 5))
echo "Result is $result"
echo "Script completed"
```

With `set -x` enabled, when you run the script using `./myscript.sh` , the output would look something like this:



```
+ echo 'Starting script...'
Starting script...
+ x=10
+ echo 'Value of x is 10'
Value of x is 10
+ result=15
+ echo 'Result is 15'
Result is 15
+ echo 'Script completed'
Script completed
```

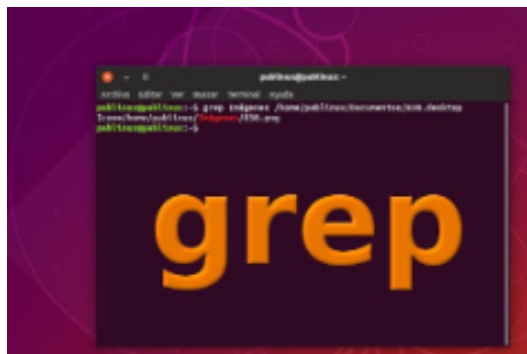
As you can see, each command is preceded by a `+` sign, indicating that it's being executed in debug mode. This helps you track the flow of execution and identify any issues in your script.

### Note:

- Be cautious when using `set -x` in production scripts or when handling sensitive information, as it can display sensitive data in the terminal.
- To turn off debugging mode, you can use `set +x`.

Debugging mode ( `set -x` ) is a valuable tool for understanding how your script works and finding errors. It's particularly useful when you're troubleshooting complex scripts or trying to identify the source of unexpected behavior.

## What is Grep command ?



### Pattern: Grep - Powerful Text Search Utility

#### Key Points:

- **Searching Text:** `grep` is a command-line utility to search text in Unix/Linux systems.
- **Basic Syntax:** It searches for a regular expression pattern in a text file and prints matching lines.
- **Usage:** `grep [options] 'pattern' file`
- **Common Options:**

- `-i` : Ignore case
  - `-R` : Recursive search
  - `-A` , `-B` , `-C` : Print lines around match
  - `-c` : Print count of matching lines
  - `-v` : Inverted search, print non-matching lines
- **Regular Expressions:** It supports regex like `*` , `.` , `^` , `$` , `[]` for flexible pattern matching.
  - **Multiple Patterns:** Use the OR `|` operator to search for multiple patterns.
  - **Case Sensitivity:** By default, `grep` searches case-sensitively.
  - **Recursive Search:** `-R` searches recursively in all files in directories.
  - **Context Display:** `-C` provides context around the matching line.
  - **Inverted Search:** `-v` inverts the match, printing non-matching lines.
  - **Versatile Tool:** Ideal for quickly searching patterns across text files.
  - **Pipe Usage:** Often combined with pipes and other commands, like `ps aux | grep .`
  - **Name Origin:** The name `grep` comes from the `ed` command `g/re/p` , meaning globally search for a regex and print.

In summary, `grep` is an essential tool for searching and filtering text patterns on the command-line in Unix/Linux systems. Its versatility, regex support, and integration with other commands

## What is find command ?

**Pattern: Find - Search for Files and Directories**



### Key Points:

- **Purpose:** `find` is used to search for files and directories based on specified criteria within a directory hierarchy.
- **Basic Syntax:**

```
find [path] [options] [expression]
```

- **Search Path:** [path] specifies the starting directory for the search. If not provided, the current directory is used.
- **Search Expression:** [expression] defines the search criteria, e.g., -name , -type , -size , etc.
- **Common Options:**
  - -name pattern : Search by filename pattern using wildcards.
  - -type type : Search by file type (e.g., f for regular files, d for directories).
  - -size [+/-]size : Search by file size (e.g., +10M for larger than 10MB).
  - -exec command {} \; : Execute a command on each found item.
  - -print : Display the path of found items.

- **Examples:**

- Search for files named example.txt :

```
find /path/to/search -name "example.txt"
```

- Search for directories:

```
find /path/to/search -type d
```

- Search for files larger than 100MB:

```
find /path/to/search -type f -size +100M
```

- Delete all .log files:

```
find /path/to/search -name "*.log" -exec rm {} \;
```

- **Advanced Expressions:**

- Combine expressions using -a (AND), -o (OR), ! (NOT).
- Group expressions using parentheses.

- **Search Entire System:**

- Use / as the search path to start from the root directory.

- **Use Cases:**

- Locating files by name, type, size.
- Performing actions on found files using -exec .

- **Efficiency Tip:** Use specific directories and criteria to limit the search scope.

- **Find vs. Grep:** `find` searches file metadata, while `grep` searches file content.
- **Complex Searches:** Can be used in complex scenarios for intricate searches.

In summary, the `find` command is a versatile tool for searching and locating files and directories based on various criteria. Its flexibility, combined with options for executing

## What is Pipe Command ?

a pipe ( `|` ) is a special character that enables the output of one command to be used as the input of another command. Pipes allow you to create a sequence of commands, where the output of the previous command becomes the input for the next command. This mechanism is referred to as "piping" or creating a "pipeline."



Here's how pipes work:

### 1. Command 1 | Command 2:

- Command 1 produces some output.
- The `|` symbol directs the output of Command 1 as the input to Command 2 .
- Command 2 processes the input from Command 1 .

### 2. Example:

```
cat file.txt | grep "keyword"
```

- `cat file.txt` displays the contents of `file.txt` .
- `|` pipes the output to `grep` .
- `grep "keyword"` searches for lines containing the keyword in the output of `cat` .

Pipes are incredibly powerful and versatile, allowing you to chain commands together to perform complex operations without the need to store intermediate results in files. This is a fundamental concept in Unix/Linux command-line scripting and is used for data processing, filtering,

transforming, and much more.

## What is ps -ef command ?

The `ps -ef` command is used to display information about running processes on a Unix-like operating system, such as Linux. This command provides a detailed list of all processes currently running on the system.

```
crio-user:~$ ps
 PID TTY TIME CMD
 14474 pts/0 00:00:00 bash
 16496 pts/0 00:00:00 nc
 16854 pts/0 00:00:00 ps
crio-user:~$
crio-user:~$
```

Here's what each part of the command does:

- `ps` : This is the command itself, which stands for "process status." It's used to query information about processes.
- `-ef` : These are options or flags that modify the behavior of the `ps` command:
- `-e` : This option selects all processes on the system, regardless of their terminal associations.
- `-f` : This option provides a "full-format" output, which displays a more detailed view of process information, including the process hierarchy (parent-child relationships).

So, when you run `ps -ef`, you're telling the system to show you information about all processes in a detailed, full-format view.

Here's an example output of `ps -ef` :

| UID  | PID | PPID | C | STIME | TTY | TIME     | CMD                      |
|------|-----|------|---|-------|-----|----------|--------------------------|
| root | 1   | 0    | 0 | Aug13 | ?   | 00:00:01 | /usr/lib/systemd/systemd |
| root | 2   | 0    | 0 | Aug13 | ?   | 00:00:00 | [kthreadd]               |
| root | 3   | 2    | 0 | Aug13 | ?   | 00:00:00 | [ksoftirqd/0]            |
| ...  |     |      |   |       |     |          |                          |

In this output, you can see information columns such as `UID` (user ID), `PID` (process ID), `PPID` (parent process ID), `CMD` (command being executed), and more. This information is helpful for understanding what processes are currently running on the system and their relationships to each other.

## what are set -e and set -o pipefail commands?

Both `set -e` and `set -o pipefail` are shell options in Unix-like operating systems, including Bash. They affect the behavior of your shell scripts by controlling error handling and exit statuses.

```
set -eo pipefail

git && foo

+ git && foo

sh -c "git && foo"

+ sh -c "git && foo"
/opt/atlassian/pipelines/agent/trp/shellScript3581893656372674292.sh: line 16: git: not found
sh: git: not found
```

### 1. `set -e` :

- When `set -e` is enabled in a shell script, it means that the script will exit immediately if any command it runs returns a non-zero exit status (indicating an error).
- This is useful to ensure that the script stops execution as soon as an error occurs, preventing further unintended or incorrect actions.
- Example:

```
#!/bin/bash
set -e

echo "This will print."
ls non_existent_directory # This will cause an error and exit the
script.
echo "This will not print."
```

### 2. `set -o pipefail` :

- When `set -o pipefail` is enabled, it means that if any command in a pipeline (commands connected by pipes) returns a non-zero exit status, the exit status of the entire pipeline will also be non-zero.
- This option helps you accurately capture errors that might occur in any part of a pipeline.
- Example:

```
#!/bin/bash
set -e
set -o pipefail

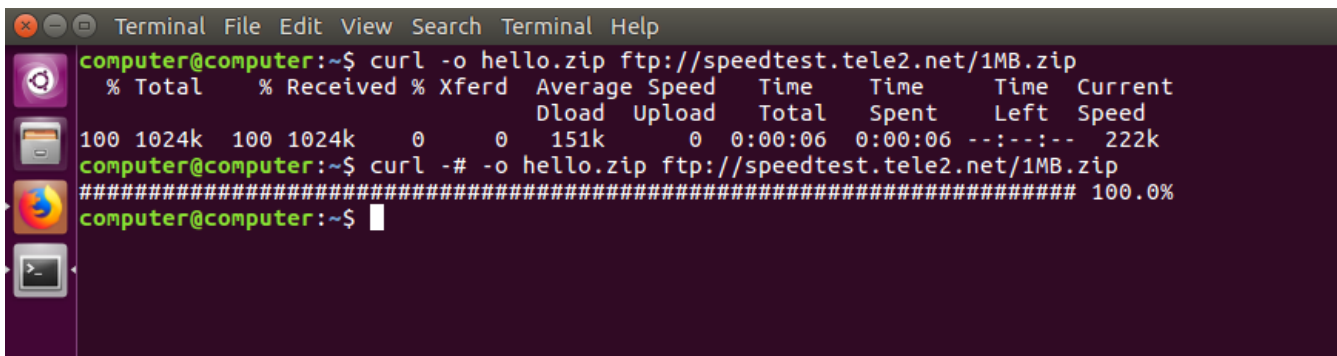
cat non_existent_file | grep "pattern" # Both commands fail, and
the script exits with non-zero status.
echo "This will not print."
```

When combining both options, the script will immediately exit on the first command that fails, even within a pipeline, and the exit status of the script will be non-zero.

It's important to use these options carefully, as they can impact the behavior of your scripts. They

## What is CURL command ?

curl is a command-line tool and library for transferring data with URLs. It supports various protocols, including HTTP, HTTPS, FTP, FTPS, SCP, SFTP, LDAP, and more. With curl, you can interact with web servers, download/upload files, and perform various operations over different network protocols directly from the command line.



```
Terminal File Edit View Search Terminal Help
computer@computer:~$ curl -o hello.zip ftp://speedtest.tele2.net/1MB.zip
% Total % Received % Xferd Average Speed Time Time Time Current
 % 0 0 151k 0 0:00:06 0:00:06 --:--:-- 222k
computer@computer:~$ curl -# -o hello.zip ftp://speedtest.tele2.net/1MB.zip
100.0%
computer@computer:~$
```

Here are some common use cases and examples of how to use curl :

1. **HTTP GET Request:** To make a simple GET request to a website:

```
curl https://www.example.com
```

2. **HTTP POST Request:** To send data in the body of a POST request:

```
curl -X POST -d "key=value" https://api.example.com/endpoint
```

3. **Download a File:** To download a file from a URL:

```
curl -o filename.ext https://www.example.com/file.ext
```

4. **Upload a File:** To upload a file using POST:

```
curl -X POST -F "file=@path/to/file" https://api.example.com/upload
```

5. **Follow Redirects:** To follow redirects and show verbose output:

```
curl -L -v https://www.example.com
```

6. **Set Headers:** To set custom headers in a request:

```
curl -H "Authorization: Bearer token" https://api.example.com/protected
```

7. **HTTP Basic Authentication:** To use HTTP Basic Auth:

```
curl -u username:password https://api.example.com
```

8. **Save Cookies and Use in Subsequent Requests:** To save cookies and use them in the next request:

```
curl -c cookies.txt https://www.example.com/login
curl -b cookies.txt https://www.example.com/dashboard
```

9. **Send Data via JSON:** To send data as JSON in a request:

```
curl -X POST -H "Content-Type: application/json" -d '{"key":"value"}' https://api.example.com/data
```

10. **Output Response Headers:** To display only the response headers:

```
curl -I https://www.example.com
```

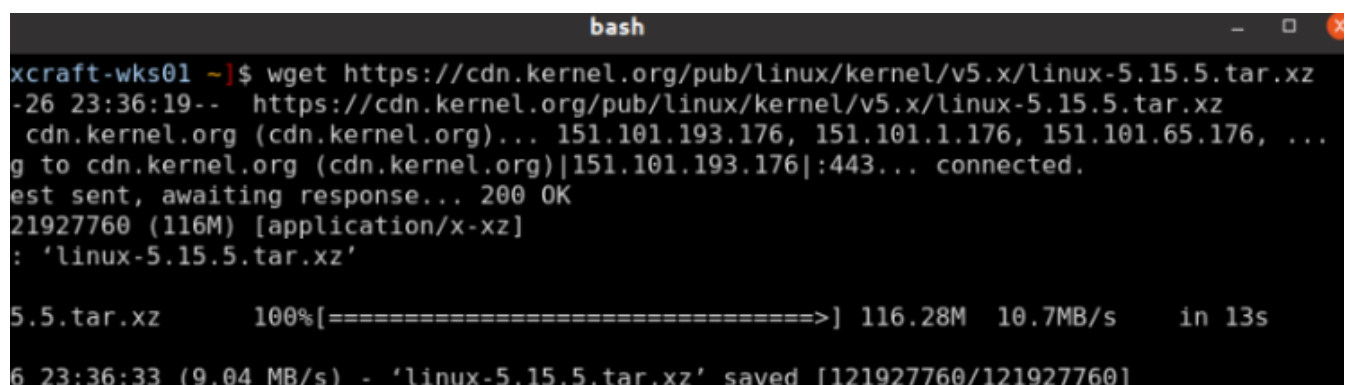
11. **Use a Specific User-Agent:** To set a custom User-Agent header:

```
curl -A "My User Agent" https://www.example.com
```

These are just a few examples of what you can do with `curl`. It's a versatile tool for interacting with web services, APIs, and various network protocols directly from the command line. You can use `man curl` to access the manual and learn more about its capabilities and options.

## What is Wget command ?

wget is another command-line tool used for downloading files from the web. It's similar to curl but specialized for downloading content. While curl is more versatile and can interact with various network protocols, wget is designed specifically for straightforward downloading tasks. Here are some common use cases and examples of how to use wget:



```
bash
xcraft-wks01 ~]$ wget https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.15.5.tar.xz
--2023-08-14 23:36:19-- https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.15.5.tar.xz
Connecting to cdn.kernel.org (cdn.kernel.org)... 151.101.193.176, 151.101.1.176, 151.101.65.176, ...
Connected to cdn.kernel.org (cdn.kernel.org)|151.101.193.176|:443... connected.
HTTP request sent, awaiting response... 200 OK
121927760 (116M) [application/x-xz]
Saving to: 'linux-5.15.5.tar.xz'

linux-5.15.5.tar.xz 100%[=====>] 116.28M 10.7MB/s in 13s

2023-08-14 23:36:33 (9.04 MB/s) - 'linux-5.15.5.tar.xz' saved [121927760/121927760]
```



Here are some common use cases and examples of how to use `wget` :

1. **Download a File:** To download a file from a URL:

```
wget https://www.example.com/file.ext
```

2. **Save to a Specific Filename:** To specify the output filename when downloading:

```
wget -O output.ext https://www.example.com/file.ext
```

3. **Continue Downloading Partially Transferred Files:** To resume a previously interrupted download:

```
wget -c https://www.example.com/largefile.ext
```

4. **Download Multiple Files:** To download multiple files using a text file with URLs:

```
wget -i urls.txt
```

5. **Mirror a Website:** To mirror a website's content for offline browsing:

```
wget --mirror https://www.example.com
```

6. **Limit Download Speed:** To limit download speed (e.g., 1MB/s):

```
wget --limit-rate=1m https://www.example.com/largefile.ext
```

7. **User-Agent and Referer Headers:** To set custom User-Agent and Referer headers:

```
wget --user-agent="My User Agent" --referer=https://www.referrer.com https://www.example.com
```

8. **Download in the Background:** To download in the background (useful for large downloads):

```
wget -b https://www.example.com/largefile.ext
```

9. **Download Only If Newer:** To download a file only if it's newer on the server:

```
wget -N https://www.example.com/file.ext
```

10. **Use a Proxy:** To download through a proxy server:

```
wget --proxy=on -e use_proxy=yes -e http_proxy=http://proxy.example.com:8080 https://www.example.com
```

These examples showcase some of the common use cases of `wget` . It's a handy tool for downloading files and even mirroring websites. For more details and options, you can refer to the `man wget` manual.

# What are sudo and su commands?

Both sudo and su are command-line tools used to execute commands as another user, typically the superuser (root), on Unix-like operating systems. However, they differ in their approach and functionality:

```
[aaronkilik@tecmint ~]$ su tecmint
Password:
[tecmint@tecmint aaronkilik]$
[tecmint@tecmint aaronkilik]$ ls
ls: cannot open directory .: Permission denied
[tecmint@tecmint aaronkilik]$
[tecmint@tecmint aaronkilik]$ cd
[tecmint@tecmint ~]$
[tecmint@tecmint ~]$ ls
bin lost
```

Difference Between "su" and "su -" in Linux

## 1. sudo (Superuser Do):

- `sudo` is used to execute a command as another user, usually the superuser (root), with elevated privileges.
- It's designed to provide finer control over who can run privileged commands and which commands they can run.
- Users need to be explicitly granted permission in the `sudoers` file (usually located at `/etc/sudoers`).
- Usage:

`sudo` command

- Example: Run `apt update` with root privileges:

`sudo apt update`

## 2. su (Switch User):

- `su` is used to switch to another user account, often the root user, by default.
- It requires the user to enter the target user's password if switching to a user other than the root user.
- Once you switch to another user, all subsequent commands are executed in the context of that user until you exit.
- Usage:

`su [username]`

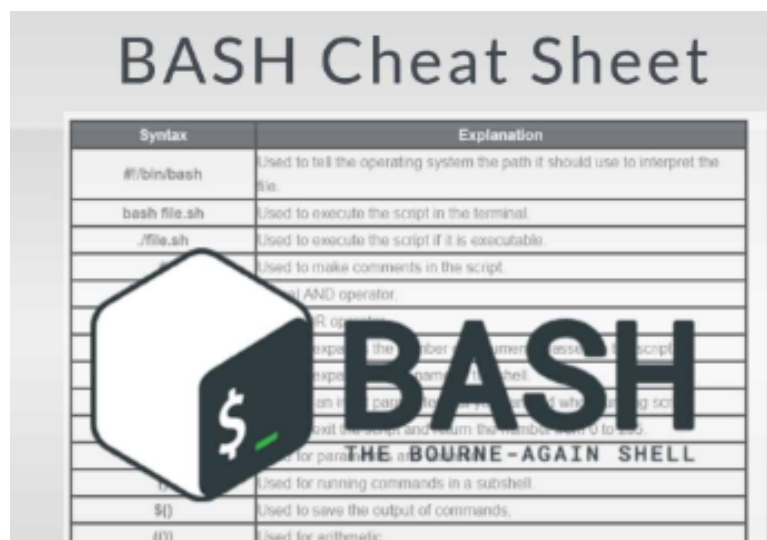
- Example: Switch to the root user:

**su**

### Key Differences:

- `sudo` is designed to provide controlled and temporary privilege escalation for specific commands, while `su` changes the entire user context.
- `sudo` allows administrators to delegate specific tasks without sharing the root password, enhancing security.
- `sudo` provides a way to log command executions for auditing purposes.
- `su` requires the target user's password (except for switching to root), while `sudo` uses the user's own password.

In summary, use `sudo` when you need to run specific commands with elevated privileges and maintain better security practices. Use `su` when you need to temporarily switch to another user's environment, often for more extended administrative tasks.



- **Echo:** Print text or variables to stdout. Example: `echo "Hello World"`
- **Read:** Read input from the user into a variable. Example: `read varname`
- **Variable Assignment:** `varname=value`
- **Arithmetic:** `expr` , `let` , `${() }` for arithmetic operations. Example: `expr 1 + 2`
- **Conditional Statements:** `if` , `elif` , `else` , `fi` . Execute code blocks based on conditions.
- **Loops:** `for` , `while` , `until` . Execute repetitive tasks.
- **Functions:** Reusable blocks of code for modularity. Define with `funcname(){} .`
- **Sed:** Find and replace text. Example: `sed 's/foo/bar' file.txt`

- **Awk**: Pattern scanning and processing. Example: `awk '{print $2}' file.txt`
- **Grep**: Search for matching expressions. Example: `grep 'error' log.txt`
- **Cut, Paste, Head, Tail, Wc**: Manipulate textual data.
- **SSH, SCP**: Securely connect to remote servers.
- **Find**: Find files based on criteria. Example: `find . -name "*.txt"`
- **Crontab**: Scheduler for cron jobs.
- **Kill, Pkill**: Send signals to processes.
- **Basename, Dirname**: Get path components.
- **Du, Df**: Check disk usage.
- **Sort, Uniq**: Sort and filter text.
- **Tee**: Redirects output to both file and stdout.
- **Xargs**: Build and execute command lines from stdin.
- **Git**: Version control system. `git clone` , `git commit` , `git push` etc.
- **Curl, Wget**: Transfer data from the web. Example: `curl -O file.txt`
- **Tar**: Archive files. Example: `tar -czvf file.tar.gz /path/to/folder`
- **Gzip, Gunzip**: Compress and decompress files.
- **Chmod**: Change file permissions. Example: `chmod 755 script.sh`
  
- **Executing External Commands**: Running Linux commands and utilities from within scripts using `command` or `$(command)` .
- **Globbering**: Using wildcards like `*` and `?` to match patterns in file/folder names.
- **Pipes**: Passing the output of one command as input to another using the `|` character.
- **Redirection**: Controlling input and output using `<` , `>` , `>>` . Example: `ls -l > files.txt`
- **Brace Expansion**: Generating arbitrary strings. Example: `mkdir logs{1,2,3}`
- **Command Substitution**: Using output as arguments. Example: `grep $(uname) file`
- **Exit Codes**: Getting the exit status of commands using `$?` .
- **Debugging**: Utilizing Bash debug flags like `-x` and `set -x` to debug scripts.

- **Signals:** Handling signals like SIGTERM, SIGKILL, SIGINT, etc., with `kill`.
- **Comments:** Lines starting with `#` are not executed.
- **Control Operators:** Using `&&`, `||` for AND and OR conditional execution.
- **Parameter Expansion:** Modifying variables using `${var}`, `${var:-default}`, etc.
- **Arrays:** Defining and accessing array variables using syntax like `myarray=(1 2 3)`.
- **Dictionary / Associative Arrays:** Declaring and using key-value arrays.
- **Case/Switch Statements:** Implementing multiway conditional branching.
- **Parsing Command Line Arguments:** Processing arguments using `$1`, `$2`, etc.
- **Subshells:** Executing in a nested sub-environment using parentheses.
- **Scheduling with Cron:** Running scripts on a schedule.
- **Logging:** Capturing log output using the `logger` command.
- **Functions:** Modularizing code into reusable functions.
- **Trapping Signals:** Gracefully handling script termination.

## Explain 'if else' in scripting detail?

The `if` and `else` statements in shell scripting are used for conditional execution. They allow you to run different sets of commands based on whether a condition evaluates to true or false.

```
1 #Initializing the variable
2 a=20
3 if [$a < 10]
4 then
5 #If variable less than 10
6 echo "a is less than 10"
7 elif [$a < 25]
8 then
9 # If variable less than 25
10 echo "a is less than 25"
11 else
12 # If variable is greater than 25
13 echo "a is greater than 25"
14 fi
```

Result

```
$bash -f main.sh
a is greater than 25
```

Here's a detailed explanation of how to use `if` and `else` statements in shell scripts:

### The `if` Statement:

The basic syntax of the `if` statement is as follows:

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

- The `[ condition ]` is an expression that evaluates to true or false. It can involve comparisons, file checks, or other conditions. You can use operators like `-eq` (equal), `-ne` (not equal), `-lt` (less than), `-gt` (greater than), etc.
- The `then` keyword indicates the start of the block of commands to execute if the condition is true.
- The commands within the `if` block are executed only if the condition evaluates to true.

### Example of the `if` Statement:

```
#!/bin/bash

count=10

if [$count -eq 10]; then
 echo "The count is 10."
fi
```

### The `if-else` Statement:

The `if-else` statement allows you to execute different sets of commands based on whether a condition is true or false. The syntax is as follows:

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

- If the condition in the `if` statement is true, the commands within the `if` block are executed.
- If the condition is false, the `else` block is executed.

### Example of the `if-else` Statement:

```
#!/bin/bash

count=5

if [$count -eq 10]; then
 echo "The count is 10."
else
 echo "The count is not 10."
fi
```

## Nested if Statements:

You can also have nested `if` statements to handle more complex conditions. Here's an example:

```
#!/bin/bash

score=85

if [$score -ge 90]; then
 echo "Grade: A"
elif [$score -ge 80]; then
 echo "Grade: B"
elif [$score -ge 70]; then
 echo "Grade: C"
else
 echo "Grade: D"
fi
```

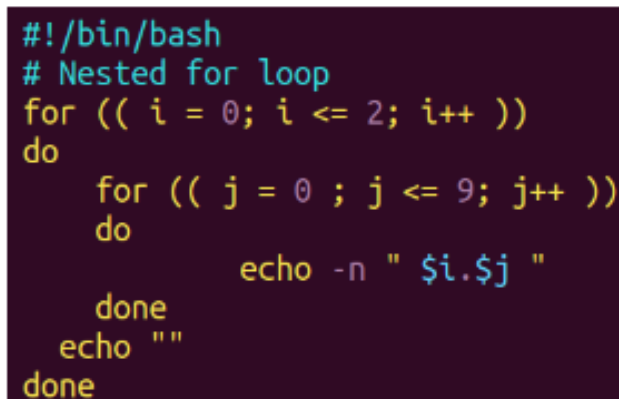
In this example, the script assigns a grade based on the value of the `score` variable.

## Summary:

In shell scripting, the `if` and `else` statements allow you to control the flow of execution based on conditions. They are fundamental tools for implementing decision-making logic in your scripts.

## Explain 'for loop' in scripting detail ?

A `for` loop in shell scripting is used to iterate over a sequence of values and execute a set of commands for each value. It's a fundamental control structure that enables you to perform repetitive tasks efficiently. Here's a detailed explanation of how to use `for` loops in shell scripts:



```
#!/bin/bash
Nested for loop
for ((i = 0; i <= 2; i++))
do
 for ((j = 0 ; j <= 9; j++))
 do
 echo -n " $i.$j "
 done
 echo ""
done
```

## Basic Syntax:

The basic syntax of a `for` loop is as follows:

```
for variable in value1 value2 ... valueN; do
 # Commands to be executed for each value
done
```

- `variable` : This is a user-defined variable that will hold the value of each iteration.
- `value1 value2 ... valueN` : These are the values over which the loop iterates. You can provide a list of values separated by spaces.
- The `do` keyword marks the beginning of the loop body.
- The commands within the loop body are executed for each value of the loop variable.
- The `done` keyword marks the end of the loop.

### Example of a `for` Loop:

```
#!/bin/bash

for fruit in apple banana orange; do
 echo "I like $fruit."
done
```

### Using a Range of Values:

You can also use a range of values for the loop, often using the `seq` command to generate sequences.

```
#!/bin/bash

for number in $(seq 1 5); do
 echo "Number: $number"
done
```

### Looping Through Files:

A common use case is to iterate through files in a directory.

```
#!/bin/bash

for file in /path/to/files/*; do
 echo "Processing $file"
 # Add your processing commands here
done
```

### Nesting `for` Loops:

You can also nest `for` loops to create multidimensional iterations.



```
#!/bin/bash

for i in {1..3}; do
 for j in {a..c}; do
 echo "i: $i, j: $j"
 done
done
```

### Using the in Keyword:

In addition to using a sequence of values, you can use a command that generates a list of items.

```
#!/bin/bash

for item in $(ls /path/to/directory); do
 echo "Item: $item"
done
```

The `for` loop is a powerful construct that allows you to perform repetitive tasks efficiently in

### Example :

#### shell script: How to analyze the health of a Node ?

Analyzing the health of a node (like a server or system) involves checking various indicators to ensure it's functioning properly. Here's a simple shell script example that demonstrates how you might perform basic health checks on a node:

```
#!/bin/bash

Check CPU usage
cpu_usage=$(top -bn 1 | grep "Cpu(s)" | awk '{print $2}')
echo "CPU Usage: $cpu_usage"

Check available memory
free_memory=$(free -m | awk '/Mem:/ {print $4}')
echo "Free Memory: $free_memory MB"

Check disk space usage
disk_usage=$(df -h / | awk '/\// {print $5}')
echo "Disk Usage: $disk_usage"

Check if a critical service is running
if pgrep "nginx" > /dev/null; then
 echo "Nginx is running"
else
 echo "Nginx is not running"
fi

Check network connectivity
ping -c 3 google.com > /dev/null
if [$? -eq 0]; then
 echo "Network is reachable"
else
 echo "Network is not reachable"
fi

Check if server load is high
load_average=$(uptime | awk '{print $10}')
echo "Load Average: $load_average"

Add more checks as needed

echo "Health analysis completed"
```

### Explanation:

break down the steps in detail for the shell script that analyzes the health of a node:

```
#!/bin/bash

Check CPU usage
cpu_usage=$(top -bn 1 | grep "Cpu(s)" | awk '{print $2}')
echo "CPU Usage: $cpu_usage"
```

- `#!/bin/bash` : This line is the shebang that indicates the interpreter to use (in this case, `/bin/bash` ).
- `cpu_usage=$(top -bn 1 | grep "Cpu(s)" | awk '{print $2}')` : This line uses the `top` command to display system processes, and `grep` and `awk` to extract the CPU usage percentage. The output is stored in the `cpu_usage` variable.
- `echo "CPU Usage: $cpu_usage"` : This line displays the CPU usage percentage.

*# Check available memory*

```
free_memory=$(free -m | awk '/Mem:/ {print $4}')
echo "Free Memory: $free_memory MB"
```

- `free -m` : This command shows memory usage in megabytes.
- `awk '/Mem:/ {print $4}'` : This uses `awk` to find the line that contains "Mem:" and extracts the fourth field (free memory) from that line.
- `echo "Free Memory: $free_memory MB"` : This line displays the amount of free memory in megabytes.

*# Check disk space usage*

```
disk_usage=$(df -h / | awk '/\// {print $5}')
echo "Disk Usage: $disk_usage"
```

- `df -h /` : This command shows disk space usage for the root directory ( `/` ) in human-readable format ( `-h` ).
- `awk '/\// {print $5}'` : This uses `awk` to find the line that contains "/", the root directory, and extracts the fifth field (disk usage percentage) from that line.
- `echo "Disk Usage: $disk_usage"` : This line displays the disk usage percentage.

*# Check if a critical service is running*

```
if pgrep "nginx" > /dev/null; then
 echo "Nginx is running"
else
 echo "Nginx is not running"
fi
```

- `pgrep "nginx" > /dev/null` : This command searches for the process named "nginx" and redirects the output to `/dev/null` to suppress it.
- `if ... then ... else ... fi` : This constructs a conditional statement. If the `pgrep` command succeeds (nginx process found), it prints "Nginx is running." Otherwise, it prints "Nginx is not running."

```
Check network connectivity
ping -c 3 google.com > /dev/null
if [$? -eq 0]; then
 echo "Network is reachable"
else
 echo "Network is not reachable"
fi
```

- `ping -c 3 google.com` : This command pings Google's server three times.
- `[ $? -eq 0 ]` : This checks the exit status of the previous command. If the exit status is 0 (indicating success), it means the network is reachable.

```
Check if server load is high
load_average=$(uptime | awk '{print $10}')
echo "Load Average: $load_average"
```

- `uptime` : This command shows the system's load average.
- `awk '{print $10}'` : This uses `awk` to extract the 10th field (load average) from the output.
- `echo "Load Average: $load_average"` : This line displays the load average.

```
Add more checks as needed
```

```
echo "Health analysis completed"
```

- `echo "Health analysis completed"` : This line indicates that the health analysis is complete.

In summary, the script checks various aspects of the node's health, including CPU usage, memory, disk space, critical service status, network connectivity, and server load. It uses a combination of commands (`top`, `free`, `df`, `ps`, `ping`, `uptime`) and text processing

- Prudhvi Vardhan