Difference between `.`, `..`, `~`

- `.` : Refers to the **current directory**.
  Example: `ls .` lists files in the current directory.

- `..` : Refers to the **parent directory** (one level up).
  Example: `cd ..` moves you up one directory.

- `~` : Refers to the **home directory** of the current user.
  Example: `cd ~` takes you to `/home/username`.

Relative vs Absolute Paths

- **Absolute path**: Starts from the **root directory `/`** and gives the full path.
  Example: `/home/user/projects/file.txt`

- **Relative path**: Path relative to the **current directory**.
  Example: `../file.txt` (go up one level and access file.txt)

Tip: `pwd` shows the absolute path of your current location.

Difference between `chmod 777` and `chmod +x`

- `chmod 777`: Gives **read (r), write (w), and execute (x)** permissions to **everyone (owner, group, others)**.
  - Example: `-rwxrwxrwx` (very open and not recommended for security).

- `chmod +x`: Only **adds execute permission** (keeps existing permissions intact).
  - Example: If file has `rw- r-- r--`, after `chmod +x`, it becomes `rwx r-- r--`.

How to combine commands using pipes `|`

The pipe `|` **passes the output of one command as input to another command**.

Example:

```
ps aux | grep apache
```

`ps aux`: Lists all processes
`grep apache`: Filters only lines containing "apache"

---

5. `kill` vs `kill -9`

- `kill <PID>`: Sends a **SIGTERM (signal 15)** → Politely asks the process to terminate (can be ignored by the process).
- `kill -9 <PID>`: Sends a **SIGKILL (signal 9)** → Forcefully kills the process immediately (cannot be ignored).
  Use `kill -9` only if normal `kill` doesn't work.

Difference between `>` and `>>`

- `>` : **Overwrite redirect**. Writes output to a file, replacing existing content.
  Example: `echo hello > file.txt` (erases previous content).
- `>>` : **Append redirect**. Adds output at the end of the file without erasing content.
  Example: `echo world >> file.txt`

Difference between `exit 0` and `exit 1`

`exit 0`: Indicates **success** (no errors).

`exit 1`: Indicates **failure or error**.

Example in scripts:

```
if [ "$file" = "" ]; then

    echo "File not found"
    exit 1   # failure
fi

    exit 0       # success
```

Difference between `su` and `sudo`

- **`su` (substitute user)**: Switches to another user account (default is root).
    - When you run `su`, you are asked for the **target user's password**.
    - Example: `su` → switches to root and opens a new shell.
- **`sudo` (superuser do)**: Runs a single command with elevated privileges.
    - It asks for the **current user's password** (if that user is in the `sudoers` list).
    - Example: `sudo apt update`
    - After command executes, you remain as your original user.

Key difference: `su` switches user accounts completely, `sudo` temporarily grants admin rights for one command.

How to use `find` command

The `find` command searches for files and directories.

Basic syntax:

```
find <path> <options> <criteria>
```

Examples:

Find all `.txt` files in current directory and subdirectories:

```
find . -name "*.txt"
```

Find files modified in the last 2 days:

```
find . -mtime -2
```

Find files and run a command on them:

```
find /var/log -name "*.log" -exec rm {} \;
```

(`{}` is replaced by each found file)

How to pass arguments in a shell script

When you run a script, arguments can be accessed using `$1`, `$2`, etc.

Example:

```
#!/bin/
echo "First argument: $1"
echo "Second argument: $2"
```

Run:`./script.sh hello world`

Output:

```
First argument: hello
Second argument: world
```

Special variables:

- `$@`: all arguments
- `$#`: number of arguments
- `$0`: script name

How to set exit codes and check them

Set exit code:

Use `exit <code>` in your script or command.
`exit 0`: success
`exit 1`: general error
`exit 2`: incorrect usage (custom meanings allowed)

Check exit code of the last command:

Use `$?`

```
ls /tmp
echo $?    # Prints 0 if successful, non-zero if failed
```

Examples of different exit codes:

- `0`: success (no error)
- `1`: general errors (permissions, syntax errors, etc.)
- `2`: misuse of shell builtins
- Codes **>128**: indicate the process was terminated by a signal (e.g., 137 = killed by `kill -9`)

Example in script:

```
if [ "$1" = "" ]; then
    echo "No argument provided"
    exit 1
fi
echo "All good"
exit 0
```

Difference between Hard Link and Soft (Symbolic) Link & How to Create Them

- **Hard Link**
    - Points directly to the **data (inode)** of a file.
    - If the original file is deleted, the data remains accessible through the hard link.
    - Cannot link directories, only files.
    - Cannot link across different filesystems.

Command:

```
ln original.txt hardlink.txt
```

    -
- **Soft (Symbolic) Link**
    - Acts as a **shortcut** (stores path to the target file).
    - If the original file is deleted, the soft link becomes broken.
    - Can link directories and files.
    - Can cross filesystems.

Command:

```
ln -s original.txt softlink.txt
```

    -

Summary: Hard links = multiple names for the same data. Soft links = pointers to the file path.

Foreground vs Background Processes (`&`, `jobs`, `fg`, `bg`)

Foreground process:
Runs directly in the terminal; you cannot use the terminal until it finishes.

Example:
```
sleep 60
```

Background process:
Runs in the background; you can still use the terminal.

Example:
```
sleep 60 &
```

- **`jobs`**: Lists background jobs.
- **`fg %job_id`**: Brings a background job to the foreground.
- **`bg %job_id`**: Resumes a paused background job.

Difference between `scp` and `rsync` for File Transfer

- **`scp` (secure copy):**

Copies files over SSH.
Always copies full files, even if only part changed.

Example:
```
scp file.txt user@remote:/path/
```

- **`rsync`:**

  Efficiently syncs files between systems.

  Copies only the differences (delta transfer).

  Supports resuming interrupted transfers.

Example:

```
rsync -avz file.txt user@remote:/path/
```

Summary: `scp` = simple copy. `rsync` = efficient sync with incremental updates.

Difference between Single Quotes `' '` and Double Quotes `" "`

Single quotes `' '`: Preserves everything literally. Variables and special characters are **not expanded**.

```
echo '$HOME'   # prints: $HOME
```

Double quotes `" "`: Variables and special characters **are expanded**.

```
echo "$HOME"   # prints: /home/username
```

---

5. How to Use `case` Statement Instead of Multiple `if`

Example:

```
#!/bin/bash
read -p "Enter a number: " num

case $num in
  1)
    echo "One"
```

```
    ;;
  2)
    echo "Two"
    ;;
  *)
    echo "Other number"
    ;;
esac
```

- Each block ends with `;;`
- `*` acts as a default case.

Difference Between Sourcing a Script (`. script.sh`) and Executing (`./script.sh`)

- **Executing (`./script.sh`)**:
    - Runs the script in a **new shell**.
    - Any variables or changes (e.g., `cd`, `export`) do **not affect the current shell**.
- **Sourcing (`. script.sh` or `source script.sh`)**:
    - Runs the script in the **current shell**.
    - Any variables, functions, or directory changes persist after the script finishes.

Example:

```
# test.sh
export VAR="hello"


./test.sh   # VAR will NOT be available after script
. test.sh   # VAR will be available in current shell
```

How to Debug a Script: bash -x script.sh

- When you run a script with bash -x, it enables **debug mode**.
- This prints **each command** and its **expanded arguments** as they are executed.
- Useful for finding errors in scripts (like wrong variable values or flow issues).

Example:

```
#!/bin/bash
name="John"
echo "Hello $name"
echo "This will fail" $non_existing_var
```

Run with debug:

```
bash -x script.sh
```

Output:

```
+ name=John
+ echo 'Hello John'
Hello John
+ echo 'This will fail'
```

This will fail

+ indicates each line being executed.

You can also enable debug **inside the script**:

```
set -x    # start debugging
...your code...
set +x    # stop debugging
```

---

2. What is Difference Between PID and PPID?

PID (Process ID)

A unique identifier assigned by the operating system to every running process.
Example:

```
echo $$
```

This prints the PID of the current shell.

PPID (Parent Process ID)

The PID of the process that started (spawned) the current process.
Example:

```
echo $PPID
```

This prints the PID of the parent process (the process that launched this one).

Example with ps:

```
ps -f
```

Output (simplified):

```
UID   PID  PPID CMD
root  100   1    /usr/bin/sshd
user  120   100  bash
user  130   120  vim
```

Here:

bash has PID 120, and its parent is sshd with PPID 100.
vim has PID 130, and its parent is bash (PPID 120).

Summary:

- PID = current process ID
- PPID = parent process ID