

LTSC_NOTES:

Table of Contents

LTSC_NOTES:.....	1
all_linux_commands.pdf.....	11
alias command in linux.....	11
expr, bc, awk commands in linux.....	12
{{}} or shell command.....	14
command to download vim text editor.....	15
command to exit vim.....	16
set command.....	16
tar command to backup files.....	18
head commqnd in linux.....	20
tr command.....	21
basename , expr command.....	22
basename command to change filename extension.....	24
How to run a script?.....	25
command to make a new script file?.....	26
Why permission is denied while ruuning a script?.....	28
chgrp command to change files group ownership.....	30
touch command to change files last modification or access time.....	32
chown command to change the files ownership.....	33
What is the meaning of recursively?.....	35
passwd command to change password.....	36
type command to know command type.....	37
bzip2 command to compress file.....	39
.....	40
gzip command to compress file.....	40
zip - Compress multiple files to a single file.....	41
cat - Concatenate files.....	43
cp -r - Copy directory tree.....	45
what is verbose output?.....	46
ftp, scp, sftp - Copy file between machines.....	47
grep -c - Count number of lines containing a pattern.....	49
wc - Count number of lines, words, and characters.....	51
ar - Create an archive of C object files.....	53
ln - Create links to a file.....	55
cut - Cut columns or fields from a file.....	57

cut -c.....	59
sleep - Delay command execution in a script.....	60
diff - Difference between two files (as sed-like instructions).....	62
``ls -l grep "d"`` - Directory list.....	64
du - Disk space utilization.....	65
top - Free space in memory and swap.....	67
meaning of swap.....	69
paste - Join two files laterally.....	69
kill - Kill job.....	71
Command to show pid of a process.....	72
`grep -E, egrep, fgrep` - Lines containing one or more of multiple patterns.....	73
sort - Lines in ASCII collating sequence.....	75
sort -n - Lines in numeric sequence.....	77
tac, rev- Lines in reverse order.....	78
sort -f - Lines sorted ignoring case.....	79
uniq -d - Lines that are repeated.....	81
uniq -u - Lines that occur only once.....	82
ssh - Log in to a remote machine.....	83
make - Maintain a group of C programs.??.....	84
tail -f - Monitor growth of a file.....	86
mv - Move files to another directory.....	87
bg - Move job to the background.....	88
fg - Move job to the foreground.....	90
sed - Multiple segments from a file.....	92
hostname - Name of the local host.....	94
uname - Operating system name.....	95
GNU.....	96
uname -r - Operating system release.....	96
export - Pass variable value to sub-shell.....	97
tr -s - Squeeze multiple spaces to a single space.....	99
expr - String length.....	100
wc -c.....	102
tr - Substitute one character for another.....	103
sed - Substitute one pattern for another.....	105
su - Superuser from a nonprivileged account.....	107
date - System date.....	108
top - System memory usage.....	110
who -r, runlevel - System run level.....	111
How to know process name under any PID?.....	113

exit, logout - Terminate shell script.....	115
ping - Test connectivity of host.....	116
bunzip2 - Uncompress .bz2 file.....	118
gunzip - Uncompress .gz file.....	119
compress multiple files and folders in tar.gz.....	120
umount - Unmount file system.....	122
who - Users and their activities.....	123
Basics of linux.....	125
Why learning Linux is helpful.....	127
Operating system = Kernel + system applications.....	127
Linux file system.....	128
Getting started.....	128
Piping.....	129
• cat text.txt wc -w.....	129
• sort file.txt uniq.....	129
• ps aux grep "chrome" awk '{print \$2}'.....	130
• echo "Hello, World!" tr '[lower:]' '[upper:]'	131
• Login to remote system: ssh.....	132
• Copying files: scp.....	132
Command Description.....	132
5. ls -ltr =>sorting the listing by time modification.....	132
7. cp file1 file2 -copy files(!!??).....	133
9. mv file1 file2 rename file1 to file2.....	135
10. rm -i file.....	135
21. grep -i [pattern] file.....	136
22. find -iname "file".....	136
23. ./file =>executes a file in current directory.....	136
How to change the permission of a file in linux.....	137
24. ../../file =>executes a file two levels above the current directory.....	138
Absolute and relative path.....	139
Echo.....	139
echo [options] [string(s)].....	139
echo "New content" >> file.txt.....	141
• text="Hello World"; echo "Length of \$text is \${#text}".....	142
• result=\$(echo "5 + 7" bc) ; echo "\$result".....	142
• result=\$(echo "sqrt(25)" bc -l) ; echo "V25 = \$result"	143
File permissions.....	143
What is hard link of a file?.....	144
How to create hardlink to a file?.....	145

Changing file permissions.....	146
File ownership.....	147
grep.....	148
▪ Counting matches: grep -c "pattern" file.txt.....	148
▪ Display line numbers: grep -n "pattern" file.txt.....	148
▪ Invert match: grep -v "pattern" file.txt.....	148
▪ display the names of files with a match: grep -l "pattern" file.....	149
▪ Recursive search: grep -r "pattern" directory/.....	149
▪ Regular expressions: grep "^[0-9]*\$" numbers.txt.....	149
▪ Matching multiple patterns: grep "pattern1\ pattern2" file.txt.....	150
▪ Show context around match: grep -C 2 "pattern" file.txt.....	150
▪ Search file types: grep "pattern" --include "* .txt" dir/.....	151
Arithmetic Computation.....	151
• expr:.....	151
• bc:.....	151
• (()): Shell arithmetic expansion for integer computations.....	151
Difference between ((x = 5 + 3)) and x=\$((5+3))	153
File Permissions and Ownership.....	154
• chown: Change file's ownership.....	154
Text Processing.....	154
• head: Beginning of file.....	154
• tr: Translate or delete characters.....	154
• cut: Cut columns or fields from a file.....	155
• sort: Sorts Lines in ASCII collating sequence.....	159
• tail -r: Lines in reverse order.....	159
• uniq -d, uniq -u: filter out or identify repeated lines.....	160
• tr -s: Squeeze multiple spaces to a single space.....	160
File Display and Manipulation.....	161
Append File Content: cat file1.txt >> file2.txt.....	162
• Directory list.....	162
Ls -l /directory.....	162
Ls -l grep "^d".....	163
• Ls -p . grep /.....	163
• du: Disk space utilization.....	164
Sort and display largest directories: du -h directory/ sort -rh head -n 10.....	164
• tail -f: Monitor growth of a file.....	165
watch : runs a command repeatedly.....	165
System and User Information.....	166
• who -r, runlevel: System run level.....	166

• who: sayak tty2 2023-08-31 18:01 (tty2).....	167
User Account and Security.....	167
• exit, logout: Terminate shell script.....	168
SSH and Remote Access.....	168
• ssh -p port_number username@remote_host.....	168
• scp, sftp: Copy file between machines.....	169
• Copy local to remote machine:(Not working!!??).....	169
System and Process Control.....	170
• uptime: Show system uptime.....	170
• htop: Interactive process viewer.....	170
• kill: Terminate process by PID:.....	171
• nohup: Run a command in the background.....	172
Users in Linux:.....	177
Absolute and Relative path:.....	177
• reboot: Reboot the system.....	198
Environment and Variables.....	198
• set: Assign values to positional parameters.....	198
• export: Pass variable value to sub-shell.....	200
Input and Output.....	202
• read: Input data to a shell script interactively.....	202
• paste: Join two files laterally.....	203
System Date and Time.....	204
• Current date: date +"%Y-%m-%d".....	204
• Future date: date -d "+3 days" +"%Y-%m-%d".....	204
L2-vim.pptx.pdf.....	206
vi -O file1 file2:.....	206
vi -o file1 file2:.....	207
Getting started.....	207
vi/vim - open vim.....	208
vi/vim - Saving and Quitting.....	208
:w bar - Save the file as "bar" (similar to "Save As" in Windows).....	209
:w! bar - Overwrite the existing file "bar" without confirmation.....	210
:n1,n2w foo - Write lines from n1 to n2 to the file named "foo".....	212
:n1,n2w>> foo - Append lines from n1 to n2 to the file named "foo".....	213
vi/vim - Copying.....	214
yy: Copy the current line.....	214
y: Copy the selected region (Vim only).....	215
vi/vim - Deleting.....	216
vi/vim - Insert text.....	216

[Ctrl-v][Ctrl-b]: Insert the text "[Ctrl-b]" (literal control characters).....	217
vi/vim - Interface to Linux.....	217
• :!cmd.....	217
• :!%.....	218
• :r !cmd -	218
:sh or [Ctrl-z] - Escape to the UNIX shell temporarily (suspend editor).....	218
vi/vim - Undo and Redo.....	219
vi/vim - Substitution.....	219
[range]: Specifies the range of lines to perform the substitution. If omitted, the substitution will be applied to the current line only.....	220
vi/vim - Search and Repeat.....	222
/pat: Non-incremental search forward for the string "pat.".....	222
:set ic: Ignore case when searching.....	223
Regular expressions.....	225
Regular expressions in text processing.....	225
• *: Zero or more occurrences of the previous character.....	225
How to execute these regular expressions?.....	226
• .: A single character.....	227
• .*: Nothing or any number of characters.....	227
go.*d: Matches "god", "good", "goooooooood", "goXd", "goYd", and so on.....	227
[^Z]: Any character except "Z.".....	228
^DM: "DM" at the beginning of a line.....	229
• sun\$: "sun" at the end of a line.....	229
g\+: One or more occurrences of the previous character.....	229
g\?: Zero or one occurrence of the previous character.....	229
GIF\ JPEG: As above - GIF or JPEG.....	229
wood\ (cock\ house\): As above - woodcock or woodhouse.....	229
wood\ (cock\ house\): As above - woodcock or woodhouse.....	230
\<pat: Pattern "pat" at the beginning of a word.....	231
pat\>: Pattern "pat" at the end of a word.....	231
\{m\}: m occurrences of the previous character.....	231
• [0-9]\{5\}:.....	231
\{m,\}: At least m occurrences of the previous character.....	231
\{m,n\}: Between m and n occurrences of the previous character.....	231
\(\exp\): exp and attaches tag \1, \2, etc. to exp.....	232
Each part (day, month, year) is captured with a separate tag: \1, \2, \3.....	232
\w: A word character (same as [a-zA-Z0-9_]).....	234
\W: A non-word character (same as [^a-zA-Z0-9_]).....	234
\w\+: Matches one or more word characters (e.g., "word", "123", "abc_").....	235
# Basic difference between \W and \W\+ :.....	235

\d: A digit (same as [0-9]).....	236
\d+\.\d\{2\}: Matches a decimal number with two decimal places.....	236
\D: A non-digit (same as [^0-9]).....	237
\s: A whitespace character.....	237
\S: A non-whitespace character.....	237
• \t\w\+\t: Matches a word surrounded by tabs.....	237
[[:alpha:]] - An alphabetic character.....	238
Form feed character:.....	238
What is control character?.....	239
[[:alpha:]]\+: Matches one or more alphabetic characters.....	240
How to print lines having only numbers.....	241
Method 1: grep -E "^[0-9.-]+\s+[0-9.-]+\\$" minima.dat.....	241
Method 2: While loop.....	242
What are the differences between literal characters, escape characters and control characters?... ..	243
Literal characters.....	243
Escape characters.....	244
Control characters.....	244
Method 3: Using for loop.....	246
History.....	249
what is the difference between bash and bourne?.....	249
GNU project:.....	251
bash overview.....	251
bash features.....	251
What is wildcard character in bash scripting.....	252
First command.....	254
bash hpc.sh arg1 arg2 arg3.....	254
First line: #!/bin/bash what does it mean ?.....	254
Script arguments.....	255
Examples with \$0, \$1, \$# etc.....	255
Another Example with \$0, \$1, \$# etc.....	257
One remarkable point:.....	257
Example.....	258
Essential elements of the bash script.....	259
• Command substitution:.....	259
• Redirection: Input and output can be redirected.....	260
• Arithmetic operations and conditional expression.....	262
Arithmetic Evaluation and Expansion with integers.....	262
Arithmetic operators.....	263
Example 1: pre and post increment.....	264

Explanation:.....	264
What is the difference between += and ++ ?.....	265
Exit status.....	266
Common Exit Status Codes.....	266
Using Exit Status.....	267
Example:.....	267
> /dev/null:.....	268
status=\$?:.....	268
Example 2.....	269
File Redirection.....	269
• 2>: Redirects standard error to a file.....	269
• <<: Takes input from script.....	270
Example: here-doc.....	270
cat << EOF > file.txt:.....	271
Constructs.....	272
For Loop:.....	272
Example:.....	273
While Loop:.....	273
Example:.....	273
If Statement:.....	274
Important points:.....	274
• [[...]] Construct -.....	274
• Example:.....	274
To find consecutive characters in a string:.....	275
operators for If Statement:.....	276
Example:.....	277
case Statement:.....	278
Example 1.....	278
Example 2.....	280
Important Note:.....	281
Bash Script Basic Syntax.....	282
How many types operators are there in bash scripting?.....	283
Arithmetic Operators:.....	283
Comparison Operators:.....	283
• Using '=' is it ok, or '==' should be used for string comparison?.....	284
Logical Operators:.....	286
Assignment Operators:.....	287
Ternary Operator (Conditional Operator):.....	288
File Operators:.....	288

Bash Scripts example:.....	288
Example : loop with ‘seq’ command.....	290
Example: Loop with {1...n}.....	290
Example : Getting users’ informations.....	291
Example: [[:upper:]] , [[:lower:]].....	292
Example: logical “&&” and comparison “-eq”.....	293
Example: ‘break’ in a infinite while loop.....	295
Example 1:.....	295
Example 2:.....	296
What is the difference between example 1 and 2; both having same output?.....	296
Simple calculator in bash.....	297
Algorithm.....	297
What is the difference between the example 2 and example 3?.....	299
Example 5: Associative array.....	301
Example 6:.....	303
Arrays.....	304
declare command.....	304
-i: Declare a variable as an integer:.....	304
-a: Declare a variable as an indexed array:.....	305
-A: Declare a variable as an associative array:.....	305
-p: Display attributes and options of variables.....	306
-x: Export a variable for child processes.....	306
Examples from lecture:.....	307
Example 2.....	308
Example 3.....	308
Example 4.....	309
Example 5:.....	309
Example 6:.....	310
Functions:.....	310
Example 1.....	311
Example 2.....	312
Example 3.....	313
Example 4.....	314
Example 5:.....	314
Working with practical examples.....	315
Retrieving a value from specified file and printing: (REVISE!?).....	315
Automated daily file backup script.....	319
Hard disk space monitoring and threshold alert script.....	322
Recursive counting of files and directories.....	324

Start, stop or monitor processes on remote machines.....	326
Job queue.....	329
MODULE 2: LECTURE 1.....	333
Using 'sed' (Stream Editor) for Text Manipulation.....	333
Benefits of 'sed' for Batch Processing and Automated Text Manipulation:.....	333
Basic syntax of invoking 'sed'.....	334
Most frequently used 'sed' commands:.....	334
Search and replace.....	335
sed -i 's/apples/oranges/g' fruit.txt &.....	337
sed 's/fruits/apples/gi' fruit.txt : Difference??.....	337
-E for Extended regular expressions.....	339
Delete "d" with sed:.....	341
Print p.....	343
sed -n '/banana/p' file.....	343
What happens if I do not give -n ?.....	344
sed -n 's/apple/date/gp' file.....	345
Append/ Insert:.....	345
sed '/banana/i This is a fruit:' file.....	345
sed '/^apple/i This starts with "apple".' file.....	346
sed '/banana/a This is a fruit:' file.....	346
sed '/^apple/a This starts with "apple".' file.....	347
sed '/banana/s/^/The fruit is:/' file.....	347
sed '/banana/s/\$/ is a fruit/' file.....	347
Multiple Sed.....	348
sed 's/\b./\U&/g' file sed 's/\$/- Fruit/' sed '/banana/d'.....	348
sed -E '/(apple banana)/s/\$/ is a fruit/; /(cashew)/s/\$/ is a nut/;.....	349
AWK:.....	350
Basic AWK Syntax.....	350
Simple AWK usage.....	350
Simple AWK usage: (with pattern and action).....	352
Simple AWK usage: (with mathematical operation).....	354
Calculating the Sum of Numbers.....	354
Calculate the average of Numbers.....	355
Perform Arithmetic Operations.....	355
AWK SCRIPTS: Examples.....	355
What is the difference between scripts with extension .awk and with .sh.....	356
.....	358
.....	358
Checking if the line is not empty (if (\$0)): This part of the script ensures that empty lines in the input file are not considered when determining the maximum and minimum values. The condition if (\$0) checks if	

the entire line (\$0) contains any characters. If it's empty, meaning there are no characters on that line, the line is skipped. This is important to avoid incorrect calculations if the file contains empty lines.....	358
Comparing the number on the line (\$1) with max and min: The script reads each line of the input file and extracts the number from the line using \$1. It then compares this number to the current maximum and minimum values (max and min). If the number is greater than the current maximum value (\$1 > max), it updates max. Similarly, if the number is less than the current minimum value (\$1 < min), it updates min. This way, the script keeps track of the maximum and minimum values encountered while processing the file.....	358
3. Given a file named grades.csv with student names and their respective scores, write an AWK script to calculate and print the letter grades for each student based on the following scale: EX: 90-100; A: 80-89	359
ETSC CLASS ASSIGNMENT.....	362
1. Population Over Time -.....	363
2. Population Distribution:.....	364
3. Population by Region -.....	364
5.1 Urban vs. Rural Population.....	366
Xm Grace Assignment.....	369
Hoek-Brown strength criterion.....	369
Parameters for assignments.....	370
some commonly used symbols in xmgrace:.....	370
Total list.....	370
\{f\}: Text Formatting.....	372
\ x: Special Characters.....	374
Assignment Procedure.....	374
Sample Question paper.....	375
diff < cat 1.txt > cat 2.txt.....	375
chown :hpc test.dat.....	376
sed '/start/,/end/ s/old/new/g' file.txt.....	377
Write the command to calculate the sum of 12, 18, and 22?.....	378
Using awk, print lines from a file where the first column matches a string HPC.....	378

all_linux_commands.pdf

alias command in linux

In Linux, the **alias** command is used to create custom shortcuts for other commands. An alias is a way to define a shorter name or abbreviation for a

longer command or command with options. This can make your workflow more efficient by reducing the amount of typing you have to do.

Here's the basic syntax for creating an alias:

bash

```
alias alias_name='original_command'
```

For example, let's say you frequently use the `ls -l` command to list files in a detailed format. You could create an alias like this:

bash

```
alias ll='ls -l'
```

After defining this alias, whenever you type `ll` in the terminal and press Enter, it will be equivalent to running `ls -l`.

However, aliases are only valid for the current session. To make them permanent and available every time you open a new terminal session, you need to add them to your shell's configuration file (such as `.bashrc`, `.bash_profile`, `.zshrc`, etc., depending on your shell).

For example, you can add the above alias to your `.bashrc` file like this:

bash

```
echo "alias ll='ls -l'" >> ~/.bashrc
```

Remember to source the configuration file or restart your terminal for the changes to take effect:

bash

```
source ~/.bashrc
```

Keep in mind that while aliases are useful, they might not work in all situations. For more complex tasks, it's often better to create shell functions or scripts. Also, be careful when creating aliases that override existing command names or create confusion, as it can lead to unexpected behavior.

expr, bc, awk commands in linux

Certainly! **expr**, **bc**, and **awk** are three commonly used commands in Linux for performing various types of calculations and text processing tasks. Here's a brief overview of each:

1. **expr**: The **expr** command is used for performing basic arithmetic operations and comparisons in shell scripts and the command line. It supports addition, subtraction, multiplication, division, and other basic mathematical operations. It's primarily used for integer arithmetic.

Example usage:

```
bash
```

```
result=$(expr 5 + 3)  
echo $result # Outputs: 8
```

```
result=$(expr 10 / 2)  
echo $result # Outputs: 5
```

2. **bc**: The **bc** (basic calculator) command is a more powerful tool for performing arbitrary-precision arithmetic. It's capable of handling floating-point arithmetic and supports a wide range of mathematical functions and expressions. It's often used when more advanced calculations are needed.

Example usage:

```
bash
```

```
echo "3.14 * 2.5" | bc # Outputs: 7.85
```

```
echo "scale=2; 10 / 3" | bc # Outputs: 3.33
```

3. **awk**: The **awk** command is a versatile text processing tool that allows you to perform pattern scanning and text manipulation based on specified rules. It's particularly useful for working with structured data, such as columns in files or output from other commands. **awk** operates on records (usually lines) and fields (usually whitespace-separated portions of records).

Example usage:

bash

```
echo "Alice 25" | awk '{ print $1 }' # Outputs:  
Alice
```

```
echo "Alice 25" | awk '{ print $2 }' # Outputs:  
25
```

awk can also be used for more complex tasks, such as filtering lines based on conditions, performing calculations, and more.

Keep in mind that these are just basic examples, and each of these commands has many more features and options for different use cases. You can refer to their respective manual pages (`man expr`, `man bc`, `man awk`) for more in-depth information on their capabilities and usage.

SA

{} or shell command

It seems like you've mentioned **{}** which is commonly used in templates and notations, particularly in contexts like text templating engines, configuration management tools, and programming frameworks like `Jinja2`, `Mustache`, or `Handlebars`.

The double curly braces (**{}**) are often used to enclose placeholders that are later replaced with actual values or data. This is common in

scenarios where you need to generate dynamic content, such as generating HTML, configuration files, or any other text-based output.

For example, in a template engine, you might have something like:

bash

```
Hello, {{name}}! Your account balance is $  
{{balance}}.
```

When this template is processed, the placeholders `{{name}}` and `{{balance}}` would be replaced with actual values.

On the other hand, a "shell command" generally refers to a command that you run in a command-line interface (shell) like the Bash shell in Linux. Shell commands are used to interact with the operating system, run programs, manage files, and perform various tasks.

If you provide more context or clarify your question, I'd be happy to provide more specific information. Are you looking to understand how `{}` is used in a particular context, or are you asking about shell commands in general?

SA

command to download vim text editor

To download and install the Vim text editor on a Linux system, you typically use your distribution's package manager. The package manager handles the installation process, including any necessary dependencies. Here are some common package manager commands for different Linux distributions:

1. Debian/Ubuntu:

bash

```
sudo apt update
```

```
sudo apt install vim
```

2. Fedora:

```
bash
```

```
sudo dnf install vim
```

3. CentOS/RHEL:

```
bash
```

```
sudo yum install vim
```

After running the appropriate command for your distribution, the package manager will download and install Vim on your system. Once the installation is complete, you can run the Vim text editor by typing `vim` in the terminal.

If you prefer a different version of Vim or want to explore more installation options, you can also compile Vim from source or use other package management methods depending on your needs.

SA

command to exit vim

To exit the Vim text editor, follow these steps:

1. Press the `Esc` key to ensure you're in the "Normal" mode (if you're in another mode).
2. Type `:q` to quit Vim. If you have unsaved changes, Vim will not allow you to quit and will show an error message.
3. If you want to save your changes and quit, type `:wq` and press `Enter`. This will write (save) the changes and then quit Vim.
4. If you want to force quit without saving changes, type `:q!` and press `Enter`.

Remember that Vim has different modes (Normal, Insert, Visual, etc.), so if you're not sure which mode you're in, pressing the `Esc` key usually brings you back to Normal mode.

If you're in the Insert mode (used for typing text), press `Esc` to switch to Normal mode, and then use the `:q`, `:wq`, or `:q!` commands as described above.

If you're new to Vim, it might take some time to get used to its unique interface and keybindings. You can explore more about Vim's functionality by typing `vimtutor` in the terminal, which provides an interactive tutorial to help you get started.

SA

set command

The `set` command in a Unix-like shell is used to configure various shell options and settings. It allows you to control how the shell behaves and how it interacts with you as a user. The `set` command can be used with different options to enable or disable specific settings.

Here are some common uses of the `set` command:

1. **View Current Settings:** Running the `set` command without any options will display a list of current shell options and their statuses.

bash

- `set`
- **Enable/Disable Options:** You can use the `set` command to enable or disable specific options. Options are often represented using shorthand letters.

For example, to enable the option that makes the shell display a message when a command finishes executing, you can use:

bash

```
set -o notify
```

To disable this option:

bash

- `set +o notify`
- **Using Long Options:** In addition to the shorthand options, you can also use long option names that are more descriptive. To enable or disable a long option, you use `set -` or `set +` followed by the long option name.

For example, to enable the "verbose" option:

bash

```
set -o verbose
```

To disable it:

bash

- `set +o verbose`
- **Setting Variables:** The `set` command can also be used to set or modify environment variables and shell variables.

bash

```
4.      set VAR_NAME=variable_value
```

5. Other Options: There are many other options you can use with the `set` command to customize your shell's behavior. These options control aspects like error handling, input handling, history, filename expansion, and more.

Keep in mind that the specific options and behaviors of the `set` command might vary slightly depending on the shell you're using. The examples provided here are based on the Bash shell, which is one of the most commonly used Unix-like shells. To learn more about the available options and their meanings, you can refer to the documentation for your specific shell by using the `man` command, like `man bash` or `man zsh`.

SA

tar command to backup files

The `tar` command is commonly used to create compressed archive files or to extract files from such archives. To back up files using the `tar` command, you can create a tarball (compressed archive) that contains the files you want to back up. Here's the basic syntax for creating a tarball:

bash

```
tar -czvf backup.tar.gz files_to_backup
```

Let's break down the options used in the above command:

- `-c`: Create a new archive.
- `-z`: Compress the archive using gzip.
- `-v`: Verbose mode, which shows the progress of archiving.
- `-f`: Specifies the filename of the archive.

Replace `backup.tar.gz` with the desired name for your backup archive and `files_to_backup` with the list of files and directories you want to include in the backup. You can specify multiple files and directories separated by spaces.

For example, to create a backup of files in the `documents` and `pictures` directories, you would run:

bash

```
tar -czvf backup.tar.gz documents pictures
```

After running this command, a `backup.tar.gz` file will be created in the current directory, containing the contents of the `documents` and `pictures` directories.

If you want to later extract the contents of the backup tarball, you can use the following command:

bash

```
tar -xzvf backup.tar.gz
```

- **-x**: Extract files from the archive.
- **-z**: Decompress the archive using gzip.
- **-v**: Verbose mode.
- **-f**: Specifies the filename of the archive.

This will extract the files and directories back into the current directory.

Remember to adjust the paths and filenames according to your specific use case. Additionally, the `tar` command has many more options and variations for more advanced use cases. You can refer to the `tar` command's manual (`man tar`) for more information.

2 / 2

SA

head command in linux

The `head` command in Linux is used to display the beginning (or the top) of a text file or input stream. By default, it displays the first 10 lines of the specified file. It's often used to quickly view the contents of a file without loading the entire file into memory.

The basic syntax of the `head` command is:

bash

```
head [options] [file]
```

Here are some common usage examples of the `head` command:

1. Display the first 10 lines of a file:

bash

```
head filename
```

2. Display a specific number of lines (e.g., first 5 lines) of a file:

bash

```
head -n 5 filename
```

3. Display a file's first few lines with line numbers:

bash

```
head -n 10 -v filename
```

4. Display the first part of a file along with the headers:

bash

```
head -n 20 -v -c filename
```

Here are a few options that you can use with the **head** command:

- **-n N**: Specify the number of lines to display. For example, **-n 20** will display the first 20 lines.
- **-c N**: Display the first N bytes instead of lines.
- **-v**: Display file headers (used in conjunction with **-n** or **-c**).

Remember that the **head** command is primarily used to view the beginning of files and text streams. If you want to see the end of a file, you can use the **tail** command (**tail -n**), and if you want to view a specific range of lines, you can use the **sed** or **awk** commands.

SA

tr command

The **tr** command in Linux is used for translating or deleting characters from standard input (usually text) and writing the result to standard output. It's often used for simple text transformations and substitutions. The name "tr" stands for "translate."

The basic syntax of the **tr** command is:

bash

```
tr [options] set1 [set2]
```

Here, `set1` represents the set of characters to be replaced, and `set2` represents the set of characters to replace them with. If `set2` is not provided, `tr` will remove the characters specified in `set1`.

Here are a few examples of how the `tr` command can be used:

- **Replacing Characters:** To replace all occurrences of the letter "a" with the letter "b" in a text file:

bash

```
• tr 'a' 'b' < input.txt
```

- **Removing Characters:** To remove all digits from a text file:

bash

```
• tr -d '0-9' < input.txt
```

- **Translating Characters:** To convert lowercase letters to uppercase in a text file:

bash

```
• tr 'a-z' 'A-Z' < input.txt
```

- **Squeezing Repeated Characters:** To squeeze consecutive repeating characters into a single character:

bash

```
• echo "helllllooo" | tr -s 'l'
```

- **Complementing Characters:** To complement a set of characters (i.e., replace all characters not in the set):

bash

```
5. echo "abc123" | tr -c 'a-z' 'X'
```

The `tr` command can be useful for various text manipulation tasks, especially when combined with input/output redirection and piping. Keep in mind that `tr` works with single characters, so it's best suited for simple

transformations. If you need more advanced text processing, you might want to explore tools like `sed` or `awk`. To learn more about the options and usage of the `tr` command, you can refer to the command's manual by typing `man tr` in the terminal.

SA

basename , expr command

The `basename` and `expr` commands are both commonly used in Linux/Unix systems for different purposes.

1. basename Command: The `basename` command is used to extract the filename or the last part of a pathname. It strips off the directory path and displays only the filename component.

Here's the basic syntax of the `basename` command:

bash

```
basename [options] pathname
```

Some examples of using the `basename` command:

- Extract the filename from a full path:

bash

- `basename /home/user/file.txt`

Output: `file.txt`

- Extract the filename from a path with trailing slashes:

bash

- `basename /home/user/directory/`

Output: `directory`

2. expr Command: The `expr` command is used for performing basic arithmetic operations, string comparisons, and evaluating expressions in shell scripts. It's versatile but mainly suited for integer arithmetic.

Here's a simple example of using the `expr` command for arithmetic:

```
bash  
result=$(expr 5 + 3)  
echo $result # Outputs: 8
```

And an example of using `expr` for string comparison:

```
bash  
if [ $(expr "$string1" = "$string2") -eq 1 ]; then  
    echo "Strings are equal"  
else  
    echo "Strings are not equal"  
fi
```

However, note that `expr` has limitations and is more basic compared to modern scripting languages and utilities. It's often recommended to use built-in shell arithmetic (`$((. . .))` for Bash) or other scripting languages like Python for more complex tasks.

Keep in mind that both `basename` and `expr` commands can vary in features and behavior across different Unix-like systems. You can check their respective manual pages (`man basename` and `man expr`) for more details and usage options specific to your system.

SA

basename command to change filename extension

The `basename` command by itself is not designed to directly change the file extension of a filename. It's primarily used to extract the base name (filename without the path) from a given path.

However, you can combine the `basename` command with other commands like `echo` and string manipulation to achieve the desired result. Here's an example of how you can use `basename` to change the file extension:

bash

```
original_filename="/path/to/your/file.txt"
```

```
new_extension=".csv"
```

```
base_name=$(basename "$original_filename")
```

```
new_filename="${base_name%.*}.${new_extension}"
```

```
echo "$new_filename"
```

In this example, the `%.*` part in `${base_name%.*}` removes the extension from the base name, and then the new extension is added using `${new_extension}`. This effectively changes the extension of the filename.

Let's say you have a file named `/path/to/your/file.txt`, and you want to change the extension to `.csv`. The script above will output `/path/to/your/file.csv`.

Keep in mind that this approach is a basic demonstration. If you need to handle more complex cases or multiple periods in the filename, you might need more robust string manipulation or a scripting language like Python or Bash for better control.

SA

How to run a script?

Running a script depends on the type of script you have and the programming or scripting language it's written in. Here are the general steps to run scripts in various scenarios:

- **1. Shell Scripts (Bash, Shell, etc.):**

If you have a shell script (a script written in a shell language like Bash), follow these steps:

a. Open a terminal. b. Navigate to the directory where your script is located using the `cd` command. c. Run the script using `./` followed by the script filename. For example, if your script is named `myscript.sh`:

`bash`

- `chmod +x myscript.sh # Make the script executable (if needed)`

`./myscript.sh # Run the script`

- **Python Scripts:**

If you have a Python script, follow these steps:

a. Open a terminal. b. Navigate to the directory where your script is located using the `cd` command. c. Run the script using the `python` command followed by the script filename. For example, if your script is named `myscript.py`:

`bash`

- `python myscript.py # Run the Python script`

Note that you might need to use `python3` instead of `python` if you have both Python 2 and Python 3 installed.

- **Other Scripting/Programming Languages:**

For other scripting or programming languages, the general approach involves using the appropriate interpreter or compiler for that language. Replace `<language>` with the actual language you're using:

a. Open a terminal. b. Navigate to the directory where your script is located using the `cd` command. c. Run the script using the appropriate interpreter/compiler. For example:

bash

3. `<language> script.<extension> # Replace <language> and <extension> with actual values`

Remember to make the script executable if needed. In some cases, you might need to adjust permissions using the `chmod` command.

Additionally, if your script relies on specific files or libraries, ensure that they're available in the same directory or are properly installed.

Keep in mind that this is a general overview. The steps might vary based on your operating system and setup.

SA

command to make a new script file?

2 / 2

To create a new script file, you can use a text editor or a command-line tool. Here's how you can create a new script file using the command line:

1. Using a Text Editor:

You can use a text editor like `nano`, `vim`, or `gedit` to create and edit your script file. For example, to create a new Bash script named `myscript.sh` using `nano`:

bash

- `nano myscript.sh`

Then you can start typing your script code in the text editor. After you're done, save the file and exit the text editor according to the editor's instructions.

• Using Command-Line Redirection:

If you want to create a simple script without opening a text editor, you can use output redirection (>) to create the file and input data. For example, to create a new Bash script named `myscript.sh`:

bash

- `echo "#!/bin/bash" > myscript.sh`

```
echo "echo 'Hello, world!'" >> myscript.sh
```

```
chmod +x myscript.sh # Make the script executable
```

The above example creates a basic Bash script that prints "Hello, world!" when executed.

- **Using a Here Document:**

You can also use a here document to create a script. This method is particularly useful for creating more complex scripts. For example, to create a Bash script named `myscript.sh`:

bash

3. `cat << EOF > myscript.sh`

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

```
echo 'Hello, world!'
```

```
EOF
```

```
chmod +x myscript.sh # Make the script executable
```

The `<< EOF` syntax indicates the start of the here document, and the lines between `EOF` are the content of the script.

Remember to make your script file executable using the `chmod` command, as shown in the examples above. Once the script file is created, you can execute it using the appropriate interpreter or by running `./script_name`.

Why permission is denied while running a script?

The "Permission denied" error when trying to run a script typically occurs because the script does not have the necessary execution permissions. In Linux and Unix-like systems, files must be marked as executable before they can be run as scripts.

Here are a few common scenarios that might lead to a "Permission denied" error and how to address them:

1. Missing Execution Permission:

If the script does not have the execute permission for the user attempting to run it, you will see a "Permission denied" error.

To fix this, you need to make the script executable using the `chmod` command:

`bash`

```
1.   chmod +x script_name
```

Replace `script_name` with the actual name of your script file.

2. Wrong Path:

If you're trying to execute the script using its path (e.g., `/path/to/script_name`), make sure the path is correct. Double-check the path and the spelling of the filename.

3. Running from the Wrong Directory:

If you're running the script from a different directory, make sure you provide the correct path to the script or navigate to the directory where the script is located using the `cd` command.

4. Permissions on the Script's Parent Directories:

If the script is located in a directory that is not accessible due to its permissions, you might encounter a "Permission denied" error. Check the

permissions of the script's parent directories using the `ls -l` command and adjust them if needed.

5. Filesystem Mounted as No-Exec:

Sometimes, filesystems are mounted with the "noexec" option, which prevents any files from being executed. Check if your filesystem has been mounted with this option.

6. File Ownership:

Ensure that you have the necessary permissions to access the script file. If the file is owned by another user or if you don't have read or execute permissions on the file, you might encounter permission issues.

7. SELinux or AppArmor Restrictions:

If you're using a system with security modules like SELinux or AppArmor, they might impose restrictions on script execution. Check your system's security policies.

8. File Corruption:

In rare cases, if the file is corrupted, it might not be executable. You might need to recreate the file or restore it from a backup.

Remember to use the `ls -l` command to check the file permissions and ownership of the script file. If you're still encountering issues after checking these aspects, additional troubleshooting might be needed based on the specifics of your situation.

SA

chgrp command to change files group ownership

The `chgrp` command is used in Linux and Unix-like systems to change the group ownership of files and directories. Here's the basic syntax of the `chgrp` command:

bash

```
chgrp [options] new_group file(s)
```

Here, `new_group` is the name of the group to which you want to change ownership, and `file(s)` represents the list of files or directories for which you want to change the group ownership.

Here are a few examples of using the `chgrp` command:

1. Change the group ownership of a single file:

bash

- `chgrp newgroup myfile.txt`

```
sayak@ubuntu:~$ cd harry
sayak@ubuntu:~/harry$ ls -l
total 12
-rwxrw-r-- 1 sayak sayak    0 Aug 14 15:25 2.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 15:38 56.txt
-rw-rw-r-- 1 sayak sayak  219 Aug 21 10:33 abc.txt
-rw-rw-r-- 1 sayak sudo     0 Aug 21 10:32 def.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 18:31 harry.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 18:31 HARRY.txt
-rw-rw-r-- 1 sayak sayak   22 Aug 16 10:24 rohan.txt
drwxr--r-- 2 sayak sayak 4096 Aug 14 15:32 this
sayak@ubuntu:~/harry$ groups
sayak sudo vboxsf
```

Change
group
ownership
multiple

bash

```
sayak@ubuntu:~/harry$ chgrp vboxsf def.txt
sayak@ubuntu:~/harry$ ls -l
total 12
-rwxrw-r-- 1 sayak sayak    0 Aug 14 15:25 2.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 15:38 56.txt
-rw-rw-r-- 1 sayak sayak  219 Aug 21 10:33 abc.txt
-rw-rw-r-- 1 sayak vboxsf    0 Aug 21 10:32 def.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 18:31 harry.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 18:31 HARRY.txt
-rw-rw-r-- 1 sayak sayak   22 Aug 16 10:24 rohan.txt
drwxr--r-- 2 sayak sayak 4096 Aug 14 15:32 this
sayak@ubuntu:~/harry$
```

chgrp

```
newgroup file1.txt file2.txt file3.txt
```

- Change the group ownership of all files in a directory:

bash

- `chgrp newgroup /path/to/directory/*`

```
sayak@ubuntu:~/harry/this$ ls -l
total 0
-rw-rw-r-- 1 sayak sudo 0 Aug 14 15:22 1.txt
-rw-rw-r-- 1 sayak sudo 0 Aug 14 15:32 2.txt
sayak@ubuntu:~/harry/this$ chgrp sayak /home/sayak/harry/this/*
sayak@ubuntu:~/harry/this$ ls -l
total 0
-rw-rw-r-- 1 sayak sayak 0 Aug 14 15:22 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 14 15:32 2.txt
sayak@ubuntu:~/harry/this$ █
```

Change the group ownership recursively for a directory and its subdirectories:

bash

4. `chgrp -R newgroup /path/to/directory`

Some common options that you can use with the `chgrp` command:

- `-R` or `--recursive`: Recursively change group ownership for directories and their contents.
- `-v` or `--verbose`: Display a message for each file or directory whose group ownership is changed.
- `--reference=file`: Change the group ownership to match that of the reference file.

Keep in mind that you need appropriate permissions to change the group ownership of files. Typically, only the owner of the file or a superuser (root) can change the group ownership. Also, ensure that the new group you specify exists on your system. You can use the `groups` command to check the existing groups.

Remember to use the `ls -l` command to verify the changes in group ownership after using `chgrp`.

SA

touch command to change files last modification or access time

The `touch` command in Linux and Unix-like systems is primarily used to create new empty files or update the access and modification timestamps of existing files. By default, it updates both the access and modification timestamps to the current time.

To update the timestamps of a file without modifying its content, you can use the `touch` command with the filename as an argument. Here's the basic syntax:

bash

```
touch filename
```

This will update both the access and modification timestamps of the file to the current time.

If you want to specifically update only the access or modification timestamp, you can use the `-a` (access time) or `-m` (modification time) options, respectively:

bash

```
touch -a filename # Update access time
```

```
touch -m filename # Update modification time
```

For example, to update only the modification time of a file named `myfile.txt`:

bash

```
touch -m myfile.txt
```

```
-rw-rw-r-- 1 sayak sayak 0 Aug 14 15:22 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 14 15:32 2.txt
sayak@ubuntu:~/harry/this$ touch 1.txt
sayak@ubuntu:~/harry/this$ ls -l
total 0
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:19 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 14 15:32 2.txt
sayak@ubuntu:~/harry/this$ touch -a 2.txt
sayak@ubuntu:~/harry/this$ ls -l
total 0
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:19 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:21 2.txt
sayak@ubuntu:~/harry/this$ █
```

Keep in mind that the ability to change timestamps depends on the permissions you have for the file. Typically, you need write permissions for the file to update its timestamps. If you're working with system files or directories, you might need superuser (root) privileges to modify timestamps.

It's important to note that while you can update timestamps using the `touch` command, you cannot set arbitrary timestamps to a specific date and time. The `touch` command updates timestamps to the current system time. If you need precise control over timestamps, you might need to use other tools or programming languages that allow manipulation of timestamps.

SA

chown command to change the files ownership

The `chown` command in Linux and Unix-like systems is used to change the ownership (both user and group) of files and directories. Here's the basic syntax of the `chown` command:

bash

```
chown [options] new_owner[:new_group] file(s)
```

Here:

- `new_owner` is the new owner's username.
- `new_group` is the new group's name (optional).

You can use both `new_owner` and `new_group`, or just one of them.

Here are some examples of using the `chown` command:

1. Change both user and group ownership of a single file:

bash

- `chown newuser:newgroup myfile.txt`

```
root@ubuntu:/home/sayak/harry/this# ls -l
total 0
-rw-rw-r-- 1 root sayak 0 Aug 21 11:19 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:21 2.txt
root@ubuntu:/home/sayak/harry/this# chown sayak:vboxsf 1.txt
root@ubuntu:/home/sayak/harry/this# ls -l
total 0
-rw-rw-r-- 1 sayak vboxsf 0 Aug 21 11:19 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:21 2.txt
root@ubuntu:/home/sayak/harry/this#
```

Change only the user ownership of a file:

bash

- `chown newuser myfile.txt`

```
-rw-rw-r-- 1 sayak vboxsf 0 Aug 21 11:19 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:21 2.txt
root@ubuntu:/home/sayak/harry/this# chown root 1.txt
root@ubuntu:/home/sayak/harry/this# ls -l
total 0
-rw-rw-r-- 1 root vboxsf 0 Aug 21 11:19 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:21 2.txt
```

Change only the group ownership of a file:

bash

- `chown :newgroup myfile.txt`

```
root@ubuntu:/home/sayak/harry/this# ls -l
total 0
-rw-rw-r-- 1 root vboxsf 0 Aug 21 11:19 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:21 2.txt
root@ubuntu:/home/sayak/harry/this# chown :sayak 1.txt
root@ubuntu:/home/sayak/harry/this# ls -l
total 0
-rw-rw-r-- 1 root sayak 0 Aug 21 11:19 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:21 2.txt
```

- Change ownership for multiple files:

bash

- `chown newuser:newgroup file1.txt file2.txt
file3.txt`

- Change ownership recursively for a directory and its contents:

bash

5. `chown -R newuser:newgroup
/path/to/directory`

Common options you can use with the `chown` command:

- `-R` or `--recursive`: Recursively change ownership for directories and their contents.
- `-v` or `--verbose`: Display a message for each file or directory whose ownership is changed.
- `--reference=file`: Change ownership to match that of the reference file.

Keep in mind that you need appropriate permissions to change the ownership of files. Typically, only the superuser (root) can change the ownership to any user or group. Ordinary users can change the ownership of only those files they own.

Also, make sure to specify the correct username and group name. You can use the `id` command to see the list of users and groups on your system.

After using `chown`, you can verify the ownership changes using the `ls -l` command.

SA

What is the meaning of recursively?

"Recursively" refers to the process of applying an operation not only to a specific item but also to all its sub-items, and their sub-items, and so on, until all items within the hierarchy have been processed. In other words, a

recursive operation is one that operates repeatedly on the sub-elements of a larger structure.

For example, when dealing with directories and files, performing an action recursively means applying that action to the main directory and all of its subdirectories, and then to subdirectories within those subdirectories, and so on. This is often used when you need to perform the same operation on all the items within a directory tree.

In the context of Unix-like systems and commands, the `-R` or `--recursive` option is frequently used to indicate that an action should be performed on the specified directory and its entire content, including all subdirectories and their contents. This is useful for tasks like copying, moving, changing permissions, or changing ownership of a directory and all its contents in a single command.

For example, the `cp` command with the `-R` option (`cp -R source_directory target_directory`) would recursively copy the contents of `source_directory` and all its subdirectories into `target_directory`. Similarly, the `chown` command with the `-R` option (`chown -R new_owner:new_group directory`) would recursively change the ownership of `directory` and all its subdirectories and files.

In summary, "recursively" means performing an operation on an item and its entire hierarchy of nested sub-items. It's a powerful concept that's commonly used in programming, system administration, and various other fields to efficiently handle complex structures.

SA

passwd command to change password

The `passwd` command in Linux and Unix-like systems is used to change a user's password. Users can change their own passwords if they know

their current password, and the root user (superuser) can change the passwords for other users.

Here's the basic syntax of the `passwd` command:

To change your own password:

bash

`passwd`

To change another user's password (as root):

bash

`sudo passwd username`

Replace `username` with the username of the user whose password you want to change.

When you run the `passwd` command, you'll be prompted to enter your current password (or the root password, if you're changing another user's password). After that, you'll be prompted to enter the new password twice to confirm it. The password you type will not be displayed on the screen for security reasons.

SA

type command to know command type.

The `type` command in Linux and Unix-like systems is used to determine the type of a command. It provides information about how a command would be interpreted if it were executed. This can include whether the command is a shell built-in, an alias, a function, or an external executable file.

Here's the basic syntax of the `type` command:

bash

`type command_name`

Replace `command_name` with the name of the command you want to inquire about.

Here are some examples of how the `type` command can be used:

1. Check if a command is a shell built-in:

bash

- `type echo`
- Check if a command is an alias:

bash

- `type ls`
- Check if a command is a function defined in your shell:

bash

- `type my_function`
- Check if a command is an external executable file:

bash

4. `type python`

```
root@ubuntu:/home/sayak/harry/this# alias ll='ls -l'
root@ubuntu:/home/sayak/harry/this# ll
total 0
-rw-rw-r-- 1 sayak vboxsf 0 Aug 21 11:19 1.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 21 11:21 2.txt
root@ubuntu:/home/sayak/harry/this# type ls -l
ls is aliased to `ls --color=auto'
bash: type: -l: not found
root@ubuntu:/home/sayak/harry/this# type ll
ll is aliased to `ls -l'
root@ubuntu:/home/sayak/harry/this# type echo
echo is a shell builtin
root@ubuntu:/home/sayak/harry/this# t
```

The output of the `type` command will provide information about the nature of the command. It might indicate if the command is a shell built-in, an alias, a function, or if it's available as an external executable (along with its path).

Keep in mind that the `type` command's behavior might slightly vary based on the shell you are using (e.g., Bash, Zsh, etc.), as different shells can have different ways of handling commands and their types.

SA

bzip2 command to compress file

The `bzip2` command in Linux and Unix-like systems is used to compress files using the Burrows-Wheeler block sorting text compression algorithm and Huffman coding. It produces files with the `.bz2` extension. The compression provided by `bzip2` is typically higher compared to other compression methods.

Here's the basic syntax of the `bzip2` command:

bash

```
bzip2 [options] filename
```

For example, to compress a file named `example.txt`, you would use:

bash

```
bzip2 example.txt
```

This will create a compressed file named `example.txt.bz2` and remove the original `example.txt` file.

Commonly used options with the `bzip2` command:

- `-d` or `--decompress`: Decompress the specified file.
- `-k` or `--keep`: Keep the original file after compression (or decompression) instead of removing it.
- `-C` or `--stdout`: Write compressed data to standard output instead of creating a file.(??!!)
- `-v` or `--verbose`: Display verbose output.

```
root@ubuntu:/home/sayak/harry/this# ls
1.txt 2.txt
root@ubuntu:/home/sayak/harry/this# bzip2 1.txt
root@ubuntu:/home/sayak/harry/this# ls
1.txt.bz2 2.txt
root@ubuntu:/home/sayak/harry/this# bzip2 -d 1.txt.bz2
root@ubuntu:/home/sayak/harry/this# ls
1.txt 2.txt
root@ubuntu:/home/sayak/harry/this# bzip2 -k 1.txt
root@ubuntu:/home/sayak/harry/this# ls
1.txt 1.txt.bz2 2.txt
root@ubuntu:/home/sayak/harry/this# bzip2 -c 1.txt
bzip2: I won't write compressed data to a terminal.
bzip2: For help, type: `bzip2 --help'.
```

For example, to decompress a file named `example.txt.bz2` and keep the compressed file:

bash

```
bzip2 -dk example.txt.bz2
```

Remember that using the `bzip2` command will replace the original file with the compressed or decompressed version, depending on the operation you perform. If you want to keep the original file intact, use the `-k` option.

The `bzip2` command provides an efficient compression method, but keep in mind that the trade-off is that compression might be slower compared to some other compression utilities. If you're looking for faster compression or different compression algorithms, you might also consider using tools like `gzip` or `xz`.

SA

gzip command to compress file

The `gzip` command in Linux and Unix-like systems is used to compress files using the GNU zip compression algorithm. It produces files with the `.gz` extension. The compression provided by `gzip` is relatively fast and provides moderate compression ratios.

Here's the basic syntax of the `gzip` command:

bash

```
gzip [options] filename
```

For example, to compress a file named `example.txt`, you would use:

```
bash
```

```
gzip example.txt
```

This will create a compressed file named `example.txt.gz` and remove the original `example.txt` file.

Commonly used options with the `gzip` command:

- `-d` or `--decompress`: Decompress the specified `.gz` file.
- `-k` or `--keep`: Keep the original file after compression (or decompression) instead of removing it.
- `-C` or `--stdout`: Write compressed data to standard output instead of creating a file.
- `-v` or `--verbose`: Display verbose output.

For example, to decompress a file named `example.txt.gz` and keep the compressed file:

```
bash
```

```
gzip -dk example.txt.gz (this command didn't work!?)
```

Remember that using the `gzip` command will replace the original file with the compressed or decompressed version, depending on the operation you perform. If you want to keep the original file intact, use the `-k` option.

`gzip` provides a good balance between speed and compression efficiency, making it a commonly used compression utility. If you're looking for higher compression ratios at the cost of potentially slower compression, you might consider using utilities like `bzip2` or `xz`.

zip - Compress multiple files to a single file

The `zip` command in Linux and Unix-like systems is used to compress files and directories into a single compressed archive file. It's a common utility for creating compressed archives that can be easily shared or transferred.

Here's the basic syntax of the `zip` command:

bash

```
zip [options] archive.zip file(s) directory(s)
```

- `archive.zip` is the name of the output compressed archive.
- `file(s)` and `directory(s)` are the list of files and directories you want to include in the archive.

For example, to create an archive named `myarchive.zip` containing files `file1.txt`, `file2.txt`, and the entire `mydirectory` directory:

bash

```
zip myarchive.zip file1.txt file2.txt mydirectory
```

```
root@ubuntu:/home/sayak/harry# ls
2.txt abc.txt harry.txt rohan.txt
56.txt def.txt HARRY.txt this
root@ubuntu:/home/sayak/harry# zip myarchive.zip 56.txt 2.txt this
adding: 56.txt (stored 0%)
adding: 2.txt (stored 0%)
adding: this/ (stored 0%)
root@ubuntu:/home/sayak/harry# ls
2.txt abc.txt harry.txt myarchive.zip this
56.txt def.txt HARRY.txt rohan.txt
```

Commonly used options with the `zip` command:

- `-r` or `--recurse-paths`: Recursively include directories and their contents.
- `-q` or `--quiet`: Quiet mode, suppress output messages.
- `-v` or `--verbose`: Display verbose output.

- **-x** or **--exclude**: Exclude files or patterns from being added to the archive.

For example, to create a compressed archive of the **mydirectory** directory and its contents, excluding files with the **.tmp** extension:

zip -r myarchive.zip mydirectory -x "*.\tmp" (Did not work properly!??)

After

```
root@ubuntu:/home/sayak/harry# ls -R
.:
2.txt  abc.txt  harry.txt  myarchive.zip  this
56.txt  def.txt  HARRY.txt  rohan.txt

./this:
1.txt  1.txt.bz2  2.txt
root@ubuntu:/home/sayak/harry# zip -r myarchive2.zip this -x "*.\txt.bz2"
    adding: this/ (stored 0%)
    adding: this/1.txt (deflated 3%)
    adding: this/2.txt (stored 0%)
    adding: this/1.txt.bz2 (stored 0%)
root@ubuntu:/home/sayak/harry# ls -R
.:
2.txt  abc.txt  harry.txt  myarchive2.zip  rohan.txt
56.txt  def.txt  HARRY.txt  myarchive.zip  this

./this:
1.txt  1.txt.bz2  2.txt
```

creating the archive, you can use the **unzip** command to extract its contents:

bash

unzip myarchive.zip

Remember that the **zip** command creates archives in the ZIP format, which is widely supported and can be extracted using various tools on different platforms. If you're working with specific compression formats like **.gz** or **.bz2**, you would use the **gzip** or **bzip2** commands, respectively, as discussed earlier.

cat - Concatenate files

The `cat` command in Linux and Unix-like systems is used to concatenate and display the contents of one or more files. The word "concatenate" means to combine multiple items end-to-end, and in this case, the `cat` command is used to display the contents of files in the terminal.

Here's the basic syntax of the `cat` command:

bash

```
cat [options] file(s)
```

You can provide one or more filenames as arguments to the `cat` command. It will then display the contents of those files on the terminal.

For example, to display the contents of a single file named `file.txt`:

bash

```
cat file.txt
```

To display the contents of multiple files, you can list their names as separate arguments:

bash

```
cat file1.txt file2.txt file3.txt
```

Commonly used options with the `cat` command:

- `-n` or `--number`: Display line numbers along with the content.
- `-b` or `--number-nonblank`: Display line numbers only for non-blank lines.
- `-A` or `--show-all`: Display non-printable characters as well (visible using `-v` or `-e`).
- `-s` or `--squeeze-blank`: Squeeze multiple adjacent blank lines into a single blank line.

For example, to display the contents of a file with line numbers:

bash

```
cat -n file.txt
```

```
root@ubuntu:/home/sayak/harry/this# cat -n 1.txt
 1 sfgtrht
 2 dfgfkj
 3 gldis
 4 flkrs
 5 38509sjkfn
 6 0375hsnayenrh
 7 edlkeiue92432@!#@#@#(8900
 8
```

`cat` is a simple yet versatile command that's often used for quickly viewing the contents of files, especially text files. Keep in mind that if you need to view specific parts of a file, you might also consider using other commands like `head`, `tail`, or text editors like `less`.

SA

cp -r - Copy directory tree.

The `cp` command in Linux and Unix-like systems is used to copy files and directories. The `-r` option (or `--recursive` option) is used to copy directories and their contents recursively, creating a copy of the entire directory tree.

Here's the basic syntax of the `cp` command with the `-r` option:

bash

```
cp -r source_directory destination_directory
```

- `source_directory` is the directory you want to copy.
- `destination_directory` is the location where the copy should be placed.

For example, to copy the entire contents of a directory named `source` to a new directory named `destination`:

bash

```
cp -r source destination
```

The `-r` option ensures that not only the files in the `source` directory are copied but also all subdirectories and their contents, maintaining the directory structure.

Commonly used options with the `cp` command:

- `-i` or `--interactive`: Prompt before overwriting files.
- `-u` or `--update`: Copy only when the source file is newer than the destination file.
- `-v` or `--verbose`: Display verbose output.

For example, to copy a directory while displaying verbose output:

`bash`

```
cp -rv source destination
```

Keep in mind that the `cp` command with the `-r` option can be used to copy entire directory trees, but it's important to use it with caution, especially when overwriting existing files or directories. Always double-check your commands and consider using the `-i` option to prompt for confirmation before overwriting files.

SA

what is verbose output?

Verbose output, often referred to as "verbosity," is a type of output generated by a program or command that provides more detailed and extensive information than the standard or default output. It's designed to give users a deeper understanding of what the program is doing behind the scenes, which can be particularly helpful for troubleshooting, debugging, or gaining insight into the program's behavior.

When a program or command runs with verbose output enabled, it might display additional details such as:

- **Progress updates:** Information about the progress of an operation, like the percentage completed or the current step being executed.
- **Debugging information:** Internal details of how the program is processing data or making decisions.
- **Actions taken:** Details about what actions are being performed, such as which files are being accessed, modified, or copied.
- **Configurations:** Displaying configuration settings or options that the program is using during its execution.
- **Additional messages:** Providing extra messages that can help the user understand what the program is doing at various points.

Verbose output can be useful in various scenarios, including:

- **Troubleshooting:** When something goes wrong, verbose output can reveal details that help identify the issue.
- **Learning:** Users can learn more about how a program works and how it interacts with various components.
- **Fine-tuning:** For programs that have multiple options and configurations, verbose output can help users understand how different options affect the program's behavior.

In command-line programs, verbosity is often controlled using command-line options like **-v**, **--verbose**, or other similar flags. For example, running a command with the **-v** option might enable verbose output.

However, verbose output can be overwhelming for simple tasks or casual use, as it tends to produce a lot of information. It's usually recommended to use verbose output when specifically troubleshooting or when you need to understand the inner workings of a program in more detail.

ftp, scp, sftp - Copy file between machines

ftp, scp, and sftp are three different methods for copying files between machines in a networked environment, typically over SSH or FTP protocols. Each method has its own features, advantages, and use cases.

1. FTP (File Transfer Protocol):

FTP is an older network protocol designed for transferring files over a network. It operates on port 21 by default. While FTP is not secure by default (data is transferred in plain text), you can use FTPS (FTP Secure) to add encryption.

To use FTP, you typically need an FTP client (like `ftp` command) on the source machine and an FTP server on the target machine. Here's a basic example using the `ftp` command:

bash

- `ftp target_machine_address`

Once connected, you can use commands like `put` to upload files and `get` to download files.

• SCP (Secure Copy Protocol):

SCP is a secure file transfer protocol that uses SSH to transfer files between machines. It's included with most SSH installations and provides encrypted and secure file copying.

To use SCP, you can use the `scp` command with the following syntax:

bash

```
scp source_file  
username@target_machine:destination_path
```

For example, to copy a local file named `file.txt` to a remote machine:

bash

```
scp file.txt  
username@target_machine:/path/to/destination/
```

You can also copy files from the remote machine to your local machine:

bash

- scp
username@source_machine:/path/to/source/file.txt .
- **SFTP (SSH File Transfer Protocol):**

SFTP is a secure alternative to FTP that uses SSH for secure file transfer. It operates over an SSH connection on port 22 and provides encrypted data transmission.

To use SFTP, you can use the `sftp` command with the following syntax:

bash

```
sftp username@target_machine
```

Once connected, you can use commands similar to those in a regular shell, such as `put` to upload files and `get` to download files.

arduino

```
3.      sftp> put local_file.txt
```

```
sftp> get remote_file.txt
```

When choosing between `scp` and `sftp`, `sftp` tends to be more versatile as it provides an interactive interface similar to a regular shell.

Both SCP and SFTP are recommended for secure file transfers, with SCP being simpler for single file transfers and SFTP providing a more feature-rich interactive environment. FTP should generally be avoided for security reasons due to its lack of encryption.

grep -c - Count number of lines containing a pattern.

The grep command in Linux and Unix-like systems is used to search for text patterns within files. The `-c` option is used with `grep` to count the number of lines that contain a specific pattern in one or more files.

Here's the basic syntax of the `grep` command with the `-c` option:

bash

```
grep -c pattern file(s)
```

- `pattern` is the text pattern you want to search for.
- `file(s)` is the list of files in which you want to search for the pattern.

For example, to count the number of lines containing the word "apple" in a file named `fruits.txt`:

bash

```
grep -c "apple" fruits.txt
```

```
root@ubuntu:/home/sayak/harry/this# cat 1.txt
sfgtrht
dfgfkj
gldis
flkrs
38509sjkfn
0375hsnayenrh
edlkeiue92432!#@#@#(8900
cd ..

root@ubuntu:/home/sayak/harry/this# grep -c "3" 1.txt
3
root@ubuntu:/home/sayak/harry/this#
```

`fruits.txt` that contain the word "apple."

You can also search for a pattern in multiple files. For instance, to count the occurrences of the word "error" in all `.log` files in a directory:

bash

```
grep -c "error" *.log
```

The -c
is
when
only
know
count of
that
the
without

```
root@ubuntu:/home/sayak/harry/this# cat 1.txt
sfgtrht
dfgfkj
gldis
flkrs
38509sjkfn
0375hsnayenrh
edlkeiue92432@!#@#@#(8900
cd ..

root@ubuntu:/home/sayak/harry/this# cat 2.txt
jfkf
l2wo395
wpwe
r,rkrtrpl20935437560-
5867986587543056954=-60=3529
-3945609506
-3956-9654/-/-+


9789565
-**/-*8+9

root@ubuntu:/home/sayak/harry/this# grep -c "3" *.txt
1.txt:3
2.txt:5
```

option
helpful
you
want to
the
lines
match
pattern,

displaying the actual content of the lines.

Keep in mind that the **grep** command is case-sensitive by default. If you want to perform a case-insensitive search, you can use the **-i** option:

bash

```
grep -ci "apple" fruits.txt
```

This command will count lines containing "apple" regardless of case (e.g., "Apple," "APPLE," etc.).

SA

wc - Count number of lines, words, and characters.

The `wc` command in Linux and Unix-like systems is used to count the number of lines, words, and characters in a file or input provided through standard input. It's a versatile command that can help you analyze text files and output.

Here's the basic syntax of the `wc` command:

bash

```
wc [options] file(s)
```

You can provide one or more filenames as arguments to the `wc` command. It will then display the counts for lines, words, and characters in the given file(s).

Commonly used options with the `wc` command:

- `-l` or `--lines`: Display the count of lines.
- `-w` or `--words`: Display the count of words.
- `-c` or `--bytes`: Display the count of bytes (characters).
- `-m` or `--chars`: Display the count of characters (equivalent to `--bytes`).
- `-L` or `--max-line-length`: Display the length of the longest line.

For example, to count the number of lines, words, and characters in a file named `sample.txt`:

bash

```
wc sample.txt
```

This will display output like this:

```
5 20 120 sample.txt
```

Where:

- 5 is the number of lines.
- 20 is the number of words.
- 120 is the number of characters.

```
root@ubuntu:/home/sayak/harry/this# cat 1.txt
sfgtrht
dfgfkj
gldis
flkrs
38509sjkfn
0375hsnayenrh
edlkeiue92432@!#@#@#(8900
cd ..
bash
root@ubuntu:/home/sayak/harry/this# wc 1.txt
 9  9 84 1.txt
```

sample.txt # Display line count

wc -w sample.txt # Display word count

wc -c sample.txt # Display character count

The **wc** command is handy for quickly obtaining statistics about the content of text files or the output of other commands. If you want to analyze multiple files at once, you can list their names as separate arguments to the **wc** command.

SA

ar - Create an archive of C object files.

The **ar** command in Linux and Unix-like systems is used to create and manipulate archive files, often referred to as "ar" files. These archives are used to bundle multiple object files together into a single file, which can be convenient for library management and distribution. The **ar** command is commonly used with C and C++ programming.

Here's the basic syntax of the **ar** command:

bash

```
ar [options] archive file(s)
```

- **archive** is the name of the archive file you want to create or modify.
- **file(s)** is the list of object files you want to include in the archive.

For example, to create an archive file named **myarchive.a** containing object files **file1.o** and **file2.o**:

bash

```
ar rcs myarchive.a file1.o file2.o
```

Commonly used options with the **ar** command:

- **r** or **--replace**: Add or replace files in the archive.
- **C** or **--create**: Create a new archive.
- **S** or **--plugin**: Use a specific plugin (not commonly used).
- **t** or **--list**: List the contents of the archive.
- **X** or **--extract**: Extract files from the archive.

For example, to list the contents of the **myarchive.a** archive:

bash

```
ar t myarchive.a
```

```
root@ubuntu:/home/sayak/harry/this# ls
1.txt  1.txt.bz2  2.txt
root@ubuntu:/home/sayak/harry/this# ar rcs myarchive.a 1.txt 2.txt
root@ubuntu:/home/sayak/harry/this# ls
1.txt  1.txt.bz2  2.txt  myarchive.a
root@ubuntu:/home/sayak/harry/this# cat myarchive.a
!<arch>
1.txt/          0          0      0      644      84
sfgtrht
dfgfkj
gldis
flkrs
38509sjkfn
0375hsnayenrh
edlkeiue92432@!#@#(8900
cd ..

2.txt/          0          0      0      644     117
jfkf
l2wo395
wpwe
r,rkrtrpl20935437560-
5867986587543056954=-60=3529
-3945609506
-3956-9654/-/-+
9789565
-**/-*8+9
```

```
root@ubuntu:/home/sayak/harry/this# ar t myarchive.a
1.txt
2.txt
root@ubuntu:/home/sayak/harry/this#
```

ar archives are often used to bundle compiled object files (**.o** files) and static libraries (**.a** files) that can be linked with other programs during compilation. They are widely used in C and C++ development for creating libraries that contain reusable code.

Keep in mind that **ar** archives are different from compressed archives like **.zip**, **.tar.gz**, or **.tar.bz2**. **ar** archives mainly deal with bundling object files and do not provide compression; they simply collect object files and related metadata.

In - Create links to a file.

The `ln` command in Linux and Unix-like systems is used to create links to files. There are two types of links that can be created using the `ln` command: hard links and symbolic (soft) links.

1. Hard Links: A hard link is a reference to an existing inode (file data and metadata) on the file system. Hard links essentially create multiple directory entries that point to the same data on disk. Hard links cannot point to directories or across different file systems.

Here's the basic syntax of the `ln` command for creating a hard link:

bash

```
ln [options] source_file hard_link_name
```

- `source_file` is the original file to which you want to create a link.
- `hard_link_name` is the name of the new hard link.

For example, to create a hard link named `hard_link.txt` that points to an existing file `original.txt`:

bash

```
ln original.txt hard_link.txt
```

2. Symbolic (Soft) Links: A symbolic link, also known as a soft link, is a special type of file that acts as a pointer to another file or directory. It can point to files or directories on different file systems and can also point to non-existent files. Symbolic links are indicated by an `l` at the beginning of the `ls -l` output.

Here's the basic syntax of the `ln` command for creating a symbolic link:

bash

```
ln -s source_file symbolic_link_name
```

- `source_file` is the target file or directory to which you want to create a symbolic link.
- `symbolic_link_name` is the name of the new symbolic link.

For example, to create a symbolic link named `sym_link.txt` that points to an existing file `original.txt`:

bash

```
ln -s original.txt sym_link.txt
```

```
root@ubuntu:/home/sayak/harry/this# ln 1.txt hard_link.txt
root@ubuntu:/home/sayak/harry/this# ls
1.txt  1.txt.bz2  2.txt  hard_link.txt  myarchive.a
root@ubuntu:/home/sayak/harry/this# cat hard_link.txt
sfgtrht
dfgfkj
gldis
flkrs
38509sjkfn
0375hsnayenrh
edlkeiue92432@!#@#(8900
cd ..

root@ubuntu:/home/sayak/harry/this# ln -s 1.txt soft_link.txt
root@ubuntu:/home/sayak/harry/this# ls
1.txt      2.txt          myarchive.a
1.txt.bz2  hard_link.txt  soft_link.txt
root@ubuntu:/home/sayak/harry/this#
```

When using symbolic links, you can delete the original file, and the symbolic link will still exist. However, if you delete the original file of a hard link, the hard link will still point to the data on disk until all hard links are deleted.

The `ln` command provides the ability to create links, and it's commonly used for organizing files, creating shortcuts, and maintaining consistent directory structures.

cut - Cut columns or fields from a file.

The **cut** command in Linux and Unix-like systems is used to extract specified columns or fields from lines in a file. It's particularly useful when working with delimited text files, such as CSV (comma-separated values) files, where each line contains fields separated by a specific delimiter (like a comma or tab).

Here's the basic syntax of the **cut** command:

bash

```
cut [options] -f fields file(s)
```

- **fields** specifies the columns or fields you want to extract. You can provide a single field number, a range of fields, or a comma-separated list of field numbers.
- **file(s)** is the name of the file(s) from which you want to extract fields.

For example, let's say you have a file named `data.csv` containing the following data:

Name, Age, Country

Alice, 25, USA

Bob, 30, Canada

Charlie, 22, UK

To extract the second field (Age) from this file, you would use:

bash

```
cut -f 2 -d ',' data.csv
```

The `-d ','` option specifies that the fields are delimited by a comma.

The output would be:

Age

25

30

22

Commonly used options with the **cut** command:

- **-d delimiter**: Specify the delimiter that separates fields.
- **-f fields**: Specify the fields or columns you want to extract.
- **-c characters**: Extract specific characters from each line
(character-based cutting).(?!)

For example, to extract both the Name and Country columns:

bash

```
cut -f 1,3 -d ',' data.csv
```

Name,Country

Alice,USA

Bob,Canada

Charlie,UK

The **cut** command is particularly handy when you need to extract specific columns from large data files, and it's commonly used in data processing and scripting tasks.

The **cut** command with the **-c** option is used to extract specific characters from lines in a file. Unlike the **-f** option, which extracts fields based on a delimiter, the **-c** option extracts characters based on their positions in the line.

```
root@ubuntu:/home/sayak/harry/this# cat 3.csv
Name,Age,Country
Alice,25,USA
Bob,30,Canada
Charlie,22,UK
root@ubuntu:/home/sayak/harry/this# cut -f 3 -d ',' 3.csv
Country
USA
Canada
UK
root@ubuntu:/home/sayak/harry/this# cut -c 2-6 3.csv
ame,A
lice,
ob,30
harli
```

-C

Here's the basic syntax of the **cut** command with the **-C** option:

bash

```
cut -c character_range file(s)
```

- **character_range** specifies the range of characters you want to extract. This can be a single character index, a range of character indices, or a comma-separated list of character indices.
- **file(s)** is the name of the file(s) from which you want to extract characters.

For example, consider a file named **text.txt** with the following content:

Hello, World!

To extract the first five characters:

bash

```
cut -c 1-5 text.txt
```

Output:

Hello

To extract characters at specific indices:

bash

```
cut -c 1,3,5 text.txt
```

Output:

```
Hlo
```

The `cut -c` option is particularly useful when you need to extract specific characters from lines, regardless of whether they are delimited by a certain character. It's commonly used in scenarios where character-based extraction is required, such as when working with fixed-width data formats.

SA

sleep - Delay command execution in a script.

The `sleep` command in Linux and Unix-like systems is used to introduce a delay or pause in the execution of a script or command. It's particularly useful when you want to add a time delay between consecutive commands, create timed loops, or simulate time-based behavior in scripts.

Here's the basic syntax of the `sleep` command:

bash

```
sleep [duration]
```

- `duration` is the amount of time to sleep, specified in seconds. It can be a whole number or a fractional value.

For example, to introduce a 5-second delay in a script:

bash

```
echo "Starting..."
```

```
sleep 5
```

```
echo "Done!"
```

```
sayak@ubuntu:~/harry>this$ cat script2.sh
echo "Starting..."
sleep 5
echo "Done!"

sayak@ubuntu:~/harry>this$ ./script2.sh
bash: ./script2.sh: Permission denied
sayak@ubuntu:~/harry>this$
```

(Why permission is denied??!)

In this example, the script will print "Starting...", then pause for 5 seconds using the `sleep` command, and finally print "Done!" after the delay.

You can also use fractions of seconds for shorter delays. For instance, to sleep for half a second:

```
bash
```

```
echo "Sleeping..."
sleep 0.5
echo "Awake!"
```

Keep in mind that the actual duration of the sleep might vary depending on system load and other factors. The `sleep` command does not guarantee precise timing, but it provides a reliable way to introduce controlled delays in scripts.

`sleep` is frequently used in shell scripts and automation tasks where you need to control the timing of actions, wait for processes to complete, or create pauses in a controlled manner.

SA

diff - Difference between two files (as sed-like instructions).

The **diff** command in Linux and Unix-like systems is used to compare the contents of two text files and display the differences between them. It can show the lines that differ and provide a unified or context diff format, which includes instructions that resemble **sed** commands for transforming one file into another.

Here's the basic syntax of the **diff** command:

bash

```
diff [options] file1 file2
```

- **file1** is the first file you want to compare.
- **file2** is the second file you want to compare.

For example, let's say you have two files named **file1.txt** and **file2.txt** with the following content:

file1.txt:

kotlin

Hello, this is file 1.

Line 2.

Line 3.

file2.txt:

kotlin

Hello, this is file 2.

Line 2 modified.

Line 3.

To compare these two files and display the differences:

bash

```
diff file1.txt file2.txt
```

The output might look like:

yaml

1c1

```
< Hello, this is file 1.
```

```
> Hello, this is file 2.
```

3c3

```
< Line 2.
```

```
> Line 2 modified.
```

In the context of the output:

- 1c1: Indicates that line 1 from `file1.txt` was changed to match line 1 from `file2.txt`.
- <: Shows the original content from `file1.txt`.
- ---: Separates the original content from the modified content.
- >: Shows the modified content from `file2.txt`.

The `diff` command has several options that allow you to customize the output format, ignore certain differences, and more. Some common options include:

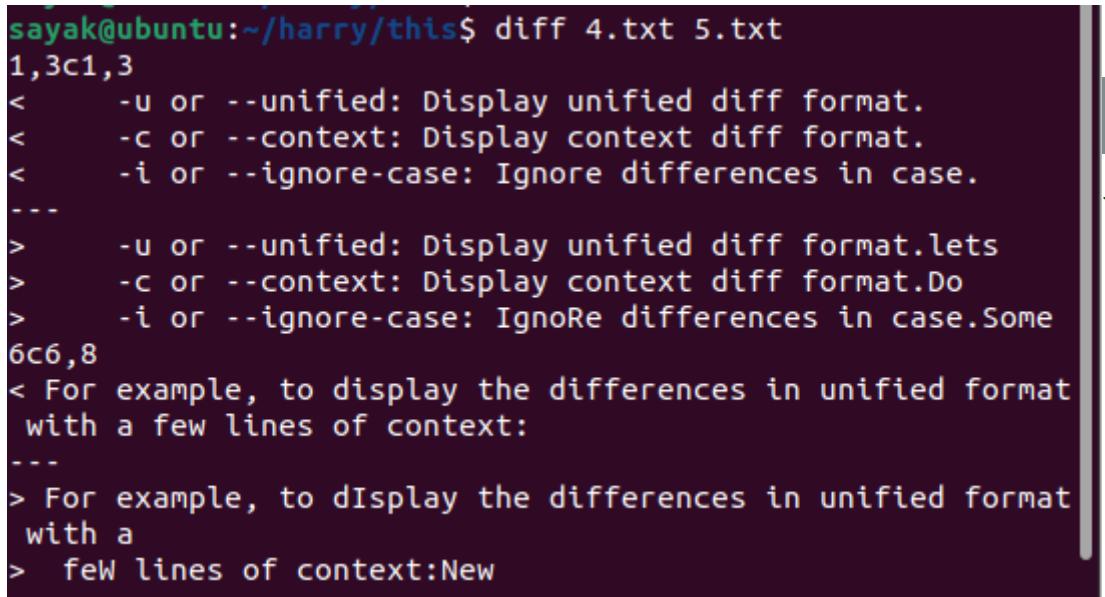
- `-u` or `--unified`: Display unified diff format.
- `-C` or `--context`: Display context diff format.
- `-i` or `--ignore-case`: Ignore differences in case.
- `-B` or `--ignore-blank-lines`: Ignore differences due to blank lines.

For example, to display the differences in unified format with a few lines of context:

bash

```
diff -u file1.txt file2.txt
```

The **diff** command is often used for comparing code changes, configurations, and any kind of textual differences between files.



```
sayak@ubuntu:~/harry/this$ diff 4.txt 5.txt
1,3c1,3
<     -u or --unified: Display unified diff format.
<     -c or --context: Display context diff format.
<     -i or --ignore-case: Ignore differences in case.
---
>     -u or --unified: Display unified diff format.lets
>     -c or --context: Display context diff format.Do
>     -i or --ignore-case: IgnoRe differences in case.Some
6c6,8
< For example, to display the differences in unified format
  with a few lines of context:
---
> For example, to dIsplay the differences in unified format
  with a
> few lines of context:New
```

SA

ls

grep "^\d" - Directory list.

The command you provided, `ls -l | grep "^\d"`, is used to list directories in a directory listing using the `ls` command and then filter out only the lines that represent directories using the `grep` command.

Here's how the command works:

1. `ls -l`: This command lists the contents of the current directory in long format. The `-l` option stands for "long" and provides detailed information about each file and directory, including permissions, owner, group, size, and timestamps.

2. | (Pipe): This symbol is used to redirect the output of the command on its left to the input of the command on its right. In this case, it takes the output of `ls -l` and pipes it as input to the `grep` command.
3. `grep "^\d"`: This command filters the input lines to match only lines that start with the character "d," indicating a directory. The `^` symbol is a regular expression anchor that matches the beginning of a line.

So, when you run the command `ls -l | grep "^\d"`, you're essentially listing the contents of the current directory in long format and then filtering out only the lines that represent directories. This can be useful to quickly see a list of directories within a directory without displaying files or other types of entries.

SA

du

```
sayak@ubuntu:~/harry$ ls -l
total 20
-rwxrw-r-- 1 sayak sayak      0 Aug 14 15:25 2.txt
-rw-rw-r-- 1 sayak sayak      0 Aug 14 15:38 56.txt
-rw-rw-r-- 1 sayak sayak    219 Aug 21 10:33 abc.txt
-rw-rw-r-- 1 sayak vboxsf     0 Aug 21 10:32 def.txt
-rw-rw-r-- 1 sayak sayak      0 Aug 14 18:31 harry.txt
-rw-rw-r-- 1 sayak sayak      0 Aug 14 18:31 HARRY.txt
-rw-r--r-- 1 root  root    797 Aug 21 19:21 myarchive2.zip
-rw-r--r-- 1 root  root   438 Aug 21 19:10 myarchive.zip
-rw-rw-r-- 1 sayak sayak     22 Aug 16 10:24 rohan.txt
drwxr--r-- 2 sayak sayak  4096 Aug 22 19:28 this
sayak@ubuntu:~/harry$ ls -l | grep "^\d"
drwxr--r-- 2 sayak sayak  4096 Aug 22 19:28 this
```

Disk space utilization.

The `du` command in Linux and Unix-like systems is used to estimate and display the disk space utilization of files and directories. It provides information about the sizes of files and directories in a specified location.

Here's the basic syntax of the `du` command:

bash

```
du [options] [file(s) or directory]
```

- **file(s) or directory** is the name of the file or directory you want to analyze. If not specified, it defaults to the current directory.

Commonly used options with the du command:

- **-h or --human-readable**: Display sizes in a human-readable format (e.g., KB, MB, GB).
- **-s or --summarize**: Display only the total size of the specified directory or file.
- **-c or --total**: Display a grand total at the end of the output.
- **-k, -m, -g, -t**: Specify the unit (KB, MB, GB, or TB) for the sizes.
- **-a or --all**: Display sizes of all files and directories, not just directories.
- **-L or --dereference**: Follow symbolic links and report the sizes of the target files.

For example, to display the disk space utilization of the current directory in human-readable format:

bash

du -h

To display the total size of a specific directory and its subdirectories:

bash

du -sh directory_name

To get a summary of the disk space used by a file and all the files and directories within it:

bash

**du -sh
file_name**

```
sayak@ubuntu:~/harry$ du -ah
0          ./def.txt
0          ./56.txt
0          ./HARRY.txt
4.0K      ./myarchive2.zip
4.0K      ./abc.txt
0          ./harry
0          ./harry.txt
4.0K      ./myarchive.zip
4.0K      ./this/hard_link.txt
4.0K      ./this/script2.sh
0          ./this/soft_link.txt
4.0K      ./this/3.csv
4.0K      ./this/4.txt
4.0K      ./this/5.txt
4.0K      ./this/myarchive.a
4.0K      ./this/2.txt
4.0K      ./this/1.txt.bz2
4.0K      ./this/script.sh
40K       ./this
0          ./2.txt
4.0K      ./rohan.txt
60K       .
```

The `du` command is commonly used to identify large files and directories that are consuming disk space. It's helpful for managing disk usage, finding storage inefficiencies, and identifying potential candidates for cleanup or archiving.

SA

top - Free space in memory and swap

The `top` command in Linux and Unix-like systems is a real-time system monitoring utility that provides information about various system resources, including CPU usage, memory usage, swap usage, and more. It displays an interactive, dynamic view of system activity and performance.

While `top` primarily focuses on displaying real-time CPU and memory statistics, it doesn't specifically provide information about free space in memory and swap. Instead, it shows how much of these resources are currently being used.

To use `top`:

1. Open your terminal.
2. Simply type `top` and press Enter.

The `top` display will provide information about CPU usage, memory usage, load averages, and more. The memory-related information includes:

- **VIRT**: The virtual memory size of a process.
- **RES**: The resident set size (portion of a process's memory that is held in RAM).
- **SHR**: The shared memory size of a process.
- **SWAP**: The amount of swap space used by a process.

To exit `top`, you can press the `q` key.

```

sayak@ubuntu:~/harry$ top
To      top - 20:01:34 up 6:32, 1 user, load average: 0.60, 0.42
Tasks: 230 total, 2 running, 228 sleeping, 0 stopped,
%Cpu(s): 0.6 us, 0.4 sy, 0.0 ni, 98.9 id, 0.0 wa, 0.0
MiB Mem : 4910.3 total, 247.0 free, 2966.9 used, 1
MiB Swap: 2680.0 total, 2679.2 free, 0.8 used. 1

          PID USER      PR  NI      VIRT      RES      SHR S %CPU
        1799 sayak     20   0  5190360  461712 146732 S  2.6
        3099 sayak     20   0  572920  54004  38820 S  0.7
        3343 sayak     20   0 2757844 266412  95252 S  0.7
        533 systemd+   20   0    14824   6784   6016 S  0.3
       2322 sayak     20   0  227516   3100   2816 S  0.3
       6642 sayak     20   0 3246572 562832 115144 S  0.3
         1 root      20   0  166720  11748   8164 S  0.0
         2 root      20   0      0      0      0 S  0.0
         3 root      0  -20      0      0      0 I  0.0
         4 root      0  -20      0      0      0 I  0.0
         5 root      0  -20      0      0      0 I  0.0
         6 root      0  -20      0      0      0 I  0.0
         8 root      0  -20      0      0      0 I  0.0
        10 root      0  -20      0      0      0 I  0.0
        11 root     20   0      0      0      0 I  0.0
        12 root     20   0      0      0      0 I  0.0
        13 root     20   0      0      0      0 I  0.0
        14 root     20   0      0      0      0 S  0.0
        15 root     20   0      0      0      0 R  0.0
        16 root     rt   0      0      0      0 S  0.0

```

monitor memory usage and swap space, you can focus on the "Mem" and "Swap" lines in the `top` display. These lines show the total memory and swap available, as well as the amount used and free. Keep in mind that "free" memory doesn't necessarily mean it's unused; the Linux kernel often uses available memory for caching and other purposes.

If you specifically want to know the free and used memory and swap space, you might consider using the `free` command. For example:

`bash`

`free -h`

This command will show the memory and swap statistics in human-readable format. It will give you a clear view of the total, used, free, and available memory and swap space on your system.

SA

meaning of swap

In the context of computer systems and operating systems, "swap" refers to a portion of the storage (usually a dedicated disk space) that is used as virtual memory to supplement the physical RAM (Random Access Memory) in a system. Swap space allows the operating system to effectively manage memory usage, especially when physical RAM is exhausted or when processes need more memory than is physically available.

23/08/2023

SA

paste - Join two files laterally

The `paste` command in Linux and Unix-like systems is used to combine the lines of two or more files side by side (laterally) in columns. It's particularly useful when you want to compare or display related data from multiple files.

Here's the basic syntax of the `paste` command:

bash

```
paste [options] file1 file2 [file3...]
```

- `file1`, `file2`, and `file3` are the names of the files you want to paste together.
- You can provide more than two files, and the `paste` command will combine them side by side.

For example, let's say you have two files named `names.txt` and `ages.txt` with the following content:

names.txt:

Alice

Bob

Charlie

ages.txt:

25

30

22

To join the lines of these two files laterally:

bash

paste names.txt ages.txt

The output will be:

Alice 25

Bob 30

Charlie 22

By default, the **paste** command uses a tab character as the delimiter between columns. You can use the **-d** option to specify a different delimiter. For example, to use a space as the delimiter:

bash

paste -d ' ' names.txt ages.txt

Alice 25

Bob 30

Charlie 22

```
sayak@ubuntu:~/harry/this$ paste -d '# ' 1.txt 2.txt
sfgtrht#jfkf
dfgfkj#l2wo395
gldis#wpwe
flkrs#r,rkrtrpl20935437560-
38509sjkfn#5867986587543056954--60=3529
0375hsnayenrh#-3945609506
edlkeiue92432@!#@#(8900#-3956-9654/-/-+
cd ..
#9789565
#-**/-*8+9
#
```

The `paste` command is commonly used for combining related data for easy comparison, generating reports, or creating tabular output. It's a simple yet effective way to merge content from multiple files into a single output.

SA

kill - Kill job.

The `kill` command in Linux and Unix-like systems is used to send signals to processes or jobs. It's commonly used to terminate processes or manage their behavior by sending specific signals.

Here's the basic syntax of the `kill` command:

bash

```
kill [options] PID
```

- PID is the Process ID of the process you want to send a signal to.

Commonly used options with the `kill` command:

- `-l` or `--list`: List available signal names.
- `-s signal` or `--signal=signal`: Specify the signal to send. Signal can be a signal name (e.g., `TERM`) or signal number (e.g., `15` for `SIGTERM`).
- `-SIG signal`: Same as `-s signal`.
- `-p` or `--pid`: Send a signal to a specific process ID without killing it.

For example, to terminate a process with a specific PID:

bash

```
kill PID
```

Or you can use signal names:

bash

```
kill -s TERM PID
```

The default signal sent by `kill` is `SIGTERM` (15), which typically requests the process to terminate gracefully. However, some processes might handle `SIGTERM` differently. If a process doesn't respond to `SIGTERM`, you can use the more forceful `SIGKILL` (9) signal:

bash

```
kill -s KILL PID
```

You can also use the numeric signal values directly, such as `kill -9 PID`.

The `kill` command is useful for managing processes, especially when you want to stop or modify their behavior. It's important to use caution when sending signals, especially `SIGKILL`, as forcefully terminating a process might lead to data corruption or incomplete operations.

```
sayak@ubuntu:~/harry>this$ pgrep thunderbird  
20110  
SA sayak@ubuntu:~/harry>this$ kill 20110  
sayak@ubuntu:~/harry>this$ pgrep thunderbird  
20436  
sayak@ubuntu:~/harry>this$ kill -9 20436  
sayak@ubuntu:~/harry>this$ █
```

Command to show pid of a process

To show the Process ID (PID) of a specific process, you can use the `pgrep` or `pidof` command. These commands allow you to search for processes by their name and provide their corresponding PID.

1. Using pgrep command:

The `pgrep` command is used to search for processes by their name and return their PIDs.

bash

```
pgrep process_name
```

For example, to find the PID of a process named `firefox`:

bash

```
pgrep firefox
```

This will return the PID of the `firefox` process if it's currently running.

2. Using pidof command:

The `pidof` command is used to find the PID of a process based on its name.

```
bash
```

```
pidof process_name
```

For example, to find the PID of a process named `chromium`:

```
bash
```

```
pidof chromium
```

```
sayak@ubuntu:~/harry>this$ pgrep thunderbird  
20664  
sayak@ubuntu:~/harry>this$ pidof thunderbird  
20748 20664
```

Both
and

pgrep
pidof

can be useful for quickly identifying the PID of a process when you know its name. If multiple processes with the same name are running, these commands will return all their PIDs separated by spaces.

SA

`grep -E, egrep, fgrep` - Lines containing one or more of multiple patterns.

The `grep` command in Linux and Unix-like systems provides different options for searching and matching patterns in text files. Among these options are `-E` (or `egrep`) and `fgrep` (or `grep -F`), which are used for more advanced pattern matching.

1. grep -E or egrep: The `-E` option (or using the `egrep` command) enables extended regular expressions, allowing you to use more complex pattern-matching syntax. This includes the use of metacharacters like `|` (OR), `()` (grouping), and others.

```
bash
```

```
grep -E pattern file(s)
```

For example, to search for lines containing either "apple" or "banana":

```
bash
```

- **grep -E "apple|banana" fruits.txt**
- **fgrep or grep -F:** The **fgrep** command (or using **grep -F**) is used for "fixed-string" searches. It treats the search pattern as a literal string rather than a regular expression. This can be useful when you want to search for exact strings without considering special characters as metacharacters.

```
bash
```

```
fgrep pattern file(s)
```

For example, to search for the exact string "Hello, World!" in a file:

```
bash
```

2. **fgrep "Hello, World!" myfile.txt**

Both **grep -E** and **fgrep** (or **grep -F**) are powerful tools for pattern matching, and the choice between them depends on the complexity of your search pattern and whether you need regular expression capabilities.

Regular expressions provide more flexibility, while fixed-string matching ensures that special characters are treated as literals.

Note that the basic **grep** command without any options performs standard pattern matching using basic regular expressions.

SA

sort - in

Lines
ASCII

```
sayak@ubuntu:~/harry/this$ cat 1.txt
Indian Institute
of technology #&! iit
Kharagpur
Directors name:
VK Tiwari
Department 344
of Civil Engineering
Hydraulics $#2389021
and & Water~~~~
resources
Enginnering Sayak
Karmakar
sayak      IIT
234567890
StudenT
Ex student pHD
PhD
Doctor of Philosophy

sayak@ubuntu:~/harry/this$ egrep "iit|3" 1.txt
of technology #&! iit
Department 344
Hydraulics $#2389021
234567890
sayak@ubuntu:~/harry/this$ fgrep "ng Sayak" 1.txt
Enginnering Sayak
sayak@ubuntu:~/harry/this$
```

collating sequence

The **sort** command in Linux and Unix-like systems is used to sort lines of text files or standard input in ascending or descending order based on the ASCII collating sequence. It can be used to alphabetically or numerically sort lines of text, making it useful for organizing and manipulating data.

Here's the basic syntax of the **sort** command:

bash

sort [options] [file(s)]

- **file(s)** is the name of the file(s) you want to sort. If not specified, **sort** reads from standard input.

Commonly used options with the **sort** command:

- **-r or --reverse:** Sort in descending order.
- **-n or --numeric-sort:** Sort numerically instead of lexicographically.
- **-k field or --key=field:** Specify a field to use as the sort key.
- **-t separator or --field-separator=separator:** Specify a custom field separator.
- **-u or --unique:** Remove duplicate lines before sorting.
- **-o output_file:** Write output to a file instead of standard output.

For example, to sort the lines of a file named `names.txt` in ascending order:

`bash`

sort names.txt

To sort numerically:

`bash`

sort -n numbers.txt

To sort lines in descending order:

`bash`

sort -r data.txt

To sort based on a specific field, considering a comma as the field separator:

`bash`

```
sayak@ubuntu:~/harry>this$ sort 1.txt
234567890
and & Water~~~~
answer
Answer
Department 344
Directors name:
Doctor of Philosophy
Enginnering Sayak
Ex student pHD
Hydraulics #2389021
Indian Institute
Karmakar
KARMAKAR
Kharagpur
Kharagpur
of Civil Engineering
of technology #&! iit
PhD
resources
sayak IIT
StudenT
VK Tiwari
& YOU tube
```

```
sayak@ubuntu:~/harry>this$ cat 6.csv
Sayak, deep, asis, Hari
10,15,7,13
kolkata, raiganj, nadia, bankura
sayak@ubuntu:~/harry>this$ sort -t ',' -k 4 6.csv
10,15,7,13
kolkata, raiganj, nadia, bankura
Sayak, deep, asis, Hari
sayak@ubuntu:~/harry>this$ sort -t ',' -k 3 6.csv
10,15,7,13
Sayak, deep, asis, Hari
kolkata, raiganj, nadia, bankura
sayak@ubuntu:~/harry>this$
```

sort -t ',' -k 2 data.csv

here, '-k 3' means that the sorting will

be based on the values found in the 3rd column of the CSV file.

The `sort` command is flexible and widely used for a variety of sorting tasks. It's particularly useful when dealing with large datasets, log files, and any situation where you need to arrange lines of text in a specific order.

SA

sort -n - Lines in numeric sequence.

The `sort -n` command is used to sort lines of text files in ascending order based on a numeric sequence. This means that the lines are sorted as if the content on each line is a number, not a string. The `-n` option tells the `sort` command to perform a numeric sort.

Here's how the command works:

bash

```
sort -n file(s)
```

- `file(s)` is the name of the file(s) you want to sort.

For example, consider a file named `numbers.txt` with the following content:

100

5

42

8

Using the command `sort -n numbers.txt` will sort the lines based on numeric values:

5

8

42

100

In this example, the lines are sorted in ascending order numerically.

Keep in mind that the `-n` option is useful when you want to sort lines containing numeric data. If you were to use the regular `sort` command without the `-n` option, the lines would be sorted in lexicographical order (as strings) rather than by their numeric values.

SA

tac, rev- Lines in reverse order.

The `tail` command, by default, is used to display the last lines of a file. The basic syntax of the `tail` command is as follows:

bash

```
tail [options] file
```

- `file` is the name of the file you want to display.

Commonly used options with the `tail` command:

- `-n num` or `--lines=num`: Display the last `num` lines of the file.
For example, `tail -n 10 file.txt` displays the last 10 lines.
- `-f` or `--follow`: Output appended data as the file grows.
- `-c num` or `--bytes=num`: Display the last `num` bytes of the file.

If you want to reverse the order of lines in a file, you can use the `tac` command (reverse of `cat`) or `rev` command (reverse each line):

Using `tac`:

bash

```
tac file.txt
```

Using rev:

bash

```
rev file.txt
```

Please note that the availability of the `tac` and `rev` commands depends on your specific Linux distribution. If they are not available on your system, you might need to use other methods or commands to achieve the desired result.

```
sayak@ubuntu:~/harry/this$ cat 2.txt
Sayak
Harry Sujoy 456
12 kamal
%rounak
harsh 89
57 ananda
-&7 nayan
sayak@ubuntu:~/harry/this$ tac 2.txt
-&7 nayan
57 ananda
harsh 89
%rounak
12 kamal
Harry Sujoy 456
Sayak
sayak@ubuntu:~/harry/this$ rev 2.txt
kayaS
654 yojus yrraH
lamak 21
kanuor%
98 hsrah
adnana 75
nayan 7&-
sayak@ubuntu:~/harry/this$ tail -n 3 2.txt
harsh 89
57 ananda
-&7 nayan
sayak@ubuntu:~/harry/this$ tail -c num 2.txt
tail: invalid number of bytes: 'num'
sayak@ubuntu:~/harry/this$ tail -c 8 2.txt
7 nayan
```

SA

sort -f - Lines sorted ignoring case.

The `sort -f` command is used to sort lines of text files while ignoring the case of letters. This means that uppercase and lowercase letters are treated as equivalent during the sorting process. The `-f` option tells the `sort` command to perform a case-insensitive sort.

Here's how the command works:

bash

```
sort -f file(s)
```

- file(s) is the name of the file(s) you want to sort.

For example, consider a file named names.txt with the following content:

apple

Banana

cherry

banana

Using the command

sort -f names.txt

will sort the lines while ignoring case:

apple

banana

Banana

cherry

In this example, the lines are sorted in ascending order while ignoring the case of the letters.

The -f option is particularly useful when you want to sort lines with the same letters but different cases (uppercase and lowercase) as if they were the same. This is commonly used when sorting strings that represent names, words, or text data where the case should not affect the sort order.

```
sayak@ubuntu:~/harry>this$ sort -f 1.txt
```

```
234567890
and & Water~~~~
answer
Answer
Department 344
Directors name:
Doctor of Philosophy
Enginnering Sayak
Ex student pHD
Hydraulics $#2389021
Indian Institute
Karmakar
KARMAKAR
Kharagpur
Kharagpur
of Civil Engineering
of technology #&! iit
PhD
resources
sayak IIT
StudenT
VK Tiwari
& YOU tube
```

```
sayak@ubuntu:~/harry>this$ sort 1.txt
```

```
234567890
and & Water~~~~
answer
Answer
Department 344
Directors name:
Doctor of Philosophy
```

uniq -d - Lines that are repeated

The `uniq -d` command is used to display lines that are repeated (duplicates) in a sorted text file. The `-d` option tells the `uniq` command to show only the lines that appear more than once consecutively.

Here's how the command works:

bash

`uniq -d file`

- `file` is the name of the sorted text file you want to analyze for duplicates.

For example, consider a file named `data.txt` with the following content:

apple

apple

banana

cherry

cherry

cherry

banana

Using the command `uniq -d` on `data.txt` will display the lines that are repeated:

apple

cherry

In this example, both "apple" and "cherry" appear more than once consecutively, so they are displayed by the `uniq -d` command.

```
sayak@ubuntu:~/harry>this$ cat 1.txt
Indian Institute
of technology #&! iit
Directors name:
VK Tiwari
Department 344
Hydraulics $#2389021
and & Water~~~~
resources
Enginnering Sayak
sayak      IIT
234567890

Student
[ student PhD
PhD
Doctor of Philosophy
Kharagpur
Kharagpur
KARMAKAR
answer
Answer
& YOU tube
Kharagpur
Enginnering Sayak
Enginnering Sayak
sayak@ubuntu:~/harry>this$ uniq -d 1.txt
Kharagpur
Enginnering Sayak
sayak@ubuntu:~/harry>this$
```

Please note that the `uniq` command expects the input to be sorted, and it only detects duplicates that appear in consecutive lines. If you want to find duplicates that are not necessarily consecutive, you might need to use other tools or techniques to achieve that, such as combining `sort` and `uniq`, or using scripting languages like `awk` or `Python`.

SA

uniq -u - Lines that occur only once

The `uniq -u` command is used to display lines that occur only once (unique lines) in a sorted text file. The `-u` option tells the `uniq` command to show only the lines that are not duplicated.

Here's how the command works:

bash

```
uniq -u file
```

- `file` is the name of the sorted text file you want to analyze for unique lines.

For example, consider a file named `data.txt` with the following content:

apple

banana

cherry

apple

grape

banana

Using the command
uniq -u data.txt

display the lines that
only once:

cherry
grape

```
sayak@ubuntu:~/harry/this$ cat 5.txt
Rahul
Rahul
Sayak
Sagar
Sagar

sayak@ubuntu:~/harry/this$ uniq -u 5.txt
Sayak
```

will
occur

In this example, "cherry" and "grape" appear only once in the file, so they are displayed by the **uniq -u** command.

It's important to note that the **uniq** command expects the input to be sorted, and it identifies unique lines based on their **exact match**. If you want to find unique lines in an unsorted file or if the lines have minor differences, you might need to preprocess the data or use other tools for more advanced processing.

SA

ssh - Log in to a remote machine.

The **ssh** command in Linux and Unix-like systems is used to establish a secure shell (SSH) connection to a remote machine. SSH is a cryptographic network protocol that allows you to securely log in to and manage remote servers and systems over an encrypted connection.

Here's the basic syntax of the **ssh** command:

bash

ssh [options] [user@]hostname

- **user** is the username you want to log in with on the remote machine (optional).
- **hostname** is the hostname or IP address of the remote machine you want to connect to.

For example, to log in to a remote machine with the username **john** and the hostname **example.com**, you would use:

bash

ssh john@example.com

?????sayak@ubuntu:~/harry>this\$ ssh sayak@10.0.2.15
ssh: connect to host 10.0.2.15 port 22: Connection refused
sayak@ubuntu:~/harry>this\$

Not working

If you don't specify a username, your local username will be used by default. Additionally, if the remote machine uses a non-standard SSH port (not the default 22), you can use the -p option followed by the port number:

bash

ssh -p 2222 john@example.com

After entering the ssh command, you'll be prompted to enter the password for the remote user (unless you're using key-based authentication). Once authenticated, you'll have a secure shell session on the remote machine, allowing you to execute commands, transfer files, and perform various administrative tasks.

SSH is widely used for remote management, secure file transfers, and secure communication between machines. It's an essential tool for system administrators and developers working with remote servers.

SA

make - Maintain a group of C programs.??

The make command in Linux and Unix-like systems is a utility used for building and maintaining groups of programs, typically written in the C programming language. It's used to automate the process of compiling and linking source code files to create executable programs.

make uses a "Makefile" to define a set of rules that specify how source code files are compiled and linked to produce the final executable

program. The Makefile contains information about dependencies between source files, compilation commands, and other build-related instructions.

Here's a basic overview of how the `make` command works:

1. **Create a Makefile:** You need to create a Makefile in the same directory as your source code files. The Makefile specifies the rules for building your program.
2. **Define Rules:** In the Makefile, you define rules that specify how to build the executable program. A rule typically includes the target (the name of the program or an intermediate file), its dependencies (source code files), and the commands to compile and link the files.
3. **Run make:** Once the Makefile is set up, you can use the `make` command to initiate the build process. `make` will read the Makefile and execute the rules to compile and link the program.

For example, consider a simple C program named `my_program.c` that needs to be compiled and linked to create an executable named `my_program`. You would create a Makefile like this:

```
make
```

```
my_program: my_program.c  
    gcc -o my_program my_program.c
```

In this Makefile, the rule specifies that the target `my_program` depends on `my_program.c`. The command `gcc -o my_program my_program.c` is used to compile and link the source code to create the executable.

To build the program, you would navigate to the directory containing the Makefile and the source code file, and then run:

```
bash
```

```
make
```

The `make` command will read the Makefile, identify the rules, and execute the necessary commands to build the executable program.

`make` is a powerful tool for managing the build process of C programs and other software projects. It automates the process of building complex projects with multiple source files, libraries, and dependencies.

SA

tail -f - Monitor growth of a file.

The `tail -f` command is used to monitor the growth of a file in real time. It displays the last few lines of a file and continues to update the display as new lines are added to the file. This is particularly useful for monitoring log files or any files that are actively being written to.

Here's how the command works:

bash

`tail -f file`

- `file` is the name of the file you want to monitor.

For example, if you want to monitor the contents of a log file named `app.log`:

bash

`tail -f app.log`

```
sayak@ubuntu:~/harry/this$ tail -f 1.txt
Answer
& YOU tube
Kharagpur
Enginnering Sayak
Enginnering Sayak
we are working
reply to me

tail: 1.txt: file truncated
Indian Institute
of technology #&! iit
Directors name:
VK Tiwari
Department 344
Hydraulics $#2389021
and & Water~~~~
resources
Enginnering Sayak
sayak      IIT
234567890

Student
[ student pHD
PhD
Doctor of Philosophy
Kharagpur
Kharagpur
KARMAKAR
answer
Answer
& YOU tube
Kharagpur
Enginnering Sayak
Enginnering Sayak
we are working
reply to me
sayan
```

As new lines are added to the app.log file, they will be displayed in your terminal in real time. This is often used to keep an eye on changes to log files, error messages, or any other files that are being updated continuously.

To stop monitoring, you can press **Ctrl + C** in the terminal where the **tail -f** command is running. This will interrupt the command and return you to the command prompt.

SA

mv - Move files to another directory.

The **mv** command in Linux and Unix-like systems is used to move files or directories from one location to another. It can also be used to rename files or directories. The basic syntax of the **mv** command is as follows:

bash

```
mv [options]
source
destination
```

- **source** is the name of the file or directory you want to move.
- **destination** is the target location where you want to move the source.

For example, to move a file named **file.txt** from the current directory to another directory called **destination**:

```
sayak@ubuntu:~/harry/that$ ls
sayak@ubuntu:~/harry/that$ cd ..
sayak@ubuntu:~/harry$ mv 7.txt that
sayak@ubuntu:~/harry$ ls -r
this      myarchive.zip   harry.txt  56.txt
that      myarchive2.zip  def.txt    2.txt
rohan.txt HARRY.txt      abc.txt
sayak@ubuntu:~/harry$ cd that
sayak@ubuntu:~/harry/that$ ls
7.txt
sayak@ubuntu:~/harry/that$ mv 5.txt that/5new.txt
mv: cannot stat '5.txt': No such file or directory
sayak@ubuntu:~/harry/that$ ls
7.txt
sayak@ubuntu:~/harry/that$ cd ..
sayak@ubuntu:~/harry$ mv 2.txt that
sayak@ubuntu:~/harry$ cd /that
bash: cd: /that: No such file or directory
sayak@ubuntu:~/harry$ cd /home/sayak/harry/that/
sayak@ubuntu:~/harry/that$ ls
2.txt  7.txt
sayak@ubuntu:~/harry/that$ cd ..
sayak@ubuntu:~/harry$ mv 56.txt that/56.2.txt
sayak@ubuntu:~/harry$ cd /home/sayak/harry/that/
sayak@ubuntu:~/harry/that$ ls
2.txt  56.2.txt  7.txt
sayak@ubuntu:~/harry/that$ cd ..
sayak@ubuntu:~/harry$ mv abc.txt def.txt that/
sayak@ubuntu:~/harry$ cd /home/sayak/harry/that/
sayak@ubuntu:~/harry/that$ ls
2.txt  56.2.txt  7.txt  abc.txt  def.txt
sayak@ubuntu:~/harry/that$ cd ..
sayak@ubuntu:~/harry$ ls
harry.txt  myarchive2.zip  rohan.txt  this
HARRY.txt  myarchive.zip   that
sayak@ubuntu:~/harry$ █
```

```
bash
```

```
mv file.txt destination/
```

If you want to rename the file while moving, you can specify the new name in the destination:

```
bash
```

```
mv file.txt destination/newfile.txt
```

To move a directory and its contents, you can do:

```
bash
```

```
mv directory_name destination/
```

You can also move multiple files at once:

```
bash
```

```
mv file1.txt file2.txt destination/
```

The `mv` command can also be used to overwrite existing files, but it will ask for confirmation before overwriting. If you want to suppress the confirmation prompt and overwrite files without asking, you can use the `-f` (force) option:

```
bash
```

```
mv -f source destination
```

Please use the `mv` command with caution, especially when using the `-f` option, as it can lead to data loss if not used carefully. Always double-check your source and destination paths before executing the command.

SA

bg - Move job to the background.

The `bg` command in Linux and Unix-like systems is used to move a suspended job to the background, allowing it to continue running while

you work on other tasks in the terminal. This is particularly useful when you have a job that was paused using the **Ctrl + Z** key combination and you want to resume it in the background.

Here's how the **bg** command works:

1. First, you need to suspend a job using the **Ctrl + Z** key combination. This will pause the currently running foreground job and return control to the terminal.
2. Once the job is suspended, you can use the **bg** command to move the suspended job to the background.

The basic syntax of the **bg** command is:

bash

bg [job_spec]

- **job_spec** is the job identifier or job number of the suspended job. If not specified, the most recently suspended job will be moved to the background.

For example, let's say you have a process running in the foreground, and you've paused it using **Ctrl + Z**:

csharp

\$ my_command

^Z

**[1]+ Stopped
my_command**

In this case, the **[1]+** indicates that the job number is 1.

```
sayak@ubuntu:~/harry>this$ tail -f 1.txt
Answer
& YOU tube
Kharagpur
Engineering Sayak
Engineering Sayak
we are working
reply to me
sayan

^Z
[1]+  Stopped                  tail -f 1.txt
sayak@ubuntu:~/harry>this$ bg %1
[1]+ tail -f 1.txt &
```

To move the suspended job to the background:

bash

bg %1

Or, you can simply use **bg** without specifying the job number if it's the most recently suspended job:

bash

bg

After using the **bg** command, the job will continue running in the background, and you'll see a message indicating that the job has been resumed in the background.

Keep in mind that some jobs might not work correctly in the background, especially those that require user interaction or that were designed to run in the foreground. It's recommended to test the behavior of the job in the background to ensure it works as expected.

SA

fg - Move job to the foreground.

The **fg** command in Linux and Unix-like systems is used to move a suspended job from the background to the foreground, allowing you to interact with and monitor the job's progress. This command is the opposite of the **bg** command, which moves a job from the foreground to the background.

Here's how the **fg** command works:

1. First, you need to have a job running in the background. You can achieve this by running a command followed by an ampersand (&) at the end, which will start the command in the background.

For example:

bash

1. my_command &

2. You can then use the `fg` command to bring a background job to the foreground.

The basic syntax of the `fg` command is:

bash

```
fg [job_spec]
```

- `job_spec` is the job identifier or job number of the background job you want to bring the foreground. If not specified, the most recently backgrounded job will be moved to foreground.

For example, let's say you have a process running in the background:

bash

```
$ my_command &  
[1] 12345
```

In this case, the job number is 1, and the process ID is 12345.

To move the background job with job number 1 to foreground:

bash

```
fg %1
```

```
sayak@ubuntu:~/harry>this$ tail -f 1.txt  
Answer  
& YOU tube  
Kharagpur  
Enginnering Sayak  
Enginnering Sayak  
we are working  
reply to me  
sayan  
  
^Z  
[1]+ Stopped tail -f 1.txt  
sayak@ubuntu:~/harry>this$ bg %1  
[1]+ tail -f 1.txt &  
sayak@ubuntu:~/harry>this$ ^C  
sayak@ubuntu:~/harry>this$ tail -f 1.txt  
Answer  
& YOU tube  
Kharagpur  
Enginnering Sayak  
Enginnering Sayak  
we are working  
reply to me  
sayan  
  
^Z  
[2]+ Stopped tail -f 1.txt  
sayak@ubuntu:~/harry>this$ bg %2  
[2]+ tail -f 1.txt &  
sayak@ubuntu:~/harry>this$ fg %2  
tail -f 1.txt  
^C  
sayak@ubuntu:~/harry>this$ fg %1  
tail -f 1.txt  
^C  
sayak@ubuntu:~/harry>this$ fg %1  
bash: fg: %1: no such job  
sayak@ubuntu:~/harry>this$ █
```

Or, if it's the most recent background job, you can simply use:

bash

fg

Here, we first run two processes (job number 1 and 2). Then we paused them (using **crtl+z**) and run them in background . Then, we take them back in the foreground and finally killed them using **crtl+c**. Finally, we can see that there is no job anymore.

After using the **fg** command, you can interact with the job like any other process. You can use standard keyboard interactions to control the job or provide input to it.

Keep in mind that some jobs might not behave correctly when moved to the foreground, especially if they were designed to run exclusively in the background. Additionally, interactive jobs might require your attention and input when brought to the foreground.

SA

sed - Multiple segments from a file.

To work with multiple segments from a file using the **sed** command, you can use ranges to specify the lines you want to operate on. Ranges allow you to specify a starting line and an ending line within which you want to perform a particular operation. Here's how you can use **sed** to work with multiple segments from a file:

bash

```
sed 'start_line,end_line operation' file.txt
```

- **start_line** is the line number or pattern where the range should start.
- **end_line** is the line number or pattern where the range should end.
- **operation** is the **sed** command you want to apply to the lines within the specified range.

For example, let's say you have a file named **data.txt** with the following content:

mathematica

Line 1

Line 2

Line 3

Line 4

Line 5

If you want to print lines 2 to 4 from the file, you can use:

bash

```
sed '2,4p'  
data.txt
```

This will output:

mathematica

Line 2

Line 3

Line 4

You can also use patterns in the range specification.

For instance, if you want to print lines from the line containing "Line 2" to the line containing "Line 4":

bash

```
sed '/Line 2/,/Line 4/p' data.txt
```

This will produce the same output as above.

Similarly, you can perform other sed operations within the specified range, such as deleting lines or substituting text.

```
sayak@ubuntu:~/harry/this$ cat 2.txt  
Sayak  
Harry Sujoy 456  
12 kamal  
%rounak  
harsh 89  
57 ananda  
-&7 nayan  
Sagardeep sharma  
Nayan kumar  
sayak@ubuntu:~/harry/this$ sed '2,4p' 2.txt  
Sayak  
Harry Sujoy 456  
Harry Sujoy 456  
12 kamal  
12 kamal  
%rounak  
%rounak  
harsh 89  
57 ananda  
-&7 nayan  
Sagardeep sharma  
Nayan kumar  
sayak@ubuntu:~/harry/this$
```

Note that, specified lines under ,sed' command are shown twice only.

Remember that `sed` ranges are inclusive, meaning both the start and end lines are included in the operation. Also, `sed` operates line by line, so if you're using patterns, they will match against each line individually.

SA

hostname - Name of the local host.

The `hostname` command in Linux and Unix-like systems is used to display or set the name of the local host (the machine you are currently working on). It provides information about the system's hostname, which is a unique name used to identify the system within a network.

Here's how the `hostname` command can be used:

1. Display Hostname:

Simply running the `hostname` command without any options will display the current hostname of your system:

bash

- **hostname**
- **Set Hostname:**

```
sayak@ubuntu:~/harry/this$ hostname
ubuntu
sayak@ubuntu:~/harry/this$ su root
Password:
root@ubuntu:/home/sayak/harry/this# hostname
ubuntu
```

You can also use the `hostname` command to change the system's hostname. However, this change will only be temporary for the current session. To change the hostname permanently, you need to modify configuration files. To set a temporary hostname:

bash

2. **sudo hostname newhostname**

Replace `newhostname` with the desired hostname.

Please note that changing the hostname might have implications for networking and system configuration, so be cautious and follow appropriate procedures if you're making changes to a production system.

uname - Operating system name.

The `uname` command in Linux and Unix-like systems is used to retrieve various system information, including the name of the operating system. It provides details about the system's kernel, machine architecture, and other system-related information.

To get the operating system name using the `uname` command, you can use the `-s` or `--kernel-name` option. Here's how you can do it:

bash

```
uname -s
```

Alternatively, you can also use the `-o` or `--operating-system` option to achieve the same result:

bash

```
uname -o
```

Running either of these commands will output the name of the operating system, which could be something like "Linux," "GNU/Linux," "FreeBSD," "Darwin," or other similar names depending on the underlying operating system.

For example, if you're using a Linux-based system, running `uname -s` might output:

Linux

Keep in mind that the exact output will depend on the specific operating system you're using.

```
sayak@ubuntu:~/harry>this$ uname -s
Linux
sayak@ubuntu:~/harry>this$ uname -o
GNU/Linux
```

GNU

It seems like you mentioned "GNU." GNU is an operating system and an extensive collection of free software, including various utilities, libraries, and tools. The GNU Project was initiated by Richard Stallman in the early 1980s with the goal of creating a free and open-source Unix-like operating system.

Many Unix-like systems, including Linux distributions, use components from the GNU Project to form complete operating systems. Commonly, these systems are referred to as "GNU/Linux" distributions. The combination of the Linux kernel with the GNU utilities and software forms a complete and functional operating system.

So, when you see "GNU" in the context of operating systems, it often refers to the GNU Project and its associated software that contributes to the functionality of various operating systems, particularly in conjunction with the Linux kernel.

SA

uname -r - Operating system release

The `uname -r` command in Linux and Unix-like systems is used to retrieve the release version of the operating system's kernel. It provides information about the specific version of the kernel that is currently running on the system.

Running the following command:

`bash`

`uname -r`

```
sayak@ubuntu:~/harry>this$ uname -r  
6.2.0-26-generic
```

Will output the release version of the kernel. The format of the output might vary slightly depending on the distribution and version, but it typically follows a pattern like X.Y.Z, where:

- X represents the major version of the kernel.
- Y represents the minor version of the kernel.
- Z represents the patch level of the kernel.

For example, if the output is:

5.4.0-77-generic

This indicates that the running kernel has a major version of 5, a minor version of 4, and a patch level of 0. The additional text (- 77 - generic) might be distribution-specific and can provide additional information about the kernel's configuration or build.

Keep in mind that the output of `uname -r` provides information about the kernel version, **not necessarily the entire operating system version**. Other commands or tools might be needed to obtain information about the full operating system release.

SA

export - Pass variable value to sub-shell.

The `export` command in Linux and Unix-like systems is used to create environment variables or pass variable values to subshells. Environment variables are variables that are available to all processes and commands run within the shell session or its child processes.

When you use the `export` command followed by a variable name, you're making that variable available to any subsequently spawned subshells. This is particularly useful when you want to share the value of a variable with commands or scripts that are executed in a subshell.

Here's how you can use the `export` command:

bash

export VARIABLE_NAME=value

- `VARIABLE_NAME` is the name of the variable you want to export.
- `value` is the value you want to assign to the variable.

For example, let's say you have a variable named `MY_VARIABLE` and you want to pass its value to a subshell:

bash

```
MY_VARIABLE="Hello from the main shell"
```

```
export MY_VARIABLE
```

Now, if you start a subshell by typing `bash`, any commands run within that subshell will have access to the exported variable:

bash

```
$ bash
```

```
$ echo $MY_VARIABLE
```

```
Hello from the main shell
```

```
sayak@ubuntu:~/harry>this$ MY_VARIABLE="Hello from the main shell"  
The sayak@ubuntu:~/harry>this$ export MY_VARIABLE  
sayak@ubuntu:~/harry>this$ bash  
sayak@ubuntu:~/harry>this$ echo $MY_VARIABLE  
Hello from the main shell
```

The `export` command is often used when you want to share variables with scripts, functions, or other shells invoked from the current shell. Keep in mind that environment variables created using `export` are only available to child processes and not to the parent shell or other unrelated shells.

If you don't use the `export` command to make a variable global, it will only be available in the current shell session and any child processes spawned from it, but not in subshells.

SA

```
sayak@ubuntu:~/harry>this$ MY_VARIABLE2="Hello from the sub main shell"  
sayak@ubuntu:~/harry>this$ echo $MY_VARIABLE2  
Hello from the sub main shell
```

tr -s - Squeeze multiple spaces to a single space.

The `tr` command in Linux and Unix-like systems is used for character translation or replacement. The `-s` option in the `tr` command stands for "squeeze," and it is used to squeeze or collapse repeated occurrences of a specified character into a single occurrence. This is often used to convert multiple consecutive occurrences of a character into a single occurrence.

To squeeze multiple spaces into a single space using the `tr` command, you can do the following:

bash

```
tr -s ' ' < input.txt > output.txt
```

- `-s` tells `tr` to squeeze the repeated occurrences of the specified character.
- `' '` is the character you want to squeeze, which is a space in this case.
- `input.txt` is the input file containing the text you want to process.
- `output.txt` is the file where the processed text will be saved.

For example, let's say you have a file named `text.txt` with the following content:
csharp

This is an example.

```
sayak@ubuntu:~/harry/this$ cat 5.txt
A cleverrr brrrown fox
jumps overrr a lazy
dog

sayak@ubuntu:~/harry/this$ tr -s 'r' < 5.txt
A clever brown fox
jumps over a lazy
dog

sayak@ubuntu:~/harry/this$ tr -s 'r' < 5.txt > 6.txt
sayak@ubuntu:~/harry/this$ cat 6.txt
A clever brown fox
jumps over a lazy
dog

sayak@ubuntu:~/harry/this$ cat 5.txt
A cleverrr brrrown fox
jumps overrr a lazy
dog

sayak@ubuntu:~/harry/this$ echo "A cleverrr brrrown fox
jumps overrr a lazy
dog" | tr -s 'r'
A clever brown fox
jumps over a lazy
dog
```

Running the `tr` command mentioned above:

bash

```
tr -s ' ' < text.txt
```

Will output:

csharp

This is an example.

```
# But, change are displayed only in the terminal. Not saved in any file in this case like previous command.
```

As you can see, the multiple spaces have been squeezed into a single space.

You can also use the `tr` command without input and output redirection by providing text directly as an argument:

bash

```
echo "This is an example." | tr -s ' '
```

This command will produce the same output as shown above.

SA

expr - String length.

The `expr` command in Linux and Unix-like systems is used for evaluating expressions. While `expr` is often used for arithmetic operations, it can also be used to find the length of a string.

To find the length of a string using the `expr` command, you can use the following syntax:

bash

```
expr length "string"
```

Replace "string" with the actual string for which you want to determine the length.

For example, if you want to find the length of the string "Hello, World!":

bash

```
expr length "Hello, World!"
```

This will output:

13

The output is the length of the string "Hello, World!".

Keep in mind that while `expr` can be used to find the length of a string, there are more modern and flexible ways to achieve this using other commands, like `echo`, `wc`, or shell parameter expansion. For example, using `echo` and `wc`:

bash

```
echo -n "Hello, World!" | wc -c
```

Or using shell parameter expansion:

bash

```
string="Hello, World!"
```

```
echo ${#string}
```

Both
these

```
sayak@ubuntu:~/harry/this$ expr length "A clever brown fox jumps over a lazy dog"
46
sayak@ubuntu:~/harry/this$ echo -n "A clever brown fox jumps over a lazy dog" | wc -c
46
sayak@ubuntu:~/harry/this$ string="A clever brown fox jumps over a lazy dog"
sayak@ubuntu:~/harry/this$ echo ${#string}
46
sayak@ubuntu:~/harry/this$ wc -c 6.txt
42 6.txt
sayak@ubuntu:~/harry/this$ cat 6.txt
A clever brown fox
jumps over a lazy
dog
```

of

alternatives might be more intuitive and provide additional functionality beyond just string length calculation.

SA

WC -C

The **wc** command in Linux and Unix-like systems is used to count lines, words, and characters in a file or input provided through standard input. The **-c** option is used with the **wc** command to specifically count the number of characters in the input.

Here's the basic syntax:

bash

wc -c file.txt

- **-c** indicates that you want to count characters.
- **file.txt** is the name of the file you want to count characters in.

For example, if you have a file named **example.txt** with the following content:

Hello, world!

Running the command:

bash

wc -c example.txt

Will output:

14 example.txt

This indicates that the file **example.txt** contains 14 characters.

You can also use **wc -c** without specifying a file to count characters from standard input. This is useful when you want to input text directly:

bash

```
echo "Hello, world!" | wc -c
```

This will give you the same result of 14.

Keep in mind that `wc -c` counts each character, including spaces, tabs, and newline characters. If you want to count words or lines, you can use other options like `-w` for words and `-l` for lines with the `wc` command.

SA

tr - Substitute one character for another.

The `tr` command in Linux and Unix-like systems is used for character translation or replacement. It allows you to perform basic character-level substitutions in text. You can use the `tr` command to replace one character with another, delete certain characters, or squeeze repeated characters.

To substitute one character for another using the `tr` command, you can use the following syntax:

bash

```
tr 'old_char' 'new_char' < input.txt > output.txt
```

- `old_char` is the character you want to replace.
- `new_char` is the character you want to substitute it with.
- `input.txt` is the input file containing the text you want to process.
- `output.txt` is the file where the processed text will be saved.

For example, let's say you have a file named `data.txt` with the following content:

Hello, world!

If you want to replace all occurrences of the letter "o" with the letter "x," you can use the `tr` command like this:

bash

```
tr 'o' 'x' < data.txt
```

This will output:

Hellx, wxrld!

You can also use the `tr` command without input and output redirection by providing text directly as an argument:

bash

```
echo "Hello, world!" | tr 'o' 'x'
```

This command will produce the same output as shown above.

Keep in mind that `tr` performs character-level substitution, so it replaces individual characters and not substrings. It's suitable for simple character transformations but may not be suitable for more complex text manipulations that require pattern matching or regular expressions.

```
# sayak@ubuntu:~/harry>this$ cat 5.txt
that, this
A cleverrr brrrown fox
jumps overrr a lazy
dog

sayak@ubuntu:~/harry>this$ tr 'rrrr' 'r' < 5.txt
A cleverrr brrrown fox
jumps overrr a lazy
dog

sayak@ubuntu:~/harry>this$ tr 'oooo' 'o' < 5.txt
A cleveooo boooown fox
jumps oveooo a lazy
dog

sayak@ubuntu:~/harry>this$ tr 'oooo' 'o' < 5.txt > 7.txt
sayak@ubuntu:~/harry>this$ cat 7.txt
A cleveooo boooown fox
jumps oveooo a lazy
dog

sayak@ubuntu:~/harry>this$ ^[[200~echo "Hello, world!" | tr
'o' 'x'
echo: command not found
sayak@ubuntu:~/harry>this$ echo "Hello, world!" | tr 'o' 'x
'
Hellx, wxrld!
```

Note
by
rule,

multiple characters can not be converted to a single character.

```
# In  sayak@ubuntu:~/harry>this$ tr 'er' 'o' < 5.txt          the  
1st   A clooooo boooown fox  
      jumps ooooo a lazy  
      dog  
  
sayak@ubuntu:~/harry>this$ cat 5.txt  
A cleverrr brrrown fox  
jumps overrr a lazy  
dog  
  
sayak@ubuntu:~/harry>this$ tr 'er' 'xy' < 5.txt  
A clxvxyyy byyyown fox  
jumps ovxyyy a lazy  
dog
```

both 'e' and 'r' converted to 'o'. in the 2nd case, 'e' was converted to 'x' and 'r' was converted to 'y'.

SA

sed - Substitute one pattern for another.

The `sed` command (stream editor) in Linux and Unix-like systems is used for text manipulation and transformation. One of its common uses is for performing substitutions, where it replaces one pattern with another in a text file. This is particularly useful for making changes to a large number of occurrences of a specific pattern throughout a file.

To substitute one pattern for another using the `sed` command, you can use the following syntax:

bash

```
sed 's/old_pattern/new_pattern/g' input.txt >  
output.txt
```

- `old_pattern` is the pattern you want to find and replace.
- `new_pattern` is the pattern you want to substitute in place of the old pattern.
- `input.txt` is the input file containing the text you want to process.
- `output.txt` is the file where the processed text will be saved.

The `s` at the beginning of the `sed` command stands for "substitute," and the `g` at the end stands for "global," indicating that the substitution should be performed globally throughout the entire line.

For example, let's say you have a file named `data.txt` with the following content:

bash

Hello, world!

This is a test.

If you want to replace all occurrences of the word "Hello" with "Hi," you can use the `sed` command like this:

bash

sed 's/Hello/Hi/g' data.txt

This will output:

bash

Hi, world!

This is a test.

You can also use the `sed` command to perform in-place substitutions by using the `-i` option:

```
sayak@ubuntu:~/harry>this$ cat 6.txt
A clever brown fox
jumps over a lazy
dog

sayak@ubuntu:~/harry>this$ sed 's/er/ar/g' 6.txt
A clevar brown fox
jumps ovar a lazy
dog

sayak@ubuntu:~/harry>this$ cat 6.txt
A clever brown fox
jumps over a lazy
dog

sayak@ubuntu:~/harry>this$ sed -i 's/er/ar/g' 6.txt
sayak@ubuntu:~/harry>this$ cat 6.txt
A clevar brown fox
jumps ovar a lazy
dog
```

bash

sed -i 's/Hello/Hi/g' data.txt

This will modify the `data.txt` file directly.

Keep in mind that `sed` supports more complex substitutions using regular expressions and provides additional options for controlling the behavior of substitutions.

SA

su - Superuser from a nonprivileged account

The `su` command in Linux and Unix-like systems is used to switch to another user account, usually the superuser (root), from a nonprivileged account. This allows you to execute commands as if you were the target user, often used when you need administrative privileges to perform tasks that require elevated permissions.

Here's the basic syntax of the `su` command:

bash

`su [options] [username]`

- `options` are various command-line options you can use with the `su` command.
- `username` is the name of the user you want to switch to. If not specified, the default user is usually the superuser (root).

By default, when you run `su` without specifying a username, you switch to the superuser account. For example:

bash

`su`

This will prompt you to enter the superuser's password, and once authenticated, you'll be granted superuser privileges.

If you want to switch to a specific user other than the superuser, you can specify their username as an argument. For example, to switch to the user named "john":

bash

su john

This will prompt you for John's password, and if entered correctly, you'll switch to John's account.

To `sayak@ubuntu:~/harry>this$ su` open a
Password: new
`root@ubuntu:/home/sayak/harry>this# su sayak`
`sayak@ubuntu:~/harry>this$` shell

with the specified user's environment (similar to logging in as that user), you can use the `-` option:

bash

su - john

This option ensures that the target user's environment variables, home directory, and other settings are loaded.

Keep in mind that using `su` to switch to the superuser or other user accounts requires appropriate permissions. Additionally, using superuser privileges should be done carefully and only when necessary, as it can have significant security implications if used improperly.

SA

date - System date.

The `date` command in Linux and Unix-like systems is used to display or set the system date and time. When used without any arguments, the `date` command displays the current system date and time.

Here's the basic usage of the `date` command:

bash

date

Running this command will output the current system date and time in a format similar to the following:

yaml

```
Fri Aug 20 15:30:00 UTC 2023
```

The output includes the day of the week, the abbreviated month name, the day of the month, the time in hours, minutes, and seconds, the time zone (UTC in this case), and the year.

You can also format the output using the + option followed by a format string. For example, to display the date in a custom format like "YYYY-MM-DD HH:MM:SS":

bash

```
date "+%Y-%m-%d %H:%M:%S"
```

This will output something like:

yaml

```
2023-08-20 15:30:00
```

```
sayak@ubuntu:~/harry>this$ date
Friday 25 August 2023 04:51:25 PM IST
sayak@ubuntu:~/harry>this$ date "+%Y-%m-%d %H:%M:%S"
2023-08-25 16:52:51
sayak@ubuntu:~/harry>this$ date "+%Y/%m/%d %H:%M:%S"
2023/08/25 16:53:20
sayak@ubuntu:~/harry>this$ date "+%Y/%m/%d %H-%M-%S"
2023/08/25 16-53-32
```

date command requires appropriate permissions to set the system date and time. Typically, you need superuser (root) privileges to modify the date and time settings.

SA

top - System memory usage.

The `top` command in Linux and Unix-like systems is used to monitor system processes and resource utilization in real-time. It provides an interactive view of various system statistics, including CPU usage, memory usage, process information, and more.

To specifically view system memory usage using the `top` command, you can run the following command:

```
bash
```

```
top
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU
26321	sayak	20	0	2899760	414540	106864	S	16.3
6852	sayak	20	0	336288	43464	32512	S	3.0
1635	sayak	9	-11	2217740	22672	17040	S	2.0
2356	sayak	20	0	12.2g	470752	153512	S	2.0
533	systemd+	20	0	14824	6784	6016	S	0.3
1185	root	20	0	502228	5356	4244	S	0.3
1799	sayak	20	0	5237508	465736	115140	S	0.3
2310	sayak	20	0	227000	3100	2688	S	0.3
2322	sayak	20	0	227516	3100	2816	S	0.3
3099	sayak	20	0	577672	55504	35456	S	0.3
3343	sayak	20	0	2858196	303856	88980	S	0.3
26289	sayak	20	0	2413440	82268	61568	S	0.3
30475	sayak	20	0	21888	4096	3328	R	0.3
1	root	20	0	168024	12900	8164	S	0.0

When you run the `top` command, you'll see a dynamic display that updates every few seconds. By default, the processes are sorted by the percentage of CPU usage. To sort the processes by memory usage, you can press the **M key** while in the `top` interface.

- 1. PID:** Process ID. This is a unique identifier assigned to each running process in the system.
- 2. USER:** The username of the user who started the process.

3. **PR:** Priority. This column shows the priority of the process. Lower values indicate higher priority. In Linux, priority ranges from -20 (highest priority) to 19 (lowest priority).
4. **NI:** Nice Value. This value represents the "niceness" of a process, which is a measure of its priority. A lower nice value indicates a higher priority.
5. **VIRT:** Virtual Memory Usage. This column shows the virtual memory size used by the process. It includes both actual memory and swap space.
6. **RES:** Resident Memory Size. This column displays the amount of physical RAM used by the process. It represents the portion of the process's memory that is currently in RAM.
7. **SHR:** Shared Memory. This column shows the portion of the process's memory that is shared with other processes. It includes shared libraries and memory-mapped files.
8. **S:** Process Status. This column indicates the current status of the process. Common statuses include "R" for running, "S" for sleeping, "Z" for zombie, and more.
9. **%CPU:** Percentage of CPU Usage. This column displays the percentage of CPU time used by the process since the last update.

To exit the `top` command, you can press the `q` key.

Keep in mind that `top` is a powerful tool for monitoring system resource usage and understanding system behavior. It provides real-time insights into how the system's resources are being utilized by various processes.

SA

who -r, runlevel - System run level

The `who -r` and `runlevel` commands in Linux and Unix-like systems are used to display the current runlevel of the system. Runlevel refers to the operational state of the system, which determines the services and

daemons that are active or inactive. Runlevels are used to manage different system states, such as booting, maintenance, and shutdown.

Here's how you can use these commands:

1. **who -r Command:**

The **who -r** command provides information about the current runlevel and the last system reboot time.

bash

who -r

The output might look like this:

arduino

- **run-level 5 2021-08-15 14:30**
last=3

In this example, the current runlevel is 5, and the last system reboot occurred on August 15th, 2021, at 14:30.

```
sayak@ubuntu:~/harry>this$ who -r
                    run-level 5 2023-08-21 10:59
• sayak@ubuntu:~/harry>this$ runlevel
N 5
sayak@ubuntu:~/harry>this$
```

runlevel Command:

The **runlevel** command provides information about the current runlevel without additional details.

bash

runlevel

The output might look like this:

mathematica

- 2. **N 5**

In this example, the current runlevel is 5.

Runlevels can vary depending on the Linux distribution and the specific configuration of the system. Common runlevels include:

- 0: Halt (system shutdown)
- 1: Single-user mode (maintenance mode)
- 2: Multi-user mode without networking
- 3: Multi-user mode with networking
- 5: Multi-user mode with networking and GUI
- 6: Reboot

The output of `who -r` or `runlevel` will reflect the current runlevel, giving you insight into the operational state of the system.

SA

How to know process name under any PID?

To find the process name associated with a specific Process ID (PID) in Linux and Unix-like systems, you can use various commands such as `ps`, `top`, or `pgrep`. Here are a few methods you can use:

1. Using `ps` Command:

You can use the `ps` command with the `-p` option to display information about a specific PID. This will include the process name.

bash

```
ps -p <PID>
```

Replace `<PID>` with the actual Process ID you want to look up.

For example:

bash

- `ps -p 1234`

- **Using `top` Command:**

While running the `top` command, you can press the `C` key to toggle displaying the process command names along with their PIDs.

1. Run `top` in the terminal.
2. Press the `c` key. (Not working for me!??)

This will change the display to show the process names.

- **Using pgrep Command:**

The `pgrep` command allows you to search for processes by name or other attributes and returns the PIDs of matching processes.

bash

```
pgrep -l -P <PID> (Not working for me!??)
```

Replace `<PID>` with the parent PID under which you want to search for processes.

For example:

bash

```
3. pgrep -l -P 1234
```

The `-l` option in `pgrep` displays the PID and process name.

Remember that the process name displayed might be truncated if it's too long. If you need more detailed information about the process, consider using the `ps` command or other process monitoring tools like `top` or `htop`.

```
sayak@ubuntu:~/harry>this$ ps -p 30807
  PID TTY      TIME CMD
 30807 ?        00:03:27 Isolated Web Co
sayak@ubuntu:~/harry>this$ pgrep -l -P 30807
sayak@ubuntu:~/harry>this$ ps -p 30807 -o comm=
Isolated Web Co
sayak@ubuntu:~/harry>this$
```

bash

```
4. ps -p <PID> -o comm=
```

Replace `<PID>` with the actual Process ID for which you want to find the process name.

For example, to find the process name of the process with PID 1234:

```
bash
```

```
ps -p 1234 -o comm=
```

This command will display the process name corresponding to the given PID.

The `-o comm=` option specifies that you want to output only the command name (process name) of the specified PID. The `=` sign at the end is used to ensure that there's no header displayed in the output.

SA

exit, logout - Terminate shell script.

The `exit` and `logout` commands in a shell script are used to terminate the execution of the script and exit the current shell session. While they serve similar purposes, they are used in slightly different contexts.

1. `exit` Command:

The `exit` command is used to terminate the execution of a shell script.

When encountered in a script, it immediately ends the script's execution and returns an exit status to the parent shell.

```
bash
```

```
exit [status]
```

- `status` is an optional integer value that represents the exit status of the script. It typically ranges from `0` (success) to non-zero values (error).

For example, to exit a script with a successful status:

```
bash
```

- `#!/bin/bash`

```
echo "Script is about to exit."
```

```
exit 0
```

- **logout Command:**

The `logout` command is used to log out of the current shell session. It's typically used in interactive shell sessions (when you're directly working in the terminal) rather than in shell scripts. When you use `logout` in an interactive session, it will log you out of the session and return you to the previous environment.

bash

2. **logout**

For example, if you're logged into a remote server via SSH and you want to log out of the session, you can simply type `logout`. (I.T, that's why my 'logout' command is not working.)

```
sayak@ubuntu:~/harry/this$ exit 0
exit
root@ubuntu:/home/sayak/harry/this# exit 0
exit
sayak@ubuntu:~/harry/this$ logout
bash: logout: not login shell: use 'exit'
sayak@ubuntu:~/harry/this$ █
```

It's important to note that the `exit` command is used within shell scripts to control the script's termination, while the `logout` command is used to end an interactive shell session. Make sure to use these commands in the appropriate context.

SA

ping - Test connectivity of host

The `ping` command in Linux and Unix-like systems is used to test the connectivity and reachability of a remote host or network device using the Internet Control Message Protocol (ICMP). It sends ICMP echo request packets to the target host and waits for ICMP echo reply packets, measuring the round-trip time and packet loss.

Here's the basic syntax of the `ping` command:

bash

ping [options] host

- **options** are various command-line options you can use with the **ping** command.
- **host** is the IP address or hostname of the remote host you want to test connectivity with.

For example, to test connectivity with a remote host with the IP address **8.8.8.8**:

bash

```
ping 8.8.8.8
```

This will continuously send ICMP echo requests to the target host and display the round-trip time, packet loss, and other statistics.

Some common options for the **ping** command include:

- **-c count**: Specifies the number of packets to send before stopping.
- **-i interval**: Specifies the interval between sending packets.
- **-t ttl**: Sets the Time to Live (TTL) value for the packets.
- **-s size**: Specifies the size of the ICMP packets.
- **-q**: Quiet mode; suppresses most output, showing only summary information.

For example, to send only three packets with a packet size of 64 bytes to a host:

bash

```
ping -c 3 -s 64 example.com
```

Keep in mind that some systems may have restrictions on using the **ping** command due to security concerns, and firewalls may block ICMP traffic.

Additionally, the response times and behavior of the **ping** command can vary depending on network conditions and the remote host's configuration.

bunzip2 - Uncompress .bz2 file.

The **bunzip2** command is used in Linux and Unix-like systems to uncompress files that have been compressed using the **.bz2** format. This format is commonly associated with the Bzip2 compression algorithm.

To uncompress a **.bz2** file using the **bunzip2** command, you can use the following syntax:

bash

```
bunzip2 filename.bz2
```

Replace **filename.bz2** with the name of the compressed file you want to uncompress.

For example, if you have a file named **example.txt.bz2** that you want to uncompress, you would run:

bash

```
bunzip2 example.txt.bz2
```

After running the command, the **.bz2** file will be uncompressed, and you will have the original file **example.txt**.

Keep in mind that **bunzip2** will replace the compressed file with the uncompressed version. If you want to keep both versions, it's a good idea to make a backup copy of the compressed file before running the

```
sayak@ubuntu:~/harry/this$ ls
1.txt      3.csv   6.csv   hard_link.txt  script.sh
1.txt.bz2  4.txt   6.txt   myarchive.a    soft_link.txt
2.txt      5.txt   7.txt   script2.sh
sayak@ubuntu:~/harry/this$ bzip2 7.txt
sayak@ubuntu:~/harry/this$ ls
1.txt      3.csv   6.csv   hard_link.txt  script.sh
1.txt.bz2  4.txt   6.txt   myarchive.a    soft_link.txt
2.txt      5.txt   7.txt   script2.sh
sayak@ubuntu:~/harry/this$ bunzip2 7.txt.bz2
sayak@ubuntu:~/harry/this$ ls
1.txt      3.csv   6.csv   hard_link.txt  script.sh
1.txt.bz2  4.txt   6.txt   myarchive.a    soft_link.txt
2.txt      5.txt   7.txt   script2.sh
sayak@ubuntu:~/harry/this$
```

bunzip2 command.

gunzip - Uncompress .gz file.

The `gunzip` command in Linux and Unix-like systems is used to uncompress files that have been compressed using the `.gz` format. This format is commonly associated with the Gzip compression algorithm.

To uncompress a `.gz` file using the `gunzip` command, you can use the following syntax:

bash

gunzip filename.gz

Replace `filename.gz` with the name of the compressed file you want to uncompress.

For example, if you have a file named `example.txt.gz` that you want to uncompress, you would run:

bash

gunzip example.txt.gz

After

```
sayak@ubuntu:~/harry/this$ ls
1.txt      3.csv   6.csv  hard_link.txt  script.sh
1.txt.bz2  4.txt   6.txt  myarchive.a    soft_link.txt
2.txt      5.txt   7.txt  script2.sh
sayak@ubuntu:~/harry/this$ gzip 2.txt
sayak@ubuntu:~/harry/this$ ls
1.txt      3.csv   6.csv  hard_link.txt  script.sh
1.txt.bz2  4.txt   6.txt  myarchive.a    soft_link.txt
2.txt.gz   5.txt   7.txt  script2.sh
sayak@ubuntu:~/harry/this$ gunzip 2.txt.gz
sayak@ubuntu:~/harry/this$ ls
1.txt      3.csv   6.csv  hard_link.txt  script.sh
1.txt.bz2  4.txt   6.txt  myarchive.a    soft_link.txt
2.txt      5.txt   7.txt  script2.sh
```

running the command, the `.gz` file will be uncompressed, and you will have the original file `example.txt`.

In some cases, you might encounter compressed files with the `.tar.gz` extension. These are often called "tarballs" and contain multiple files and

directories that have been compressed using Gzip. To uncompress and extract the contents of a `.tar.gz` file, you can use the `tar` command along with `gunzip`:

bash

```
tar -xzf filename.tar.gz
```

The `-x` option indicates extraction, `-z` indicates that the file is Gzipped, and `-f` specifies the filename. Replace `filename.tar.gz` with the actual filename of the tarball you want to extract.

SA

compress multiple files and folders in tar.gz

To compress multiple files and folders into a single `.tar.gz` archive, you can use the `tar` command along with the Gzip compression option. Here's the syntax:

bash

```
tar -czf archive_name.tar.gz file1 file2 folder1  
folder2
```

- `-c`: Create a new archive.
- `-z`: Use Gzip compression.
- `-f`: Specify the name of the archive file.

Replace `archive_name.tar.gz` with the desired name for your compressed archive. List the files and folders you want to include in the archive after the archive name.

For example, to compress two files named `file1.txt` and `file2.txt`, and a folder named `documents`:

bash

```
tar -czf myarchive.tar.gz file1.txt file2.txt  
documents
```

This will create a compressed archive named `myarchive.tar.gz` containing `file1.txt`, `file2.txt`, and the contents of the `documents` folder.

If you want to

```
sayak@ubuntu:~/harry/this$ ls
1.txt      3.csv  6.csv  hard_link.txt  script.sh
1.txt.bz2   4.txt  6.txt  myarchive.a    soft_link.txt
2.txt      5.txt  7.txt  script2.sh
sayak@ubuntu:~/harry/this$ tar -czf archive_name.tar.gz 1.t
xt 5.txt script2.sh
sayak@ubuntu:~/harry/this$ ls
1.txt      4.txt  7.txt          script2.sh
1.txt.bz2  5.txt  archive_name.tar.gz  script.sh
2.txt      6.csv  hard_link.txt    soft_link.txt
3.csv      6.txt  myarchive.a
sayak@ubuntu:~/harry/this$ tar -xzf archive_name.tar.gz
sayak@ubuntu:~/harry/this$ ls
1.txt      4.txt  7.txt          script2.sh
1.txt.bz2  5.txt  archive_name.tar.gz  script.sh
2.txt      6.csv  hard_link.txt    soft_link.txt
3.csv      6.txt  myarchive.a
```

compress an entire directory and its contents, you can specify the directory name as the argument to the `tar` command:

bash

`tar -czf myarchive.tar.gz mydirectory`

This will compress the entire `mydirectory` along with all its subdirectories and files into the `myarchive.tar.gz` archive.

Remember that the order of the files and directories in the command matters; they will be included in the archive in the order you list them. Also, be mindful of the relative paths when specifying files and directories.

umount - Unmount file system.

The `umount` command in Linux and Unix-like systems is used to unmount (dismount) a mounted file system. When you mount a file system, it becomes accessible at a specific mount point within the directory structure. The `umount` command allows you to safely detach the file system from the mount point, making it unavailable for further access.

(But, how can I access those files although after running the particular command using superuser power??!!).

```
sayak@ubuntu:~/harry/this/soham$ su root
Password:
root@ubuntu:/home/sayak/harry/this/soham# ls
8.txt 9.txt pramita
root@ubuntu:/home/sayak/harry/this/soham# umount /home/sayak/harry/this/soham
umount: /home/sayak/harry/this/soham: not mounted.
root@ubuntu:/home/sayak/harry/this/soham# vim 8.txt
root@ubuntu:/home/sayak/harry/this/soham# su sayak
sayak@ubuntu:~/harry/this/soham$ vim 8.txt
sayak@ubuntu:~/harry/this/soham$
```

Here's the basic syntax of the `umount` command:

bash

umount [options] mount_point

- `options` are various command-line options you can use with the `umount` command.
- `mount_point` is the directory where the file system has been mounted and you want to unmount.

For example, to unmount a file system that is mounted at the directory `/mnt/mydisk`:

bash

umount /mnt/mydisk

After executing this command, the file system mounted at `/mnt/mydisk` will be unmounted, and the contents of that file system will no longer be accessible at that location.

It's important to note that you need appropriate permissions to unmount a file system. **Typically, only the superuser (root) has the necessary privileges to unmount file systems.** Additionally, before unmounting a file system, ensure that no processes or applications are actively using files on that file system, as unmounting a mounted file system that is in use can result in data corruption.

For more advanced options and considerations, you can refer to the `umount` command's manual page by running:

```
bash
```

```
man umount
```

SA

who - Users and their activities.

The `who` command in Linux and Unix-like systems is used to display information about users who are currently logged into the system and their activities. It provides details such as the username, terminal or remote host they are logged in from, login time, and more.

Here's the basic syntax of the `who` command:

```
bash
```

```
who [options]
```

- **options** are various command-line options you can use with the `who` command.

When you run the `who` command without any options, it will display a list of currently logged-in users along with their details, including username, terminal or remote host, login time, and duration of the session.

For example:

yaml

username1 pts/0 **2023-08-20 10:00 (:0)**

username2 tty1 **2023-08-20 09:30**

In this example, **username1** is logged in from the **pts/0** terminal since 10:00 AM, and **username2** is logged in on **tty1** since 9:30 AM.

Some common options for the **who** command include:

- **-H**: Display column headers.
- **-q**: Display only the number of logged-in users.
- **-u**: Display the username along with the other details.

For example,

```
sayak@ubuntu:~/harry>this/soham$ who
sayak    tty2          2023-08-21 11:00 (tty2)
sayak@ubuntu:~/harry>this/soham$ who -H
NAME      LINE      TIME           COMMENT
sayak    tty2      2023-08-21 11:00
sayak@ubuntu:~/harry>this/soham$ who -q
sayak
# users=1
sayak@ubuntu:~/harry>this/soham$ who -u
sayak    tty2      2023-08-21 11:00  old          1702 (t
ty2)
sayak@ubuntu:~/harry>this/soham$
```

example, to display the number of logged-in users:

bash

who -q

The **who** command is useful for checking the current user sessions on a system, especially on multi-user environments or remote servers. It can help you monitor who is using the system and what they are doing.

Basics of linux

Table of Contents

<u>Basics of linux</u>	1
<u>Why learning Linux is helpful</u>	3
<u>Operating system = Kernel + system applications</u>	3
<u>Linux file system</u>	4
<u>Getting started</u>	4
<u>Piping</u>	4
• cat text.txt wc -w.....	5
• sort file.txt uniq.....	5
• ps aux grep "chrome" awk '{print \$2}'.....	6
• echo "Hello, World!" tr [:lower:] [:upper:]'.....	7
<u>• Login to remote system: ssh</u>	8
<u>• Copying files: scp</u>	8
<u>Command Description</u>	8
5. ls -ltr => sorting the listing by time modification.....	8
7. cp file1 file2 -copy files(!!??).....	9
9. mv file1 file2 rename file1 to file2.....	11
10. rm -i file.....	11
21. grep -i [pattern] file.....	12
22. find -iname "file".....	12
23. ./file => executes a file in current directory.....	13
How to change the permission of a file in linux.....	13
24. ../../file => executes a file two levels above the current directory.....	14
<u>Absolute and relative path</u>	15
<u>Echo</u>	15
echo [options] [string(s)].....	15
echo "New content" >> file.txt.....	17
• text="Hello World"; echo "Length of \$text is \${#text}".....	18
• result=\$(echo "5 + 7" bc) ; echo "\$result".....	19
• result=\$(echo "sqrt(25)" bc -l) ; echo "\sqrt{25} = \$result".....	19
<u>File permissions</u>	20
<u>What is hard link of a file?</u>	21
How to create hardlink to a file?.....	21
<u>Changing file permissions</u>	23
<u>File ownership</u>	23
<u>grep</u>	24
▪ Counting matches: grep -c "pattern" file.txt.....	24
▪ Display line numbers: grep -n "pattern" file.txt.....	25
▪ Invert match: grep -v "pattern" file.txt.....	25
▪ display the names of files with a match: grep -l "pattern" file.....	25
▪ Recursive search: grep -r "pattern" directory/.....	26
▪ Regular expressions: grep "^[0-9]*\$" numbers.txt.....	26
▪ Matching multiple patterns: grep "pattern1 pattern2" file.txt.....	27
▪ Show context around match: grep -C 2 "pattern" file.txt.....	27
▪ Search file types: grep "pattern" --include "*.txt" dir/.....	28
<u>Arithmetic Computation</u>	28
• expr:.....	28

• bc:.....	28
• (()): Shell arithmetic expansion for integer computations.....	28
Difference between ((x = 5 + 3)) and x=\$((5+3))	30
File Permissions and Ownership.....	30
• chown: Change file's ownership.....	31
Text Processing.....	31
• head: Beginning of file.....	31
• tr: Translate or delete characters.....	31
• cut: Cut columns or fields from a file.....	32
• sort: Sorts Lines in ASCII collating sequence.....	36
• tail -r: Lines in reverse order.....	36
• uniq -d, uniq -u: filter out or identify repeated lines.....	37
• tr -s: Squeeze multiple spaces to a single space.....	37
File Display and Manipulation.....	38
Append File Content: cat file1.txt >> file2.txt.....	38
• Directory list.....	39
Ls -l/directory.....	39
ls -l grep "^\d".....	39
• ls -p . grep /.....	39
• du: Disk space utilization.....	40
Sort and display largest directories: du -h directory/ sort -rh head -n 10.....	40
• tail -f: Monitor growth of a file.....	41
watch : runs a command repeatedly.....	42
System and User Information.....	42
• who -r, runlevel: System run level.....	42
• who: sayak tty2 2023-08-31 18:01 (tty2).....	43
User Account and Security.....	44
• exit, logout: Terminate shell script.....	44
SSH and Remote Access.....	44
• ssh -p port number username@remote host.....	44
• scp, sftp: Copy file between machines.....	45
• Copy local to remote machine:(Not working!!??).....	45
System and Process Control.....	46
• uptime: Show system uptime.....	46
• htop: Interactive process viewer.....	46
• kill: Terminate process by PID:.....	47
• nohup: Run a command in the background.....	48
• reboot: Reboot the system.....	49
Environment and Variables.....	49
• set: Assign values to positional parameters.....	49
• export: Pass variable value to sub-shell.....	51
Input and Output.....	52
• read: Input data to a shell script interactively.....	52
• paste: Join two files laterally.....	54
System Date and Time.....	54
• Current date: date +"%Y-%m-%d".....	54
• Future date: date -d "+3 days" +"%Y-%m-%d".....	55

Why learning Linux is helpful

- Command line proficiency
- Server management
- Security and stability
- Development platform
- Flexibility and customization

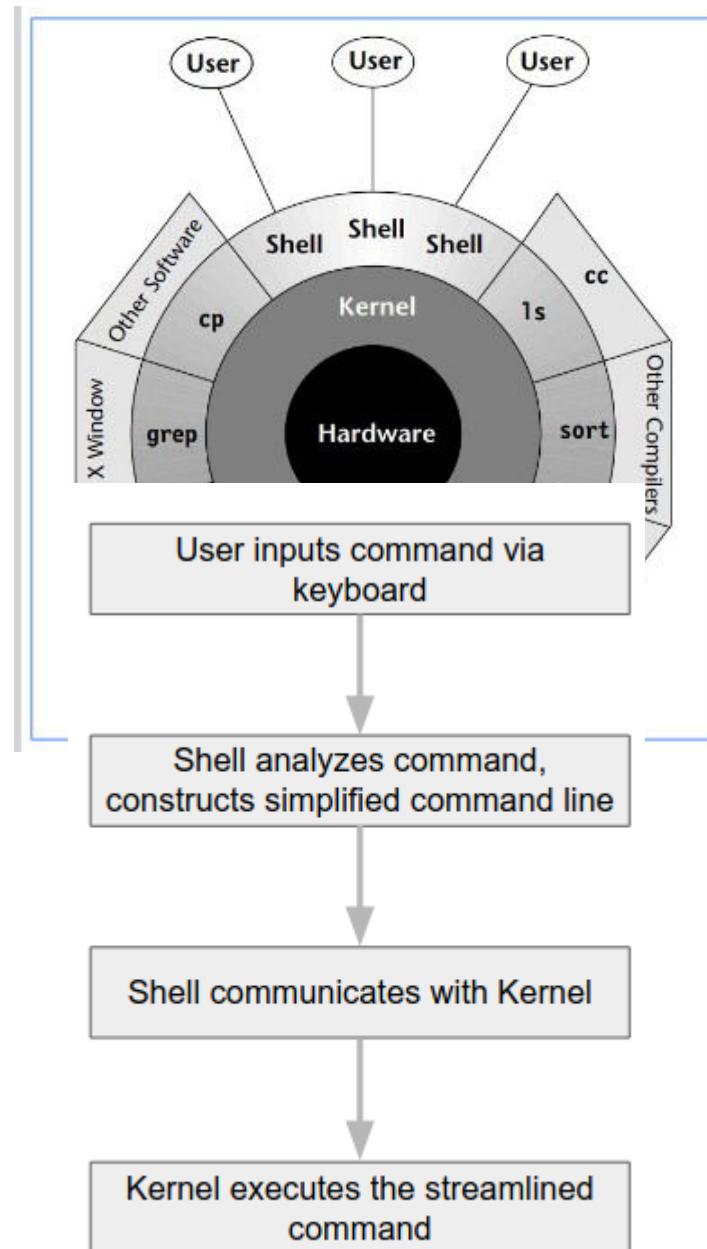
A Multi-user system

- A multitasking system
- A Repository of applications
- Pattern matching
- Programming facility
- Documentation

Linux is derived from Unix; developed by Linus Torvalds in 1991

Operating system = Kernel + system applications

- Kernel is core of the OS
- Kernel interacts with machine's hardware while shell interacts with the user
- When programs are not running, the kernel has to do the work. Controls the execution of the program
- Manages system memory, schedule processes, decides their priorities, etc.
- Computers require an interpreter to understand user



commands - shell

- Interface between the user and the kernel
- Files & process
- Files - containers for storing static information
- Process - program in execution

Linux file system

- Directory in Linux is similar to a “Folder” in Windows OS
- Files are organized into directories and sub-directories
- Directories are separated by forward slash (/)

Getting started

- All Linux commands are **CASE SENSITIVE**
- Type one or two characters of a command and HIT TAB (twice) for knowing the list of all commands starting with these characters.

man

Manual - Provides help information for the command.

- “*” character means match everything
- “?” character means match any one character
- [0-4] or [m-s] matches a range of characters

Piping

- ls -l | grep "file"

- **cat text.txt | wc -w**

The command "cat text.txt | wc -w" is used to count the number of words in the "text.txt" file using the **wc** (word count) command in Unix-like operating systems. The **cat** command is used to display the contents of the file, and the output is then piped using the **|** symbol to the **wc -w** command, which counts the words in the text.

Keep in mind that the command assumes that "text.txt" is in the same directory as where you're executing the command, or you provide the correct path to the file.

```
sayak@ubuntu:~/harry>this$ cat 2.txt | wc -w
56
```

- **sort file.txt | uniq**

The command "sort file.txt | uniq" is used to sort the lines in the "file.txt" and then filter out duplicate lines using the **uniq** command in Unix-like operating systems. Here's what each part of the command does:

1. **sort file.txt**: This part of the command sorts the lines in the "file.txt" in ascending order. By default, **sort** treats the entire line as a sorting key.
2. **uniq**: This command filters adjacent **duplicate** lines from its input. Since the input is sorted, it removes only consecutive duplicate lines. If you have **non-consecutive duplicates**, you might need to sort the file before using **uniq**.

It's worth noting that if you want to get a list of unique lines from a file without sorting them, you could use **sort file.txt | uniq -u**. This would first sort the lines and then display only the lines that appear once.

```

sayak@ubuntu:~/harry/this$ cat 2.txt
i - Insert Mode
• Transitions Vim to the insert mode for text input.
• Esc - Normal Mode
• Returns Vim to the normal mode
• Returns Vim to the normal mode

• :w - Write Write
• Saves changes made to the file.
• :q - Quit
• Exits Vim.
• :wq - Write and Quit Quit
• Saves changes and exits Vim.
• Saves changes and exits Vim.

sayak@ubuntu:~/harry/this$ sort 2.txt | uniq

• Esc - Normal Mode
• Exits Vim.
i - Insert Mode
• :q - Quit
• Returns Vim to the normal mode
• Saves changes and exits Vim.
• Saves changes made to the file.
• Transitions Vim to the insert mode for text input.
• :wq - Write and Quit Quit
• :w - Write Write
sayak@ubuntu:~/harry/this$ sort 2.txt | uniq -u

• Esc - Normal Mode
• Exits Vim.
i - Insert Mode
• :q - Quit
• Saves changes made to the file.
• Transitions Vim to the insert mode for text input.
• :wq - Write and Quit Quit
• :w - Write Write
sayak@ubuntu:~/harry/this$

```

#(Note that , these lines are being sorted according to the alphabetic order of the first character of each line).

• ***ps aux | grep "chrome" | awk '{print \$2}'***

The command "ps aux | grep 'chrome' | awk '{print \$2}'" is used to find and display the process IDs (PIDs) of all processes that have "chrome" in their command line using a series of Unix-like commands. Here's what each part of the command does:

1. **ps aux**: This command lists all the running processes on the system. The options "aux" specify that you want to see information

about all processes (including those of other users), in a detailed format.

2. `grep 'chrome'`: This part of the command searches for the term "chrome" in the output of the `ps aux` command. It filters out the lines that do not contain "chrome".

3. `awk '{print $2}'`: This part of the command uses the **awk** utility to extract and print the second field (column) from the filtered lines. In this context, the second field represents the process ID (PID) of the matching processes.

```
sayak@ubuntu:~/harry>this$ ps aux | grep "firefox" | awk '{print $2}'  
2356  
2529  
2544  
2671  
6850  
6852  
44392  
46564  
59114  
59123  
59156  
60266
```

So, when you run this entire command, you'll get a list of process IDs of all processes that have "chrome" in their command line, typically indicating Chrome browser processes.

- **`echo "Hello, World!" | tr [:lower:] [:upper:]`**

The `tr` command is used for character-level translation or deletion in Unix-like systems, and '`[:lower:]`' and '`[:upper:]`' are character classes representing lowercase and uppercase letters respectively. The command replaces lowercase letters with their corresponding uppercase counterparts.

```
sayak@ubuntu:~/harry>this$ echo "Hello, World!" | tr [:lower:] [:upper:]  
HELLO, WORLD!
```

ctrl+c halts the current command

- **ctrl+z** stops the current command and resume it with `fg` in the foreground or `bg` in the background

- **Login to remote system: ssh**
- ssh -X remote_username@remote_hostname OR
Eg: **ssh -X sandeep@10.3.55.120**
-X: This is an option for SSH that enables X11 forwarding. X11 forwarding allows you to run graphical applications on the remote server and have their graphical output displayed on your local machine. This is particularly useful when you want to run GUI applications on a remote server but view their interfaces on your local system.

```
sayak@ubuntu:~/harry>this$ ssh -X root@10.0.2.15
ssh: connect to host 10.0.2.15 port 22: Connection refused
```

- **Copying files: scp**

- scp -r remote_username@remote_hostname:~/file1 . OR
- scp -r file1 [remote_username@remote_hostname:~](#)

Command Description

1. **mkdir dir** -create directory
2. **cd dir** -change directory
3. **pwd** -present working directory
4. **ls** -list files and directories

5. ls -ltr => sorting the listing by time modification

The **ls -ltr** command is used to list the files and directories in the current directory in a long format, sorted by their modification time in ascending order (oldest to newest).

Here's what each part of the command does:

- **ls:** This is the command used to list the files and directories.

- -l: This option specifies the long format listing, which includes additional information about the files and directories, such as permissions, ownership, size, and modification time.
- -t: This option is used to sort the listing by modification time, with the most recently modified files appearing at the bottom.
- -r: This option reverses the sorting order, so with -t and -r together, the most recently modified files will appear at the top.

So, when you run `ls -ltr`, you'll get a list of files and directories in the current directory, displayed in long format, sorted by modification time in descending order (newest to oldest).

```
sayak@ubuntu:~/harry/this$ ls -ltr
total 40
-rw-r--r-- 1 root  root    330 Aug 22 16:18 myarchive.a
lrwxrwxrwx 1 root  root      5 Aug 22 16:29 soft_link.txt -> 1.txt
-rw-rw-r-- 1 sayak vboxsf  356 Aug 24 10:17 hard_link.txt
-rw-rw-r-- 1 sayak sayak  504 Aug 27 12:39 archive_name.tar.gz
drwxrwxr-x 3 sayak sayak 4096 Aug 27 13:10 soham
-rw-rw-r-- 1 sayak sayak   318 Aug 27 16:17 1_new.txt
-rw-rw-r-- 1 sayak sayak   115 Aug 27 16:59 4.txt
-rw-rw-r-- 1 sayak sayak   267 Aug 27 16:59 1.txt
-rw-rw-r-- 1 sayak sayak    83 Aug 27 17:24 5.txt
-rw-rw-r-- 1 sayak sayak   329 Aug 28 10:23 3.txt
-rw-rw-r-- 1 sayak sayak   349 Aug 29 17:40 2.txt
```

touch file -creates an empty file.

7. `cp file1 file2` -copy files(!!??)

The command `cp file1 file2` is used to copy the contents of `file1` to a new file named `file2`. It creates a new file named `file2` with the same content as `file1`.

If `file2` already exists, it will be overwritten by the content of `file1`. If `file2` does not exist, it will be created. If you want to copy both the contents and the attributes (permissions, timestamps, etc.) of `file1` to `file2`, you can use the `-p` flag like this:

bash

```
cp -p file1 file2
```

This will ensure that the attributes of `file1` are preserved in `file2`.

If you want to copy multiple files to a directory, you can specify the destination directory as the last argument:

bash

```
cp file1 file2 destination_directory/
```

This will copy both `file1` and `file2` to the `destination_directory`.

8. `cp -r dir1 dir2` -copy directory dir1 to directory dir2

```
sayak@ubuntu:~$ mkdir harry3
sayak@ubuntu:~$ cp -r this harry3
cp: cannot stat 'this/this/1.txt': Permission denied
cp: cannot stat 'this/this/2.txt': Permission denied
cp: cannot stat 'this/this/1.txt.bz2': Permission denied
sayak@ubuntu:~$ cd harry3
sayak@ubuntu:~/harry3$ ls
this
```

#(Here, some files can not be copied due to permissions. But, ‘this’ named folder with other files got transferred.)

9. mv file1 file2 rename file1 to file2

```
sayak@ubuntu:~/harry>this$ ls
10.txt    2.txt  5.txt          myarchive.a
1_new.txt  3.txt  archive_name.tar.gz  soft_link.txt
1.txt     4.txt  hard_link.txt   soham
sayak@ubuntu:~/harry>this$ cat 5.txt
12 kaaaaaaaaamamjkanlkkjkyu
%rounghjugfyhtytssak
hmataarsh 89
Sayak
Harry Sujoy 456
sayak@ubuntu:~/harry>this$ mv 5.txt 11.txt
sayak@ubuntu:~/harry>this$ cat 11.txt
12 kaaaaaaaaamamjkanlkkjkyu
%rounghjugfyhtytssak
hmataarsh 89
Sayak
Harry Sujoy 456
sayak@ubuntu:~/harry>this$ cat 4.txt
• Returns Vim to the normal mode
• :w - Write
• Saves changes made to the file.
• :q - Quit
• Exits Vim.
sayak@ubuntu:~/harry>this$ mv 11.txt 4.txt
sayak@ubuntu:~/harry>this$ 4.txt
4.txt: command not found
sayak@ubuntu:~/harry>this$ cat 4.txt
12 kaaaaaaaaamamjkanlkkjkyu
%rounghjugfyhtytssak
hmataarsh 89
Sayak
Harry Sujoy 456
```

(# Here, in the first case, when ‘file 2’ got a new name, ‘mv’ command simply renamed the file. But, ‘file 2’ is named with an existing file, that existing file is overwritten).

10. rm -i file

=> remove file1 (interactive mode: Ask for confirmation before deleting)

```
11. sayak@ubuntu:~/harry>this$ ls
1_new.txt  3.txt          hard_link.txt  soham
1.txt      4.txt          myarchive.a
2.txt      archive_name.tar.gz  soft_link.txt
sayak@ubuntu:~/harry>this$ rm -i 1_new.txt
rm: remove regular file '1_new.txt'? y
sayak@ubuntu:~/harry>this$ ls
1.txt  3.txt  archive_name.tar.gz  myarchive.a  soham
2.txt  4.txt  hard_link.txt      soft_link.txt
```

rmdir dir1 =>remove directory

12. **history** => prints all recent commands to stdout

13. **cat file** => prints contents of the file
14. **head file** => outputs first 10 lines
15. **tail file** => outputs last 10 lines
16. **top** => display all running processes
17. **kill -9 PID** => kill the process with process id
18. **bg** => list all stopped jobs or resume the stopped job in the background
19. **fg** => Brings the most recent job to the foreground
20. **date** => displays today date and time

21. grep -i [pattern] file

=> search for a pattern in file and print it to stdout.

```
sayak@ubuntu:~/harry/this/soham$ cat 1.txt
i - Insert Mode
• Transitions Vim to the insert mode for text input.
• Esc - Normal Mode
• Returns Vim to the normal mode
• :w - Write Sayak
• Saves changes made to the file.
• :q - Quit sayak
• Exits Vim.
• :wq SayaK - Write and Quit
• Saves changes SAYAK and exits Vim.
sayak@ubuntu:~/harry/this/soham$ grep -i 'sayak' 1.txt
• :w - Write Sayak
• :q - Quit sayak
• :wq SayaK - Write and Quit
• Saves changes SAYAK and exits Vim.
```

find

iname “file”

=>search for a file in current directory recursively

```
sayak@ubuntu:~$ find -iname pramita
./harry/this/soham/pramita
sayak@ubuntu:~$
```

23. *./file => executes a file in current directory*

```
sayak@ubuntu:~/harry/this$ cat 2.txt
i - Insert Mode
• Transitions Vim to the insert mode for text input.
• Esc - Normal Mode
• Returns Vim to the normal mode
• Returns Vim to the normal mode
sayak@ubuntu:~/harry/this$ ./2.txt
bash: ./2.txt: Permission denied
sayak@ubuntu:~/harry/this$ chmod u+x,g+x 2.txt
sayak@ubuntu:~/harry/this$ ./2.txt
./2.txt: line 1: i: command not found
./2.txt: line 2: •: command not found
./2.txt: line 3: •: command not found
./2.txt: line 4: •: command not found
./2.txt: line 5: •: command not found
sayak@ubuntu:~/harry/this$
```

#(As the permission for execution of the file is denied, so we need to change the permission using chmod command).

How to change the permission of a file in linux

To change the permissions of a file in Linux, you can use the **chmod** command. The **chmod** command allows you to modify the read, write, and execute permissions for the owner, group, and others. Here's the basic syntax:

bash

```
chmod permissions file
```

The **permissions** part is a combination of letters and symbols that represent the desired permission settings, and **file** is the name of the file you want to modify.

Here are some examples of how to use the **chmod** command:

1. Numeric Representation: You can use numeric values to set permissions.

The numeric values correspond to the following permissions:

- 4: Read (r)
- 2: Write (w)
- 1: Execute (x)

For example, to give read and write permissions to the owner, and read-only permissions to the group and others, you would use:

bash

chmod 644 file

2. Symbolic Representation: You can also use symbolic representation to set permissions using letters:

- u: Owner
- g: Group
- o: Others
- a: All (equivalent to ugo)

For example, to give execute permission to the owner and the group, you would use:

bash

chmod u+x, g+x file

Remember, it's important to be careful when changing file permissions, especially on system files, to avoid unintended consequences.

Note that changing permissions might require administrative privileges (using sudo).

24. ../../file => executes a file two levels above the current directory

```
sayak@ubuntu:~/harry/this$ ../../12.txt
bash: ../../12.txt: Permission denied
sayak@ubuntu:~/harry/this$ chmod u+x,g+x 12.txt
chmod: cannot access '12.txt': No such file or directory
sayak@ubuntu:~/harry/this$ cd
sayak@ubuntu:~$ chmod u+x,g+x 12.txt
sayak@ubuntu:~$ cd /home/sayak/harry/this/
sayak@ubuntu:~/harry/this$ ../../12.txt
../../12.txt: line 1: Sayak: command not found
../../12.txt: line 2: Govinda: command not found
../../12.txt: line 3: Krishna: command not found
../../12.txt: line 4: Durga: command not found
sayak@ubuntu:~/harry/this$ █
```

#(Here, we have a file named '12.txt' in the home directory. We want to execute that file while we are in two directory ahead. But, we did not have necessary permission to execute. So, we need to go to the home directory to change the permission. Then, we were able to execute.)

25. **zip file.zip file1 file2** or **unzip file.zip** =>compress/uncompress the files

Absolute and relative path

```
# sayak@ubuntu:~$ cd /home/sayak/harry/this/soham
one sayak@ubuntu:~/harry/this/soham$ cd pramita
sayak@ubuntu:~/harry/this/soham/pramita$ cd ../../../../that/
sayak@ubuntu:~/harry/that$
```

First
is the

absolute path. Second line is the relative path. 3rd line of relative path is used, when we need to change the branch of the folder. We should use that many number of ‘..’ that we want to go backward.

Echo

echo [options] [string(s)]

SA

The echo command in Unix-like operating systems is used to print its arguments (strings or variables) to the standard output (usually the terminal). Here's the basic syntax:

bash

```
echo [options] [string(s)]
```

- [options]: These are optional flags that modify the behavior of the echo command.
- [string(s)]: These are the strings or variables you want to print.

Here are a few examples of using the echo command:

1. Print a simple string:

bash

```
echo "Hello, World!"
```

2. Print the value of a variable:

bash

name="Alice"

**echo "Hello,
\$name"**

3. Print without a newline (useful for appending to the same line):

bash

echo -n "This is on "

echo "the same line"

The -n option with the first echo prevents it from adding a newline character at the end of the printed string. As a result, the second echo continues on the same line as the first one, and the output appears as a single line.

#How it looks like
n"and and (“-“-e”).
4.
sayak@ubuntu:~/harry/that\$ echo -n "This is on "
echo "the same line"
This is on the same line
sayak@ubuntu:~/harry/that\$ echo "This is on "
echo "the same line"
This is on
the same line
sayak@ubuntu:~/harry/that\$ echo -e "Line 1\nLine 2"
Line 1
Line 2
sayak@ubuntu:~/harry/that\$ █

looks
with (“-
“-e”)
without
n"and

Interpret escape sequences (such as \n for newline):

bash

echo -e "Line 1\nLine 2"

The -e option allows you to use escape sequences to represent special characters, such as newline (\n), tab (\t), and others, in the output. ***Without the -e option, the escape sequences would be treated as literal characters.***

The echo command is commonly used for printing messages and information in shell scripts and command-line environments. Keep in mind that the behavior of the echo command can vary slightly between different Unix-like systems due to implementation differences.

echo "New content" >> file.txt

SA

The command echo "New content" >> file.txt is used to append the text "New content" to the end of the file named "file.txt". The >>

operator is used for appending content to a file, and if the file does not exist, it will be created.

After running this command, the content "New content" will be

added as a new line at the end of the "file.txt". If "file.txt" already contains content, the new text will be added on a new line below the existing content.

Keep in mind that if you want to overwrite the entire file content with the new text instead of appending, you can use a single > operator like this:

bash

```
echo "New content" > file.txt
```

This will replace the existing content of "file.txt" with "New content".

```
sayak@ubuntu:~/harry/that$ cd ../this
sayak@ubuntu:~/harry/this$ cat 2.txt
i - Insert Mode
• Transitions Vim to the insert mode for text input.
• Esc - Normal Mode
• Returns Vim to the normal mode
• Returns Vim to the normal mode
sayak@ubuntu:~/harry/this$ echo "new content" >> 2.txt
sayak@ubuntu:~/harry/this$ cat 2.txt
i - Insert Mode
• Transitions Vim to the insert mode for text input.
• Esc - Normal Mode
• Returns Vim to the normal mode
• Returns Vim to the normal mode
new content
sayak@ubuntu:~/harry/this$ echo "new content" > 2.txt
sayak@ubuntu:~/harry/this$ cat 2.txt
new content
sayak@ubuntu:~/harry/this$
```

- **echo -e "Column 1\tColumn 2\tColumn 3"**

The `-e` option enables the interpretation of escape sequences, and `\t` represents a tab character.

```
sayak@ubuntu:~/harry>this$ echo -e "Column 1\tColumn 2\tColumn 3"
Column 1           Column 2           Column 3
```

- **text="Hello World" ; echo "\$text" | tr '[:lower:]' '[:upper:]'**

```
sayak@ubuntu:~/harry>this$ text="Hello World" ; echo "$text" | tr '[:lower:]' '[:upper:]'
HELLO WORLD
```

- **text="Hello World"; echo "Length of \$text is \${#text}"**

`text="Hello World";`: This assigns the string "Hello World" to the variable `text`.

`echo "Length of $text is ${#text}"`: This command prints the length of the text stored in the `text` variable. `${#text}` is used to retrieve the length of the variable's value.

```
sayak@ubuntu:~/harry>this$ text="Hello World"; echo "Length of $text
is ${#text}"
World": command not found
Length of Hello World is 11
```

- **result=\$(echo "5 + 7" | bc) ; echo "\$result"**

`result=$(echo "5 + 7" | bc) ;` This command calculates the result of the arithmetic expression "5 + 7" using the `bc` (basic calculator).
The `$()` syntax is used to capture the output of the command and store it in the `result` variable.

`echo "$result"`: This command prints the value stored in the `result` variable.

```

sayak@ubuntu:~/harry/this$ result=$(echo "5 + 7" | bc) ; echo "$result"
12
sayak@ubuntu:~/harry/this$ result=$(echo "5 + 7") ; echo "$result"
5 + 7
sayak@ubuntu:~/harry/this$ result=(echo "5 + 7") ; echo "$result"
echo
sayak@ubuntu:~/harry/this$ result=(echo "5 + 7" | bc) ; echo "$result"
"
bash: syntax error near unexpected token `| '
sayak@ubuntu:~/harry/this$ result=(echo "5 + 7") ; echo "result"
result

```

(# Note the different outputs in various cases).

- **result=\$(echo "sqrt(25)" | bc -l) ; echo " $\sqrt{25}$ = \$result"**

The `-l` flag tells `bc` to load the standard math library, which provides additional mathematical functions and constants beyond basic arithmetic operations.

When you run `bc` with the `-l` flag, you can perform more advanced mathematical calculations using functions like `s()` for sine, `c()` for cosine, `l()` for natural logarithm, and so on. Additionally, constants like `e` (Euler's number) and `pi` (π) are available for use.

```

sayak@ubuntu:~/harry/this$ result=$(echo "sqrt(25)" | bc -l) ; echo " $\sqrt{25}$  = $result"
 $\sqrt{25}$  = 5.000000000000000000000000000000
sayak@ubuntu:~/harry/this$ result=$(echo "sqrt(25)" | bc) ; echo " $\sqrt{25}$  = $result"
 $\sqrt{25}$  = 5

```

Here, ‘sqrt’ works without “`bc -l`” flag, but sine, cosine like functions will not work. See below:

```

sayak@ubuntu:~/harry/this$ result=$(echo "s(pi)" | bc) ; echo "sin(180) = $result"
Runtime error (func=(main), adr=4): Function s not defined.
sin(180) =
sayak@ubuntu:~/harry/this$ result=$(echo "s(pi)" | bc -l) ; echo "sin(180) = $result"
sin(180) = 0

```

File permissions

- File Permissions are represented using characters:

- r: Read permission
- w: Write permission
- x: Execute permission
- Type ls -l to see file permissions.

Examples:

- rw-r--r--: Owner has read and write permissions, group and others have read-only permissions.
- rwxr-x---: Owner has read, write, and execute permissions, group has read and execute permissions, others have no permissions.
- rwxrwxrwx: All users have full read, write, and execute permissions. This is not recommended for security reasons, as it gives unrestricted access.

- Read-only file
 - -r--r--r-- 1 owner group 100 Aug 1 file.txt
- Executable File:
 - -r-xr-xr-x 1 owner group 100 Aug 1 script.sh
- Full Access for Owner:
 - **-rwx----- 1 owner group 100 Aug 1 file.txt**

- * 1: This indicates the number of hard links to the file. Hard links are additional references to the same file content. The value "1" here indicates that there is only one hard link to this file.
 - 100: This is the file's size in bytes.
 - Aug 1: This indicates the date of the last modification to the file. In this case, it's August 1st.

SA

What is hard link of a file?

A hard link is a feature in Unix-like operating systems that allows multiple directory entries (filenames) to point to the same inode (data structure that stores file information and data blocks) on the disk. In other words, it's a way to create multiple references to the same physical file content. All hard links to a file share the same data blocks on the disk.

Hard links can exist within the **same directory or across different directories**. They can even exist in different filesystems on the same machine, as long as those filesystems support hard links.

It's important to note that hard links can only be created for regular files and not for directories or special files. Also, you cannot create a hard link to a file located on a different filesystem.

Hard links provide a way to have multiple filenames pointing to the same content, which can be useful for managing files efficiently and maintaining consistency across different parts of the filesystem.

How to create hardlink to a file?

SA

To create a hard link to a file in Unix-like operating systems, you can use the `ln` command followed by the source file name and the desired name for the hard link. The basic syntax is as follows:

bash

```
ln source_file hard_link_name
```

Here's how you can create a hard link:

1. Open a terminal.
2. Navigate to the directory where you want to create the hard link, using the `cd` command.
3. Use the `ln` command to create the hard link. For example, let's say you have a file named `original.txt` and you want to create a hard link named `mylink.txt`:

bash

3. `ln original.txt mylink.txt`

After running this command, the hard link `mylink.txt` will be created in the same directory as `original.txt`. Both files will point to the same inode on the disk, meaning changes made to one file will be reflected in the other. Here below, we made some changes in `mylink.txt` in vim text editor. No change was done in `10.txt`, still the change is reflected.

```
sayak@ubuntu:~/harry>this/soham$ cat 10.txt
i - Insert Mode
• Transitions Vim to the insert mode for text input.
• Esc - Normal Mode
• Returns Vim to the normal mode
•
sayak@ubuntu:~/harry>this/soham$ ln 10.txt mylink.txt
sayak@ubuntu:~/harry>this/soham$ vim mylink.txt

[No write since last change]
sayak@ubuntu:~/harry>this/soham$ exit
exit
sayak@ubuntu:~/harry>this/soham$ cat 10.txt
i - Insert Mode
• Transitions Vim to the insert mode for text input.
• Esc - Normal Mode
• Returns Vim to the normal mode
• I have made some change in mylink.txt. It reflects in
  10.txt also
```

Remember that you cannot create a hard link to a directory or a special file. Hard links can only be created for regular files.

From listing files, see and

```
sayak@ubuntu:~/harry>this/soham$ ls -l
total 20
-rw-rw-r-- 2 sayak sayak 197 Aug 30 19:35 10.txt
-rw-rw-r-- 1 sayak sayak 291 Aug 30 09:44 1.txt
-rw-rw-r-- 1 sayak sayak 35 Aug 27 13:10 8.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 27 12:59 9.txt
-rw-rw-r-- 2 sayak sayak 197 Aug 30 19:35 mylink.txt
drwxrwxr-x 2 sayak sayak 4096 Aug 27 12:57 pramita
```

the of the you can 10.txt

`mylink.txt` both files has 2 hardlinks.

Changing file permissions

- Use the 'chmod' command to modify permissions
- `chmod [options] permissions file(s)`

- Examples:
- Add execute permission: **chmod u+x file.txt**
- Remove read for group/others: **chmod go-r file.txt**
- Set rwx for owner, rx for group&other: chmod 755 script.sh
here r=4, w=2 and x=1

File ownership

- Files have an owner and a group associated with them.
- Owners create files, groups provide shared access.
- Use chown to change ownership
- **chown [options] new_owner:new_group files**
- Examples:
- chown hpc:ccds file.txt
- chown :ccds file.txt
- chown root:root dir/
- chown -R user:group dir/

```
# [Note
10.txt
has 2
sayak@ubuntu:~/harry/this/soham$ groups
sayak sudo vboxsf
sayak@ubuntu:~/harry/this/soham$ ls -l
total 20
-rw-rw-r-- 2 sayak sayak 197 Aug 30 19:35 10.txt
-rw-rw-r-- 1 sayak sayak 291 Aug 30 09:44 1.txt
-rw-rw-r-- 1 sayak sayak 35 Aug 27 13:10 8.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 27 12:59 9.txt
-rw-rw-r-- 2 sayak sayak 197 Aug 30 19:35 mylink.txt
drwxrwxr-x 2 sayak sayak 4096 Aug 27 12:57 pramita
sayak@ubuntu:~/harry/this/soham$ chown root:sudo 10.txt
chown: changing ownership of '10.txt': Operation not permitted
sayak@ubuntu:~/harry/this/soham$ chown sayak:vboxsf 10.txt
sayak@ubuntu:~/harry/this/soham$ ls -l
total 20
-rw-rw-r-- 2 sayak vboxsf 197 Aug 30 19:35 10.txt
-rw-rw-r-- 1 sayak sayak 291 Aug 30 09:44 1.txt
-rw-rw-r-- 1 sayak sayak 35 Aug 27 13:10 8.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 27 12:59 9.txt
-rw-rw-r-- 2 sayak vboxsf 197 Aug 30 19:35 mylink.txt
drwxrwxr-x 2 sayak sayak 4096 Aug 27 12:57 pramita
```

that, as
this file

hardlinks, so group ownership of two files ‘10.txt’ and ‘mylink.txt’ both have changed.]

grep

- grep [options] pattern [file...]
- Examples:
 - **Basic text search:** grep "keyword" file.txt
 - **Case-Insensitive search:** grep -i "word" file.txt
 - **Counting matches:** grep -c "pattern" file.txt

The command grep -c "pattern" file.txt is used to count the number of lines in the file "file.txt" that match the specified pattern. The -c flag tells grep to display the count of matching lines rather than the actual lines themselves.

- **Display line numbers:** grep -n "pattern" file.txt

The command grep -n "pattern" file.txt is used to search for lines in the file "file.txt" that match the specified pattern and display those lines along with their line numbers. The -n flag tells grep to include line numbers in the output.

- **Whole word match:** grep -w "word" file.txt

- **Invert match: grep -v "pattern"**

The command `grep -v "pattern"` file.txt is used to search for lines in the file "file.txt" that do not match the specified pattern. The `-v` flag tells grep to perform an inverted match, meaning it will exclude lines that contain the specified pattern.

```
sayak@ubuntu:~/harry/files_module2_l1$ cat fruit.txt
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
Fruits are delicious.

sayak@ubuntu:~/harry/files_module2_l1$ grep -i "fruits"
" fruit.txt
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
Fruits are delicious.
sayak@ubuntu:~/harry/files_module2_l1$ grep -c "fruits"
" fruit.txt
2
sayak@ubuntu:~/harry/files_module2_l1$ grep -n "fruits"
" fruit.txt
1:This is a sample file with some fruits.
2:I like fruits. Who does not like fruits?
sayak@ubuntu:~/harry/files_module2_l1$ grep -w "fruits"
" fruit.txt
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
sayak@ubuntu:~/harry/files_module2_l1$ grep -v "fruits"
" fruit.txt
Fruits are delicious.

sayak@ubuntu:~/harry/files_module2_l1$ grep -l "fruits"
" *.txt
fruit.txt
more_fruits.txt
```

- **display the names of files with a match: grep -l "pattern" file**

The command `grep -l "pattern" file` is used to search for files that contain the specified pattern and display only the names of those files. The `-l` flag tells grep to list the names of files with at least one matching line.

For example, if you have multiple text files in a directory and you want to find and list the names of files that contain the word "apple", you would use:

bash

```
grep -l "apple" *.txt
```

▪ Recursive search: grep -r "pattern" directory/

```
sayak@ubuntu:~/harry/files_module2_l1$ cd ..
sayak@ubuntu:~/harry$ grep -r "fruits" files_module2_l
1/
files_module2_l1/fruit.txt:This is a sample file with
some fruits.
files_module2_l1/fruit.txt:I like fruits. Who does not
like fruits?
files_module2_l1/more_fruits.txt:This is a sample file
with some fruits, such as apples, bananas, oranges, m
angoes, etc.
sayak@ubuntu:~/harry$
```

Regular expressions: grep "^[0-9]*\$" numbers.txt

The command grep "^[0-9]*\$" numbers.txt uses a regular expression to search for lines in the file "numbers.txt" that **consist entirely of digits**. Here's what the regular expression and the command do:

```
sayak@ubuntu:~/harry$ cd this
sayak@ubuntu:~/harry/this$ cat 4.txt
• 12 kaaaaaaamamamjkanlkkjkkyu
%rounghjugfyhtytssak
hmataarsh 89
Sayak
456
sayak@ubuntu:~/harry/this$ grep "^[0-9]*$" 4.txt
456
```

grep:
This is
the

command used for searching patterns in files.

- "^[0-9]*\$": This is a regular expression pattern that matches lines containing only digits. Let's break down the pattern:
 - ^: Anchors the pattern to the beginning of the line.
 - [0-9]*: Matches zero or more occurrences of digits (0-9).
 - \$: Anchors the pattern to the end of the line.
- numbers.txt: This is the name of the file in which you want to perform the search.

In summary, this command will search the "numbers.txt" file and **display lines that consist entirely of digits (0-9)**, with no other characters or spaces.

- **Matching multiple patterns: grep "pattern1\|pattern2" file.txt**

"pattern1\|pattern2": This is a regular expression pattern that matches lines containing either "pattern1" or "pattern2".

```
sayak@ubuntu:~/harry/this$ cd ../files_module2_l1/
sayak@ubuntu:~/harry/files_module2_l1$ grep "fruits\|F
ruits" fruit.txt
This is a sample
I like fruits.
Fruits are delici
sayak@ubuntu:~/harry/files_module2_l1$ cat more_fruits
.txt
This is a sample file with some fruits, such as apples
, bananas, oranges, mangoes, etc.
I do not like apple pie.
I really like apple juice, orange juice, but not banan
a juice.
Yesterday I got 3 bananas and 4 oranges.

sayak@ubuntu:~/harry/files_module2_l1$ grep -C 2 "pie"
more_fruits.txt
This is a sample file with some fruits, such as apples
, bananas, oranges, mangoes, etc.
I do not like apple pie.
I really like apple juice, orange juice, but not banan
a juice.
Yesterday I got 3 bananas and 4 oranges.
```

- **Show context around match: grep -C 2 "pattern" file.txt**

-C 2: This option specifies that grep should display 2 lines of context (both before and after) around each matching line.

- **Search file types: grep "pattern" --include "*.txt" dir/**

--include "* .txt": This option specifies that only files with a ".txt" extension should be included in the search. The wildcard * matches any filename before the ".txt" extension.

(Something wrong! Not working!!)

- **Exclude file types: grep "pattern" --exclude "* .txt" dir/**

(Something wrong! Not working!!)

Arithmetic Computation

- **expr:**

Evaluate arithmetic expressions and perform string operations.

Example: expr 5 + 3 outputs 8 (note spaces between 5,+ ,3)

- **bc:**

An arbitrary precision calculator language.

Example: echo "5 + 3" | bc outputs 8.

- awk, sed: NEXT LECTURE

- **(()): Shell arithmetic expansion for integer computations.**

In Unix-like shells, such as Bash, the (()) double parentheses construct is used for shell arithmetic expansion, which allows you to perform integer computations directly within the shell. This feature is commonly used to perform calculations, comparisons, and assignments involving integers. It's especially useful when scripting or working within the command line.

Here are some examples of how the (()) construct is used for integer computations:

1. Basic Arithmetic:

You can perform basic arithmetic operations like addition, subtraction, multiplication, and division.

```
result=$((5 + 3))
```

```
echo $result #
```

Output: 8

```
sayak@ubuntu:~$ result=$((5+3)); echo $result
8
sayak@ubuntu:~$ x=10;y=3;result=$((x*y));echo $result
30
sayak@ubuntu:~$ a=15
b=20
if ((a > b)); then
    echo "a is greater than b"
else
    echo "a is not greater than b"
fi
a is not greater than b
sayak@ubuntu:~$ count=5
((count++))
echo $count
6
sayak@ubuntu:~$ count=5; ((count++)); echo $count
6
sayak@ubuntu:~$ value=$(( (5 > 3) && (4 < 2) ))
echo $value
0
sayak@ubuntu:~$
```

The dollar sign \$ in the context of Unix-like shells, such as Bash, is used to access the value of a variable. When you use \$ followed by a variable name, the shell substitutes the variable's value in its place.

- **Variables and Assignment:**

You can use variables and assign results to them.

```
x=10
```

```
y=3
```

```
result=$((x * y))  
echo $result # Output: 30
```

- **Comparison:**

You can perform integer comparisons and use the result in conditionals.

```
a=15  
b=20  
if ((a > b)); then  
    echo "a is greater than b"  
else  
    echo "a is not greater than b"  
fi
```

```
sayak@ubuntu:~$ a=15  
b=20  
if ((a > b)); then  
    echo "a is greater than b"  
else  
    echo "a is not greater than b"  
fi  
a is not greater than b  
sayak@ubuntu:~$ a=15;b=20;if ((a > b)); th  
en    echo "a is greater than b"; else  
    echo "a is not greater than b"; fi  
a is not greater than b
```

```
echo "a is not greater than b"
```

Here, you can see how we write the bash script in 1 command line using semi-colons only.

- **Increment and Decrement:**

You can increment and decrement variables.

```
count=5
```

```
((count++))  
echo $count # Output: 6
```

- **Using Logical Operators:**

You can use logical operators for calculations.

```
value=$(( ( 5 > 3) && (4 < 2) ))  
echo $value # Output: 0 (false)
```

Remember that the `(())` construct is specifically for integer arithmetic. For floating-point arithmetic or more complex calculations, you might need to use other tools or scripting languages like `bc` or Python.

Difference between `((x = 5 + 3))` and `x=$((5+3))` .

`((x = 5 + 3))` followed by `echo $x` outputs 8. OR

`x=$((5+3))` followed by `echo $x` outputs 8.

The `((x = 5 + 3))` approach uses the `(())` construct directly for arithmetic assignment, while the `x=$((5 + 3))` approach uses the `$(())` construct to perform arithmetic calculation and then assign the result to the variable `x`.

```
sayak@ubuntu:~$ ((x=5+3));echo $x  
8  
sayak@ubuntu:~$ x=$((5+3));echo $x  
8
```

File and Ownership

Permissions

- **chgrp:** Change file's group ownership.

```
chgrp new_group_name file_or_directory
```

- **touch:** Change file's last modification or access time.

• **chown: Change file's ownership**

```
chown new_user:new_group file_or_directory
```

```
# Change only the user
```

```
chown newuser: myfile.txt
```

```
# Change only the group
```

```
chown :newgroup myfile.txt
```

- **chmod:** Change file's permissions.

Text Processing

- **head: Beginning of file.**

```
# (If we write "head
sayak@ubuntu:~/harry/files_module2_l1$ head -2 fruit.txt
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
sayak@ubuntu:~/harry/files_module2_l1$ head -n 2 fruit.txt
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
```

fruit.txt" by default it will write first 10 lines.)

- **tr: Translate or delete characters**

```
# Convert all uppercase letters to lowercase
```

```
sayak@ubuntu:~/harry/files_module2_l1$ echo "HELLO WORLD" | tr '[[:upper:]]' '[[:lower:]]'
hello world
# sayak@ubuntu:~/harry/files_module2_l1$ echo "HELLO WORLD" | tr '[A-Z]' '[a-z]'
hello world
```

Remove all spaces from a string

```
sayak@ubuntu:~/harry/files_module2_l1$ cat fruit.txt | tr -d ' '
This is a sample file with some fruits.
# I like fruits. Who does not like fruits?
Fruits are delicious.
```

Replace all vowels with the letter "X"

```
sayak@ubuntu:~/harry/files_module2_l1$ cat fruit.txt | tr '[aeiou]' 'X'
ThXs Xs X sXmplX fXlX wXth sXmX frXXts.
# I lXkX frXXts. WhX dXXs nXt lXkX frXXts?
FrXXts XrX dXlXcXXXs.
```

Squeeze all duplicate characters in a string

```
sayak@ubuntu:~/harry>this$ echo "Thhiisss iisss mmyyy  
nameeee" | tr -s 'a-z'  
This is my name  
• sayak@ubuntu:~/harry>this$ cat 4.txt  
12 kaaaaaaaaamamjkanlkkjkkyu  
%rounghjugfyhtytssak  
hmataarsh 89  
SSayak  
4555776  
DDDear  
sayak@ubuntu:~/harry>this$ cat 4.txt | tr -s 'a-z' 'A  
-Z'  
12 KAMAMJKANLKJKYU  
%ROUNGHJUGFYHTYTSAK  
HMATARSH 89  
SAYAK  
4555776  
DEAR
```

cut: Cut columns or fields from a file.

The **cut** command is used to extract columns or fields from lines of text files. It's particularly useful when dealing with structured data where fields are separated by a specific delimiter, like a comma or tab. Here are some common use cases and examples of the **cut** command:

Basic Usage:

bash

- **cut -d ',' -f 1 file.csv**
- **cut**: The command itself.
- **-d ','**: Specifies the delimiter as a comma (',').
- **-f 1**: Specifies that you want to extract the first field.
- **file.csv**: The input file from which you want to extract fields.

```
sayak@ubuntu:~/harry/files_module2_l1$ cut -d ',' -f  
1 fruit.txt  
This  
I  
Fruits
```

- Extract Multiple Fields:

bash

- **cut -d '\$\t' -f 2,3 file.tsv**

- **-d '\$'\t'**: Specifies the delimiter as a tab character (**\$'\t'**).
- **-f 2,3**: Specifies that you want to extract the second and third fields.
- **file.tsv**: The input file from which you want to extract fields.

```
sayak@ubuntu:~/harry/files_module2_l1$ cut -d ' ' -f  
1,2 fruit.txt  
This is  
I like  
Fruits are
```

- **Range of Fields:**

bash

- **cut -d ':' -f 2-4 file.txt**
- **-d ':'**: Specifies the delimiter as a colon (**'.'**).
- **-f 2-4**: Specifies that you want to extract fields from the second to the fourth (inclusive).
- **file.txt**: The input file from which you want to extract fields.

```
sayak@ubuntu:~/harry/files_module2_l1$ cut -d ' ' -f  
1-3 fruit.txt  
This is a  
I like fruits.  
Fruits are delicious.
```

- **Complementing Fields:**

bash

- **cut -d ' ' --complement -f 2 file.txt**
- **-d ' '**: Specifies the delimiter as a space.
- **--complement**: Instructs **cut** to complement the field list (extract all fields except the specified ones).
- **-f 2**: Specifies that you want to exclude the second field.

- **file.txt**: The input file from which you want to extract fields.

```
sayak@ubuntu:~/harry/files_module2_11$ cut -d ' ' --complement -f 2 fruit.txt
This a sample file with some fruits.
I fruits. Who does not like fruits?
Fruits delicious.
```

- **Specifying Delimiter with Tab Characters:**

bash

- **cut -d \$'\t' -f 3 file.txt**
- **-d \$'\t'**: Specifies the delimiter as a tab character (\$'\t').
- **-f 3**: Specifies that you want to extract the third field.
- **file.txt**: The input file from which you want to extract fields.

- **Extracting Specific Characters:**

bash

- **cut -c 5-10 file.txt**
- **-c 5-10**: Specifies that you want to extract characters from the fifth to the tenth position.
- **file.txt**: The input file from which you want to extract characters.

```
sayak@ubuntu:~/harry/files_module2_11$ cut -c 5-10 fruit.txt
is a ke fru ts are
```

Combining with Other Commands:

bash

```
grep "keyword" input.txt | cut -d ',' -f 2
```

- **grep "keyword" input.txt | cut -d ',' -f 2**: Searches for lines containing "keyword" in **input.txt**.

- | : The pipe symbol, which takes the output of the command on the left and uses it as input for the command on the right.
- `cut -d ',' -f 2`: Extracts the second field from each line using a comma as the delimiter.

Remember, these commands are examples, and you can modify them according to your specific use case, data format, and extraction needs.

```
sayak@ubuntu:~/harry/files_module2_l1$ grep "fruits"
fruit.txt | cut -d ',' -f 2
is
like
```

• **sort: Sorts Lines in ASCII collating sequence.**

`sort [options] file`

Option	Description
--------	-------------

- | | |
|-----------------|--|
| <code>-n</code> | Sort the lines numerically. |
| <code>-f</code> | Fold case, so that "ABC" and "abc" are considered equal. |
| <code>-b</code> | Ignore leading and trailing spaces. |
| <code>-r</code> | Sort the lines in reverse order. |

```
sayak@ubuntu:~/harry/files_module2_l1$ cat 12.txt
123
2 heart
Hand schools
hunter
96 23 part123
= house
sayak@ubuntu:~/harry/files_module2_l1$ sort -n 12.txt
123
2 heart
Hand schools
= house
hunter
96 23 part123
123
sayak@ubuntu:~/harry/files_module2_l1$ sort -f 12.txt
123
2 heart
96 23 part123
Hand schools
= house
hunter
sayak@ubuntu:~/harry/files_module2_l1$ sort -r 12.txt
hunter
= house
Hand schools
96 23 part123
2 heart
123
```

- k Sort by the specified field. The field is specified by a number, starting with 1 for the first field.(?!)
- t The delimiter that separates the fields. The default delimiter is a space. (?!)

- **tail -r: Lines in reverse order.**

```
tail -r filename.txt or
```

```
some_command | tail -r
```

```
sayak@ubuntu:~/harry/files_module2_1
1$ tail -r fruit.txt
tail: invalid option -- 'r'
Try 'tail --help' for more information.
```

#(Please note that the `tail -r` command might not be available on all Unix-like systems, as it's not a standard POSIX option for the `tail` command. If it's not available on your system, you can achieve similar results using other commands or scripting.)

- **uniq -d, uniq -u: filter out or identify repeated lines**

1. `uniq -d`: This option displays only the lines that are repeated (i.e., lines that appear more than once in the input). It removes all lines that occur only once. Here's how you can use it:

```
bash
```

```
uniq -d filename.txt
```

2. `uniq -u`: This option displays only the lines that occur only once (i.e., lines that do not have duplicates in the input).

It removes all lines that are repeated. Here's how you can use it:

```
bash
```

```
uniq -u filename.txt
```

This command will display only the lines in `filename.txt` that occur exactly once.

- **tr -s: Squeeze multiple spaces to a single space.**

```
sayak@ubuntu:~/harry/files_module2_l1$  
echo "Rohan    is    a good    boy." | tr  
      -s ' '  
Rohan is a good boy.  
sayak@ubuntu:~/harry/files_module2_l1$  
cat 13.txt  
apple          banana  
      apple    mango    grapes  
  
sayak@ubuntu:~/harry/files_module2_l1$  
cat 13.txt | tr -s ' '  
apple banana  
      apple mango grapes
```

File Display and Manipulation

- cat: Concatenate files.
- Display File Content: `cat file.txt`
- **Concatenate Multiple Files:** `cat file1.txt file2.txt > combined.txt`

```
sayak@ubuntu:~/harry/files_module2_l1$ cat fruit.txt more_fruits.txt
> combined.txt
sayak@ubuntu:~/harry/files_module2_l1$ cat combined.txt
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
Fruits are delicious.

This is a sample file with some fruits, such as apples, bananas, oranges, mangoes, etc.
I do not like apple pie.
I really like apple juice, orange juice, but not banana juice.
Yesterday I got 3 bananas and 4 oranges.

This is a sample file with some fruits.
I like fruits. Who does not like fruits?
Fruits are delicious.
```

Append File Content: cat file1.txt >> file2.txt

```
sayak@ubuntu:~/harry/files_module2_l1$ cat 12.txt
123
2 hear t
Hand schools
hunter
96 23 part123
= house
sayak@ubuntu:~/harry/files_module2_l1$ cat 13.txt
apple banana
apple mango grapes

sayak@ubuntu:~/harry/files_module2_l1$ cat 12.txt >> 13.txt
sayak@ubuntu:~/harry/files_module2_l1$ cat 13.txt
apple banana
apple mango grapes

123
2 hear t
Hand schools
hunter
96 23 part123
= house
cat file.txt |
head -n 10
```

• Directory list

Ls -l /directory

When you want to list the content of any directory other than your current working directory.

```
sayak@ubuntu:~/harry/files_module2_l1$ ls -l /home/sayak/Documents/
total 7212
-rw-rw-r-- 1 sayak sayak 1935839 Aug 19 11:31 LINUX.odt
-rw-rw-r-- 1 sayak sayak 2058630 Sep 1 11:57 LTSC_LINUX_COMMANDS.odt
-rw-rw-r-- 1 sayak sayak 121167 Aug 29 16:31 LTSC_REGULAR_EXPRESSION.odt
-rw-rw-r-- 1 sayak sayak 2902282 Aug 27 13:20 LTSC_SELFCMADE.odt
-rw-rw-r-- 1 sayak sayak 357388 Aug 28 09:51 LTSC-VIM.odt
```

ls -l | grep "^\d"

list all the directories in the current directory.

• **ls**

```
sayak@ubuntu:~$ ls -l | grep "^\d"
drwxr-xr-x 2 sayak sayak 4096 Aug 11 12:02 Desktop
drwxr-xr-x 2 sayak sayak 4096 Sep 1 11:57 Documents
drwxr-xr-x 2 sayak sayak 4096 Aug 31 12:32 Downloads
drwxrwxr-x 5 sayak sayak 4096 Aug 31 12:32 harry
drwxr-xr-x 2 root root 4096 Aug 19 10:39 harry2
drwxrwxr-x 3 sayak sayak 4096 Aug 30 08:48 harry3
drwxr-xr-x 2 sayak sayak 4096 Aug 11 12:02 Music
drwxr-xr-x 3 sayak sayak 4096 Aug 14 15:35 Pictures
drwxr-xr-x 2 sayak sayak 4096 Aug 11 12:02 Public
drwx----- 4 sayak sayak 4096 Aug 11 12:06 snap
drwxr-xr-x 2 sayak sayak 4096 Aug 11 12:02 Templates
drwxr-xr-x 3 root root 4096 Aug 21 19:40 this
drwxr-xr-x 2 sayak sayak 4096 Aug 11 12:02 Videos
```

grep /

the command `ls -p . | grep /` will list all the files and directories in the current directory, including those with a trailing slash.

The `ls` command lists the contents of the current directory. The `-p` option tells `ls` to print the full path to each file or directory, including the trailing slash.

The `grep` command searches for lines that match a pattern. The `/` pattern matches a line that contains a slash.

```
sayak@ubuntu:~/harry$ ls -p . | grep /
files_module2_l1/
that/
this/
```

- **du: Disk space utilization.**

du [options] directory

Option	Description
--------	-------------

- | | |
|----|---|
| -a | Recursively list all files and directories, including hidden files. |
| -h | Display sizes in human-readable format, such as KB, MB, GB, and TB. |
| -s | Display the total disk space utilization for the specified directory. |
| -x | Exclude directories from the calculation. |

```
sayak@ubuntu:~/harry$ du -a this
12      this/.1.txt.swp
4       this/archive_name.tar.gz
4       this/hard_link.txt
0       this/soft_link.txt
4       this/1.txt
12      this/.2.txt.swp
4       this/4.txt
4       this/3.txt
4       this/myarchive.a
4       this/soham/pramita
4       this/soham/8.txt
4       this/soham/1.txt
4       this/soham/10.txt
0       this/soham/9.txt
20      this/soham
4       this/2.txt
76      this

sayak@ubuntu:~/harry$ du -h this
4.0K    this/soham/pramita
20K     this/soham
76K     this

sayak@ubuntu:~/harry$ du -s this
76      this

sayak@ubuntu:~/harry$ du -x this
4       this/soham/pramita
20      this/soham
76      this

sayak@ubuntu:~/harry$ █
```

Sort and display largest directories: du -h directory/ | sort -rh | head -n 10

The command `du -h directory/ | sort -rh | head -n 10` is used to show 10 largest directories within the specified directory, sorted in descending order of size.

1. `du -h directory/`: The `du` (disk usage) command is used to estimate file and directory space usage. The `-h` option makes the output human-readable, displaying sizes in a more understandable format (e.g., "1.2G" for 1.2 gigabytes). `directory/` should be replaced with the path to the directory you want to analyze. This command generates a list of directories and their respective sizes.
2. `|`: The pipe (`|`) symbol is used to take the output from the previous command and pass it as input to the next command.

3. **sort -rh**: The **sort** command is used to sort the output of **du** in reverse order (-r) based on the human-readable sizes (-h). This means that the largest directories will appear at the top of the list.

4. | : Another pipe symbol is used to pass the sorted list to the next command.

5. **head -n 10**: The **head** command is used to display the first 10 lines of the sorted list. In this context, it will show the 10 largest directories.

```
sayak@ubuntu:~/harry$ du -h this/ | sort -rh | head -n 3
76K    this/
20K    this/soham
4.0K   this/soham/pramita
sayak@ubuntu:~/harry$
```

- **-f**: **tail**

Monitor growth of a file.

the **tail -f** command can be used to monitor the growth of a file. The **tail** command displays the last few lines of a file, and the **-f** option tells **tail** to follow the file and display new lines as they are added to the file.

Option	Description
n	
-n	Specifies the number of lines to display.
-c	Specifies the number of bytes to display.
-s	Specifies the number of seconds to sleep between updates.

For example, the following command will display the last 10 lines of the file **log.txt** and update every second:

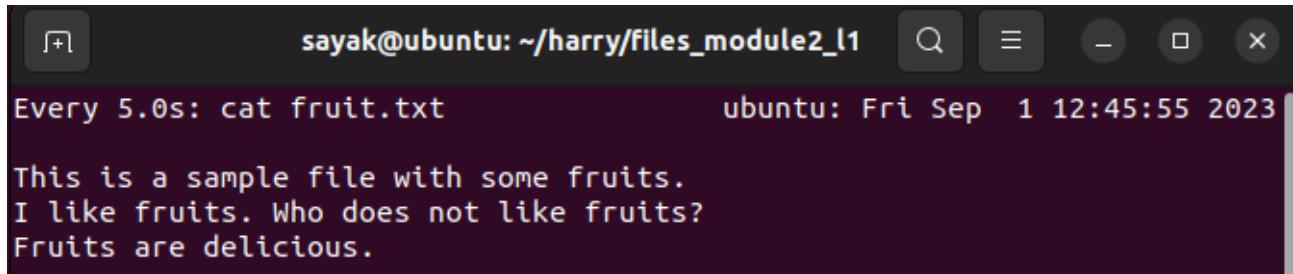
```
tail -f -n 10 -s 1 log.txt
```

watch : runs a command repeatedly.

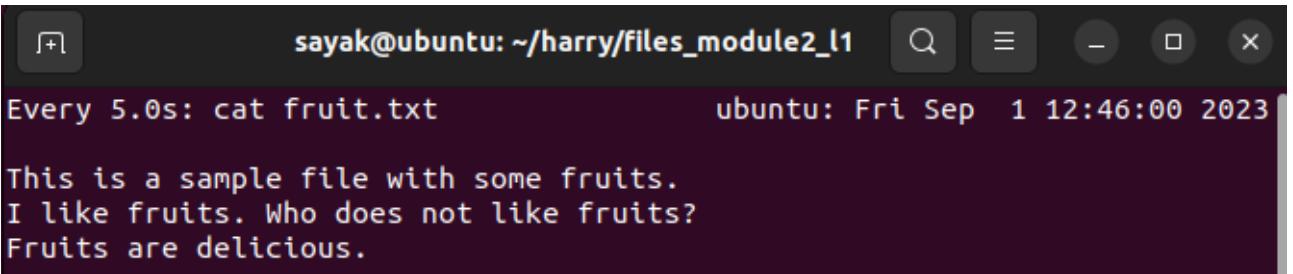
- Watch changes for every second: **watch -n 5 "cat file.txt"**

The **watch** command runs a command repeatedly and displays the output of the command. The **-n** option tells **watch** to run the command every **n** seconds.

```
# sayak@ubuntu:~/harry/files_module2_l1$ watch -n 5 "cat fruit.txt"
```



```
sayak@ubuntu: ~/harry/files_module2_l1
Every 5.0s: cat fruit.txt                                         ubuntu: Fri Sep 1 12:45:55 2023
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
Fruits are delicious.
```



```
sayak@ubuntu: ~/harry/files_module2_l1
Every 5.0s: cat fruit.txt                                         ubuntu: Fri Sep 1 12:46:00 2023
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
Fruits are delicious.
```

(note that, if you write ‘watch’ command and press ‘enter’ this type of interface will come and shows updated file in every ‘n’ seconds (here, n=5). You can kill the process, by pressing ctrl+c.)

System and User Information

- **hostname:** Name of the local host.
- **uname:** Operating system name.
- **uname -r:** Operating system release.

```
sayak@ubuntu:~/harry/files_module2_l1$ hostname
ubuntu
sayak@ubuntu:~/harry/files_module2_l1$ uname
Linux
sayak@ubuntu:~/harry/files_module2_l1$ uname -r
6.2.0-26-generic
sayak@ubuntu:~/harry/files_module2_l1$ who
sayak    tty2          2023-08-31 18:01 (tty2)
sayak@ubuntu:~/harry/files_module2_l1$ who -r
                           run-level 5 2023-08-31 18:01
sayak@ubuntu:~/harry/files_module2_l1$
```

- **who -r, runlevel:** System run level.

1. Runlevel 0: Halt or Shutdown

- This runlevel indicates that the system is shutting down or halting. It is used to power off or halt the system gracefully.

2. Runlevel 1 or S (Single-User Mode):

- Single-user mode is typically used for system maintenance and recovery tasks. In this mode, only a minimal set of services and processes are running, and you have a single-user shell prompt.

3. Runlevel 2:

- This runlevel is often used for multi-user mode without networking services. It's a text-based mode with no graphical interface.

4. Runlevel 3:

- Multi-user mode with networking enabled. In many Unix systems, this is the default runlevel for server installations.

5. Runlevel 4:

- This runlevel is often left unused or reserved for custom configurations, as it doesn't have a standardized meaning.

6. Runlevel 5:

- Multi-user mode with networking enabled and a graphical user interface (GUI). This is typically the default runlevel for desktop systems.

7. Runlevel 6: Reboot

- This runlevel is used to initiate a system reboot. When the system enters runlevel 6, it will restart itself.

• **who: sayak tty2 2023-08-31 18:01 (tty2)**

Users and their activities.

tty2: This is the terminal or console where the user's session is running. "tty" stands for "teletypewriter," and it represents a text-based terminal. In this case, it's **tty2**, which could refer to the second text terminal.

User Account and Security

- passwd: Change own password.
- su: Superuser from a nonprivileged account.

- **exit, logout: Terminate shell script.**

exit: The **exit** command is used to terminate a shell session or script.

When you run **exit**, the shell will exit, and if it's the last shell session, it will also log you out of the system if you're in a terminal session. If you're running a script, it will terminate the script and return to the parent shell.

(# for example, if you want to temporarily exit vim you type “**:sh**” in the command line and if you return back to vim, you write “**exit**” command.)

bash

- **exit**
- **logout**: The **logout** command is typically used in a shell to log out of the current session. It's commonly used in interactive shell sessions to exit the shell and log out of the system.

bash

2. **logout**

The specific behavior of these commands may vary depending on your shell and how it's configured. In many cases, running either **exit** or **logout** will effectively terminate the shell session and return you to the previous level or log you out of the system.

SSH and Remote Access

- **ssh**: Log in to a remote machine.

- **ssh username@remote_host**

- **ssh -p port_number username@remote host**

```
sayak@ubuntu:~/harry/files_module2_l1$ ssh -p 22 sayak@10.0.2.15
ssh: connect to host 10.0.2.15 port 22: Connection refused
```

- **scp, sftp: Copy file between machines.**

SCP stands for **Secure Copy**. It is a command-line tool that uses SSH (Secure Shell) to copy files securely between two machines. SCP encrypts

the data being transferred, so it is not possible for unauthorized users to intercept the data.

SFTP stands for **Secure File Transfer Protocol**. It is a network protocol that uses SSH to transfer files securely between two machines. SFTP is similar to SCP, but it provides a more interactive experience. For example, you can use SFTP to list the contents of a directory or to navigate between directories.

- ***Copy local to remote machine:(Not working!!??)***

scp local_file.txt username@remote_host:/path

```
sayak@ubuntu:~/harry/files_module2_l1$ scp /home/sayak/Documents/LTSC_LIN  
UX_COMMANDS.odt sayak@10.0.2.15:/home/sayak/Downloads/  
ssh: connect to host 10.0.2.15 port 22: Connection refused  
lost connection  
sayak@ubuntu:~/harry/files_module2_l1$
```

- **Copy Remote machine to Local:**

scp username@remote_host:/remote_path/file.txt local_path/

System and Process Control

- **uptime: Show system uptime**

```
sayak@ubuntu:~/harry/files_module2_l1$ uptime
 15:54:49 up  6:09,  1 user,  load average: 0.26, 0.46, 0.64
sayak@ubuntu:~/harry/files_module2_l1$ uptime -s
2023-09-01 09:45:43
sayak@ubuntu:~/harry/files_module2_l1$ uptime -p
up 6 hours, 16 minutes
sayak@ubuntu:~/harry/files_module2_l1$ uptime -u
uptime: invalid option -- 'u'

Usage:
  uptime [options]

Options:
  -p, --pretty    show uptime in pretty format
  -h, --help      display this help and exit
  -s, --since     system up since
  -V, --version   output version information and exit
```

Uptime: This means columnwise i) current time, ii) the system has been running for 6 hr 9 min, iii) 1 user is currently logged in, and the average load on the system is 0.26, 0.46, 0.64. (# The average load is a measure of the number of processes that are waiting to be executed by the CPU. The average load is calculated over a period of time, such as 1 minute, 5 minutes, or 15 minutes.)

uptime -s: shows the time from which the system is running.

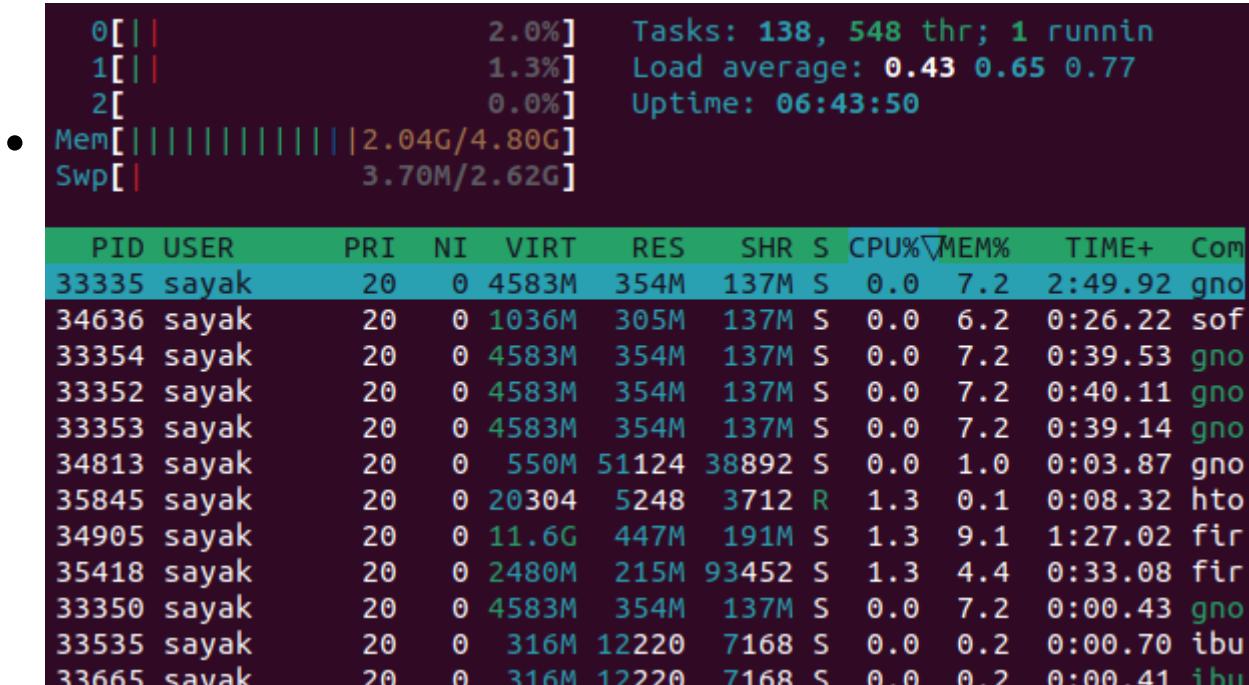
Uptime -p: Show the time since the last boot.

- **htop: Interactive process viewer**

Here are some of the features of htop:

- Live updating list of all processes
- CPU usage, memory usage, and other statistics
- Interactive interface with mouse support
- Ability to kill, change priority, and view full command line of processes
- Configurable columns and colors
- Tree view of processes
- Support for multiple monitors

The default view shows the processes sorted by CPU usage. To kill a process, press K and then enter the process ID. To change the priority of a process, press P and then enter the new priority. To view the full command line of a process, press F. Press q to escape.



kill: Terminate process by PID:

kill PID or

kill -9 PID

The **-9** option specifies the SIGKILL signal, which is a non-ignorable signal that will force the process to terminate. The **kill** command without any options will send the SIGTERM signal to the process. The SIGTERM signal is a graceful termination signal that allows the process to clean up before it terminates. Use the **kill** command with caution, as it can have unintended consequences.

- **bg** & **fg**: Move job to the background or foreground.

- **nohup**: Run a command in the background

The nohup command is used to run a command in the background and ignore the HUP (hangup) signal. The HUP signal is sent to a process when

the controlling terminal of the process is closed. This means that the **nohup** command will continue to run even if you close the terminal window where you started it.

The basic syntax of the nohup command is:

nohup command [ar] **LINUX**

TUTORIALS

#Unix is the foundation is linux.

Two types of interfaces are there. Command line interface and Graphical user interface. GUI is useful for task managing and system manager.

sayak@ubuntu:~\$ ls #ls command lists the directory contents.

```
Desktop Downloads Pictures snap Videos  
Documents Music Public Templates
```

**sayak@ubuntu:~\$ pwd #pwd means print working directory
/home/sayak**

In windows, we have different drives. But, in LINUX, we have tree structure of the files.

sayak@ubuntu:~\$ cd / # "/" is called the root node.

sayak@ubuntu:\$ ls

```
bin dev lib libx32 mnt root snap sys var  
boot etc lib32 lost+found opt run srv tmp  
cdrom home lib64 media proc sbin swapfile usr #Bin, dev , lib  
libx32 are also some of the nodes within the root nodes. They may be  
some files, directories etc.
```

pwd shows the parents of the current directory and 'ls' represents the childrens of the current directory.

**sayak@ubuntu:/ \$ #Here, sayak is the name of the user. And
ubuntu is the name of the virtual box machine. '\$' sign
means I am a regular user. '/' means I am in the '/' or root
directory.**

**sayak@ubuntu:/ \$ cd bin # If we click 'cd bin' then the next
line comes. Means we are in the bin directory now.**

sayak@ubuntu:/bin\$

**# If you write 'cd bin' it represents the relative path and 'cd\bin' it
represents the absolute path. So, cd<some directory name>' takes you to
that certain directory. Then if you write 'ls' it will print all the files under
that directory only.**

sayak@ubuntu:/bin\$ cd .. # If you write ‘cd ..’ command, it will take you back shown as below.

sayak@ubuntu:~\$ cd Downloads/

sayak@ubuntu:~/Downloads\$ #If we press tab after entering ‘Dow..’ it will auto-complete the directory ‘Downloads’.

sayak@ubuntu:~\$ cd harry

sayak@ubuntu:~/harry\$ mkdir this

using these commands, we can go to ‘harry’ directory and can make ‘this’ named folder within ‘harry’ directory.

sayak@ubuntu:~/harry\$ pwd

/home/sayak/harry

to know the absolute path of something, we need “pwd” command.

sayak@ubuntu:~/harry\$ touch 1.txt

sayak@ubuntu:~/harry\$ touch 2.txt # ‘touch’ command makes two empty files named ‘1.txt’ &‘2.txt’ within ‘harry’ directory.

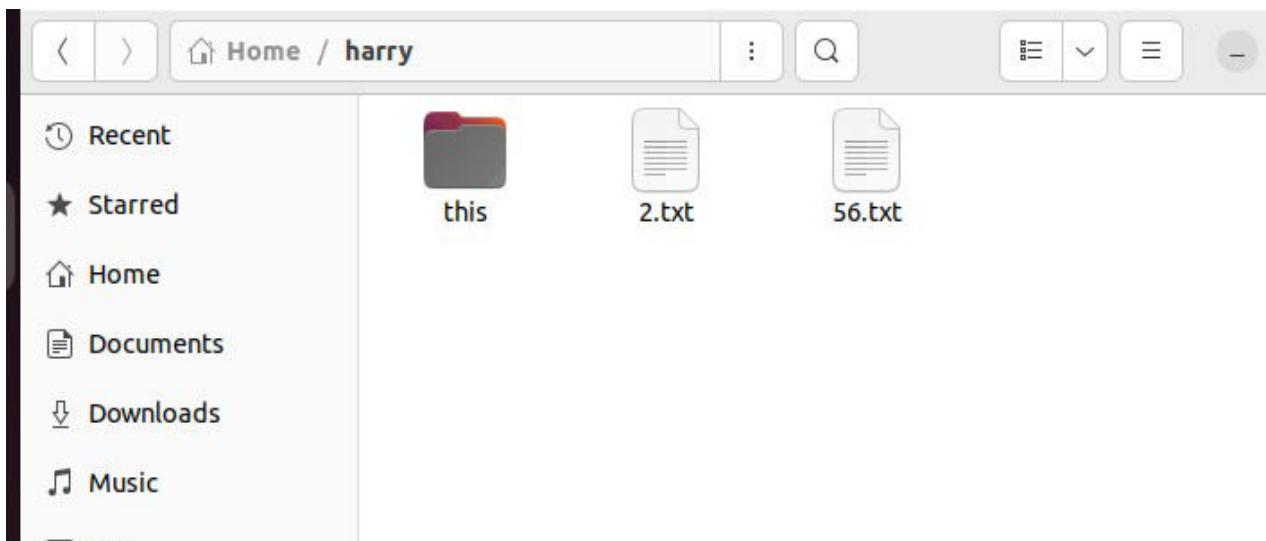
sayak@ubuntu:~/harry\$ mv 1.txt this/ # ‘mv’ command moves ‘1.txt’ file in ‘this’ named directory.

sayak@ubuntu:~/harry\$ cp 2.txt this/ # ‘cp’ command copies ‘2.txt’ file in ‘this’ named directory.

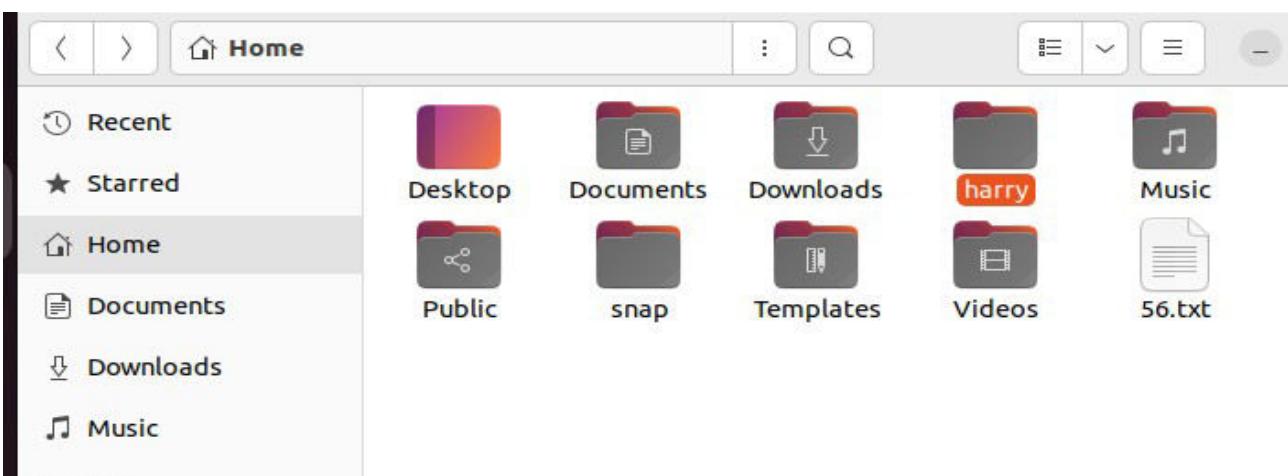
```
sayak@ubuntu:~/harry$ pwd  
/home/sayak/harry  
sayak@ubuntu:~/harry$ touch 56.txt  
sayak@ubuntu:~/harry$ cp /home/sayak/harry/56.txt /home/sayak/
```

By doing this, we can check the current directory. Then creating a new blank file in that (56.txt). we can copy that file after providing the full path where it is located (/home/sayak/harry/56.txt) and then providing the path where it is need to be copied (/home/sayak/).

After creating file:



After moving in the desired location:



Users in Linux:

1. Regular user -> Cannot see to others home directory.

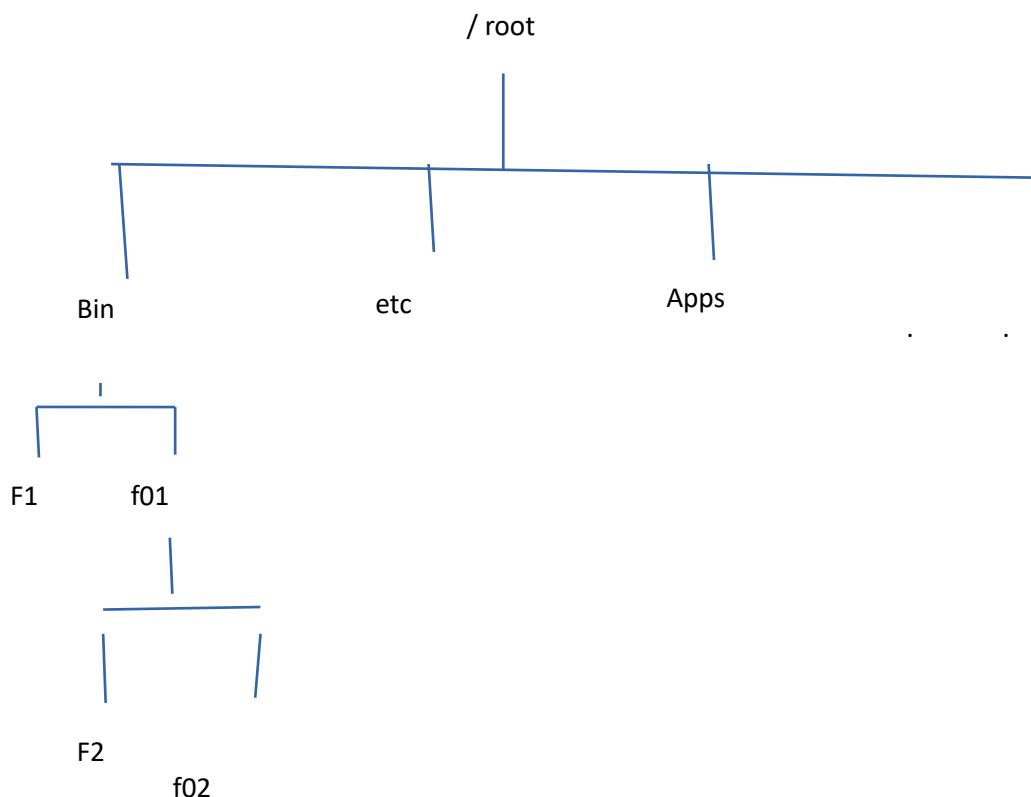
2. Root user or super user -> Can do anything to anyone's home directory. Full access. Called admin.

If we write 'sudo' before any command, the system can identify you as a root user.

If we write 'sudo su' before any command, for all the works it will obey you as a super user. But, it is not recommended. As it is very riskful.

3. Service user:

Absolute and Relative path:



Let , our present working directory is f01. If we want to go to f02 , we just need to write ‘cd f02’. So we nned to write relative path only.

But , if we are in ‘Apps’ directory and need to go to ‘f01’, We should write the full path (i.e. cd */bin/f01*). *It is called the absolute path.*

```
sayak@ubuntu:~/harry$ sudo apt-get update
[sudo] password for sayak:
Get:1 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]
Hit:2 http://in.archive.ubuntu.com/ubuntu jammy InRelease
Get:3 http://in.archive.ubuntu.com/ubuntu jammy-updates InRelease [119 kB]
Get:4 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Packages [657 kB]

Get:28 http://in.archive.ubuntu.com/ubuntu jammy-backports/main amd64 DEP-11 Metadata [4,888 B]
Get:29 http://in.archive.ubuntu.com/ubuntu jammy-backports/universe amd64 DEP-11 Metadata [15.5 kB]
Fetched 8,098 kB in 8s (1,056 kB/s)
Reading package lists... Done
```

‘Sudo apt-get update’ this command makes the list of the softwares that are available to be updated.

But, ‘Sudo apt-get upgrade’ this command installs all the upgraded versions of the softwares.

```
sayak@ubuntu:~/harry$ ls
2.txt 56.txt this
sayak@ubuntu:~/harry$ ls -R
.:
2.txt 56.txt this

./this:
1.txt 2.txt
```

#Here , ‘ls’ lists the files and folders under the present directory. But, ‘ls -R’ lists the files and folders under the present directory and the all consecutive subdirectories also.

```
sayak@ubuntu:~/harry$ ls
2.txt 56.txt this
sayak@ubuntu:~/harry$ touch .harry
sayak@ubuntu:~/harry$ ls -a
. . 2.txt 56.txt .harry this
sayak@ubuntu:~/harry$
```

If ‘.’ is present before a file name, it becomes a new hidden file. To include the hidden file in the list, ‘ls -a’ command is used.

```
sayak@ubuntu:~/harry$ ls -l
total 4
-rw-rw-r-- 1 sayak sayak    0 Aug 14 15:25 2.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 15:38 56.txt
drwxrwxr-x 2 sayak sayak 4096 Aug 14 15:32 this
sayak@ubuntu:~/harry$ ls -t
56.txt  this  2.txt
sayak@ubuntu:~/harry$ ls -r
this  56.txt  2.txt
sayak@ubuntu:~/harry$
```

#ls -l. The -l option signifies the long list format. This shows a lot more information presented to the user than the standard command. You will see the file permissions (for user, for groups, for other public), the number of links, owner name, owner group, file size, time of last modification, and the file or directory name.

ls -t. sort by time & date.

#The option "ls -r" lists all files and directories in reverse order. The files and directories are all arranged in reverse alphabetical order.

sayak@ubuntu:~/harry\$ clear # 'clear' command clears the terminal and make it to look fresh.

sayak@ubuntu:~/harry\$ history #Prints the history of all commands.

sayak@ubuntu:~/harry\$ printf "This is a ball.\n"

This is a ball. # This is the way to print a string.

```
sayak@ubuntu:~/harry$ ls
2.txt 56.txt this
sayak@ubuntu:~/harry$ pwd
/home/sayak/harry
sayak@ubuntu:~/harry$ touch harry.txt
sayak@ubuntu:~/harry$ ls
2.txt 56.txt harry.txt this
sayak@ubuntu:~/harry$ touch HARRY.txt
sayak@ubuntu:~/harry$ ls
2.txt 56.txt harry.txt HARRY.txt this
```

In LINUX, file names are case sensitive. So, we can make two or more files with same name (word-cases are different) in a folder. But, we cannot do it in windows. But, this is confusing; so not recommended.

sayak@ubuntu:~/harry\$ sudo apt install snail # Command to install an app package named 'snail'.

```
sayak@ubuntu:~/harry$ ls -l
total 4
-rw-rw-r-- 1 sayak sayak    0 Aug 14 15:25 2.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 15:38 56.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 18:31 harry.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 18:31 HARRY.txt
drwxrwxr-x 2 sayak sayak 4096 Aug 14 15:32 this
sayak@ubuntu:~/harry$
```

Here, drwx=> directory read write execute.

-rwe=> file read write execute.

To set permissions for owner, group and public we use Chmod calculator. ‘chmod’ is a command which changes the permissions of a file.

Chmod Calculator

An awesome Chmod Calculator to convert Linux file permissions between different formats.

Owner	Group	Public
Read <input checked="" type="checkbox"/>	Read <input checked="" type="checkbox"/>	Read <input checked="" type="checkbox"/>
Write <input checked="" type="checkbox"/>	Write <input checked="" type="checkbox"/>	Write <input type="checkbox"/>
Execute <input checked="" type="checkbox"/>	Execute <input type="checkbox"/>	Execute <input type="checkbox"/>

Linux Permissions:

Chmod Calculator
Chmod Calculator is a free utility to calculate the numeric (octal) or symbolic value for a set of file or folder permissions in Linux servers.

How to use
Check the desired boxes or directly enter a valid numeric value (e.g. 777) or symbolic notation (e.g. rwxrwxrwx) to see its value in other formats.

‘Chmod’ calculator works on the conversion of binary numbers of three groups(owner, group, public) to an octal number. Here, 111=7 { $(111)_2 = (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = (7)_{10}$ }, 110=6, 100=4. So, Linux permission is 764.

```
sayak@ubuntu:~/harry$ chmod 764 2.txt
sayak@ubuntu:~/harry$ ls -l
total 4
-rwxrw-r-- 1 sayak sayak    0 Aug 14 15:25 2.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 15:38 56.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 18:31 harry.txt
-rw-rw-r-- 1 sayak sayak    0 Aug 14 18:31 HARRY.txt
drwxrwxr-x 2 sayak sayak 4096 Aug 14 15:32 this
sayak@ubuntu:~/harry$
```

Here, the file permission of 2.txt file has been changed.

```

sayak@ubuntu:~/harry$ ps
    PID TTY          TIME CMD
      2514 pts/0    00:00:00 bash
      14065 pts/0   00:00:00 ps
sayak@ubuntu:~/harry$ ps -a
    PID TTY          TIME CMD
      1684 tty2    00:00:00 gnome-session-b
      14078 pts/0   00:00:00 ps

```

‘ps’ lists all the command processes which are currently running. ‘ps -a’ lists all the background processes also.

sayak@ubuntu:~/harry\$ top #This ‘top’ command shows which of the running processes takes most of the spaces as shown below.

```

top - 16:18:54 up 7:08, 1 user, load average: 0.45, 0.79, 0.90
Tasks: 210 total, 1 running, 209 sleeping, 0 stopped, 0 zombie
%Cpu(s): 10.3 us, 1.0 sy, 0.0 ni, 88.3 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
MiB Mem : 2964.3 total, 104.0 free, 1997.1 used, 863.3 buff/cache
MiB Swap: 2680.0 total, 2493.0 free, 187.0 used. 727.8 avail Mem

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
10923	sayak	20	0	2864220	372932	104572	S	21.2	12.3	9:12.82	Isolated Web Co
10367	sayak	20	0	3886940	455572	171244	S	1.7	15.0	7:08.41	firefox
2267	sayak	20	0	227516	2972	2688	S	0.7	0.1	0:57.17	VBoxClient
1094	root	20	0	298324	2560	2432	S	0.3	0.1	0:12.29	VBoxDRMClient
1769	sayak	20	0	4657880	401284	107228	S	0.3	13.2	18:46.87	gnome-shell
2496	sayak	20	0	575848	48288	33428	S	0.3	1.6	0:31.07	gnome-terminal
12780	sayak	20	0	2548912	173608	86240	S	0.3	5.7	0:09.27	Isolated Web Co
1	root	20	0	166724	10856	7400	S	0.0	0.4	0:02.24	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par_gp
5	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	slub_flushwq
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	netns
8	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H-events_highpri
10	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq

sayak@ubuntu:~/harry\$ kill 10367 #This ‘kill<space><PID>’ command closes any application. In this case, this command closes ‘firefox’ application.

Sayak@ubuntu:~/harry\$ ls

2.txt 56.txt harry.txt HARRY.txt this

sayak@ubuntu:~/harry\$ vim harry.txt # This command will open ‘harry.txt’ file in vim editor.

```
#This [sayak@ubuntu:~/harry$ cd /
sayak@ubuntu:$ ls
bin dev lib libx32 mnt root snap
boot etc lib32 lost+found opt run srv
cdrom home lib64 media proc sbin swapfile
sayak@ubuntu:$ cd etc
sayak@ubuntu:/etc$ ls
acpi machine-id
adduser.conf magic
alsa magic.mime
alternatives mailcap
anacrontab mailcap.order
apg.conf manpath.config
apm mime.types
apparmor mke2fs.conf
apparmor.d ModemManager
apport modprobe.d
apostrophe.conf modules
```

‘cd<space><dir_name>’ command changes the current directory to a new directory named ‘dir_name’.

sayak@ubuntu:/etc\$ cd ..

sayak@ubuntu:\$ # ‘cd ../’ command takes us one step back.

```
sayak@ubuntu:/var$ cd l
lib/ local/ lock/ log/
sayak@ubuntu:/var$ cd lib/
sayak@ubuntu:/var/lib$ cd /var/lib
sayak@ubuntu:/var/lib$
```

write ‘cd

**# If we
l’ and
for two**

times, it will shows all the files in the current directory starting with ‘l’. Then we can choose any of these files. If we already know the full path of our desired file, we can write the full path ‘cd /var/lib’.

sayak@ubuntu:\$ # ‘\$’ sign means normal user.

sayak@ubuntu:# # ‘#’ sign means super/root user.

sayak@ubuntu:~/harry\$ mkdir Harry

sayak@ubuntu:~/harry\$ # This command makes a new sub-directory named ‘Harry’ in the current folder named ‘harry’.

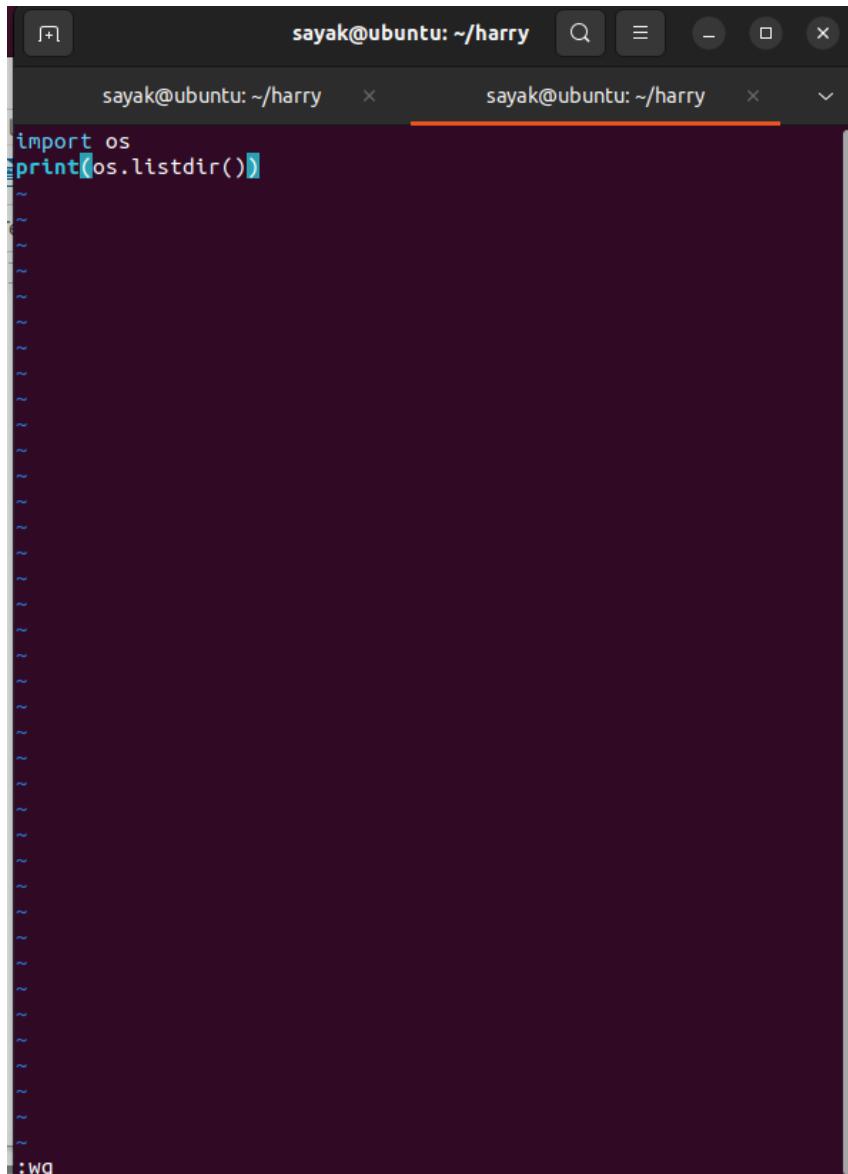
sayak@ubuntu:~/harry\$ rmdir Harry/

sayak@ubuntu:~/harry\$ #‘rmdir’ command removes or deletes the ‘Harry’ named subdirectory.

sayak@ubuntu:~/harry\$ vim main.py # This command opens/creates a python file named ‘main.py’ in vim editor.

Then write the desired python code. After that, press ‘Esc’ and write ‘:wq’ to save and vim editor and go back to the terminal.

If we press ‘:q!’, then it will exit without saving the file(Discard and exit). This is important if the file somehow been corrupted while editing.



The screenshot shows a terminal window with two tabs, both labeled "sayak@ubuntu: ~/harry". The left tab displays the Python code:

```
import os
print(os.listdir())
```

The right tab shows the terminal prompt. At the bottom of the terminal window, the command `:wq` is visible, indicating the user is preparing to save and exit the Vim editor.

```
sayak@ubuntu:~/harry$ ls
2.txt  harry.txt  main-backup.py  this
56.txt  HARRY.txt  main.py
sayak@ubuntu:~/harry$ mv main-backup.py backup.py
sayak@ubuntu:~/harry$ ls
2.txt  56.txt  backup.py  harry.txt  HARRY.txt  main.py  this
sayak@ubuntu:~/harry$
```

#This ‘mv main-backup.py backup.py’ command will move the file within the same folder. That means the file name should be just renamed from ‘main-backup.py’ to ‘backup.py’.

```
sayak@ubuntu:~/harry$ mkdir rohan
```

```
sayak@ubuntu:~/harry$ mv backup.py rohan/backup.py
```

```
sayak@ubuntu:~/harry$ cd rohan/
```

```
sayak@ubuntu:~/harry/rohan$ ls
```

```
backup.py
```

Here, we made a directory named ‘rohan’ and moved ‘backup.py’ in that folder.

```
sayak@ubuntu:~/harry/rohan$ cd ..
sayak@ubuntu:~/harry$ ls
2.txt  56.txt  harry.txt  HARRY.txt  main.py  rohan  this
sayak@ubuntu:~/harry$ rm main.py
sayak@ubuntu:~/harry$ ls
2.txt  56.txt  harry.txt  HARRY.txt  rohan  this
sayak@ubuntu:~/harry$
```

#‘rm’ command used to delete files, not the directories. Here, it removes ‘main.py’ from the folder named ‘rohan’.

```
# sayak@ubuntu:~/harry$ ls
2.txt  56.txt  harry.txt  HARRY.txt  rohan  this
sayak@ubuntu:~/harry$ rm -r rohan/
sayak@ubuntu:~/harry$ ls
2.txt  56.txt  harry.txt  HARRY.txt  this
sayak@ubuntu:~/harry$
```

Using ‘rm -r <directory name>/’ this command you can delete a folder or a directory.

```
# ayak@ubuntu:~/harry$ ls
.txt 56.txt harry.txt HARRY.txt this
ayak@ubuntu:~/harry$ touch rohan.txt
ayak@ubuntu:~/harry$ ls
.txt 56.txt harry.txt HARRY.txt rohan.txt this
```

Using ‘touch’ command you can make a new file.

```
# sayak@ubuntu:~/harry$ ls
2.txt 56.txt harry.txt HARRY.txt rohan.txt this
sayak@ubuntu:~/harry$ vim rohan.txt
sayak@ubuntu:~/harry$ cat rohan.txt
rohan is
a
good boy.
```

Here we opened the file ‘rohan.txt’ in vim text editor and using ‘cat’ command we display its contents in the terminal.

sayak@ubuntu:~/harry\$ man ls #Using this ‘man’ command, you can open the manual which open all the details about a command. Just press ‘q’ to escape from the screen and go back to the terminal.

ls(1)	User Commands	ls(1)
NAME	ls - list directory contents	
SYNOPSIS	ls [OPTION]... [FILE]...	
DESCRIPTION	List information about the FILEs (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor --sort is specified. Mandatory arguments to long options are mandatory for short options too.	
-a, --all	do not ignore entries starting with .	
-A, --almost-all	do not list implied . and ..	
--author	with -l, print the author of each file	
-b, --escape	print C-style escapes for nongraphic characters	
--block-size=SIZE	with -l, scale sizes by SIZE when printing them; e.g., '--block-size=M'; see SIZE format below	
-B, --ignore-backups	do not list implied entries ending with ~	
-c	with -lt: sort by, and show, ctime (time of last modification of file status information); with -l: show ctime and sort by name; otherwise: sort by ctime, newest first	

sayak@ubuntu:~/harry\$ htop # 'htop' is an interactive process between the viewer and system monitor. It displays cpu, monitor etc storage occupied. Press 'q' to escape the screen and return to the terminal. Shown below:

System Resource Usage Summary									
0[]	26.7%	Tasks: 140, 528 thr; 2 runn							
1[5.2%	Load average: 0.40 0.53 0.6							
Mem[1.89G/2.89G	Uptime: 12:47:53							
Swp[443M/2.62G								
PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%\\MEM%	TIME+
14927	sayak	20	0	3011M	365M	74628	S	25.8 12.3	1h03:59
14495	sayak	20	0	3774M	313M	128M	S	2.0 10.6	20:07.09
1769	sayak	20	0	4572M	310M	73340	S	1.3 10.5	43:28.08
14630	sayak	20	0	3774M	313M	128M	S	1.3 10.6	0:48.97
22053	sayak	20	0	20220	5248	3712	R	1.3 0.2	0:00.13
1796	sayak	20	0	4572M	310M	73340	S	0.7 10.5	13:49.19
2097	sayak	20	0	218M	25632	20980	S	0.7 0.8	4:16.51
2259	sayak	20	0	221M	2948	2432	S	0.7 0.1	0:35.08
2496	sayak	20	0	566M	45888	27956	S	0.7 1.5	1:05.72
14623	sayak	20	0	3774M	313M	128M	S	0.7 10.6	1:44.55
14670	sayak	20	0	3774M	313M	128M	S	0.7 10.6	1:03.97
14943	sayak	20	0	3011M	365M	74628	S	0.7 12.3	1:39.06
1	root	20	0	162M	10728	7272	S	0.0 0.4	0:02.90
183	root	19	-1	48352	16128	14592	S	0.0 0.5	0:01.17
238	root	20	0	27064	5980	4188	S	0.0 0.2	0:00.63
508	systemd-o	20	0	14824	6272	5504	R	0.0 0.2	1:12.86
509	systemd-r	20	0	26716	10428	8192	S	0.0 0.3	0:07.24
526	systemd-t	20	0	89376	6784	6016	S	0.0 0.2	0:00.22
533	systemd-t	20	0	89376	6784	6016	S	0.0 0.2	0:00.00
561	root	20	0	242M	7612	6716	S	0.0 0.3	0:01.09
562	root	20	0	2812	1792	1792	S	0.0 0.1	0:01.70
565	avahi	20	0	7624	3712	3328	S	0.0 0.1	0:00.18
566	root	20	0	18148	2816	2688	S	0.0 0.1	0:00.10
567	messagebu	20	0	11088	6144	3712	S	0.0 0.2	0:08.95
568	root	20	0	263M	15832	13272	S	0.0 0.5	0:02.43
588	root	20	0	82696	3840	3584	S	0.0 0.1	0:02.28
589	root	20	0	49712	12160	7680	S	0.0 0.4	0:00.15
590	root	20	0	245M	11096	7536	S	0.0 0.4	0:02.33
591	root	20	0	242M	7296	6656	S	0.0 0.2	0:00.03
592	syslog	20	0	217M	4992	3840	S	0.0 0.2	0:00.17
594	root	20	0	927M	21632	8832	S	0.0 0.7	0:07.51
595	root	20	0	239M	6656	6144	S	0.0 0.2	0:00.03
596	root	20	0	23632	7088	6144	S	0.0 0.2	0:00.34

F1Help F2Setup F3Search F4Filter F5Tree F6SortByF7Nice -F8Nice +

```
# sayak@ubuntu:~/harry$ ls
2.txt 56.txt harry.txt HARRY.txt rohan.txt this
sayak@ubuntu:~/harry$ ls -lart
total 28
-rwxrw-r-- 1 sayak sayak 0 Aug 14 15:25 2.txt
drwxrwxr-x 2 sayak sayak 4096 Aug 14 15:32 this
-rw-rw-r-- 1 sayak sayak 0 Aug 14 15:38 56.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 14 17:53 .harry
-rw-rw-r-- 1 sayak sayak 0 Aug 14 18:31 harry.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 14 18:31 HARRY.txt
-rw-r--r-- 1 sayak sayak 12288 Aug 15 16:51 .harry.txt.swp
-rw-rw-r-- 1 sayak sayak 22 Aug 16 10:24 rohan.txt
drwxrwxr-x 3 sayak sayak 4096 Aug 16 10:24 .
drwxr-x--- 17 sayak sayak 4096 Aug 16 10:30 ..
sayak@ubuntu:~/harry$ chmod 744 this
sayak@ubuntu:~/harry$ ls -lart
total 28
-rwxrw-r-- 1 sayak sayak 0 Aug 14 15:25 2.txt
drwxr--r-- 2 sayak sayak 4096 Aug 14 15:32 this
-rw-rw-r-- 1 sayak sayak 0 Aug 14 15:38 56.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 14 17:53 .harry
-rw-rw-r-- 1 sayak sayak 0 Aug 14 18:31 harry.txt
-rw-rw-r-- 1 sayak sayak 0 Aug 14 18:31 HARRY.txt
-rw-r--r-- 1 sayak sayak 12288 Aug 15 16:51 .harry.txt.swp
-rw-rw-r-- 1 sayak sayak 22 Aug 16 10:24 rohan.txt
drwxrwxr-x 3 sayak sayak 4096 Aug 16 10:24 .
drwxr-x--- 17 sayak sayak 4096 Aug 16 10:30 ..
sayak@ubuntu:~/harry$
```

lart' command lists all files (also the hidden files) with their permissions. Using 'chmod' command user can change the (user-group-public) permission of a file or a folder. Here, you can see, the permission of 'this' folder has been changed.

To change the ownership of a file:

sayak@ubuntu:~\$ sudo apt install apache2 #This software is installed when we want to give permission to the people to visit our websites. Here, user uses the 'sudo' command when he want to use his power of the root user. 'apache2' creates an user named 'www-data'.

```
sayak@ubuntu:~$ ls -lart
total 140
drwxr-xr-x  3 root  root  4096 Aug 11 11:50 ..
-rw-r--r--  1 sayak sayak   807 Aug 11 11:50 .profile
-rw-r--r--  1 sayak sayak  3771 Aug 11 11:50 .bashrc
-rw-r--r--  1 sayak sayak   220 Aug 11 11:50 .bash_logout
drwx-----  3 sayak sayak  4096 Aug 11 12:02 .local
drwxr-xr-x  2 sayak sayak  4096 Aug 11 12:02 Videos
drwxr-xr-x  2 sayak sayak  4096 Aug 11 12:02 Templates
drwxr-xr-x  2 sayak sayak  4096 Aug 11 12:02 Public
drwxr-xr-x  2 sayak sayak  4096 Aug 11 12:02 Music
drwxr-xr-x  2 sayak sayak  4096 Aug 11 12:02 Downloads
drwxr-xr-x  2 sayak sayak  4096 Aug 11 12:02 Desktop
drwx-----  4 sayak sayak  4096 Aug 11 12:06 snap
-rw-r--r--  1 sayak sayak     0 Aug 11 12:48 .sudo_as_admin_su
ccessful
-rw-----  1 sayak sayak 12288 Aug 11 12:52 .swp
drwx-----  2 sayak sayak  4096 Aug 12 14:31 .ssh
drwxr-xr-x  3 sayak sayak  4096 Aug 14 15:35 Pictures
drwx----- 13 sayak sayak  4096 Aug 14 15:37 .cache
-rw-rw-r--  1 sayak sayak     0 Aug 14 16:04 56.txt
-rw-----  1 sayak sayak    20 Aug 16 10:30 .lessht
drwx-----  2 sayak sayak  4096 Aug 16 19:46 .gnupg
drwxrwxr-x  3 sayak sayak  4096 Aug 16 19:47 harry
```

#using ‘ls -lart’ we can see the given permissions of all files and sub-folders in the current folder. Now, let us want to change the ownership of the folder of folder ‘harry’

Using this command, “chown www-data harry” we can change the ownership of the file from ‘sayak’ to ‘www-data’. So ‘ls -lart gives:

```
drwxrwxr-x 3 www-data sayak 4096 Aug 16 19:47 harry
```

“chown www-data:www-data harry” using this command, We can change the group also. So ‘ls -lart gives:

```
drwxrwxr-x 3 www-data www-data 4096 Aug 16 19:47 harry
```

To zip and unzip any file.

```
sayak@ubuntu:~$ touch harry.txt
sayak@ubuntu:~$ gzip harry.txt
sayak@ubuntu:~$ ls
56.txt  Documents  harry      Music      Public    Templates
Desktop  Downloads  harry.txt.gz  Pictures   snap     Videos
sayak@ubuntu:~$ gunzip harry.txt.gz
sayak@ubuntu:~$ ls
56.txt  Documents  harry      Music      Public    Templates
Desktop  Downloads  harry.txt  Pictures   snap     Videos
sayak@ubuntu:~$
```

First we created an empty txt file. Then,

it was zipped using ‘gzip’ command and unzipped by gunzip command.

sayak@ubuntu:~\$ ssh sayak@10.0.2.15 # ‘ssh username@ipaddress’ this command is used to connect to a remote server.

sayak@ubuntu:~\$ scp energy-calc sayak@10.0.2.15:~/home/sayak
‘scp’ is used when we want to copy something from a remote server. The format is scp <file name> <username>@<ip address>:<location where you want to save the file>. (Not sure!! Need to be verified).

```
# sayak@ubuntu:~$ ping google.com
PING google.com (142.250.183.14) 56(84) bytes of data.
^C
--- google.com ping statistics ---
45 packets transmitted, 0 received, 100% packet loss, time 45076ms

sayak@ubuntu:~$
```

‘ping’ command checks the network connectivity. You can cancel this command with ‘crtl+c’.

To display network connection information:

```
sayak@ubuntu:~$ netstat
Command 'netstat' not found, but can be installed with:
sudo apt install net-tools
sayak@ubuntu:~$ sudo apt install net-tools
[sudo] password for sayak:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following NEW packages will be installed:
  net-tools
0 upgraded, 1 newly installed, 0 to remove and 15 not upgraded.
Need to get 204 kB of archives.
After this operation, 819 kB of additional disk space will be used.
Get:1 http://in.archive.ubuntu.com/ubuntu jammy/main amd64 net-tools amd64 1.60+ubuntu5 [204 kB]
Fetched 204 kB in 2s (94.7 kB/s)
Selecting previously unselected package net-tools.
(Reading database ... 170783 files and directories currently installed.)
Preparing to unpack .../net-tools_1.60+git20181103.0eebece-1ubuntu5_amd64.deb ...
Unpacking net-tools (1.60+git20181103.0eebece-1ubuntu5) ...
Setting up net-tools (1.60+git20181103.0eebece-1ubuntu5) ...
Processing triggers for man-db (2.10.2-1) ...
sayak@ubuntu:~$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0  ubuntu:33626            239.237.117.34.bc:https ESTABLISHED
tcp      0      0  ubuntu:49148            ubuntu-mirror-1.ps:http  TIME_WAIT
tcp      0      0  ubuntu:39896            55.65.117.34.bc.g:https ESTABLISHED
tcp      0      0  ubuntu:45340            bom07s35-in-f1.1e:https TIME_WAIT
tcp      0      0  ubuntu:53590            bom07s33-in-f22.1:https TIME_WAIT
tcp      0      0  ubuntu:59010            180.149.55.233:https   ESTABLISHED
tcp      0      0  ubuntu:46286            bom07s45-in-f22.1:https TIME_WAIT
tcp      0      0  ubuntu:41706            180.149.55.233:https   TIME_WAIT
```

#'netstat' command helps To display network connection information.

First it was installed by 'sudo apt install net-tool' command.

```
sayak@ubuntu:~$ ps
  PID TTY          TIME CMD
 8013 pts/0    00:00:00 bash
 9932 pts/0    00:00:00 ps
sayak@ubuntu:~$ ps -ef
UID      PID  PPID   C STIME TTY          TIME CMD
root         1      0  0 14:12 ?        00:00:05 /sbin/init splash
root         2      0  0 14:12 ?        00:00:00 [kthreadd]
root         3      2  0 14:12 ?        00:00:00 [rcu_gp]
root         4      2  0 14:12 ?        00:00:00 [rcu_par_gp]
root         5      2  0 14:12 ?        00:00:00 [slub_flushwq]
root         6      2  0 14:12 ?        00:00:00 [netns]
root         8      2  0 14:12 ?        00:00:00 [kworker/0:0H-events_
root        10      2  0 14:12 ?        00:00:00 [mm_percpu_wq]
root        11      2  0 14:12 ?        00:00:00 [rcu_tasks_kthreadd]
```

To display running processes:

‘ps’ command is used to display the running processes. Using ‘ps -ef’ we can get the detailed information.

```
sayak@ubuntu:~$ ps -ef | grep sbin
root          1      0  0 14:12 ?          00:00:05 /sbin/init splash
root         581      1  0 14:12 ?          00:00:00 /usr/sbin/acpid
root         585      1  0 14:12 ?          00:00:00 /usr/sbin/cron -f -P
root         587      1  0 14:12 ?          00:00:00 /usr/sbin/NetworkManager --no-daemon
root         605      1  0 14:12 ?          00:00:00 /usr/sbin/irqbalance
--foreground
syslog       609      1  0 14:12 ?          00:00:00 /usr/sbin/rsyslogd -n -iNONE
root         617      1  0 14:12 ?          00:00:00 /sbin/wpa_supplicant -u -s -0 /run/wpa_supplicant
root         706      1  0 14:12 ?          00:00:00 /usr/sbin/ModemManager
#
```

if we use ‘ps -ef | grep <some string>’; we can get the specific running processes with that particular string. Here, the string is ‘sbin’.

To kill any current running process:

```

sayak@ubuntu:~$ top

top - 17:51:53 up 3:39, 1 user, load average: 0.79, 0.52, 0.66
Tasks: 226 total, 1 running, 225 sleeping, 0 stopped, 0 zombie
%Cpu(s): 6.9 us, 0.8 sy, 0.0 ni, 92.1 id, 0.0 wa, 0.0 hi, 0.2 si,
MiB Mem : 4910.3 total, 306.7 free, 2255.2 used, 2348.4 buff/cac
MiB Swap: 2680.0 total, 2679.7 free, 0.3 used. 2317.8 avail Me

      PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+
  3148 sayak      20   0 3040408 414344 111116 S 17.9  8.2  38:49.53
  1726 sayak      20   0 4952020 479820 147524 S  5.0  9.5  39:18.51
  2671 sayak      20   0 4071508 517064 215528 S  1.7 10.3 25:23.36
  7840 sayak      20   0 571568 53552 39856 S  1.3  1.1  0:21.05
10424 sayak      20   0 21856 4096 3328 R  1.0  0.1  0:00.28
  2225 sayak      20   0 227000 2972 2560 S  0.7  0.1  0:11.01
  2226 sayak      20   0 293052 3228 2944 S  0.7  0.1  0:31.99
  526 systemd+    20   0 14824 6784 6016 S  0.3  0.1  0:22.66
  2852 sayak      20   0 2477172 151284 85472 S  0.3  3.0  0:11.71
 10429 root       20   0      0      0      0 I  0.3  0.0  0:00.02
  1 root       20   0 167964 13016 8152 S  0.0  0.3  0:05.38

```

```

sayak@ubuntu:~$ kill 3148

```

first we run ‘top’ command to show the current running processes. Then ‘kill <process id /PID>’ this command is used to kill a particular process.

Control system services and settings.

```
sayak@ubuntu:~$ systemctl status apache2
● apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor>
   Active: active (running) since Fri 2023-08-18 18:05:38 IST; 2min 14>
     Docs: https://httpd.apache.org/docs/2.4/
   Process: 11021 ExecStart=/usr/sbin/apachectl start (code=exited, sta>
 Main PID: 11025 (apache2)
    Tasks: 55 (limit: 5794)
   Memory: 5.0M
      CPU: 47ms
     CGroup: /system.slice/apache2.service
             ├─11025 /usr/sbin/apache2 -k start
             ├─11026 /usr/sbin/apache2 -k start
             └─11027 /usr/sbin/apache2 -k start

sayak@ubuntu:~$ systemctl stop apache2
sayak@ubuntu:~$ systemctl start apache2
sayak@ubuntu:~$ systemctl restart apache2
sayak@ubuntu:~$
```

#‘systemctl status apache2’, ‘systemctl stop apache2’, ‘systemctl start apache2’, ‘systemctl restart apache2’ these command are used to check status, to stop, start and restart any services respectively.

To create a new user:

sayak@ubuntu:~\$ sudo useradd harry # User named ‘harry’ added to the partocular system.

[sudo] password for sayak: # Provide the root user’s password.

sayak@ubuntu:~\$ passwd harry # Set password for the new user harry.

sayak@ubuntu:~\$ ssh harry@10.0.2.15

harry@10.0.2.15’s password: # In the way, harry can login to the system from any other device with username, ip address and password.

(Did not work for me!!!!)

```
userdel harry # To delete a user named harry.
```

Switch user:

```
sayak@ubuntu:~$ su harry
Password:
su: Authentication failure
sayak@ubuntu:~$ su root
Password:
root@ubuntu:/home/sayak# ls
56.txt  Documents  harry      Music      Public   Templates
Desktop  Downloads  harry.txt  Pictures  snap     Videos
root@ubuntu:/home/sayak#
```

‘su harry’ command is used to become another user harry. Writing ‘su rrot’ you can return back to the root user.

Display disk space usage:

```
root@ubuntu:/home/sayak# df
Filesystem      1K-blocks    Used Available Use% Mounted on
tmpfs            502820     1552    501268   1% /run
/dev/sda3        25106692  13231584  10574424  56% /
tmpfs            2514088      0    2514088   0% /dev/shm
tmpfs             5120       4      5116   1% /run/lock
/dev/sda2        524252     6216    518036   2% /boot/efi
tmpfs            502816     144    502672   1% /run/user/1000
/dev/sr0          52244     52244           0 100% /media/sayak/VBox_GAs_7.
0.10
root@ubuntu:/home/sayak#
```

‘df’ command is used.

Display disk usage by file or directory;

```
# root@ubuntu:/home/sayak# du
4      ./Templates
4      ./mozilla/extensions
8      ./mozilla
4      ./Public
8      ./config/gtk-3.0
8      ./config/nautlius
12     ./config/dconf
84     ./config/pulse
8      ./config/htop
4      ./config/gnome-session/saved-session
8      ./config/gnome-session
12     ./config/ibus/bus
16     ./config/ibus
16     ./config/evolution/sources
20     ./config/evolution
4      ./config/procps
```

‘du’

command is used here.

To mount a file in a other location

```
sayak@ubuntu:~/harry3$ sudo mount -o loop  
/home/sayak/harry3/def.txt /home/sayak/Documents # Use command  
'sudo mount -o loop <path to your file> <point of mounting>'. (But, did  
not work for me!!!).
```

mount: /home/sayak/Documents: wrong fs type, bad option, bad superblock on /dev/loop12, missing codepage or helper program, or other error.

To see the date:

```
sayak@ubuntu:~$ date
```

Saturday 19 August 2023 11:30:50 AM IST

For example, the following command will run the **top** command in the background and ignore the HUP signal:

```
nohup top &
```

You can also use the nohup command to redirect the output of the command to a file. To do this, use the **> filename** option. For example, the following command will run the **ls** command in the background and redirect the output to the file **nohup.out**:

```
nohup ls > nohup.out &
```

The nohup command is a useful tool for running long-running commands or commands that you need to run even after you have closed the terminal window.

Here are some additional things to keep in mind when using the nohup command:

```
sayak@ubuntu:~/harry/this$ touch 13.txt  
sayak@ubuntu:~/harry/this$ nohup ls > 13.txt  
&  
[1] 36386  
sayak@ubuntu:~/harry/this$ nohup: ignoring i  
nput and redirecting stderr to stdout  
^C  
[1]+ Done nohup ls > 13.  
txt  
sayak@ubuntu:~/harry/this$ cat 13.txt  
13.txt  
1.txt  
2.txt  
3.txt  
4.txt  
archive_name.tar.gz  
hard_link.txt  
myarchive.a  
soft_link.txt  
soham  
sayak@ubuntu:~/harry/this$ █
```

- The nohup command will only ignore the HUP signal. Other signals, such as the **SIGKILL** signal, will still terminate the process.
- The nohup command does not prevent the process from being killed by the system administrator.
- The nohup command can be used to run commands in the background, but it does not create a new process group. This means that **the process will still be affected by signals sent to the foreground process group.**

• **reboot: Reboot the system**

`reboot [options]`

- **-f:** Forces the system to reboot even if there are unsaved changes.
- **-i:** Reboots the system into single-user mode.
- **-h:** Displays help information about the `reboot` command.

Environment and Variables

• **set: Assign values to positional parameters**

Positional parameters are variables that store values provided to a script or function when it is executed. **They are referred to as positional because their values are determined by the position of arguments passed to the script on the command line.** Here are some key points to understand:

1. Accessing Positional Parameters:

- Positional parameters are accessed using special variables like `$1`, `$2`, `$3`, and so on.
- `$1` represents the first positional parameter, `$2` represents the second, and so forth.
- The value of a positional parameter is the argument passed at that position on the command line when the script is executed.

2. Setting Positional Parameters with `set`:

- The `set` command is used to assign values to positional parameters.
- You can set the values of positional parameters by providing them as arguments to the `set` command. For example:

bash

- `set "John" "Doe" "30"`

In this case, "John" is assigned to \$1, "Doe" to \$2, and "30" to \$3.

- **Usage in Scripts:**

- Positional parameters are commonly used in shell scripts to process input arguments.
- They allow you to work with data passed to the script, such as filenames, user inputs, or any other information you need to process.

- **Example Script:**

- Here's an example of a simple script that uses positional parameters to display a greeting:

bash

- `#!/bin/bash`

```
# Assign values to
# positional parameters
set "John" "Doe" "30"
```

```
sayak@ubuntu:~/harry>this$ cat positional_pa
rameters.sh
#!/bin/bash

# Assign values to positional parameters
set "John" "Doe" "30"

# Access and use positional parameters
echo "Hello, $1 $2! You are $3 years old."

sayak@ubuntu:~/harry>this$ bash positional_p
arameters.sh
Hello, John Doe! You are 30 years old.
sayak@ubuntu:~/harry>this$
```

```
# Access and use positional parameters
echo "Hello, $1 $2! You are $3 years old."
```

When you run this script, it will output: "Hello, John Doe! You are 30 years old."

2. Temporary Nature:

- It's important to note that positional parameters are temporary within the script's execution. They are set when the script is run and can be accessed within the script.
- Once the script exits, the values of positional parameters are lost, and they are not stored as permanent variables.

3. Argument Count:

- The total number of positional parameters is represented by `$#`. For example, if you pass four arguments to a script, `$#` will be 4.

Positional parameters are a fundamental concept in shell scripting and are widely used for tasks like processing command-line arguments, performing data manipulation, and interacting with users. They provide a convenient way to work with input values in your scripts.

• **export: Pass variable value to sub-shell**

In Bash, you can use the `export` command to pass a variable's value to a sub-shell and make it available to subprocesses. When you export a variable, it becomes part of the environment of the sub-shell. Here's how you can do it:

1. Exporting a variable:

```
sayak@ubuntu:~/harry$ vim 14.sh
sayak@ubuntu:~/harry$ cat 14.sh
# Assign a value to a variable
VAR_NAME="value"

# Export the variable
export VAR_NAME

# Run a sub-shell
(echo "Sub-shell: \$VAR_NAME is $VAR_NAME")

sayak@ubuntu:~/harry$ bash 14.sh
Sub-shell: $VAR_NAME is value
sayak@ubuntu:~/harry$
```

In this example, `export VAR_NAME` makes the `VAR_NAME` variable available to the sub-shell, and you can access it using `$VAR_NAME` within the sub-shell.

• Verify the variable in a subprocess (child process):

You can also verify that the variable is available to subprocesses (child processes) by running a script or command from the sub-shell:

bash

```
2.      # Run a sub-shell  
  
(echo "Sub-shell: \$VAR_NAME is $VAR_NAME")
```

```
# Run a command in the sub-shell  
  
(echo "Sub-shell: Executing a command")  
  
../some_script.sh # Replace with the actual script or  
command you want to run
```

Any script or command executed from the sub-shell will inherit the exported environment variable.

Just keep in mind that when you export a variable, it becomes part of the environment of the sub-shell and its subprocesses. If you want to pass the value to a completely independent sub-shell (not just a subprocess), you can use command substitution or process substitution, as mentioned in the previous response.

- **Persistence:** Environment variables set using `export` persist for the duration of the current shell session. If you close the shell or start a new session, the environment variables won't be available unless you set them again.
- **Unsetting Environment Variables:** You can unset (remove) an environment variable using the `unset` command. Once unset, the variable will no longer be available to child processes.
- `bash`
- `unset VAR_NAME`

Input and Output

• **read: Input data to a shell script interactively**

The `read` command is a built-in utility in Bash that reads text from standard input. It can be used to read user input in an interactive shell script. The general syntax for the `read` command is:

```
read [options] variable
```

Some of the most commonly used options for the `read` command are:

- `-p`: Prompt the user for input. The prompt text is specified after the `-p` option.
- `-s`: Read the input silently. This means that the user's input will not be echoed to the screen.
- `-t`: Specify a timeout for the read operation. If the user does not enter any input within the timeout period, the `read` command will return an error.

For example, the following shell script will prompt the user for their name and store it in the variable `name`:

```
sayak@ubuntu:~/harry$ vim 15.sh
sayak@ubuntu:~/harry$ cat 15.sh
read -p "Enter your name: " name
echo "Hello, $name!"
sayak@ubuntu:~/harry$ bash 15.sh
Enter your name: Sayak
Hello, Sayak!
sayak@ubuntu:~/harry$ █
```

If the file is in non-executable form, use command, `chmod +x 15.sh`

Then, you could run the script by typing the following command:

```
./15.sh or bash 15.sh
```

Here's an example of using the `-s` option with the `read` command to securely read a password from the user:

In this script:

1. The `-s` option is used with `read` to suppress the display of the entered password.
2. `-p` is used with `read` to provide a custom prompt for the user.
3. After the user enters the password, the script displays a message indicating that the password was entered (the `-e` option for `echo` is used to interpret escape sequences like `\n` for a newline).

```
sayak@ubuntu:~/harry$ vim 16.sh
sayak@ubuntu:~/harry$ cat 16.sh
#!/bin/bash

# Prompt the user for a password without displaying it on the screen
read -s -p "Enter your password: " password

# Display a message indicating that the password was entered
echo -e "\nPassword entered.

# Now you can use the 'password' variable for further processing
# For example, you might use it to authenticate the user or perform other actions.

sayak@ubuntu:~/harry$ bash 16.sh
Enter your password:
Password entered.
sayak@ubuntu:~/harry$
```

In Bash, the `read` command with the `-p` option is used to display a prompt to the user, and `-t` option is used to set timeout for reading input. (# In this case, I did not provide any input did within the specified 10 s time, so variable \$name was empty in the first case.)

```
sayak@ubuntu:~/harry$ cat 15.sh
read -p "Enter your name: " -t 10 name
echo "Hello, $name!"
sayak@ubuntu:~/harry$ bash 15.sh
Enter your name: Hello, !
sayak@ubuntu:~/harry$ bash 15.sh
Enter your name: Sayak
Hello, Sayak!
sayak@ubuntu:~/harry$
```

• **paste:** files

```
sayak@ubuntu:~/harry/files_module2_l1$ paste
fruit.txt some_more_fruits
This is a sample file with some fruits. apple
I like fruits. Who does not like fruits?
Fruits are delicious. banana

cherry

date

elderberry

fig
```

Join two laterally

System Date and Time

- date
- **Current date: date +"%Y-%m-%d"**

The "+" character is used to tell the `date` command to format the output according to the specified format string.

- **Current time: date +"%H:%M:%S"**

- **Future date:
`date -d "+3 days" +"%Y-%m-%d"`**

The `-d` option is used to specify a date. In this case, the date is "3 days", which means that the date is 3 days after the current date.

```
sayak@ubuntu:~/harry/files_module2_l1$ date
-d "+3 days" +"%Y-%m-%d"
2023-09-05
sayak@ubuntu:~/harry/files_module2_l1$ date
-d "+3 days"
Tuesday 05 September 2023 10:33:34 AM IST
sayak@ubuntu:~/harry/files_module2_l1$ date
-d +"%Y-%m-%d"
date: invalid date '+%Y-%m-%d'
sayak@ubuntu:~/harry/files_module2_l1$ date
+"%Y-%m-%d"
2023-09-02
sayak@ubuntu:~/harry/files_module2_l1$ date
Saturday 02 September 2023 10:36:14 AM IST
```

The "+" character is used to tell the `date` command to format the output according to the specified format string. In this case, the format string is "%Y-%m-%d"

- Display year: `date +%Y`
- `start_date="2023-01-01"`
- `end_date=$(date +"%Y-%m-%d")`
- `diff=$(((date -d "$end_date" +%s) - $(date -d "$start_date" +%s)))`

+%s)) / 86400))

echo "Days between \$start_date and \$end_date: \$diff" (Not working!!)

Table of Contents

L2-vim.pptx.pdf	1
vi -O file1 file2:	1
vi -o file1 file2:	2
Getting started.....	2
vi/vim - open vim.....	2
:w bar - Save the file as "bar" (similar to "Save As" in Windows).....	3

L2-vim.pptx.pdf

vi -O file1 file2:

Open multiple files ("file1" and "file2") in vertical split mode.

```
sayak@ubuntu: ~/harry/this
```

```
Indian Institute of technology #&! iit
Directors name:
VK Tiwari
Department 344
Hydraulics $#2389021
and & Water~~~~
resources
Enginnering Sayak
sayak IIT
234567890

StudenT
[ student pHD
PhD
Doctor of Philosophy
Kharagpur
Kharagpur
KARMAKAR
answer
Answer
& YOU tube
Kharagpur
Enginnering Sayak
Enginnering Sayak
we are working
reply to me
sayan

Sayak
Harry Sujoy 456
12 kaaaaaaaaamamjkanlkkjkyu
%rounghjugfyhtytssak
hmataarsh 89
57 ananda
-&7 nayan
Sagardmecae sharma
Nayae kumar
Subarna lekha
deepen
aaaaaaaaaaaaaaaaayan
swert
poeeeeeeeeeeeeeeeeeeev
kjhjjjjjjjjjjjjjjjjjjjsksl
:wq
~
```

```

sayak@ubuntu: ~/harry/this
+1
sayak@ubuntu: ~/harry/this
Indian Institute
of technology #&! iit
Directors name:
VK Tiwari
Department 344
Hydraulics $#2389021
and & Water~~~~
resources
Enginnering Sayak
sayak    IIT
234567890

StudenT
[ student PhD
PhD
Doctor of Philosophy
Kharagpur
Kharagpur
KARMAKAR
answer
1.txt [RO]
Sayak
Harry Sujoy 456
12 kaaaaaaaaamamjkanlkkjkyy
%rounghjugfyhtytssak
hmataarsh 89
57 ananda
-&7 nayan
Sagardmecae sharma
Nayae kumar
Subarna lekha
deepen
aaaaaaaaaaaaaaaaayyan
swert
poeeeeeeeeeeeeeeeeeeeeev
kjhhjjjjjjjjjjjjjjjjjsksl
:wq

~
~
~
2.txt

```

vi -o file1 file2:

Open files in horizontal split mode.

Getting started

- i - Insert Mode
- Transitions Vim to the insert mode for text input.
- Esc - Normal Mode
- Returns Vim to the normal mode
- :w - Write
- Saves changes made to the file.
- :q - Quit
- Exits Vim.
- :wq - Write and Quit
- Saves changes and exits Vim.

vi/vim - open vim

- vi +100 foo: Open the file "foo" and position the cursor at line 100.

- vi +/pat foo: Open the file "foo" and position the cursor at the first occurrence of the pattern "pat."
- vi + foo: Open the file "foo" and position the cursor at the end of the file.
- vi -R foo: Open the file "foo" in read-only mode, preventing any changes to the file.
- vi -O file1 file2: Open multiple files ("file1" and "file2") in vertical split mode.
- vi -o file1 file2: Open files in horizontal split mode.

vi/vim - Saving and Quitting

- :w - Save file and remain in editing mode.
- :w bar - Save the file as "bar" (similar to "Save As" in Windows).
- :w! bar - Overwrite the existing file "bar" without confirmation.
- :n1,n2w foo - Write lines from n1 to n2 to the file named "foo".
- :n1,n2w>> foo - Append lines from n1 to n2 to the file named "foo".
- :x - Save the file and quit editing mode (same as :wq).
- :q - Quit editing mode when no changes are made to the file.
- :q! - Quit editing mode and abandon changes without confirmation

SA

:w bar - Save the file as "bar" (similar to "Save As" in Windows).

In the Vim text editor, the :w command is used to save the current file, while **:w filename** is used to save the current file with a new name, effectively creating a copy of the file with the specified name.

The syntax for saving a file with a new name in Vim is as follows:

bash

:w new_filename

Here's how it works:

1. Open the file you want to save in Vim.
2. Make any changes to the file if needed.

3. To save the file with a new name, use the :w command followed by the desired new filename. For example, if you want to save the file as "bar":

bash

3. :w bar

This command will save the current content of the file with the new name "bar."

```
# sayak@ubuntu:~/harry>this$ ls
1.txt      2.txt      hard_link.txt  soft_link.txt
1.txt.bz2  archive_name.tar.gz  myarchive.a  soham
sayak@ubuntu:~/harry>this$ vim 1.txt
sayak@ubuntu:~/harry>this$ ls
1_new.txt  1.txt.bz2  archive_name.tar.gz  myarchive.a  soham
1.txt      2.txt      hard_link.txt      soft_link.txt
sayak@ubuntu:~/harry>this$ █
```

new file named 1_new.txt has been created after exiting with command :w
1_new.txt

Keep in mind that this action is similar to using "Save As" in other text editors, as it saves the content of the file under a new name. The original file's content remains unchanged, and a new file is created with the specified name.

SA

:w! bar - Overwrite the existing file "bar" without confirmation

In the Vim text editor, the :w! command is used to force-save the current file, even if the file is read-only or if there are unsaved changes. When you add a filename after :w!, it indicates that you want to overwrite an existing file with the content of the current file, without asking for confirmation.

The syntax for overwriting an existing file without confirmation is as follows:

bash

:w! filename

Here's how it works:

1. Open the file you want to save in Vim.
2. Make any changes to the file if needed.
3. To overwrite an existing file without confirmation, use the :w! command followed by the filename you want to overwrite. For example, if you want to overwrite the file "bar":

bash

3. :w! bar

This command will overwrite the existing "bar" file with the content of the current file, even if the "bar" file is read-only or if there are unsaved changes.

Be cautious when using the :w! command, as it will overwrite the existing file without any confirmation. Make sure you're using the command intentionally and understand the consequences.

```
sayak@ubuntu:~/harry/this$ cat 2.txt
Sayak
Harry Sujoy 456
12 kaaaaaaaaamamjkanlkkjkkyu
%rounghjugfyhtyttssak
hmataarsh 89
57 ananda

sayak@ubuntu:~/harry/this$ vim 1.txt
sayak@ubuntu:~/harry/this$ cat 2.txt
i - Insert Mode
• Transitions Vim to the insert mode for text input.
• Esc - Normal Mode
• Returns Vim to the normal mode
• :w - Write
• Saves changes made to the file.
• :q - Quit
• Exits Vim.
• :wq - Write and Quit
• Saves changes and exits Vim.
sayak@ubuntu:~/harry/this$ █
```

#The content of “2.txt” file has beeен changed with content of “1.txt” when we wrote “:w! 2.txt” in the command line of “1.txt” in vim text editor.

:n1,n2w foo - Write lines from n1 to n2 to the file named "foo".

In the Vim text editor, the `:n1, n2w filename` command is used to write a range of lines from the current buffer to a file named "filename".

The lines specified by the range **n1** to **n2** (inclusive) will be written to the specified file.

Here's how the command works:

1. Open the file in Vim that contains the lines you want to write to another file.
2. Press `:` to enter command-line mode.
3. Enter the range of lines you want to write followed by the `w` command and the filename. For example, if you want to write lines 5 to 10 to a file named "output.txt":

bash

3. **`:5,10w output.txt`**

This command will write lines 5 to 10 (inclusive) to the file "output.txt".

Keep in mind that the range **n1, n2** specifies the line numbers within the current buffer. The lines within the specified range will be written to the file, and the original buffer content remains unchanged.

```
sayak@ubuntu:~/harry/this$ cat 3.txt
Sayak
Harry Sujoy 456
12 kaaaaaaamamjkanlkkjkyu
%rounghjugfyhtyttssak
hmataarsh 89
57 ananda

sayak@ubuntu:~/harry/this$ vim 3.txt
sayak@ubuntu:~/harry/this$ cat 5.txt
12 kaaaaaaamamjkanlkkjkyu
%rounghjugfyhtyttssak
hmataarsh 89
```

This command is useful when you want to extract specific lines from a file and save them to a **new file**. (In my case, I have opened “3.txt” file in the

vim text editor and wrote **:3,5w 5.txt** in the command line. Then a new file named '5.txt' was created).

SA

:n1,n2w>> foo - Append lines from n1 to n2 to the file named "foo".

In the Vim text editor, the **:n1, n2w>> filename** command is used to append a range of lines from the current buffer to the **end** of a file named "filename" (the file is not necessarily to be a new file, it may be an existing file). The lines specified by the range **n1** to **n2** (inclusive) will be appended to the specified file.

Here's how the command works:

1. Open the file in Vim that contains the lines you want to append to another file.
2. Press **:** to enter command-line mode.
3. Enter the range of lines you want to append followed by the **w** command, **>>** to indicate appending, and the filename. For example, if you want to append lines 5 to 10 to a file named "output.txt":

bash

3. **:5,10w>> output.txt**

This command will append lines 5 to 10 (inclusive) to the end of the file "output.txt".

Keep in mind that the range **n1, n2** specifies the line numbers within the current buffer. The lines within the specified range will be appended to the file, and the original buffer content remains unchanged.

This command is useful when you want to add specific lines from a file to the end of another file without affecting the original buffer content.

vi/vim - Copying

- yy: Copy the current line.
- 6yy: Copy **next** six lines.
- y: Copy the selected region (Vim only).
- p: Paste the copied text on the right.
- P: Paste the copied text on the left or above.

SA

yy: Copy the current line.

In the Vim text editor, the yy command is used to copy the current line. This is also known as the "yank" command, which is Vim's terminology for copying text.

Here's how the yy command works:

1. Open a file in Vim.
2. Move the cursor to the line you want to copy.
3. Press yy.

(Remember, while pressing 'yy' you should be in normal mode; not in insert mode).

After pressing yy, the current line will be copied into Vim's internal register, allowing you to paste it elsewhere in the file using the p command (# I noticed right click+ paste does not work) (for paste) or other commands.

For example, if you want to copy line 5:

1. Move the cursor to line 5.
2. Press yy.

To paste the copied line, move the cursor to the desired location and press p.

Keep in mind that you can also specify a range of lines to copy. For example, “:2,5y” would copy lines 2 to 5. (it should be written in ‘command line’ in vim.)

Vim's copy (yank) and paste commands are powerful and versatile tools for manipulating text within the editor.

SA

y: Copy the selected region (Vim only).

In the Vim text editor, the `y` command is used to copy the selected region of text when you are in Visual mode. Visual mode allows you to select text by highlighting it, and then you can perform various operations on the selected text, such as copying, cutting, or deleting.

Here's how the `y` command works in Visual mode:

1. Open a file in Vim.
2. Move the cursor to the starting position of the text you want to select.
3. Enter Visual mode by pressing `v` (for character-wise selection), `V` (for line-wise selection), or `Ctrl + v` (for block-wise selection).
4. Move the cursor to highlight the region of text you want to copy. (# the cursor should be moved by right and left arrows in the keyboard, not by the mouse).
5. Press `y`.

After pressing `y`, the selected text will be copied into Vim's internal register, allowing you to paste it elsewhere in the file using the `p` command (for paste) or other commands.

For example, to copy a block of text using block-wise Visual mode:

1. Move the cursor to the starting position of the block.
2. Press `Ctrl + v` to enter block-wise Visual mode.
3. Use the arrow keys to highlight the desired block.
4. Press `y`.

To paste the copied text, move the cursor to the desired location and press `p`.

Visual mode in Vim provides a flexible way to select and manipulate text within the editor.

vi/vim - Deleting

- x: Delete the character under the cursor.
- 6x: Delete the character under the cursor and the five characters to the right. (# Remember, everything will be done under visual mode, not insert or editing mode).
- X: Delete the previous character (to the left of the cursor).
- dd: Delete the current line.
- 4dd: Delete four lines from the current line.
- dw: Delete the word under the cursor.
- d: Delete the selected region (Vim only).
- J: Join the current line with the next line.
- kJ: Join the current line with the previous line.

vi/vim - Insert text

- i: Insert text to the left of the cursor.
- I: Insert text at the beginning of the line.
- [Ctrl-v][Ctrl-b]: Insert the text "[Ctrl-b]" (literal control characters).
- [Ctrl-v][Esc]: Insert the Escape key (literal escape character).
- a: Append text to the right of the cursor.
- A: Append text at the end of the line. => These are two ways to enter from 'normal mode' to 'insert mode'.
- o: Open a new line below the current line and enter insert mode.
- O: Open a new line above the current line and enter insert mode.

[Ctrl-v][Ctrl-b]: Insert the text "[Ctrl-b]" (literal control characters).

(What is the utility of this command!!??)

In Vim, if you want to insert literal control characters, you can use the following steps:

1. Open the file in Vim.

2. Enter insert mode by pressing **i**.
3. Press **Ctrl + v** followed by the control character you want to insert.
For example, to insert the literal control character **Ctrl + B**, press **Ctrl + v** and then **Ctrl + b**.
4. Continue editing or press **Esc** to exit insert mode.

Using the **Ctrl + v** combination followed by a control character allows you to insert that control character in its literal form.

For instance, to insert the text "[Ctrl-b]" (representing the literal **Ctrl + B** control character), follow these steps:

1. Open the file in Vim.
2. Press **i** to enter insert mode.
3. Press **Ctrl + v** followed by **Ctrl + b**.
4. Type the remaining characters: **[Ctrl-b]**.
5. Press **Esc** to exit insert mode.

This will insert the text "[Ctrl-b]" in the file, where the "[Ctrl-b]" text represents the literal **Ctrl + B** control character.

vi/vim - Interface to Linux

• **:!cmd**

Run a UNIX command cmd.

- Example: **:!ls** - Execute the ls command in the shell. (# First, press 'esc' and then type this command in the command line of vim.)

• **:!%**

- Execute the current file as a shell or Perl script.

• **:r !cmd -**

Read in the output of command cmd below the current line.

Example: **:r !head -n 3 foo** - Read the first three lines of "foo" below the current line.

:r !cat 5.txt - this command displays the content of file named '5.txt' in the vim interface only.

- **:sh or [Ctrl-z]** - Escape to the UNIX shell temporarily (suspend editor)

SA

:sh or [Ctrl-z] - Escape to the UNIX shell temporarily (suspend editor).

In Vim, you can temporarily escape to the UNIX shell using the **:sh** command or the **[Ctrl-z]** key combination. This allows you to access the shell environment and execute commands, and you can return to the Vim session afterward.

Here's how both methods work:

1. Using :sh Command:

In normal mode, you can enter the **:sh** command to temporarily escape to the shell. This opens a new shell session within your terminal. You can execute shell commands, navigate the file system, and perform other shell tasks.

To return to Vim, simply type **exit** or **Ctrl-d** to exit the shell session.

vim

- **:sh**

• Using [Ctrl-z] Key Combination:

In normal mode, you can press **[Ctrl-z]** to temporarily suspend Vim and return to your shell prompt. This effectively pauses the Vim session and returns you to the terminal. You can then execute shell commands as you normally would.

To return to the suspended Vim session, simply type the **fg** command (short for "foreground") and press Enter. This will bring Vim back to the foreground.

bash

2. [Ctrl-z]

fg

Both methods allow you to access the shell environment without fully exiting Vim. This can be useful for performing quick tasks in the shell before returning to your editing session. Keep in mind that while you're in the shell, your Vim session is effectively paused, and any changes made outside of Vim will not be reflected until you return to Vim and refresh the content.

vi/vim - Undo and Redo

- **u** - Undo last editing command
- **[Ctrl-r]** - Redo last undo (vim only)
- **U** - Undo all changes made to current **line**.

All this things are done in ‘normal’ mode, not in ‘insert’ mode.

vi/vim - Substitution

:[range]s/pattern/replacement/[flags]

- [range]: Specifies the range of lines to perform the substitution. If omitted, the substitution will be applied to the current line only.
- pattern: The text pattern to search for (can be a regular expression).
- replacement: The text to replace the matched pattern.
- flags: Optional flags that modify the behavior of the substitution.

Examples

1,\$s/s1/s2/g - Replace string "s1" with "s2" globally in all lines from 1 to the end. (# “\$” sign means this operation is applicable for the total document and “g” means it will be done globally without any confirmation message).

1,\$s/s1/s2/gc - Interactive replacement - asks for confirmation for each match. [#“replace with an (y/n/a/q/l/^E/^Y)?” this type of confirmation message will come and you have to change every word manually by clicking ‘y’ each time.]

[range]: Specifies the range of lines to perform the substitution. If omitted, the substitution will be applied to the current line only.

The text you provided seems to describe the usage of a range in the context of the substitution command in Vim. Let's break down the explanation:

In Vim, the "substitution" refers to the process of replacing a specific pattern with another text. This is often done using the :s command (substitute command). The [range] mentioned in your description specifies the range of lines to which the substitution will be applied.

Here's how it works:

- 1. Open a File in Vim:** Open the file you want to edit in Vim.
- 2. Enter Command-Line Mode:** Press : while in normal mode to enter command-line mode.
- 3. Specify the Substitution:** Use the :s command to perform the substitution. The basic syntax is as follows:

vim

:[range]s/pattern/replacement/[flags]

- [range]: Specifies the range of lines where the substitution will be applied. If omitted, the substitution will be applied to the current line only.
- pattern: The pattern you want to search for and replace.
- replacement: The text you want to replace the pattern with.
- [flags]: Optional flags that modify the substitution behavior.

For example, to replace the word "old" with "new" on the current line, you would use:

vim

:s/old/new/g

To perform the substitution on a specific range of lines, specify the range before the **s** command. For example, to replace "old" with "new" on lines 5 to 10:

```
vim
```

3. **:5,10s/old/new/g**

4. **Press Enter:** After entering the substitution command, press Enter to execute it.

```
• X: Delete the previous character (to the left of the cursor).  
• dd: Delete the current line  
• 4dd: Delete four curves from the current line.  
• dw: Delete the word under the cursor.  
• d: Delete the selected region (Vim only).  
• J: Join the current curve with the next line.  
kJ: Join the current curve with the previous line.  
~  
~  
:30,37s/an/the/g█
```

```
• X: Delete the previous character (to the left of the cursor).  
• dd: Delete the current line  
• 4dd: Delete four curves from the current line.  
• dw: Delete the word under the cursor.  
• d: Delete the selected region (Vim only).  
• J: Join the current curve with the next line.  
kJ: Join the current curve with the previous line.  
~  
~
```

```
13 substitutions on 8 lines          37,2          All
```

Vim's substitution functionality is quite powerful and flexible. The [range] allows you to apply substitutions to specific lines or even the entire file, depending on your needs. You can also use flags to control the behavior of the substitution, such as case-insensitivity or global substitution on each line.

vi/vim - Search and Repeat

- **/pat:** Non-incremental search forward for the string "pat." (# It means ,if you press 'n' it will go to the next word).
- **?pat:** Non-incremental search backward for the string "pat."(# It means ,if you press 'n' it will go to the previous word).

- **n**: Repeat the last string search in the same/forward direction.
- **N**: Repeat the last string search in the opposite/backward direction.
- **:set ic**: Ignore case when searching.

SA

/pat: Non-incremental search forward for the string "pat."

In Vim, the `/pat` command is used for a **non-incremental forward search**. This means you can search for a specific pattern or string in the text, and Vim will locate the next occurrence of that pattern in the direction you specify.

Here's how it works:

- 1. Open a File in Vim:** Open the file you want to search in Vim.
- 2. Press /:** Press the `/` key while in normal mode to enter the search mode.
- 3. Type the Pattern:** After pressing `/`, type the pattern or string you want to search for. For example, if you want to search for the string "pat," you would type:

`vim`

3. **/pat**

4. **Press Enter:** After typing the pattern, press Enter. Vim will move the cursor to the next occurrence of the pattern in the forward direction.
- 5. Press n to Repeat:** If you want to find the next occurrence of the same pattern, you can press `n`. Pressing `n` again will move to the next occurrence, and so on.
- 6. Press N to Reverse:** To move in the opposite direction and find the previous occurrence of the pattern, you can press `N`.

This type of search is non-incremental, which means that you type the entire pattern you want to search for before the search begins. Contrast this

with incremental search (triggered by pressing ? instead of /), where you can see the search results updating as you type.

Vim's search commands are a fundamental part of its text manipulation capabilities, allowing you to quickly locate and navigate through the text.

SA

:set ic: Ignore case when searching

In Vim, the :set ic command is used to enable case-insensitive searching. When this option is set, Vim will perform searches without considering the case of the letters. This means that search results will match regardless of whether the letters are uppercase or lowercase.

Here's how it works:

- 1. Open a File in Vim:** Open the file you want to search in Vim.
- 2. Enter Command-Line Mode:** Press : while in normal mode to enter command-line mode.
- 3. Enable Case-Insensitive Searching:** Type the following command and press Enter:

vim

- **:set ic**

This command enables case-insensitive searching for the current session. You won't see any output after executing the command.

- **Perform a Search:** After enabling case-insensitive searching, you can perform a search as usual. For example, if you're looking for the word "example," you can enter:

vim

- 4. /example

Vim will match occurrences of "example" regardless of their case (e.g., "Example," "eXAmple," etc.).

To disable case-insensitive searching and return to the default behavior, you can use the `:set noic` command:

`vim`

:set noic

Vim provides various options and settings to customize your editing experience, including searching. The case-insensitive search option can be especially useful when you want to locate text without being concerned about letter casing.

Regular expressions

Table of Contents

Regular expressions.....	1
Regular expressions in text processing.....	1

Regular expressions in text processing

Regular expressions are powerful tools for pattern matching and text processing used in various editors and filters, such as:

- Vi/Vim
- Emacs
- grep and egrep
- sed
- awk
- perl

• *: Zero or more occurrences of the previous character.

- go*l: Matches "gl", "gol", "gool", "goool", and so on.
- 12*34: Matches "1234", "12334", "123334", and so on.
- g*: Nothing or "g," "gg," "ggg," etc.
- go*g: Matches "gg", "gog", "goog", "gooooog", and so on.
- 12*g: Matches "12g", "12gg", "12ggg", and so on.

SA

How to execute these regular expressions?

To execute regular expressions and search for patterns in text using the * quantifier, you typically use tools or text editors that support regular

expressions. One common tool that supports regular expressions is the **grep** command-line utility. Additionally, text editors like Vim, Notepad+, and others also support regular expressions for searching and manipulating text.

Here's how you can use regular expressions with the **grep** command and in the Vim text editor:

Using grep Command:

The **grep** command is used to search for patterns in files. To use regular expressions with **grep**, you need to use the **-E** flag to enable extended regular expressions. Here's an example:

bash

```
grep -E 'ab*' file.txt
```

```
sayak@ubuntu:~/harry/this$ grep -E 'zo*k' 3.txt
: Insert text to the left of the cursor to the left of the
cursor.zk how are you now zook jfkjri zoooook ghtyrt ortyv zk
sayak@ubuntu:~/harry/this$ █
```

In
this

example, replace '**ab***' with the regular expression you want to use, and **file.txt** with the name of the file you want to search in.

Using Vim Text Editor:

In the Vim text editor, you can use regular expressions when searching by using the **/** command followed by the regular expression.[# For example: To execute regular expression “zo*k”, First, press ‘esc’ to go to normal mode. Then, type “/zo*k” and press ‘enter’. Then press “n” to go to each word individually.] For example:

vim

/ab*

This will search for the pattern **ab*** in the current file. You can also use regular expressions with the **:s** command for substitutions and other text manipulation tasks.

Remember that the exact syntax and behavior of regular expressions can vary between tools and text editors. It's a good practice to refer to the documentation of the specific tool or editor you are using to understand how to use regular expressions effectively.

• .: A single character.

- **c.t**: Matches "cat", "cet", "cft", but not "ct" or "catt".
- **a.b**: Matches "abb", "acb", "aab", but not "ab" or "aabb".

[# All the commands should be written in the command box with a "/" in front of them. For example: /c.t]

• .*: Nothing or any number of characters.

- **a.*b**: Matches "ab", "aabb", "aXb", "aYb", "aZb", and so on.
- **go.*d**: Matches "god", "good", "oooooooood", "goXd", "goYd", and so on.
- **[abc]**: Matches "a," "b," or "c."
- **[aeiou]**: Matches any lowercase vowel.
- **[123]**: Matches "1," "2," or "3."
- **[1-3]**: Matches a digit between 1 and 3. (Write **/[1-3]** in the command box in vim and continue to press 'n'.)
- **[4-6]**: Matches "4," "5," or "6."
- **[0-9]**: Matches any digit.

SA

go.*d: Matches "god", "good", "oooooooood", "goXd", "goYd", and so on.

The regular expression go . *d matches a pattern in text that starts with "go," followed by zero or more occurrences of any character (.), and ends with "d."

Here's how the components of the regular expression work:

- go: This part of the pattern matches the literal characters "go."
- . *: This is a quantifier that matches zero or more occurrences of any character (except newline). The . matches any single character, and

- * indicates that the preceding element (the dot . in this case) can occur zero or more times.
- d: This part of the pattern matches the literal character "d."

Using the examples you provided:

- "god" matches because it starts with "go" and ends with "d."
- "good" matches for the same reason.
- "oooooooood" matches because it starts with "go," contains multiple occurrences of "o," and ends with "d."
- "goXd" matches because it starts with "go" and ends with "d," and there is a character between "go" and "d."
- "goYd" matches for the same reason.

This regular expression can be used with various tools and programming languages that support regular expressions for searching and pattern matching. Remember that the . * quantifier is greedy, meaning it will match as many characters as possible while still allowing the rest of the pattern to match. If you want a non-greedy match, you can use . * ?.

[^Z]: Any character except "Z."

- [^aeiou]: Matches any consonant.
- [^0-9]: Matches any non-digit character.
- [^a-zA-Z]: A non-alphabetic character.
- [^a-zA-Z0-9]: Matches any non-alphanumeric character. (# /[^A-Za-z0-9] also gives the same result. Change in sequence does not effect anything).
- [^xyz]: Matches any character except "x," "y," or "z."

^DM: "DM" at the beginning of a line.

- ^Start: Matches "Start" at the beginning of a line. Write /**^Start** in the command box in vim and continue to press 'n' to see all the results.)
- ^XYZ: Matches "XYZ" at the beginning of a line.

- **sun\$**: "sun" at the end of a line.

- run\$: Matches "run" at the end of a line.
- abc\$: Matches "abc" at the end of a line.

- **g\+:** One or more occurrences of the previous character.

- **go\+d**: Matches "god", "good", "goooood", and so on.
- **a\+:** Matches "a", "aa", "aaa", and so on.

- **g\?:** Zero or one occurrence of the previous character.

- **colou\?r**: Matches "color" and "colour".
- **x\?y**: Matches "y" and "xy".

- **GIF\|JPEG**: As above - GIF or JPEG.

- The image format can be **GIF\|JPEG**: Matches "GIF" or "JPEG". [# Eg: To execute regular expression "Sayak\|the"; First, press 'esc' to go to normal mode. Then, type "/Sayak\|the" in the command line and press 'enter'. Then press "n" to go to each results (Sayak or the) individually.]
- The file extension is **jpg\|jpeg**: Matches "jpg" or "jpeg".

- **wood\|(cock\|house\|)**: As above - woodcock or woodhouse.

- The bird is **wood\|(cock\|house\|)**: Matches "woodcock" or "woodhouse".
- The furniture is made of **wood\|(cock\|house\|)**: Matches "woodcock" or "woodhouse".

wood\(\text{cock}\|\text{house}\): As above - woodcock or woodhouse.

The regular expression `wood\(\text{cock}\|\text{house}\)` matches the patterns "woodcock" or "woodhouse." This is achieved using the alternation (`|`) operator within a grouped expression.

Here's how the components of the regular expression work:

- `wood`: This part of the pattern matches the literal characters "wood."
- `\(\text{cock}\|\text{house}\)`: This is a grouped expression that uses the alternation operator (`|`). It matches either the pattern "cock" or the pattern "house." The backslashes `\` are used to escape the parentheses for grouping, and the `|` denotes alternation.

Using the examples you provided:

- "woodcock" matches because it matches the pattern "wood" followed by "cock."
- "woodhouse" matches because it matches the pattern "wood" followed by "house."
- "woodpecker" does not match because it does not exactly match either "woodcock" or "woodhouse."
- "wood" does not match because it doesn't include "cock" or "house" after "wood."

The alternation operator allows you to specify multiple alternative patterns within a single regular expression, effectively providing a way to match any of the specified patterns. This is a powerful feature of regular expressions that can help you match various possibilities in your text.

\<pat: Pattern "pat" at the beginning of a word.

- \<cat: Matches "catch", "caterpillar", but not "copycat". (Write "`\<cat`" in the command line of vim and press n continuously to go to next match.)
- \<in: Matches "in", "inside", but not "tin".

pat\>: Pattern "pat" at the end of a word.

- **cat\>:** Matches "cat", "scat", but not "catch".

\{m\}: m occurrences of the previous character.

- **a\{3\}:** Matches "aaa" (three consecutive 'a' characters).

- **[0-9]\{5\}:**

Matches any five-digit number.

[0-9]: This character class matches any digit between 0 and 9.

\{5\}: This quantifier specifies that the preceding element (the character class **[0-9]**) should occur exactly 5 times.

\{m,\}: At least m occurrences of the previous character.

- **b\{2,\}:** Matches "bb", "bbb", "bbbb", and so on. (Write “/b\{2,\}” in the command line of vim, click ‘enter’ and press ‘n’ continuously to go to next match.)

- **x\{4,\}:** Matches "xxxx", "xxxxx", "xxxxxx", and so on.

\{m,n\}: Between m and n occurrences of the previous character.

- **c\{2,4\}:** Matches "cc", "ccc", and "cccc".

- **y\{3,5\}:** Matches "yyy", "yyyy", and "yyyyy".

\(exp\): exp and attaches tag \1, \2, etc. to exp.

- **\(\d\d\)-\(\d\d\)-\(\d\d\d\d\):**

■ Matches a date in the format "dd-mm-yyyy".

■ Each part (day, month, year) is captured with a separate tag:

\1, \2, \3

Each part (day, month, year) is captured with a separate tag: \1, \2, \3

Exactly! In the regular expression `\(\d\d\)-\(\d\d\)-\(\d\d\d\d\)`, each part of the date (day, month, year) is captured with a separate tag using `\1`, `\2`, and `\3`. These tags are called backreferences and allow you to refer to the captured portions of the text later in the regular expression or in replacement text.

Here's how the tags work:

- `\1`: This refers to the content captured by the first capturing group, which is `\(\d\d\)`. In the context of the regular expression, it represents the day component of the date.
- `\2`: This refers to the content captured by the second capturing group, which is also `\(\d\d\)`. It represents the month component of the date.
- `\3`: This refers to the content captured by the third capturing group, which is `\(\d\d\d\d\)`. It represents the year component of the date.

For example, if you want to use these captured parts in replacement text, you can do something like this:

`vim`

```
:%s/\(\d\d\)-\(\d\d\)-\(\d\d\d\d\)/Year: \3, Month: \2, Day: \1/g
```

In this `:s` (substitute) command in Vim, the backreferences `\1`, `\2`, and `\3` are used to refer to the captured day, month, and year components, respectively. This would replace the matched date with a string like "Year: 2023, Month: 05, Day: 01" based on the captured values.

Backreferences are a powerful feature in regular expressions that allow you to manipulate and format text based on the captured groups.

```
10-09-3078
```

```
Shatarup Sayak
```

```
2034-90
```

```
We re living in 28-08-2023
```

```
Tomorrow 29-08-2023
```

```
Lets go! 30-23-2025
```

```
:%s/\(\d\d\)-\(\d\d\)-\(\d\d\d\d)/Year: \3, Month: \2, Day: \1/g
```

```
Year: 3078, Month: 09, Day: 10
Shatarup Sayak
2034-90

We're living in Year: 2023, Month: 08, Day: 28
Tomorrow Year: 2023, Month: 08, Day: 29
Let's go! Year: 2025, Month: 09, Day: 30
```

\w:
A

4 substitutions on 4 lines

34,1

Bot

word character (same as [a-zA-Z0-9_]).

- **\w\+:** Matches one or more word characters (e.g., "word", "123", "abc_").
- **@\w\+:** Matches email addresses like "example@example.com". (# It is matching @username , Sagar.@com , sayak@gmail.com; but not matching sayak@. So, it is observed that, if @ is in first or in middle of the string , then it is matching. But if it is in the last, it is not matching.)

\W: A non-word character (same as [^a-zA-Z0-9_]).

- **\W\+:** Matches one or more non-word characters (e.g., "@", "\$", "#").
- **\W\d\W:** Matches a non-word character followed by a digit followed by another non-word character.

SA

\w\+: Matches one or more word characters (e.g., "word", "123", "abc_").

The regular expression \w\+ matches one or more word characters. Word characters include letters, digits, and underscores.

Here's how the components of the regular expression work:

- `\w`: This matches any word character, which includes uppercase and lowercase letters, digits, and underscores.
- `\w+`: This quantifier specifies that the preceding element (in this case, `\w`) should occur one or more times.

Using the examples you provided:

- "word" matches because it consists entirely of word characters.
- "123" matches because it consists entirely of digits.
- "abc_" matches because it contains letters and an underscore.

However, strings like "hello!" (exclamation mark is not a word character) or "@username" (contains a special character) would not match the pattern. (# For me, cursor is going to 'u' and skipping '@' in '@username'!!!)

This regular expression is useful when you want to match words, identifiers, or strings that consist of letters, digits, and underscores.

Basic difference between \W and \W\+ :

`\W`: A non-word character (same as `[\^a-zA-Z0-9_]`).

- `\W\+`: Matches one or more non-word characters (e.g., "@", "\$", "#").

If you write '`\W`' in the command line, press 'enter' and press enter continuously, the cursor will move one by one non-word character. But, if you do the same with '`\W\+`', cursor will move with string of non-word characters, not through each character.

`\d`: A digit (same as [0-9]).

- `\d\{3}`: Matches any three-digit number (e.g., "123", "456").
- `\d\+\.\d\{2}`: Matches a decimal number with two decimal places (e.g., "10.50", "3.14").

\d+\.\d{2}: Matches a decimal number with two decimal places.

The regular expression `\d+\.\d{2}` matches a decimal number with exactly two decimal places. It's used to match numbers in a specific format, where the number consists of one or more digits before a decimal point, followed by exactly two digits after the decimal point.

Here's how the components of the regular expression work:

- `\d+`: This matches one or more digits. The `\+` quantifier specifies that the preceding element (in this case, `\d`, which represents a digit) should occur one or more times.
- `\.`: This matches a literal period (decimal point).
- `\d{2}`: This matches exactly two digits. The `\{2\}` quantifier specifies that the preceding element (in this case, `\d`) should occur exactly two times.

Using the examples you provided:

- "123 .45" matches because it has one or more digits before the decimal point and exactly two digits after the decimal point.
- "0 .00" matches for the same reason.
- "456 .123" does not match because it has more than two digits after the decimal point.
- "12 .3" does not match because it has only one digit after the decimal point.

This regular expression is useful when you want to match decimal numbers with a specific number of decimal places, such as for financial or numeric validation purposes.

\D: A non-digit (same as [^0-9]).

- `\D{2,}`: Matches two or more consecutive non-digit characters.
- `\D\S`: Matches a non-digit character followed by a non-whitespace character.

\s: A whitespace character.

- \s+: Matches one or more whitespace characters (e.g., spaces, tabs, newlines).
- \d\s: Matches a whitespace character before and after a digit.

\S: A non-whitespace character.

- \S{4}: Matches any four consecutive non-whitespace characters.
- \t: A tab.

\t\w\t: Matches a word surrounded by tabs.

\t: This matches a tab character.

\w+: This matches one or more word characters (letters, digits, or underscores).

\t: This matches another tab character.

- ^\t\d\t\$: Matches a line that starts and ends with tabs, with digits in between.

[:alpha:] - An alphabetic character.

- [:lower:] - A lowercase alphabetic character.
- [:upper:] - An uppercase alphabetic character.
- [:digit:] - A numeric character.
- [:alnum:] - An alphanumeric character.
- [:space:] - A whitespace character, including form feed.
- [:blank:] - A space or tab.
- [:punct:] - A punctuation character (not a space, letter, digit, or control character).

Form feed character:

The form feed character essentially instructs the output device to move to a new page.

if you have a specific use case where you want to include a form feed character in your text, you can follow these general steps:

1. Text Editors: In some text editors, you might be able to enter a form feed character directly using a keyboard shortcut or a special command. However, the behavior can vary between different editors. Here's how you might attempt to enter a form feed character in a few common text editors:

- **Vim:** In insert mode, you can press `Ctrl-V` followed by `Ctrl-L`.
- **Emacs:** You can use `C-q C-l`.
- **Notepad++:** You can use the "Edit" > "Character Panel" to select and insert special characters.

2. Programming Languages: In programming languages, you can include a form feed character in strings using escape sequences:

- **C/C++, Python, Java, JavaScript, Ruby, PHP:** Use `\f` inside a string.

Note: Due to the declining use of the form feed character and the diversity of systems and applications, you might find that its behavior is inconsistent or not supported in some contexts.

What is control character?

A control character, also known as a control code or non-printable character, is a character in a character encoding system that does not represent a printable character (such as a letter or a digit) but instead has a control function. Control characters are used to control devices or to provide specific instructions for formatting or manipulating text.

Some common control characters and their functions include:

- **Tab (\t):** Moves the cursor to the next tab stop. In text editors, this character can be used to align text in columns.

- **Newline (\n):** Moves the cursor to the beginning of the next line. This character is used to start a new line of text.
- **Carriage Return (\r):** Moves the cursor to the beginning of the current line. It's often used in combination with newline to start a new line.
- **Form Feed (\f):** Initiates a page break or advances the cursor to the beginning of the next page. Historically used in printing.
- **Backspace (\b):** Moves the cursor back one position. It's used to delete the previous character.
- **Escape (\e or \033):** Signals the start of an escape sequence used for controlling terminal behavior or printing special characters.

Control characters are typically **represented using escape sequences**, where a backslash \ is followed by a character or code that represents the control function. For example, \n represents a newline character, and \t represents a tab character.

In modern computing, control characters are often used in programming languages, regular expressions, and text manipulation tasks. However, their direct influence on device control and communication has become less prominent with the advancement of graphical user interfaces and display technologies.

[:alpha:]]\+: Matches one or more alphabetic characters

(e.g., "abc", "DEF").

- **[:lower:]]\{3\}:** Matches exactly three lowercase alphabetic characters (e.g., "abc", "xyz").
- **[:upper:]][:digit:]]:** Matches an uppercase letter followed by a digit (e.g., "A1", "B9").
- **[:alnum:]]*:** Matches zero or more alphanumeric characters (e.g., "123", "abc").

[:space:]: Matches any whitespace character, including spaces and tabs.

- **[:blank:]**\+: Matches one or more spaces or tabs.
- **[:print:]**{5,10}: Matches printable characters with a length between 5 and 10. (#Not working!!)
- **[:punct:]**: Matches punctuation characters like ".", "!", "?", etc.

BASH SCRIPTING

How to print lines having only numbers

```
sayak@ubuntu:~/harry/files_module2_l1$ cat minima.dat
97.11 70.15
-82.63 79.14
-177.00 106.10
97.11 106.10
-78.14sand110.59
-177.00 178.00
97.11 178.00
97.11 178.00ssa
1010 101000
```

Method 1: grep -E "^[0-9.-]+\s+[0-9.-]+\\$" minima.dat

The `grep` command you provided is used to search for lines in a text file (`minima.dat`) that match a specific regular expression pattern. Let's break down the pattern and explain how it works:

- `grep`: This is the command itself, used for searching text using regular expressions.
- `-E`: This option tells `grep` to use extended regular expressions. Extended regular expressions provide more powerful pattern matching capabilities than basic regular expressions. Similarly you can use `egrep` in place of `grep -E`.
`egrep "^[0-9.-]+\s+[0-9.-]+\$"`
`minima.dat`
- `"^[0-9.-]+\s+[0-9.-]+\$"`: This is the regular expression pattern enclosed in double quotes. Let's break it down step by step:
 - `^`: Anchors the pattern to the beginning of a line. It ensures that the pattern matches from the start of each line.

[0-9 . -]+: This part of the pattern matches one or more characters that are either digits (0-9), a period (.), or a hyphen (-). In other words, it matches positive or negative numbers with optional decimal points.

- \s+: This part of the pattern matches one or more whitespace characters (spaces or tabs). It allows for one or more spaces or tabs between the two numbers.
- [0-9 . -]+: Similar to the first part, this matches one or more characters that are digits, a period, or a hyphen. It's used to match the second number in the line.
- \$: Anchors the pattern to the end of a line. It ensures that the pattern matches up to the end of each line.

Method 2: While loop

```
while IFS= read -r line; do
if [[ $line =~ ^[0-9.-]+\s+[0-9.-]+$ ]]; then echo
"$line" ; fi
done < "minima.dat"
```

- **while**: This keyword initiates a loop that continues as long as a specified condition is true.
- **IFS=**: This sets the Internal Field Separator (IFS) to an empty string. Setting IFS to an empty string ensures that leading and trailing whitespace characters are preserved in each line read from the file. Without this, leading and trailing spaces would be trimmed by default.
- **read -r line**: The **read** command is used to read a line from the file. The **-r** option tells **read** to interpret backslashes (\) as literal characters and not as escape characters. The line read from the file is stored in the **line** variable.

2. `if [[$line =~ ^[0-9.-]+\s+[0-9.-]+$]]; then:`

This line checks if the contents of the `line` variable match the regular expression pattern. Let's break down the components:

- `[[...]]`: Double square brackets are used for conditional tests in Bash.
- `$line`: This variable contains the line read from the file.
- `=~`: This operator is used for regular expression matching.
- `^[0-9.-]+\s+[0-9.-]+$`: This is the regular expression pattern being tested. It matches lines that contain two numbers (possibly with decimal points and negative signs) separated by one or more spaces or tabs. This pattern is the same as the one used in your previous `grep` command.

3. `echo "$line"`: If the line matches the pattern, it is printed to the standard output. This effectively filters and prints only the lines from "minima.dat" that contain numerical data in the specified format.

4. `done < "minima.dat"`: This marks the end of the while loop and specifies the input source for the loop, which is the "minima.dat" file.

In summary, this Bash script reads lines from the "minima.dat" file, checks each line to see if it matches a specific numerical format using a regular expression, and if it does, it echoes (prints) the matching lines to the standard output. This is a common way to filter and process lines that meet certain criteria within a text file.

What are the differences between literal characters, escape characters and control characters?

Literal characters

are the characters that are represented by themselves. For example, the character `a` is a literal character.

Escape characters

are characters that are preceded by a backslash (\). They are used to represent special characters or sequences of characters. For example, the escape character \n represents a newline character.

- | | |
|---|---|
| <ul style="list-style-type: none">• Escape character | <ul style="list-style-type: none">• Meaning |
| <ul style="list-style-type: none">• \n• \t• \r• \'• \"• \\• \x20• \033 | <ul style="list-style-type: none">• Newline character• Tab character• Carriage return character• Single quote character• Double quote character• Backslash character• Space character• Control character ESC |

Control characters

are characters that are used to control the behavior of the computer. They are not displayed on the screen.

To type control characters in Vim:

1. **Enter Insert Mode:** Press **i** or **I** (for insert mode) to start typing text.
2. **Press Ctrl:** Hold down the **Ctrl** key on your keyboard.
3. **Press the Corresponding Key:** While holding down **Ctrl**, press the key that corresponds to the control character you want to insert.

Here is a list of some commonly used control characters and their functions in Vim:

1. **Ctrl-C**: Interrupts the current operation or command and returns to **Normal mode**. This is similar to pressing the Esc key.
2. **Ctrl-D**: In Normal mode, it scrolls the screen half a page down. In Insert mode, it inserts a literal Ctrl-D character.
3. **Ctrl-E**: In Normal mode, it scrolls **the screen up by one line** without moving the cursor. In Insert mode, it inserts a literal Ctrl-E character.
4. **Ctrl-F**: In Normal mode, it scrolls the screen forward (down) one page. In Insert mode, it inserts a literal Ctrl-F character.
5. **Ctrl-G**: Displays information about the current file, including its name, the cursor's position, and the number of lines.
6. **Ctrl-H**: Equivalent to the Backspace key, it deletes the character to the left of the cursor.
7. **Ctrl-I**: Equivalent to the Tab key, it inserts a tab character.
8. **Ctrl-J**: Equivalent to the Enter key, it moves to the next line.
9. **Ctrl-K**: In Insert mode, it is often used to enter digraphs, which are combinations of characters to create special symbols or characters.
10. **Ctrl-L**: Redraws the screen and clears any artifacts or display issues.
11. **Ctrl-M**: Equivalent to the Enter key, it moves to the next line.
12. **Ctrl-N**: In Insert mode, **it enables keyword completion based on words in the current file**.
13. **Ctrl-P**: In Insert mode, it triggers keyword completion in the reverse order of Ctrl-N.
14. **Ctrl-Q**: Resumes the terminal's flow control, useful when Ctrl-S (stop) has been used to pause terminal output.
15. **Ctrl-R**: In Insert mode, it is used for **inserting the contents of a register (clipboard or yanked text)**.
16. **Ctrl-S**: Pauses terminal output (similar to the Pause/Break key).

17. **Ctrl-T**: In Normal mode, it indents the current line (shift text right). In Insert mode, it inserts a tab character.
18. **Ctrl-U**: In Normal mode, it scrolls the screen up half a page. In Insert mode, it inserts a literal Ctrl-U character.
19. **Ctrl-V**: In Insert mode, it is used to insert a literal control character. For example, Ctrl-V followed by Ctrl-M inserts a literal carriage return character.
20. **Ctrl-W**: In Normal mode, it is used for various window-related commands, such as splitting windows and navigating between them.
21. **Ctrl-X**: In Insert mode, it is often used in conjunction with other keys for text manipulation, such as Ctrl-X Ctrl-E for increasing a number under the cursor.
22. **Ctrl-Y**: In Normal mode, it scrolls the screen up by one line without moving the cursor. In Insert mode, it inserts a literal Ctrl-Y character.
23. **Ctrl-Z**: Suspends Vim and returns to the terminal. You can resume Vim by typing `fg` in the terminal.
24. **Ctrl-J**: In Normal mode, it is used to jump to the definition of a function or tag under the cursor when working with code.
25. **Ctrl-[**: In Normal mode, it is equivalent to pressing the Esc key, and it returns to Normal mode.

These control characters provide various functionalities within Vim, and their behavior can change depending on the mode you're in (Normal, Insert, Visual, etc.) and the context in which they are used. You can customize many of these bindings in your Vim configuration if needed.

Method 3: Using for loop

```
for line in $(cat "minima.dat"); do
if [[ $line =~ ^[0-9.-]+\s+[0-9.-]+\$ ]]; then
```

```
echo "$line"
```

```
fi
```

```
done
```

1. `for line in $(cat "minima.dat"); do`: This line starts a **for loop** that reads lines from the "minima.dat" file. Here's what each part does:

- `for line in . . .`: Initiates a loop where `line` is a variable that will hold each line of the file during each iteration.
- `$(cat "minima.dat")`: Uses command substitution to execute the `cat` command, which reads the contents of the "minima.dat" file. The result is a stream of lines from the file, which are then processed one by one in the loop.
- `do`: Marks the beginning of the loop body.

2. `if [[$line =~ ^[0-9.-]+\s+[0-9.-]+$]]; then`:

This line checks whether the current line (`$line`) matches a specific regular expression pattern. Here's what each part does:

- `[[. . .]]`: Double square brackets are used for conditional tests in Bash.
- `$line`: This variable contains the current line from the file being processed in each iteration of the loop.
- `=~`: This operator is used for regular expression matching.
- `^[0-9.-]+\s+[0-9.-]+$`: This is the regular expression pattern being tested. It matches lines that contain two numbers (possibly with decimal points and negative signs) separated by one or more spaces or tabs. For example, it would match lines like "1.23 -45.67" or "0.5 7.89."

3. `echo "$line"`: If the line matches the pattern, this command echoes (prints) the line to the standard output, effectively filtering

and printing only the lines from "minima.dat" that contain numerical data in the specified format.

4. **done**: Marks the end of the **for** loop.

History

- 1979: Bourne shell - /bin/sh
- Only a few additional features are added over the time to the shell
- 1980: built-in `test` command
- 1984: hashing and shell functions
- 1989: job control features
- Competitors to Bourne shell are csh and ksh
- 1989: GNU developed Bourne-Again SHell (bash)
- written from scratch by incorporating the features from other existing shells.

SA

what is the difference between bash and bourne?

Bash (Bourne-Again Shell) and the original Bourne Shell (sh) have some key differences, with Bash being an extended and more feature-rich version of the Bourne Shell. Here are some of the main differences:

1. Compatibility:

- **Bourne Shell (sh):** It is the original Unix shell and has basic functionality. Scripts written in the Bourne Shell are generally more portable across different Unix-like systems, but they may lack some advanced features.
- **Bash (Bourne-Again Shell):** Bash is designed to be compatible with the Bourne Shell, which means most Bourne Shell scripts can run in Bash without modification. Bash is the default shell on many Linux distributions and provides additional features and capabilities.

2. Interactive Features:

- **Bourne Shell (sh):** The Bourne Shell lacks some user-friendly interactive features like command history, command-line editing, and tab completion.
- **Bash (Bourne-Again Shell):** Bash offers command history (you can cycle through previously entered commands using the up and down arrow keys), command-line editing (you can edit

and navigate the current command line easily), and **tab completion** (you can complete command and file names by pressing the Tab key).

3. Built-in Functions and Features:

- **Bourne Shell (sh):** Bourne Shell has a limited set of built-in functions and features.
- **Bash (Bourne-Again Shell):** Bash includes numerous built-in functions and features, such as **arithmetic operations, advanced string manipulation, arrays, associative arrays,** and more. These make it a more powerful scripting and interactive shell.

4. Scripting and Programming:

- **Bourne Shell (sh):** The Bourne Shell is primarily designed for basic scripting and executing system commands.
- **Bash (Bourne-Again Shell):** Bash extends the Bourne Shell's scripting capabilities with features like improved **conditional statements, more advanced loops, and enhanced error handling,** making it a more versatile choice for complex scripts and programming tasks.

5. Default Shell:

- **Bourne Shell (sh):** On some Unix systems, /bin/sh may still refer to the Bourne Shell, but on many modern systems, it's often replaced by a symlink to a more capable shell like Bash.
- **Bash (Bourne-Again Shell):** Bash is the default shell on many Linux distributions and is widely used as an interactive shell and scripting language.

In summary, while Bash retains compatibility with the Bourne Shell and can run Bourne Shell scripts, it offers many additional features and improvements that make it a more user-friendly and powerful choice for interactive use and scripting tasks.

GNU project:

The GNU in bash scripting refers to the fact that Bash is a GNU project. The GNU Project is a free software project that aims to create a free operating system called GNU. Bash is one of the many free software programs that are part of the GNU Project.

bash overview

- bash is a powerful scripting language. It is open source
- Bourne shell subset of Bash, with additional features
- Learning Bash and shell scripting is learning Unix/Linux
- Vital role in automating tasks and system administration
- A sequence of commands becomes a script file, with added command-line options
 - Define variable, functions, loops, etc enable efficient execution of more complex tasks
 - Command-line options are used to enable different options to be passed to the commands

bash features

- Input/output redirection
- Wildcard characters
- variables
- Built-in command set for writing programs
- Job control
- Command-line editing
- History

- Integer arithmetic
- Arrays and arithmetic expressions
- for loop

SA

What is wildcard character in bash scripting

In Bash scripting, a wildcard character is a special symbol that represents one or more characters in filenames or strings. Wildcards are used to match multiple files or strings based on a pattern, which can save you from having to specify every individual filename or string explicitly. Wildcards are often used with commands like `ls`, `cp`, `rm`, and in various parts of shell scripting.

Here are some commonly used wildcard characters in Bash:

1. *** (Asterisk):** The asterisk wildcard matches zero or more characters.

For example:

- `*.txt` matches all files ending with ".txt."
- `file*` matches all filenames starting with "file."

2. **? (Question Mark):** The question mark wildcard matches a single character. For example:

- `file?.txt` matches "file1.txt," "file2.txt," but not "file12.txt."

3. **[] (Square Brackets):** Square brackets allow you to specify a character set or range. For example:

- `[aeiou]` matches any single vowel.
- `[0-9]` matches any single digit.

4. **[^] (Caret Inside Square Brackets):** When the caret (^) is placed at the beginning of a character set inside square brackets, it negates the set. For example:

- `[^0-9]` matches any character that is not a digit.

5. { } (**Curly Braces**): Curly braces allow you to specify a list of options or a range. For example:

- {apple,banana,cherry} matches any of the listed words.
- {1..5} matches the numbers 1 through 5.

6. ** (**Double Asterisks**): Double asterisks are used for recursive globbing. They match directories and their subdirectories. This feature is available in modern versions of Bash. For example:

- **/*.txt matches all ".txt" files in the current directory and its subdirectories.

Here are some examples of how wildcard characters are used in Bash scripting:

bash

```
# List all text files in the current directory
```

```
ls *.txt
```

```
# Remove all files starting with "temp"
```

```
rm temp*
```

```
# Copy all files with  
names like "file1.txt"  
or "file2.txt" to a  
new directory
```

```
cp file?.txt  
new_directory/
```

```
# Find all files with  
numbers in their names
```

```
find . -type f -name "*[0-9]*"
```

```
sayak@Ubuntu:~/harry$ ls  
sayak@Ubuntu:~/harry$ cd ..  
sayak@Ubuntu:~$ mv ?.txt harry/  
sayak@Ubuntu:~$ cd harry  
sayak@Ubuntu:~/harry$ ls  
1.txt 2.txt 3.txt 4.txt 5.txt 6.txt  
sayak@Ubuntu:~/harry$ find . -type f -name "*[0-9]*"  
.4.txt  
.3.txt  
.6.txt  
.2.txt  
.5.txt  
.1.txt  
sayak@Ubuntu:~/harry$ █
```

```
# Here, we moved all the files of <any single digit>.txt format from  
"home" directory to the empty directory named "harry"
```

Wildcards are powerful tools for working with groups of files or strings that follow a pattern, and they make shell scripting more flexible and efficient.

```
find . -type f -name "*[0-9]*"
```

The command `find . -type f -name "*[0-9]*"` will find all regular files (-type f) in the current directory (.) whose name contains at least one digit . First * means there may be anything before the digit character and the last * means there may be anything after the digit characters.

First command

`bash [options] [arguments]`

- `bash hpc.sh`
- `bash hpc.sh arg1 arg2 arg3`
- First line: `#!/bin/bash`
- Comments: These lines start with a # symbol

SA

bash hpc.sh arg1 arg2 arg3

you've provided a command line in the format of running a Bash script called "hpc.sh" with three arguments: "arg1," "arg2," and "arg3."

First line: `#!/bin/bash` what does it mean ?

The first line of a Bash script, `#!/bin/bash`, is called a **shebang line**.

The shebang line tells the operating system which program to use to interpret the script. In this case, the shebang line tells the operating system to use the Bash shell to interpret the script.

The `#!` is a special character that tells the operating system that the line is a shebang line. The `/bin/bash` is the path to the Bash shell executable.

Script arguments

- Arguments are values passed to the script on the command line
- Arguments are accessed using special variables: `$0`, `$1`, `$2`, ...

Examples with `$0`, `$1`, `$#` etc.

Suppose you have a Bash script named `example.sh`, and you run it with two arguments:

bash

```
./example.sh arg1 arg2
```

Here's how you can use the special variables in the script:

bash

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

```
# $0: Script Name
```

```
echo "Script name: $0"
```

```
# $1 and $2: Positional Arguments
```

```
echo "First argument: $1"
```

```
echo "Second argument: $2"
```

```
# $# Number of Arguments
```

```
echo "Number of arguments: $"
```

```

# $* All Arguments as a Single String
echo "All arguments as a single string: $*"

# $@ All Arguments as Separate Strings
echo "All arguments as separate strings:"

for arg in "$@"; do
    echo "$arg"
done

#(for arg in "$@"; do: This line starts a for loop that iterates through each element in the $@ array. $@ contains all the script's arguments as separate strings.)

```

Now, let's break down what this script does:

1. **\$0**: This variable contains the script's name, which is "example.sh" in this case.
2. **\$1 and \$2**: These variables represent the positional arguments "arg1" and "arg2," respectively.
3. **\$#**: This variable holds the number of arguments passed to the script, which is 2 in this case.
4. **\$***: This variable contains all arguments as a single string, so it would be "arg1 arg2."
5. **\$@**: This variable contains all arguments as separate strings, so it would be an array with two elements: "arg1" and "arg2."

```

sayak@Ubuntu:~/harry$ vim example.sh
sayak@Ubuntu:~/harry$ bash example.sh potato sweets pizza
Script name: example.sh
First argument: potato
Second argument: sweets
Number of arguments: 3
All arguments as a single string: potato sweets pizza
All arguments as separate strings:
potato
sweets
pizza

```

```
#(Here, we run the same script with some different arguments. Note the  
differences.)
```

Another Example with \$0, \$1, \$# etc.

```
#!/bin/bash

x=$1

y=$2

echo "No. of
arguments: "$#

echo "All arguments
as a separate string"

for i in "$@"; do
echo $i

done

echo "-----"

echo "All arguments as a single string"

for i in "$*"; do
echo $i

done
```

```
sayak@Ubuntu:~/harry$ vim 8.sh
sayak@Ubuntu:~/harry$ bash 8.sh king Queen
No. of arguments: 2
All arguments as a separate string
king
Queen
-----
All arguments as a single string
king Queen
```

One remarkable point:

```
x=$1

y=$2

echo "No. of arguments: $""

echo "All arguments as a separate string"

for i in "$@"; do
```

```

echo $i
done
echo "-----"
echo "All arguments as a single string"
for i in $*; do
echo $i
done

```

sayak@Ubuntu:~/harry\$ bash 8.sh king Queen
No. of arguments: 2
All arguments as a separate string
king
Queen

All arguments as a single string
king
Queen

#When we run the same script without quation in \$@ and \$*, it is not coming in the single string. But \$@ is working properly.
Why?!

Example

```

#!/bin/bash
x=$1
y=$2
if [ "$#" -ne 2 ]; then
echo ./$0 var1 var2
echo "exiting . . . "
exit
fi

```

sayak@Ubuntu:~/harry\$ vim 9.sh
sayak@Ubuntu:~/harry\$ bash 9.sh Apple orange mango
.9.sh var1 var2
exiting ...
sayak@Ubuntu:~/harry\$ bash 9.sh Apple orange
sayak@Ubuntu:~/harry\$

The script will not be executed if number of arguments is equal to 2.

Essential elements of the bash script

- Input/output: read input from user or command line or a file and display output on terminal
- read, echo or printf
- **Control structures:**
 - if statements
 - case statements
 - for loops
 - while loops
- **Functions:** Set of commands can be grouped to define as a function, and reuse it at multiple times in the script.

• **Command substitution:**

Command substitution is a powerful feature in Bash (and many other Unix-like shells) that allows you to capture the output of one command and use it as input for another command or store it in a variable. There are two common syntaxes for command substitution:

1. **Backticks (`...`):** You can enclose a command within backticks, and the shell will execute that command and replace the backticked expression with its standard output. For example:

bash

```
current_date=`date`  
echo "Today is $current_date"
```

In this example, the `date` command is executed, and its output (the current date and time) is stored in the `current_date` variable.

2. **\$()**: You can also use the **\$()** syntax for command substitution, which is more modern and often preferred due to better readability and nesting capabilities. It works the same way as backticks. For example:

bash

```
current_date=$(date)  
echo "Today is $current_date"
```

Both of these examples will produce the same result, where the current date and time is captured from the `date` command and stored in the `current_date` variable.

- **Exit status and error handling:**

- **Redirection: Input and output can be redirected**

the following are the redirection operators used in Bash scripting:

- < - This is the input redirection operator. It redirects the input of a command to a file.
- > - This is the output redirection operator. It redirects the output of a command to a file.
- >> - This is the append output redirection operator. It redirects the output of a command to a file, appending the output to the end of the file.
- 2> - This redirects the standard error output of a command to a file.
- 2>&1 - This redirects both the standard error output and the standard output of a command to a file.

Here are some examples of redirection in Bash scripting:

- **cat < file.txt** - This command redirects the input of the `cat` command to the file `file.txt`.

```
sayak@Ubuntu:~/harry$ vim file.txt  
sayak@Ubuntu:~/harry$ cat < file.txt  
Sayak is a good boy
```

This generates the same output as 'cat file.txt'. That means "file.txt" serves as the input for the `cat` command

- **ls > output.txt** - This command redirects the output of the ls command to the file output.txt.

```
sayak@Ubuntu:~/harry$ touch output.txt
sayak@Ubuntu:~/harry$ ls > output.txt
sayak@Ubuntu:~/harry$ cat output.txt
1.txt
•
2.txt
•
3.txt
•
4.txt
•
5.txt
•
6.txt
•
7.sh
•
8.sh
•
9.sh
example.sh
file.txt
output.txt
```

"ls" command lists all the files and directories
of pwd and that list is saved in output.txt file.

- **echo "This is a test" >> output.txt** - This command appends the text "This is a test" to the end of the file output.txt.

```
sayak@Ubuntu:~/harry$ echo "this is a test" >> output.txt
sayak@Ubuntu:~/harry$ cat output.txt
1.txt
2.txt
3.txt
4.txt
5.txt
6.txt
7.sh
8.sh
9.sh
example.sh
file.txt
output.txt
this is a test
sayak@Ubuntu:~/harry$
```

- **2> error.log** - This command redirects the standard error output of the command command to the file error.log.

- **command 2>&1** - This command redirects both the standard error output and the standard output of the **command** command to the file **output.txt**.
- **Arithmetic operations and conditional expression**

Arithmetic Evaluation and Expansion with integers

- Syntax:

```
$((expression))
```

where expression is a valid arithmetic expression

```
#!/bin/bash

x=5; y=10; z=2

addition=$((x + y))

multiplication=$((x *y))

division=$((y / x))

result=$(( (x + y) * z ))

square=$((z*z))

echo "sum= $addition"

echo "product= $multiplication"

echo "result= $result"
```

```
#!/bin/bash
x=5; y=10; z=2
addition=$((x + y))
multiplication=$((x *y))
division=$((y / x))
result=$(( (x + y) * z ))
square=$((z*z))
echo "sum= $addition"
echo "product= $multiplication"
echo "result= $result"
echo "square= $square"
```

```
sayak@Ubuntu:~/harry$ vim Arithmatic.sh
sayak@Ubuntu:~/harry$ bash Arithmatic.sh
sum= 15
product= 50
result= 30
square= 4
```

Arithmetic operators

- + / - * // : addition/subtraction/multiplication/integer division
- ** : exponentiation
- % : modulo (remainder)
- -= : subtraction assignment
- += : addition assignment
- *= : multiplication assignment
- /= : integer division assignment
- parameter++ : post-increment
- parameter-- : post-decrement

```
sayak@Ubuntu:~/harry$ vim operator.sh
sayak@Ubuntu:~/harry$ bash operator.sh
5*2 = 10
5/2 = 2
5%2 = 1
5**2 = 25
20
20
21
```

Example 1: pre and post increment

```
echo "5*2 = $"(( 5*2 ))  
echo "5/2 = $"(( 5/2 ))  
echo "5%2 = $"(( 5%2 ))  
echo "5**2 = $"(( 5**2 ))  
num=10  
(( num += 10 ))  
echo $num  
echo $((num++))  
echo $num
```

Explanation:

Here's what happens step by step:

1. **num=10**: You assign the value 10 to the variable "num."
2. **((num += 10))**: This line increments the value of "num" by 10, so "num" becomes 20.
3. **echo \$num**: This prints the current value of "num," which is 20.
4. **echo \$((num++))**: This line prints the current value of "num" (which is 20) and then increments "num" by 1. However, the post-increment operator (**num++**) means that the increment occurs after the value is used in the expression. So, it prints 20 and then increments "num" to 21.
5. **echo \$num**: Finally, this line prints the updated value of "num," which is 21.

What is the difference between += and ++ ?

The += and ++ operators are used in programming languages, including Bash, to perform different operations:

1. += (Plus-Equals Operator):

- The += operator is used for addition and assignment.
- It takes the current value of a variable, adds a specified value to it, and then assigns the result back to the same variable.
- It's used to increment or update the value of a variable by adding another value to it.
- Example:

bash

- num=5

```
num+=3    # Equivalent to num=num+3
```

```
echo $num  # Output: 8
```

• ++ (Increment Operator):

- The ++ operator is used for incrementing a variable by 1.
- It is typically used to increase the value of a variable by 1.
- There are two forms of the ++ operator: pre-increment (++var) and post-increment (var++).
 - Pre-increment (++var) increments the variable and then uses its new value in an expression.
 - Post-increment (var++) uses the current value of the variable in an expression and then increments it.
- Example:

bash

- num=5

```
let "num++"  # Post-increment
```

```
echo $num  # Output: 6
```

In summary, `+=` is used for addition and assignment, allowing you to increment a variable by a specified value, while `++` is specifically used for incrementing a variable by 1, either before or after its current value is used in an expression.

```
num=20
```

```
num=20
```

```
echo $((num++))
```

```
sayak@Ubuntu:~/harry$ vim pre_increment.sh
```

```
sayak@Ubuntu:~/harry$ vim post_increment.sh
```

```
sayak@Ubuntu:~/harry$ bash post_increment.sh
```

```
20
```

```
num=20
```

```
echo $((num++))
```

```
echo $num
```

```
sayak@Ubuntu:~/harry$ vim 2_post_increment.sh
```

```
sayak@Ubuntu:~/harry$ bash 2_post_increment.sh
```

```
20
```

```
21
```

Exit status

- Every command exits with a numeric status
- 0 - true or success
- non zero value denotes a particular type of error
- For eg: when you type the command `ls`, it returns an exit status (not displayed on standard output). You can display the exit status using the command echo \$?

```
sayak@Ubuntu:~/harry$ cd soham
```

```
sayak@Ubuntu:~/harry/soham$ echo $?
```

```
0
```

```
sayak@Ubuntu:~/harry/soham$ ls
```

```
sayak@Ubuntu:~/harry/soham$ echo $?
```

```
0
```

Common Codes

Exit Status

- **0 - Success:** Command executed successfully without errors.
- **1 - General Errors:** Often used to indicate that something went wrong, but not specific.

```
sayak@Ubuntu:~/harry$ chmod 000 2.txt
```

```
sayak@Ubuntu:~/harry$ cat 2.txt
```

```
cat: 2.txt: Permission denied
```

```
sayak@Ubuntu:~/harry$ echo $?
```

```
1
```

```
# But, I thought, the error code should be 126?!
```

2 - Misuse of Shell Builtins: Incorrect usage of shell built-in commands.

```
sayak@Ubuntu:~/harry/soham$ ls -qwer
ls: invalid line width: 'er'
sayak@Ubuntu:~/harry/soham$ echo $?
2
```

126 - Command Not Executable: Permission issues or command not found.

- **127 - Command Not Found:** The command couldn't be found or isn't executable.

```
1
sayak@Ubuntu:~/harry$ abc
• Command 'abc' not found, but there are 17 similar ones.
sayak@Ubuntu:~/harry$ echo $?
127
```

128+x - Fatal Errors: Signals and process interruptions (x represents the signal number).

- **130 - Script Terminated:** User interrupted the script using Ctrl+C.
- 255 - Exit Status Out of Range: Used when the exit status exceeds the valid range (0-255).

Using Exit Status

- Use the special variable **\$?** to access the exit status of the last command.
- **\$?** contains the exit status of the most recently executed command.
- You can use conditional statements and control flow based on exit status.

Example:

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

```
ls file.dat > /dev/null
```

```
status=$?  
echo $status
```

ls file.dat:

The ls command is typically used to list files and directories in a directory. In this case, it's listing a single file named "file.dat." If the file exists, the ls command will **display its name in the terminal.**

> /dev/null:

1. The > symbol is used for output redirection. In this context, it redirects the stdout of the ls command to /dev/null. /dev/null is a special file in Unix-like operating systems that acts as a black hole for data. Anything written to /dev/null is effectively discarded and has no visible output.

So, when you run `ls file.dat > /dev/null`, the following happens:

- If the "file.dat" file exists, its name will be listed by the ls command, but you won't see it because the **output is immediately sent to /dev/null and discarded.** In other words, the listing output is suppressed, and you won't see it on your terminal.
- If the "file.dat" file does not exist, there will be no output, and you won't see any error messages either because both the standard output and standard error are redirected to /dev/null.

This is commonly used when you want to check for the existence of a file without displaying any output to the user or when you're only interested in the exit status of the ls command (which can be checked using \$?) to determine whether the file exists or not.

status=\$?:

This line stores the exit status of the last command (in this case, the ls command) in the status variable. The exit status is a numerical value

that indicates whether the command was successful (usually 0) or encountered an error (non-zero).

- `echo $status`: Finally, this line prints the value of the `status` variable, which contains the exit status of the `ls` command.

Example 2

```
#!/bin/bash

ls file.dat > /dev/null

if [ $? -ne 0 ]; then
echo "file not found"
fi
```

File Redirection

- `>`: Redirects standard output to a file, overwriting if exists.

```
echo "NewFile" > output.txt
```

- `>>`: Appends standard output to a file.

```
echo "Appended" >> output.txt
```

• **2>: Redirects standard error to a file.**

```
ls non_existent 2> error.txt
```

```
sayak@Ubuntu:~/harry$ cat 1.txt
** : exponentiation
• % : modulo (remainder)
• -= : subtraction assignment
• += : addition assignment
• *= : multiplication assignment
sayak@Ubuntu:~/harry$ ls 100.txt 2> 1.txt
sayak@Ubuntu:~/harry$ cat 1.txt
ls: cannot access '100.txt': No such file or directory
```

- **&>:** Redirects both output and error to the same file.

```
ls non_existent &> output_err.txt
```

- **>/dev/null:** Redirects output to null device, discarding it.

```
ls non_existent > /dev/null
```

```
sayak@Ubuntu:~/harry$ ls 1.txt > /dev/null  
sayak@Ubuntu:~/harry$
```

- **<:**

Redirects input from a file.

- Example: while read line; do ... done < input.txt

```
sayak@Ubuntu:~/harry$ cat < 10.txt  
When I was One,  
I had just begun.  
When I was Two,  
I was nearly new.  
When I was Three  
I was hardly me.  
When I was Four,  
I was not much more  
.
```

- **<<: Takes input from script**

```
sayak@Ubuntu:~/harry$ cat < 11.sh  
while read line; do ... done < 10.txt  
  
sayak@Ubuntu:~/harry$ cat << 11.sh  
>
```

Here, < sign is working , but not << ; why??

Example: here-doc

```
cat << EOF > file.txt
```

Multiline text

EOF

This is a way to create and populate a file named "file.txt" with multiline text using a "here document" or "here-doc" in Bash. Here's what each part of this command does:

cat << EOF > file.txt:

The `cat` command is used here as a means to write content to a file. The `<< EOF` part is a here-doc delimiter, indicating the start of a multiline text block. The `EOF` at the end of the here-doc is used to mark the end of the text block. The `>` operator is used for output redirection, indicating that the content of the here-doc should be written to the file "file.txt."

Multiline Text: Between the `<< EOF` and `EOF` delimiters, you can include multiple lines of text. These lines will be written to "file.txt" as the content of the file. In your example, there's a single line with the text "Multiline text."

So, when you run this command, it will create a file named "file.txt" if it doesn't exist, or it will overwrite the contents of "file.txt" if it does exist, and the content of the file will be:

```
arduino
```

```
Multiline text
```

The here-doc syntax is a convenient way to create or populate files with multiline text directly from the command line or within a shell script. You can use any delimiter you like in place of "EOF," but "EOF" is commonly used for this purpose. Below another example is given:

```
sayak@Ubuntu:~/harry$ cat << EOF > hc.txt
> This is a multiline text block.
> The delimiter (EOF) marks the end.
~ EOF
sayak@Ubuntu:~/harry$ cat << EOF > for.sh
> for num in 1 2 3 4 5; do
> echo "Number: $num"
> done
> EOF
sayak@Ubuntu:~/harry$ bash for.sh
Number:
Number:
Number:
Number:
Number:
```

```
for num in 1 2 3 4 5; do
echo "Number: $num"
done
```

#~~Here, a problem is occurring for the above case. Variables like \$num (literally everything with \$ sign) is not going to for.sh file. So this error occurs in here-doc file. So I think writing bash scripts in vim is better.~~

No, it will work if you put “\\$” in the place of only “\$”. See below:

```
sayak@Ubuntu:~/harry$ cat << EOF > while.sh
while [[ \$count -le 5 ]]; do
echo "Count: \$count"
((count++))
done
EOF
sayak@Ubuntu:~/harry$ bash while.sh
Count:
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
sayak@Ubuntu:~/harry$
```

Constructs

For Loop:

```
for variable in value1 value2 ...; do
# code to execute for each value
```

```
done
```

Example:

```
for num in 1 2 3 4 5; do  
echo "Number: $num"  
done
```

```
sayak@Ubuntu:~/harry$ vim for.sh  
sayak@Ubuntu:~/harry$ bash for.sh  
Number: 1  
Number: 2  
Number: 3  
Number: 4  
Number: 5  
sayak@Ubuntu:~/harry$ █
```

While Loop:

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

Example

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

```
sayak@Ubuntu:~/harry$ bash while.sh  
Count:  
Count: 1  
Count: 2  
Count: 3  
Count: 4  
Count: 5  
sayak@Ubuntu:~/harry$ █
```

Always remember the space
between “[“ and “\$count”.

If Statement:

```
if [ condition ]; then  
# code to execute if condition is true  
elif [ another_condition ]; then  
# code to execute if another_condition is true  
else  
# code to execute if none of the conditions are true  
fi
```

Important points:

SPACES around the [,] is important.

Elif and else portions are optional to construct an ‘if’ statement.

• **[[...]] Construct -**

construct is an extended conditional expression in Bash.

- offers enhanced features compared to the [...] command.
- Supports logical operators: && (AND), || (OR)
- Supports pattern matching using * and ?
- Regular expressions: =~

• **Example:**

- `if [[$x -lt $y]]; then echo "$x is less than $y"; fi`

```
# sayak@Ubuntu:~/harry$ cat << EOF > if1.sh
x=4
y=5
if [[ $x -lt $y ]]; then echo "$x is less than $y"; fi

EOF
sayak@Ubuntu:~/harry$ bash if1.sh
4 is less than 5
sayak@Ubuntu:~/harry$
```

Using “here-doc” format, we wrote and executed the bash file in the terminal only, no need to open vim. Only “\” required before “\$”.

- if [[\$x -lt \$y && \$string == "Hello, World!"]]; then echo "\$x is less than \$y AND string matches" ; fi

```
sayak@Ubuntu:~/harry$ cat << EOF > if2.sh
x=4
y=5
string="Hello, World!"
if [[ $x -lt $y && $string == "Hello, World!" ]]; then
echo "$x is less than $y AND string matches" ; fi

EOF
sayak@Ubuntu:~/harry$ bash if2.sh
4 is less than 5 AND string matches
sayak@Ubuntu:~/harry$
```

find consecutive characters in a string:

- if [[\$string =~ o{2}]]; then echo "String contains two consecutive o's"; fi

```
sayak@Ubuntu:~/harry$ cat << EOF > cc.sh
string=" He is a good boy."
if [[ $string =~ o{2} ]]; then echo "String contains two consecutive o's"; fi

EOF
sayak@Ubuntu:~/harry$ bash cc.sh
String contains two consecutive o's
sayak@Ubuntu:~/harry$
```

operators for If Statement:

- -eq: Equal to
- -ne: Not equal to
- -lt: Less than
- -le: Less than or equal to
- -gt: Greater than
- -ge: Greater than or equal to
- ==: Equal to
- !=: Not equal to
- =~: Regular expression match operator
- &&: Logical AND
- ||: Logical OR
- !: Logical NOT
- -e: File exists
- -f: File exists and is a regular file
- -d: File exists and is a directory
- -s: File is not empty
- -r: File is readable
- -w: File is writable
- -x: File is executable
- -z: String is empty
- -n: String is not empty

Example:

```
#!/bin/bash
read -p "Enter the path to an existing file: "
file
if [[ -e "$file" ]]; then
if [[ -f "$file" ]]; then
echo "$file is a regular file."
fi
if [[ -d "$file" ]]; then
echo "$file is a directory."
fi
if [[ -s "$file" ]]; then
echo "$file is not empty."
fi
else
echo "$file does not exist."
exit 1
fi
```

#when you encounter

`exit 1` in a script or command, it usually means that the program or script encountered an error or failed to execute its intended task. The specific meaning of the error might vary depending on the context in which `exit 1` is used. Here, “`exit 1`” is used in case when the code can not find any file with the user provided name.

Output:

```
Enter the path to an existing file: /home/sayak/harry
/home/sayak/harry is a directory.
/home/sayak/harry is not empty.
sayak@Ubuntu:~/harry$ bash check.sh
Enter the path to an existing file: /home/sayak/harry
/1.txt
/home/sayak/harry/1.txt is a regular file.
/home/sayak/harry/1.txt is not empty.
sayak@Ubuntu:~/harry$ bash check.sh
Enter the path to an existing file: /home/sayak/arun.
txt
/home/sayak/arun.txt does not exist.
sayak@Ubuntu:~/harry$
```

case Statement:

```
case "$variable" in
pattern1)
# code to execute for pattern1
;;
pattern2)
# code to execute for pattern2
;;
*)
# code to execute for other patterns
;;
esac
```

Example 1

```
#!/bin/bash
read -p "Enter choice: " choice
case $choice in
1) ls -l ;;
2) ps -f ;;
3|4) date ;;
```

```
5) who ;;  
*) break ;;  
esac
```

2) **ps -f** ; ;: If the value of **choice** is "2," it will execute the **ps -f** command, which displays a **detailed list of processes**.

- 3|4) **date** ; ;: If the value of **choice** is either "3" or "4," it will execute the **date** command, which displays the current date and time.
- *) **break** ; ;: If the value of **choice** doesn't match any of the specified cases (i.e., it's not 1, 2, 3, 4, or 5), the script will break out of the **case** statement and likely terminate or continue with other parts of your script.

```

sayak@Ubuntu:~/harry$ cat << EOF > cons.sh
> read -p "Enter choice: " choice
> case \$choice in
1) ls -l ;;
2) ps -f ;;
3|4) date ;;
5) who ;;
*) break ;;
esac
> EOF
sayak@Ubuntu:~/harry$ bash cons.sh
Enter choice: 4
Monday 04 September 2023 11:26:58 AM IST
sayak@Ubuntu:~/harry$ 3
3: command not found
sayak@Ubuntu:~/harry$ cat << EOF > cons.sh
read -p "Enter choice: " choice
case \$choice in
1) ls -l ;;
2) ps -f ;;
3|4) date ;;
5) who ;;
*) break ;;
esac
EOF
sayak@Ubuntu:~/harry$ bash cons.sh
Enter choice: 4
Monday 04 September 2023 11:27:36 AM IST
sayak@Ubuntu:~/harry$ bash cons.sh
Enter choice: 3
Monday 04 September 2023 11:27:42 AM IST
sayak@Ubuntu:~/harry$ bash cons.sh
Enter choice: 2
UID          PID      PPID  C STIME TTY          TIME CMD
sayak        4598      4580  0 Sep03 pts/0      00:00:01 bash
sayak        27331     4598  0 11:27 pts/0      00:00:00 bash cons.sh
sayak        27332     27331  0 11:27 pts/0      00:00:00 ps -f
sayak@Ubuntu:~/harry$ bash cons.sh
Enter choice: 5
sayak    tty2          2023-09-03 10:15 (tty2)
sayak@Ubuntu:~/harry$ bash cons.sh
Enter choice: *
cons.sh: line 7: break: only meaningful in a 'for', 'while', or 'until' loop
sayak@Ubuntu:~/harry$ 

```

Example 2

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

```
while true; do
read -p "Enter choice: " choice
```

```
case $choice in
```

```
1) ls -l ;;
```

```
2) ps -f ;;
```

```
3|4) date ;;
```

```
5) who ;;
```

```
*) break ;;
```

```
esac
```

```
done
```

```
sayak@Ubuntu:~/harry$ vim Cons2.sh
sayak@Ubuntu:~/harry$ bash Cons2.sh
Enter choice: 3
Monday 04 September 2023 11:50:29 AM IST
Enter choice: 5
sayak      tty2          2023-09-03 10:15 (tty2)
Enter choice: 7
sayak@Ubuntu:~/harry$ █
```

while true; do: This starts an infinite loop. The script will continue to execute the loop until the **break** statement is encountered.

- **read -p "Enter choice: "** **choice:** This line prompts the user to enter a choice and stores the input in the variable **choice**.
- **case \$choice in:** This begins a **case** statement, which evaluates the value of the **choice** variable to determine which action to perform.

done: This closes the **while** loop.

Important Note:

But, if you skip the “**break**” statement:

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

```
while true; do
```

```
    read -p "Enter choice: " choice
```

```
case $choice in
 1) ls -l ;;
 2) ps -f ;;
 3|4) date ;;
 5) who ;;
[REDACTED]
esac
done
```

in the above script, then the output will be like:

```
sayak@Ubuntu:~/harry$ vim Cons2.sh
sayak@Ubuntu:~/harry$ bash Cons2.sh
Enter choice: 3
Monday 04 September 2023 11:53:58 AM IST
Enter choice: 5
sayak    tty2          2023-09-03 10:15 (tty2)
Enter choice: 7
Enter choice: 4
Monday 04 September 2023 11:54:27 AM IST
Enter choice: *
Enter choice: █
```

So the process will not be terminated automatically even if you provide wrong choice like 7, * etc. You should type ctrl+c to kill the process.

Bash Script Basic Syntax

```
%%bash

#!/bin/bash

# Variable declaration
variable_name=value

# Conditional statements
if [ condition ]; then
    # block 1
elif [ another_condition ]; then
    # block 2
else
    # block 3
fi

# Loops
for item in list; do
    # block 4
done

while [ condition ]; do
    # block 5
done

# Functions
function_name() {
    # block 6
}
```

07/09/2023

How many types operators are there in bash scripting?

Arithmetic Operators:

Arithmetic operations can be performed using the `$(())` construct.

- Addition +
- Subtraction -
- Multiplication *
- Division /
- Modulus %

Example:

bash

```
• # Arithmetic operations  
result=$((5 + 3))
```

Comparison Operators:

- Bash uses various operators to compare values, often used within conditional statements.
- Equal to == (for strings) or -eq (for numbers)
- Not equal to != (for strings) or -ne (for numbers)
- Greater than -gt
- Less than -lt
- Greater than or equal to -ge
- Less than or equal to -le
- These operators are frequently used with if statements to make decisions based on comparisons.

Example:

bash

- # Comparison operators

```
if [ "$a" -eq "$b" ]; then
    echo "a is equal to b"
fi
```

Example:

bash

- # String comparison operators

```
if [ "$str1" = "$str2" ]; then
    echo "str1 is equal to str2"
fi
```

- ***Using '=' is it ok, or '==' should be used for string comparison?***

In Bash scripting, you can use either = or == for string comparison, but there is a subtle difference between the two:

1. Using = for String Comparison:

- The = operator is used for string comparison in Bash.
- When you use =, it performs a simple string comparison, treating the strings as text.
- If the strings are equal, the condition evaluates to true.

Example:

bash

- str1="Hello"

```
str2="Hello"
```

```
if [ "$str1" = "$str2" ]; then
    echo "str1 is equal to str2"
fi
```

- **Using == for String Comparison:**

- The == operator is also used for string comparison in Bash, but it has an additional feature.
- When you use ==, it performs a string comparison, similar to =, but it can also perform pattern matching when used within double square brackets [[. . .]].

Example:

bash

```
2.      str1="Hello"
```

```
str2="Hello"
```

```
if [[ "$str1" == "$str2" ]]; then
    echo "str1 is equal to str2"
fi
```

Both = and == can be used for simple string comparisons, and in your example, using = is perfectly fine if you only want to check whether str1 is equal to str2.

However, if you need to perform more complex string comparisons involving pattern matching or if you want to work with variables that may contain spaces or special characters, using [[. . .]] with == is often preferred. It provides more flexibility and robustness in string comparisons.

In summary, for simple string equality checks, either = or == can be used in Bash (Verified by me !), but == within [[. . .]] offers additional features for more complex string comparisons.

Logical Operators:

- These operators are used to work with Boolean values and perform logical operations.
- **Logical AND &&**

Example:

bash

```
# Logical operators

if [ "$condition1" -eq 1 ] && [ "$condition2" -eq 2 ];
then

    echo "Both conditions are true"

fi
```

```
# sayak@ubuntu:~/harry/this$ bash comparison.txt
comparison.txt: line 1: condition1: command not found
comparison.txt: line 1: condition2: command not found
sayak@ubuntu:~/harry/this$ cat << EOF > comparison.txt
condition1=1; condition2=2;      if [[ "\$condition1" -eq 1 ]] &
& [[ "\$condition2" -eq 2 ]]; then
        echo "Both conditions are true"
    fi

EOF
sayak@ubuntu:~/harry/this$ bash comparison.txt
Both conditions are true
```

Observation: If we provide space in front or back of the assignment operator “=”, the script does not work! Why??

- **Logical OR ||**

The || operator combines two conditions and returns true if at least one of the conditions is true.

```
if [ "$day" == "Saturday" ] || [ "$day" ==  
"Sunday" ]; then  
    echo "It's the weekend!"  
fi
```

- **Logical NOT !**

The ! operator negates a condition, changing true to false and vice versa.

It is used to reverse the result of a condition.

Example:

bash

```
if ! [ "$status" == "error" ]; then  
    echo "No errors found."  
fi
```

Example:

bash

```
read -p "Enter a number: " num  
if ! [ "$num" % 2 -eq 0 ]; then  
    echo "The number is odd."  
else  
    echo "The number is even."  
fi
```

(NOT working.Why??!!)

Assignment Operators:

Bash uses the = operator to assign values to variables.

- Simple assignment =

Example:

```
bash
```

- # Assignment operator

```
variable="Hello, World!"
```

Ternary Operator (Conditional Operator):

- Ternary operator is not available in bash, but you can simulate it using if-else statements.

Example:

```
bash
```

```
# Simulated ternary operator  
result=$((condition ? value_if_true : value_if_false))
```

File Operators:

- Bash has operators to check file properties.
- -e checks if a file exists.
- -f checks if a file is a regular file.
- -d checks if a file is a directory.
- -r checks if a file is readable.
- -w checks if a file is writable.
- -x checks if a file is executable.

Bash Scripts example:

Examples

```
#!/bin/bash  
  
for i in apple banana grape; do  
    echo 'Fruit: $i'  
done  
  
for j in red white brown; do  
    echo "Color: $j"  
done  
  
echo "Loop finished"
```

There are a few issues in your Bash script.
Let's correct them:

1. You should use double quotes " to enable variable expansion in echo statements.

2. In the second loop, you have a typo.
It should be echo instead of ecno.

The correct script is as following:

```
sayak@ubuntu:~/harry/this$ cat << EOF > FRUIT.sh
> for i in apple banana grape; do
    echo "Fruit: \$i"
done

for j in red white brown; do
    echo "Color: \$j"
done

echo "Loop finished"
> EOF
sayak@ubuntu:~/harry/this$ bash FRUIT.sh
Fruit: apple
Fruit: banana
Fruit: grape
Color: red
Color: white
Color: brown
Loop finished
sayak@ubuntu:~/harry/this$
```

Using “**here-doc**” format, we wrote and executed the bash file in the terminal only, no need to open vim. Only “\” required before “\$”.

Example : loop with 'seq' command

```
sayak@ubuntu:~/harry/this$ cat << EOF
> count.sh
> #!/bin/bash
count=3
for ((i = 1; i <= count; i++))
do
echo "Iteration: \$i"
done
for i in \$(seq 1 5); do
echo "Number: \$i"
echo "End of loop"
done
echo "Loops finished"
> EOF
sayak@ubuntu:~/harry/this$ bash count.sh
Iteration: 1
Iteration: 2
Iteration: 3
Number: 1
End of loop
Number: 2
End of loop
Number: 3
End of loop
Number: 4
End of loop
Number: 5
End of loop
Loops finished
```

There is a small issue with the placement of the "End of loop" message, which should be outside the inner loop to avoid it being printed multiple times. Look, in this format it is coming for multiple times.

\\$(seq 1 5): This part generates a sequence of numbers using the seq command. In this case, it generates numbers from 1 to 5. The **\\$(. . .)** syntax is used for command substitution, which means the output of the seq command (the sequence of numbers) will be used as the list for the loop.

Example: Loop with {1...n}

```
sayak@ubuntu:~/harry/this$ cat << EOF > num.sh
for num in {1..5}; do
    echo "Number: \$num"
done

EOF
sayak@ubuntu:~/harry/this$ bash num.sh
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Example : Getting users' informations

```
sayak@ubuntu:/etc$ chmod 700 passwd # I dont have the permission  
chmod: changing permissions of 'passwd': Operation not permitted
```

to extract the informations.
Still i am explaining below.

```
#!/bin/bash  
  
while read -r username rest; do  
echo $username  
  
done </etc/passwd
```

while read -r username rest; do: This line starts a **while** loop. The **read** command is used to read lines of text. In this case, it reads a line from the **/etc/passwd** file and splits it into two fields: **username** and **rest**. The **-r option is used** with **read** to preserve backslashes, ensuring that special characters in the input are treated as literal characters.

echo "\$username": Inside the loop, this line echoes the **username** variable, which contains the username extracted from the current line of the **/etc/passwd** file.

The **/etc/passwd** file is a system file on Unix-like operating systems that contains information about user accounts on the system, including usernames, user IDs, home directories, and more. Your script focuses on extracting and displaying the usernames from this file.

Example: [[[:upper:]] , [[[:lower:]]]

```
sayak@ubuntu:~/harry>this$ cat << EOF > case.sh
#!/bin/bash
read -p "Enter your choice: " choice
case "$choice" in
    [[[:upper:]]])
        echo "Uppercase letter."
        ;;
    [[[:lower:]]])
        echo "Lowercase letter."
        ;;
    *)
        echo "Something else."
        ;;
esac

EOF
sayak@ubuntu:~/harry>this$ bash case.sh
Enter your choice: W
Uppercase letter.
sayak@ubuntu:~/harry>this$ bash case.sh
Enter your choice: a
Lowercase letter.
sayak@ubuntu:~/harry>this$ bash case.sh
Enter your choice: Da
Something else.
```

Pionts to be noted:

The `case` statement should be followed by the variable or expression you want to match, enclosed in double quotes, not single ("\$choice" in this case).

Each pattern within the `case` statement should be followed by `;;` to indicate the end of that particular case.

Explanation:

`read -p "Enter your choice: " choice`: This line prompts the user to enter a choice and stores the input in the `choice` variable.

`case "$choice" in`: This line starts the `case` statement and specifies the variable or expression to be matched (in this case, the value of the `choice` variable, enclosed in double quotes).

`[[[:upper:]]]` is a character class that represents all uppercase letters.

- If the input matches an uppercase letter, the script echoes "Uppercase letter."
- The `;;` indicates the end of this case block.

*): This is the default case, which is executed if none of the previous patterns match. It uses the wildcard * to match any other input.

Example: logical “&&” and comparison “-eq”

Examples

```
#!/bin/bash
x=10
y=5

if [ $x==$y]; then echo "x is equal to y"; fi

if [ $x -gt $y && $x -lt 20 ]; then
    echo "x is between 5 and 20"
fi
```

```
#!/bin/bash
x=10
y=5

# Use spaces inside square brackets for
# proper syntax.
if [ "$x" -eq "$y" ]; then
    echo "x is equal to y"
fi

# Use double brackets for logical AND,
# and include individual comparisons inside.
if [ "$x" -gt "$y" ] && [ "$x" -lt 20 ]
; then
    echo "x is between 5 and 20"
fi

~
```

Left figure have some problem. Right one is the correct one. “==” is used for strings and “-eq” is used for numbers.

```
#!/bin/bash
x=10
y=5

# Use spaces inside square brackets for
# proper syntax.
if [ $x -eq $y ]; then
    echo "x is equal to y"
fi

# Use double brackets for logical AND,
# and include individual comparisons inside.
if [ $x -gt $y ] && [ $x -lt 20 ]; then
    echo "x is between 5 and 20"
fi
```

This also works. Variables are without double quotes.

```
if [ $x -gt $y ] && $x -lt 20 ]; then
```

But, if there is no square bracket in between curly bracket it throws error

```
sayak@ubuntu:~/harry>this$ bash operator.sh
operator.sh: line 11: [: missing `]'
```

Here's a summary of the corrections made:

1. Added spaces inside the square brackets [] in the **if** statements for proper syntax.
2. Used the correct operator **-eq** for integer equality comparison in the first **if** statement.
3. Used double brackets [[]] for logical AND (**&&**) in the second **if** statement and included individual comparisons within separate brackets.

Example: ‘break’ in a infinite while loop.

Example 1:

```
sayak@ubuntu:~/harry>this$ cat << EOF > break0.sh
#!/bin/bash

while :
do
    read -p "Enter a number (0 to exit): " num
    echo "You entered: \$num"

    if [[ \$num -eq 0 ]]; then
        echo "Exiting the loop."
        break
    fi
done

EOF
sayak@ubuntu:~/harry>this$ bash break0.sh
Enter a number (0 to exit): 0
You entered: 0
Exiting the loop.
sayak@ubuntu:~/harry>this$ bash break0.sh
Enter a number (0 to exit): 56
You entered: 56
Enter a number (0 to exit): 234
You entered: 234
Enter a number (0 to exit): 0
You entered: 0
Exiting the loop.
sayak@ubuntu:~/harry>this$ 
```

The loop starts with **while** : to create an infinite loop. The loop will continue until the **break** statement is encountered when the user enters 0.

In Bash scripting, the colon (:) is a special shell command that is essentially a no-op, meaning it does nothing. When you use **while** :, you're creating an infinite loop because the : command always returns a

success status (exit code 0), and as a result, the loop condition is always true.

If you omit the colon or write `while` without a command or condition after it, (see next example) you will encounter a syntax error. See below:

```
# sayak@ubuntu:~/harry/this$ bash break0.sh
break0.sh: line 4: syntax error near unexpected token `do'
break0.sh: line 4: `do'
sayak@ubuntu:~/harry/this$
```

Added break inside the loop to exit it when the user enters 0.

Example 2:

```
sayak@ubuntu:~/harry/this$ cat << EOF > num0.sh
#!/bin/bash
num=-1
while [[ \$num -ne 0 ]]; do
read -p "Enter a number (0 to exit): " num
echo "You entered: \$num"
done

EOF
sayak@ubuntu:~/harry/this$ bash num0.sh
Enter a number (0 to exit): 0
You entered: 0
sayak@ubuntu:~/harry/this$ bash num0.sh
Enter a number (0 to exit): 8
You entered: 8
Enter a number (0 to exit): 67
You entered: 67
Enter a number (0 to exit): 908
You entered: 908
Enter a number (0 to exit): 0
You entered: 0
sayak@ubuntu:~/harry/this$ cat << EOF > break0.sh
```

What is the difference between example 1 and 2; both having same output?

For Example 1: This code block doesn't explicitly initialize the `num` variable with an initial value. Instead, it relies on the user's first input to set the value of `num`.

- It uses `while` : to create an infinite loop, which means the loop starts immediately without any predefined initial value for `num`.

- The loop checks if `num` is equal to 0 inside the loop and breaks out of the loop if the condition is met.

For example 2: This code block initializes the `num` variable with an initial value of -1 before entering the loop. This means that the `loop will execute at least once`, even if the user enters 0 on their first input.

Simple calculator in bash

Algorithm

Start

|

|__ Read user's choice (multiplication, division, etc)

|__ Read first number

|__ Read second number

| |

| |__ If choice is X (addition / subtraction / multiplication

/ division)

| |__ Else (invalid choice)

|

|__ 7. Print result

|__ End

Example 1

Integers calculator

Important points to be noted:

- Usage of `If` statement
- Works for integer numbers

```
#!/bin/bash

echo "Simple Integer Calculator"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read -p "Enter your choice (1/2/3/4): " choice

read -p "Enter the first number: " num1
read -p "Enter the second number: " num2

if [ "$choice" -eq 1 ]; then
    result=$((num1 + num2))
elif [ "$choice" -eq 2 ]; then
    result=$((num1 - num2))
elif [ "$choice" -eq 3 ]; then
    result=$((num1 * num2))
elif [ "$choice" -eq 4 ]; then
    if [ "$num2" -eq 0 ]; then
        result="Error: Division by zero"
    else
        result=$((num1 / num2))
    fi
else
    echo "Invalid choice"
    exit 1
fi

echo "Result: $result"
```

Example 2

Integers calculator

Important points to be noted:

- Usage of `case` statement
- Works for integer numbers

```
#!/bin/bash

echo "Simple Integer Calculator"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read -p "Enter your choice (1/2/3/4): " choice

read -p "Enter the first number: " num1
read -p "Enter the second number: " num2

case $choice in
    1) result=$((num1 + num2)) ;;
    2) result=$((num1 - num2)) ;;
    3) result=$((num1 * num2)) ;;
    4)
        if [ "$num2" -eq 0 ]; then
            result="Error: Division by zero"
        else
            result=$((num1 / num2))
        fi
    ;;
    *) echo "Invalid choice"; exit 1 ;;
esac

echo "Result: $result"
```

Example 3

Real numbers calculator

Important points to be noted:

- Usage of `case` statement
- Works for both integers and real numbers

```
#!/bin/bash

echo "Simple Calculator"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read -p "Enter your choice (1/2/3/4): " choice

read -p "Enter the first number: " num1
read -p "Enter the second number: " num2

case $choice in
    1) result=$(echo "$num1 + $num2" | bc) ;;
    2) result=$(echo "$num1 - $num2" | bc) ;;
    3) result=$(echo "$num1 * $num2" | bc) ;;
    4)
        if [ $(echo "$num2 == 0" | bc) -eq 1 ]; then
            result="Error: Division by zero"
        else
            result=$(echo "scale=2; $num1 / $num2" | bc)
        fi
        ;;
    *) echo "Invalid choice"; exit 1 ;;
esac

echo "Result: $result"
```

What is the difference between the example 2 and example 3?

```
result=$(echo "$num1+$num2" | bc);;
```

```
result= $(( num1+num2 ));
```

What are the differences between these two lines?

The two lines you provided perform arithmetic operations using different syntax in Bash:

1. Using bc for Floating-Point Arithmetic:

bash

- `result=$(echo "$num1+$num2" | bc)`
- This line uses the `bc` (basic calculator) command to perform floating-point arithmetic. It takes the values of `num1` and `num2`,

concatenates them with the + operator inside the echo command, and then pipes the resulting expression to bc. bc evaluates the expression and returns the result as a floating-point number.

- **Using Shell Arithmetic with ((. . .)):**

bash

```
result=$((num1+num2))
```

- This line uses shell arithmetic within double parentheses ((. . .)) to perform integer arithmetic. It directly adds the values of num1 and num2 together and stores the result as an integer. It will round down any fractional part.

Example 4

Real numbers calculator

Important points to be noted:

- Simple code

```
#!/bin/bash

declare -A operations=(
    [1]="+"
    [2]="-"
    [3]="*"
    [4]="/"
)

echo "Simple Real Number Calculator"
echo "1. Addition"
echo "2. Subtraction"
echo "3. Multiplication"
echo "4. Division"
read -p "Enter your choice (1/2/3/4): " choice

read -p "Enter the first number: " num1
read -p "Enter the second number: " num2

operation="${operations[$choice]}"
result=$(echo "$num1 $operation $num2" | bc -l)

echo "Result: $result"
```

The code in example 4, is creating an associative array in Bash.

- **declare -A operations:** This line declares an associative array named **operations**. The **-A** option is used to specify that the variable **operations** should be treated as an associative array.
- **[1]="+"**, **[2]="-"**: These lines define key-value pairs in the associative array. In this example, **key 1** is associated with the value **+**, and **key 2** is associated with the value **-**.

An associative array allows you to store and retrieve values using keys. You can access the values using the keys you defined. In this example, **\$ {operations[\$choice]}** retrieves the value associated with user input **choice**, which may be **+, -, *, /**.

Example 5: Associative array

```
#!/bin/bash

# Declare an associative array to map numerical day of the week to its name
declare -A days_of_week=
[1]="Monday"
[2]="Tuesday"
[3]="Wednesday"
[4]="Thursday"
[5]="Friday"
[6]="Saturday"
[7]="Sunday"
)

# Main script
read -p "Enter a numerical day of the week (1-7): "
numeric_day

# Check if the input is a valid numeric day (between 1 and 7)
if [[ $numeric_day =~ ^[1-7]$ ]]; then
    day_name="${days_of_week[$numeric_day]}"
    echo "Day of the week: $day_name"
else
    echo "Invalid input. Please enter a number between 1 and 7."
fi
```

```
sayak@ubuntu:~/harry/this$ bash weekdays.sh
Enter a numerical day of the week (1-7): 5
Day of the week: Friday
sayak@ubuntu:~/harry/this$ bash weekdays.sh
Enter a numerical day of the week (1-7): 0
Invalid input. Please enter a number between 1 and 7.
sayak@ubuntu:~/harry/this$
```

Here's how this script works:

1. It declares an associative array named `days_of_week`, which maps numerical representations (1-7) to the names of days of the week.
2. In the main script, it reads a numerical day of the week from the user.
3. It checks if the input is a valid numeric day (between 1 and 7) using a regular expression.
4. If the input is valid, it retrieves the corresponding day name from the associative array and displays it. If the input is invalid, it displays an error message.

Example 6:

```
#!/bin/bash

# Declare an associative array to map operators to their corresponding functions
declare -A operators=(
    [ "+"]="+"
    [ "-"]="-"
    [ "*"]= "*"
    [ "/"]= "/"
)

# Function to perform arithmetic operation
perform_operation() {
    local operator="$1"
    local num1="$2"
    local num2="$3"

    case "$operator" in
        "+") result=$((num1 + num2));;
        "-") result=$((num1 - num2));;
        "*") result=$((num1 * num2));;
        "/") result=$((num1 / num2));;
        *) echo "Invalid operator"; exit 1;;
    esac

    echo "Result: $num1 $operator $num2 = $result"
}

# Main script
read -p "Enter the first number: " num1
read -p "Enter the operator (+, -, *, /): " operator
read -p "Enter the second number: " num2

# Check if the operator exists in the associative array
if [ -n "${operators[$operator]}"]; then
    perform_operation "${operators[$operator]}" "$num1" "$num2"
else
    echo "Invalid operator"
    exit 1
fi
```

```
sayak@ubuntu:~/harry/this$ vim arithmatic.sh
sayak@ubuntu:~/harry/this$ bash arithmatic.sh
Enter the first number: 12
Enter the operator (+, -, *, /): -
Enter the second number: 15
Result: 12 - 15 = -3
sayak@ubuntu:~/harry/this$
```

Here's what this script does:

1. It declares an associative array named `operators`, which maps arithmetic operators like `+`, `-`, `*`, and `/` to their corresponding symbols.
2. It defines a function `perform_operation` that takes an operator, two numbers, and performs the corresponding arithmetic operation using a `case` statement.
3. In the main script, it reads the first number, operator, and second number from the user.
4. It checks if the entered operator exists in the associative array.
This is a `unary test operator in Bash`. When used with `[]` (test brackets), it checks if the string inside the brackets is `non-empty` (i.e., it has one or more characters).

If it does, it calls the `perform_operation` function to calculate and display the result. If the operator is not found, it displays an error message.

Arrays

declare command

- `-i`: Declare a variable as an integer.
- `-a`: Declare a variable as an indexed array.
- `-A`: Declare a variable as an associative array.
- `-p`: Display attributes and options of variables.
- `-x`: Export a variable for child processes.

-i: Declare a variable as an integer:

- When you use `declare -i`, you declare a variable as an integer. This means that Bash will treat the variable as a numeric value, and it will perform arithmetic operations on it accordingly. If you attempt to assign a non-numeric value to an integer variable, Bash will try to interpret it as a number.

```
sayak@ubuntu:~/harry>this$ cat << EOF > integer.sh
declare -i num=5
echo \$num
EOF
sayak@ubuntu:~/harry>this$ bash integer.sh
5
sayak@ubuntu:~/harry>this$ cat << EOF > integer.sh
declare -i num="abc"
echo \$num
EOF
sayak@ubuntu:~/harry>this$ bash integer.sh
0
```

-a: Declare a variable as an indexed array:

- With `declare -a`, you declare a variable as an indexed array. Indexed arrays in Bash are arrays where elements are accessed using numeric indices. This is the default behavior for arrays in Bash, so you don't always need to use `-a`.

```
sayak@ubuntu:~/harry>this$ cat << EOF > index.sh
declare -a my_array=("apple" "banana" "cherry")
echo \${my_array[0]}
echo \${my_array[1]}
> echo \${my_array[2]}
> EOF
sayak@ubuntu:~/harry>this$ bash index.sh
apple
banana
cherry
sayak@ubuntu:~/harry>this$
```

-A: Declare a variable as an associative array:

- Using `declare -A`, you declare a variable as an associative array. Associative arrays are arrays where elements are accessed using keys instead of numeric indices. Keys can be strings or numbers, allowing you to map values to specific keys.

```

sayak@ubuntu:~/harry/this$ cat << EOF > associative.sh
declare -A fruit_prices=
[apple]=1.00
[banana]=0.75
)
read -p "Enter the fruit name (apple/banana): " choice
echo "price of the fruit: \${fruit_prices[\$choice]}"
EOF
sayak@ubuntu:~/harry/this$ bash associative.sh
Enter the fruit name (apple/banana): apple
price of the fruit: 1.00
sayak@ubuntu:~/harry/this$ bash associative.sh
Enter the fruit name (apple/banana): banana
price of the fruit: 0.75
sayak@ubuntu:~/harry/this$ █

```

Display attributes and options of variables

```

sayak@ubuntu:~/harry/this$ cat << EOF > display.sh
this> declare -i num=42
declare -a my_array=("apple" "banana" "cherry")

declare -p num my_array
> EOF
sayak@ubuntu:~/harry/this$ bash display.sh
declare -i num="42"
declare -a my_array=([0]="apple" [1]="banana" [2]="cherry")
sayak@ubuntu:~/harry/this$ █

```

example:

- `declare -i num="42"` indicates that the variable `num` is an integer (-i), and its current value is 42.
- `declare -a my_array=([0]="apple" [1]="banana" [2]="cherry")` indicates that the variable `my_array` is an indexed array (-a), and it contains elements with indices **[0]**, **[1]**, and **[2]** with their corresponding values.

The `-p` option is useful for debugging and inspecting variables in scripts.

-x: Export a variable for child processes

The process list is like below:

In

Create a Bash script

Create a child script

Make both scripts executable(if permission denied)

Run the parent script

Get Output : Variable is defined in the main script. Main script has the execution command of child script.

Though the child script is written separately, running main script gives the output of child script also.

```
sayak@ubuntu:~/harry/this$ cat << EOF > main.sh
# Export a variable
declare -x MY_VARIABLE="Hello"

# Run the child script
./child_script.sh

EOF
sayak@ubuntu:~/harry/this$ cat << EOF > child_script.sh
# Access the exported variable
echo "Child script: MY_VARIABLE = \$MY_VARIABLE"
EOF
sayak@ubuntu:~/harry/this$ bash main.sh
Child script: MY_VARIABLE = Hello
sayak@ubuntu:~/harry/this$ █
```

Examples from lecture:

```
#!/bin/bash

declare -i age=25

declare -i quantity=10

result=$((age +
quantity))

echo "Total: $result"
```

```
sayak@ubuntu:~/harry/this$ cat << EOF > age.sh
#!/bin/bash
declare -i age=25.9
declare -i quantity=10.2
result=\$((age + quantity))
echo "Total: $result"
EOF
sayak@ubuntu:~/harry/this$ bash age.sh
age.sh: line 2: declare: 25.9: syntax error: invalid
arithmetic operator (error token is ".9")
age.sh: line 3: declare: 10.2: syntax error: invalid
arithmetic operator (error token is ".2")
Total:
sayak@ubuntu:~/harry/this$
```

Clearly, variables do not
work for non- integers.

Example 2

```
sayak@ubuntu:~/harry/this$ cat << EOF > color.sh
#!/bin/bash
declare -a colors=("red" "green" "blue")
echo "First color: \${colors[0]}"
echo "Second color: \${colors[1]}"
EOF
sayak@ubuntu:~/harry/this$ bash color.sh
First color: red
Second color: green
sayak@ubuntu:~/harry/this$
```

Example 3

```
#!/bin/bash
declare -a numbers=("2.5" "3.14" "1.618")
sum=$(echo "${numbers[0]} + ${numbers[1]} + ${numbers[2]}" | bc -l)
echo "Sum of numbers: $sum"
echo "Display all variables: ${numbers[@]}"
declare -p numbers
~

sayak@ubuntu:~/harry/this$ vim numbers.sh
sayak@ubuntu:~/harry/this$ bash numbers.sh
Sum of numbers: 7.258
Display all variables: 2.5 3.14 1.618
declare -a numbers=([0]="2.5" [1]="3.14" [2]="1.618")
sayak@ubuntu:~/harry/this$
```

The **-l** option is used with **bc** to load the math library. When you use **-l**, **bc** switches to "standard math library" mode, which enables additional mathematical functions and improved precision for floating-point arithmetic.

`\${numbers[@]}` is an array expansion syntax that, when used within double quotes, expands to all the elements in the **numbers** array. In other words, it lists all the values stored in the **numbers** array.

declare -p numbers: This line uses the **declare** command with the **-p** option to display the attributes and values of the **numbers** array.

Example 4

```
#!/bin/bash
declare -A fruits
fruits["apple"]="red"
fruits["banana"]="yellow"
fruits["grape"]="purple"
echo "Color of apple: ${fruits["apple"]}"
echo "Color of banana: ${fruits["banana"]}"
declare -p fruits
```

```
sayak@ubuntu:~/harry/this$ vim fruits.sh
sayak@ubuntu:~/harry/this$ bash fruits.sh
Color of apple: red
Color of banana: yellow
declare -A fruits=([grape]="purple" [apple] =
"red" [banana]="yellow" )
sayak@ubuntu:~/harry/this$
```

```
#!/bin/bash
declare -A fruits=(
[apple]="red"
[banana]="yellow"
[grape]="purple"
)
echo "Color of apple: ${fruits[apple]}"
echo "Color of banana: ${fruits[banana]}"
declare -p fruits
```

```
sayak@ubuntu:~/harry/this$ vim fruits_modifed.sh
sayak@ubuntu:~/harry/this$ bash fruits_modifed.sh
Color of apple: red
Color of banana: yellow
declare -A fruits=([grape]="purple" [apple] =
"red" [banana]="yellow" )
```

Here, take a look. Associative arrays are written here in two different ways. But both are giving the same output.

Example 5:

```
sayak@ubuntu:~/harry/this$ cat << EOF > age2.sh
declare -a colors=("red", "green", "blue")
echo "First color: \${colors[0]}"
EOF
sayak@ubuntu:~/harry/this$ bash age2.sh
First color: red,
```

As there are some comma (,) inside the array declaration, So it is displaying “red ,” . You should remove the commas.

Example 6:

```
echo sayak@ubuntu:~/harry/this$ cat << EOF > temp.sh
#!/bin/bash
temperatures_celsius=(20 25 30 15 10 35 22 18 28 32)
echo "Celsius Fahrenheit"
echo "-----"
for celsius in "${temperatures_celsius[@]}"; do
fahrenheit=$(echo "scale=2; ($celsius * 9/5) + 32" | bc)
echo "$celsius°C $fahrenheit°F"
done
EOF
sayak@ubuntu:~/harry/this$ bash temp.sh
Celsius Fahrenheit
-----
20°C 68.00°F
25°C 77.00°F
30°C 86.00°F
15°C 59.00°F
10°C 50.00°F
35°C 95.00°F
22°C 71.60°F
18°C 64.40°F
28°C 82.40°F
32°C 89.60°F
sayak@ubuntu:~/harry/this$
```

"Celsius Fahrenheit" and echo "-----":

These lines are used to print a header for the table, creating a visual separation between the Celsius and Fahrenheit columns.

for celsius in "\${temperatures_celsius[@]}"; do:

This line starts a loop that will iterate over each value in the temperatures_celsius array. The **celsius variable** will hold each value during each iteration.

Scale=2:

This sets the **precision for the decimal places to 2**. In other words, it specifies that the result should have two digits after the decimal point.

Functions:

functions

BASIC SYNTAX:

```
function funcName {  
# commands  
}
```

OR

```
funcName() {  
# commands  
}
```

→ Returns the exit status.

WITH ARGUMENTS

```
funcName arg1 arg2
```

Example 1

```
sayak@ubuntu:~/harry>this$ cat << EOF > fun.sh  
> #!/bin/bash  
# Define a simple function  
hello() {  
echo "Hello, world!"  
}  
# Call the function  
hello  
> EOF  
sayak@ubuntu:~/harry>this$ bash fun.sh  
Hello, world!
```

Example 2

calling

```
sayak@ubuntu:~/harry/this$ cat << EOF > test.sh
> function test() {
>     value='I like CD61203 subject'
>
>     # Set the initial value
>     value='I do not like CD61203 subject'
>     # Print the current value
>     echo \$value
>     # Call function
>     test
>     # Print the updated value
>     echo \$value
> }
> EOF
sayak@ubuntu:~/harry/this$ bash test.sh
temp.sh
sayak@ubuntu:~/harry/this$ bash test.sh
I do not like CD61203 subject
I like CD61203 subject
sayak@ubuntu:~/harry/this$
```

Before
the test

function, there's an initial variable assignment `value='I do not like CD61203 subject'`. This initializes the `value` variable in the **global scope** with a specific string.

The code then uses `echo $value` to print the global `value` variable's content, which is "I do not like CD61203 subject." This is done before calling the `test` function.

The `test` function is called using `test`. However, within the function, the `value` variable is assigned a different value, "**I like CD61203 subject.**" This **assignment doesn't affect the global value variable because it operates within the function's local scope.**

After calling the function, it shows the output for variable named "value" within the function's scope.

Example 3

```
# sayak@ubuntu:~/harry>this$ cat << EOF > test2.sh
code
> function test() {
value='I like CD61203 subject'
echo \$value
}
# Set the initial value
value='I do not like CD61203 subject'
# Print the current value
echo \$value
# Call function and print the value
echo $(test)
> EOF
sayak@ubuntu:~/harry>this$ bash test2.sh
I do not like CD61203 subject
I like CD61203 subject
sayak@ubuntu:~/harry>this$
```

In the
you

provided, the first echo \$value did not work as expected because it is placed before the test function is called.

Also note that, previously we called and printed the function in the following way:

test

echo \$value

Now we did that in one line:

echo \$(test)

Example 4

```
sayak@ubuntu:~/harry>this$ cat << EOF > test3.sh
> function test() {
value=23
return \$value
}
# Set the initial value
value=100
# Print the current value
echo \$value
# Call function
test
# print the value
echo \$?
> EOF
sayak@ubuntu:~/harry>this$ bash test3.sh
100
23
```

To

summarize, the code sets a **global variable** to 100, calls a **function that sets a local variable** to 23, and then prints the exit status of the function, which is the value returned by the return statement, resulting in "23" being printed to the standard output.

echo \$? is used to **print the exit status of the last executed command**. In this case, the last executed command is the test function, which returned 23. So, echo \$? will print "23" to the standard output.

Example 5:

```
sayak@ubuntu:~/harry>this$ cat << EOF > add.sh
> #!/bin/bash
add
# Define a function that returns a value
add() {
result=\$(( \$1 + \$2 ))
return \$result
}
# Call the function and capture the return value
add 5 3
sum=\$?
echo "The sum is: \$sum"
> EOF
sayak@ubuntu:~/harry>this$ bash add.sh
The sum is: 8
```

The

function is defined. It takes two arguments, \$1 and \$2, and calculates their

sum by assigning the result to the result variable using `result=$(($1 + $2))`.

You call the add function with the arguments 5 and 3 by executing `add 5 3`.

The return value of the function, which is stored in `$?`, is captured and assigned to the sum variable using `sum=$?`.

```
sayak@ubuntu:~/harry/this$ cat << EOF > add.sh
#!/bin/bash
# Define a function that returns a value
add() {
result=$(( $1 + $2 / $3 ))
return $result
}
# Call the function and capture the return value
add 5 3 3
sum=$?
echo "The sum is: $sum"
EOF
sayak@ubuntu:~/harry/this$ bash add.sh
The sum is: 6
```

Similarly, the code works for three arguments of the function.

```
sayak@ubuntu:~/harry/this$ cat << EOF > add.sh
#!/bin/bash
# Define a function that returns a value
add() {
result=$(( $1 + $2 / $3 ))
return $result
}
# Call the function and capture the return value
add 5 3 6
sum=$?
echo "The sum is: $sum"
EOF
sayak@ubuntu:~/harry/this$ bash add.sh
The sum is: 5
```

The code does not work for non-integer values. So, “5” is displayed in place of “5.5”.

working with prac

Retrieving a value from specified file and printing: (REVISE!?)

You are provided with a directory structure where the last subdirectory contains

a file named 'out'. Write a bash script to extract the final energy for each case.

The output should be printed in the following format:

Output should be printed in the following format:

Header

Dir1

Sub-dir1 ener_val1

Sub-dir2 ener_val2

...

Dir2

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

# Directories structure with 'out' file in the last
# subdirectory

dir=$PWD/base

# Pattern to match

x='Geometry converged'

# Loop through each directory

for i in `ls -d $dir/*/`; do

pushd $i > /dev/null

dir_name=$(basename $i)

ener_found=0

# Loop through subdirectories within each directory

echo "$dir_name"

for j in `ls -d [0-9]*/`; do

grep "$x" $j/out > /dev/null

if [ $? -eq 0 ]; then

ener=$(grep 'Total Energy' $j/out | tail -n 1 | tr -s ' '
| cut -d ' ' -f 3)

echo "$j $ener"

ener_found=1

fi

done

# If no energy was found, print a message
```

```
if [ $ener_found -eq 0 ]; then
echo "$dir_name"
echo "No energy found"
fi
popd > /dev/null
done
```

This Bash script appears to be designed for a specific task within a directory structure containing 'out' files. Let me explain its functionality:

#!/bin/bash: Specifies that this script should be executed using the Bash shell.

dir=\$PWD/base: Sets the dir variable to the **current working directory** (\$PWD) followed by the subdirectory 'base'. This is the root directory for the search.

x='Geometry converged': Defines a pattern stored in the variable x, which the script will search for within the 'out' files.

The script then enters a loop to iterate through each subdirectory within the 'base' directory using **for i in ls -d \$dir/*; do**.

pushd \$i > /dev/null: Changes the current working directory to the current subdirectory (\$i) and redirects any output to /dev/null to suppress it.

dir_name=\$(basename \$i): Extracts the name of the current subdirectory and stores it in the variable dir_name. {\$i contains the full path of the current subdirectory being processed in the outer loop. **basename** is a command that extracts the final component of a path. In this case, it extracts the name of the directory without the full path. **\$(...)** is used for command substitution. It allows the result of the basename command to be assigned to the variable dir_name}.

ener_found=0: Initializes a variable ener_found to 0, which will be used to track whether the pattern was found in any 'out' files.

Inside the inner loop, **for j in ls -d [0-9]*;/ do**, the script iterates through subdirectories within the current subdirectory, expecting them to have names consisting of digits.

grep "\$x" \$j/out > /dev/null: Searches for the pattern stored in x within the 'out' file located in the current subdirectory (\$j). The > /dev/null part suppresses the output, and if the pattern is found, it sets ener_found to 1.

if [\$? -eq 0]; then: Checks if the exit status of the previous grep command (which indicates whether the pattern was found) is equal to 0, indicating a successful match.

ener=\$(grep 'Total Energy' \$j/out | tail -n 1 | tr -s ' ' | cut -d ' ' -f 3): grep to search for the string 'Total Energy' in the file located at \$j/out.

'Total Energy', tail -n 1 is used to extract the last line from the output.

tr -s ' ': This part of the command uses tr (translate or delete characters) to squeeze multiple consecutive spaces into a single space. cut -d ' ' -f 3: Finally, cut is used to split the line into fields based on spaces (' '), and -f 3 specifies that we want the third field. This is typically done to isolate and

extract the numerical value of the total energy and finally stores it to ener variable.

echo "\$j \$ener": Prints the subdirectory name and the energy value.

ener_found=1: Sets ener_found to 1, indicating that energy information was found in this subdirectory.

After the inner loop, the script checks if ener_found is still 0. If so, it means no energy information was found in any subdirectories of the current directory, and it prints a message indicating that.

Finally, **popd > /dev/null** is used to return to the parent directory after processing the subdirectories, and any output is again redirected to /dev/null to suppress it.

This script essentially searches for a specific pattern ('Geometry converged') in 'out' files within a directory structure, extracts energy information, and prints it if found. If no energy information is found, it notifies the user.

Download link of file:

https://www.dropbox.com/sh/d18w4jmye9gmayo/AAA74Aiz_fC9sXj9sshWZDT6a?dl=0

Automated daily file backup script

Create a script that performs file backups daily after 3:00 am. The script must

exclusively back up files that are either new or have been modified.

The backup

location should be set to /home1/user/backup, while the source files are located

at /home/user/work.

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

```
source_dir="/home/user/work" # This line sets the source_dir variable to the path of the source directory that you want to back up.
```

```
backup_dir="/home1/user/backup" # where the contents of source_dir will be copied.
```

```
log_file="/var/log/backup.log" # This line sets the log_file variable to the path of the log file where backup-related messages will be recorded.
```

```
current_time=$(date +"%Y-%m-%d %H:%M:%S")
```

```
echo "Backup started at $current_time" >> "$log_file" # This line appends a message to the log file, indicating the start time of the backup process.
```

```
# Check if the backup directory exists, if not, create it
```

```
if [ ! -d "$backup_dir" ]; then
```

```
mkdir -p "$backup_dir"
```

```
fi
```

```
# !: This is the logical NOT operator. It negates the condition that follows it.
```

-d "\$backup_dir": This checks whether the path stored in the variable \$backup_dir refers to a directory (-d checks if it's a directory).

mkdir -p: p: This is an option or flag for mkdir. When you use -p, it tells mkdir to create any parent directories that don't exist. In other words, it

ensures that the entire directory path specified will be created, including any intermediate directories that may be missing.

```
# Sync the source directory to the backup directory  
rsync -av --update --delete "$source_dir/" "$backup_dir/"  
>> "$log_file" 2>&1
```

This is the core of the backup operation using rsync:

-av: These options enable **verbose (-v) mode** and **archive mode (-a)**, which preserves file permissions, ownership, timestamps, and recursively copies directories.

--update: This option tells rsync to **skip files that are newer** in the destination (backup) directory, which can save time.

--delete: This option ensures that files in the backup directory are deleted if they no longer exist in the source directory.

\$source_dir/ and **"\$backup_dir/** specify the source and destination directories.

>> "\$log_file" appends the rsync output (including any error messages) to the log file.

2>&1 redirects standard error (file descriptor 2) to standard output (file descriptor 1), so both standard output and standard error are captured in the log file. It ensures that any error messages produced by the rsync command are also appended to the log file.

```
echo "Backup completed at $(date +'%Y-%m-%d %H:%M:%S')"  
>> "$log_file"
```

This line appends a message to the log file, indicating the completion time of the backup process.

Hard disk space monitoring and threshold alert script

Write a script to keep track of the available space on your hard drives.
The script

should issue a warning when the available space falls below a specified threshold

value.

```
#!/bin/bash

# Set the threshold value in percentage
threshold_percentage=10

# Loop through mounted filesystems and check disk space
while read -r fs size used avail percentage mount; do

# read command used to read a line of input and split it into multiple
variables.

fs size, used, avail, percentage, mount; these variables will hold the value
of the 1st, 2nd, 3rd, 4th, 5th and 6th word or field from the input line
respectively.

if [[ "$fs" =~ ^/dev/ ]]; then

# This line checks if the filesystem entry ($fs) begins with '/dev/'. It filters
out non-device filesystems (e.g., virtual filesystems like /dev/sda1 are
typically device-related).

available_percentage=$(echo "$percentage" | tr -d '%')

# This line removes the '%' character from the percentage value
($percentage) and stores it in the available_percentage variable as a
numeric value. ("tr" stands for translate or delete characters).

disk=$(echo "$fs" | tr -d '/')
```

```
# This line removes the '/' character from the filesystem name ($fs) to get a  
clean disk identifier, which is stored in the disk variable.
```

```
if [ "$available_percentage" -lt  
"$threshold_percentage" ]; then  
  
echo "Warning: Available space on $disk is less than  
threshold_percentage%."  
  
fi  
  
fi  
  
done < <(df -h)
```

```
# This part of the script redirects the output of the df -h command into the  
while loop. The “df -h” command lists information about mounted  
filesystems in a human-readable format, information from this command  
is read line by line in the while loop.
```

Recursive counting of files and directories

**Write a script that reports the total number of files and directories.
The counting should be performed recursively.**

```
#!/bin/bash

if [ $# -ne 1 ]; then echo "Usage: $0 <directory_path>"; exit 1; fi
```

This line is part of a script and serves as an input validation check. It verifies whether the script has been provided with the correct number of command-line arguments and provides a usage message if the condition is not met.

\$#: This is a special variable in Bash that represents the **number of command-line arguments** passed to the script or function.

echo "Usage: \$0 <directory_path>";: This part of the code prints a usage message to the standard output. \$0 represents the **name of the script itself**, and <directory_path> is a **placeholder for the expected argument**. This message informs the user of the correct usage of the script.

exit 1;: This command immediately exits the script with an exit status code of 1. An exit status code of 1 typically indicates an error or an unsuccessful execution of the script.

```
directory="$1"
```

This line assigns the first (and only) command-line argument to the variable directory, which represents the directory path to be analyzed.

```
file_count=0; dir_count=0
```

These lines initialize two variables, file_count and dir_count, to keep track of the number of files and directories found.

```
# Use 'find' to loop through items in the directory and subdirectories
```

```
while IFS= read -r item; do
```

IFS=: This part sets the Internal Field Separator (IFS) to an empty string. In this context, it ensures that leading and trailing whitespace is preserved when reading lines. By setting IFS to an empty string, you prevent Bash from trimming whitespace from the beginning and end of each line.

read -r item: This command reads a line of input and stores it in the **item variable**. Let's break down the options and parameters:

-r: This is an option for the read command. It stands for "raw" and is used to prevent backslashes from being treated as escape characters. This ensures that the input is read verbatim, without interpreting any escape sequences.

item: This is the variable where the content of each line is stored as it's read.

```
if [ -f "$item" ]; then
((file_count++))
elif [ -d "$item" ]; then
((dir_count++))
fi
done < <(find "$directory")
# The loop continues processing each item found by find until there are no more items.
echo "Total number of files: $file_count"
echo "Total number of directories: $dir_count"
```

Start, stop or monitor processes on remote machines

Write a script that allows you to manage processes on remote machines using

SSH, enabling you to start, stop, or monitor processes. It should do the following

1. Function to start a process remotely
2. Function to stop a process remotely
3. Function to monitor a process remotely

```
#!/bin/bash
u="your_remote_user"
h="remote_machine_address"

# You should replace "your_remote_user" with your actual remote
username and "remote_machine_address" with the actual address or
hostname of the remote machine.

s(){ p=$1;ssh $u@$h "nohup $p &";echo "Started $p on
$h";}

# s(): This part begins the definition of the Bash function named s. The
function starts with an opening curly brace {.
```

p=\$1;: This line **assigns the first argument** passed to the function (denoted as \$1) to a variable named p. This argument is expected to be the **name or command of the process** to be started remotely.

ssh \$u@\$h "nohup \$p &";: This line uses the ssh command to **establish an SSH connection** to a remote machine. Here's what's happening:

\$u and \$h are variables representing the remote username and remote machine's address, respectively. These variables should be defined earlier in the script with the actual values.

Inside the double quotes, the command "nohup \$p &" is executed remotely. It consists of the following:

nohup: This is a command used to run another command in such a way that it's immune to hangups (HUP signals). In other words, it allows the process to continue running even if the SSH session ends.

\$p: This is the process name or command that was passed as an argument to the function, representing the process to be started.

&: This is used to run the process in the background so that the SSH session is not blocked.

echo "Started \$p on \$h";: This line prints a message indicating that the specified process (\$p) has been started on the remote machine (\$h).

```
t(){ p=$1;ssh $u@$h "pkill -f $p";echo "Stopped $p on $h";}
```

pkill -f \$p: This is a command used to search for and terminate processes based on their names or command lines. The -f option allows searching for a process by specifying a part of its command line.

The result of this command is that it searches for a process matching the specified name or command (\$p) and sends a termination signal to that process, effectively stopping it.

```
m(){ p=$1;ssh $u@$h "pgrep -fl $p";}
```

'f' option allows searching by command line and 'l' allows listing all the processes that matches name specified with '\$p'.

```
while true; do
```

The script enters an infinite loop using while true (until user does not provide an invalid choice).

```
clear;
```

```
echo "Remote Process Management";  
echo "1. Start a process";  
echo "2. Stop a process";  
echo "3. Monitor a process";  
echo "4. Exit";  
read -p "Enter your choice: " c  
case $c in  
1) read -p "Enter the name of the process to start: " p;  
s "$p";;  
# Prompts the user to enter the name of the process to start (user provided  
process name is stored in the variable 'p' ) and calls the s() function to start  
the specified process on the remote machine.  
2) read -p "Enter the name of the process to stop: " p; t  
"$p";;  
3) read -p "Enter the name of the process to monitor: " p; m "$p";;  
4) echo "Exiting...";exit 0;;  
# 'exit 0' means successfully exits. 'Exit 1' means some error occurred.  
*) echo "Invalid choice. Please select a valid option.";;  
esac;  
# If the user enters an invalid choice, the script displays an error message.  
read -p "Press Enter to continue...";  
# The script waits for the user to press Enter to continue after each action.  
done
```

Job queue

Write a bash script that handles a job queue, allowing the execution of up to 4 jobs concurrently. Upon the completion of any job, the script should initiate the next job in the queue.

```
#!/bin/bash

ncores=4

#limit the number of concurrently running jobs based on the number of available CPU cores (ncores).

jobs=("Job1" "Job2" "Job3" "Job4" "Job5" "Job6" "Job7"
"Job8" "Job9" "Job10")

# An array called jobs is defined.

run_job() {

# This starts a function definition for run_job.

sleep_time=$((1 + $RANDOM % 5))

# It generates a random sleep time between 1 and 5 seconds. sleep_time is a variable that represents the duration of time that a job should sleep or pause before completing its execution. $RANDOM is a built-in Bash variable that generates a random integer between 0 and 32,767 (the maximum value of a signed 16-bit integer).

echo "Running $1"; sleep "$sleep_time"; echo "Completed $1"

}

# echo "Running $1": It prints a message indicating that the job is running. 1st argument (i.e. $1) represents job name?!Explanation!?
```

```
total_jobs="${#jobs[@]}"
```

`${#jobs[@]}`: This part of the line is a Bash construct used to determine the number of elements in an array named jobs. It's called an "array length" expression.

jobs: In the script, the jobs array is defined earlier and contains a list of job names or identifiers.

`@`: The `@` symbol, when used with `{}$`, is a special character that tells Bash to treat each element of the array individually.

`#`: The `#` symbol is used within `{}$` to calculate the length (i.e., the number of elements) of the array.

```
submitted_jobs=0
```

```
running_jobs=0
```

```
index=0
```

```
# Initializes:
```

The submitted_jobs variable to keep track of the number of jobs submitted;

The running_jobs variable to keep track of the number of jobs currently running;

The index variable, which is used to keep track of the current job being processed.

```
#The script then enters a while loop to manage job submission and concurrency.
```

```
while [ "$submitted_jobs" -lt "$total_jobs" ]; do  
    job="${jobs[$index]}"
```

```
# Above line assigns the value of the element in the jobs array at the current index to the job variable. It essentially selects the next job from the array to be processed.
```

```
run_job "$job" &
```

```
# This line calls the run_job function, passing the value of the job variable as an argument. The & at the end of the line runs the run_job function in the background, allowing multiple jobs to be executed concurrently.
```

```
((running_jobs++))
```

```
((index++))
```

```
# These two lines increment the running_jobs and index counters. The running_jobs counter keeps track of the number of jobs currently running, while the index counter is used to select the next job from the jobs array in the next iteration of the loop.
```

```
while [ "$running_jobs" -ge "$ncores" ]; do
```

```
wait -n
```

```
# The -n option makes wait wait for any background job to finish. Essentially, it waits for at least one of the background jobs to complete. This helps in managing the concurrency, as it ensures that there are never more than ncores jobs running at the same time.
```

```
running_jobs=$((running_jobs - 1))
```

```
# After a job completes (which is what wait -n waits for), this line decrements the running_jobs counter by 1 to reflect the fact that one job has finished and a new job can be started.
```

Done

```
# This marks the end of the while loop that waits for jobs to complete until the number of running jobs falls below the specified limit.
```

```
submitted_jobs=$((submitted_jobs + 1))
```

```
# After the loop, this line increments the submitted_jobs counter to keep track of the number of jobs that have been submitted for execution.
```

```
done
```

```
wait
```

```
# After the main loop completes, the script waits for any remaining  
background jobs to finish using wait.
```

```
echo "All jobs submitted"
```

MODULE 2: LECTURE 1

Using 'sed' (Stream Editor) for Text Manipulation

- command-line utility commonly found in Unix-like operating systems such as Linux and macOS.
- It is primarily used for performing text manipulation tasks on a stream of text input, such as files, pipelines, or standard input.
- 'sed' allows you to filter, transform, and edit text based on specified patterns and commands, making it an invaluable tool for various text processing scenarios.

Benefits of 'sed' for Batch Processing and Automated Text Manipulation:

- **Efficiency:** processes text line by line; you can apply the same operation to multiple lines.
- **Non-Interactive:** easy to integrated into scripts and automated workflows.
- **Speed:** Only command line, no GUI. Ideal for large datasets or for processing text in real-time.
- **Regular Expressions:** performs complex search and replace operations; advanced text transformations; high degree of flexibility.
- **Simultaneous Editing:** multiple editing commands in sequence to the same text stream; Perform a series of transformations on the text data without the need for multiple passes.
- **Batch Editing:** Directly change the original files without creating backups - useful to make systematic changes to a large number of files.

- **Combining with Other Tools:** 'sed' can be combined with other command-line utilities like 'grep', 'awk', and 'sort' to create powerful text processing pipelines, enabling you to achieve complex manipulations and analyses.

Basic syntax of invoking 'sed'

```
sed [options] 'pattern(s) command' input_file(s)
```

sed: invokes the 'sed' utility.

options: optional flags that modify the behavior of 'sed'. Get the options from '**man sed**'

-n; -e; -f; -i; -l , etc.

'pattern(s) command': Define your pattern(s) and commands (syntax: enclose it in single quotes)

input_file(s): file(s) you want to process; (unless from standard input or data piped from another command).

Most frequently used 'sed' commands:

s: **Substitute** a pattern with another string.

d: **Delete** lines that match the pattern.

p: **Print** lines that match the pattern.

a: **Append** text after the matched lines.

i: **Insert** text before the matched lines

Search and replace

```
sed 's/fruits/apples/' fruit.txt
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat fruit.txt
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
Fruits are delicious.
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed 's/fruits/apples/' fruit.txt
This is a sample file with some apples.
I like apples. Who does not like fruits?
Fruits are delicious.
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat fruit.txt
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
Fruits are delicious.
```

```
# Here, note that, it only changes the output. Does not change the content  
of the original file. Make sure you have the "fruit.txt" file in the same  
directory where you run the sed command, or provide the full path to the  
file if it's located in a different directory.
```

```
sed 's/fruits/apples/g' fruit.txt
```

s/fruits/apples/g': This is a **sed** substitution command with the g flag. It tells **sed** to find all occurrences of the text "fruits" in each line of the input file and replace them with "apples." The g flag stands for "global" and ensures that all occurrences on each line are replaced, not just the first one.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed 's/fruits/apples/g' fruit.txt  
This is a sample file with some apples.  
I like apples. Who does not like apples?  
Fruits are delicious.
```

```
sed 's/fruits/apples/gi' fruit.txt
```

The **sed** command you provided is performing a case-insensitive global substitution of "fruits" with "apples" in the "fruit.txt" file.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed 's/fruits/apples/gi' fruit.txt  
This is a sample file with some apples.  
I like apples. Who does not like apples?  
apples are delicious.
```

```
sed 's/fruits/oranges/gi' fruit.txt > orange.txt
```

redirects the output of the **sed** command to a new file named "orange.txt."

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed 's/fruits/oranges/gi' fruit.txt > orange.txt
sayak@Ubuntu:~/harry/files_module2_l1$ cat orange.txt
This is a sample file with some oranges.
I like oranges. Who does not like oranges?
oranges are delicious.
```

sed -i 's/fruits/oranges/g' fruit.txt

This command changes the original file.

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat fruit.txt
This is a sample file with some fruits.
I like fruits. Who does not like fruits?
Fruits are delicious.

sayak@Ubuntu:~/harry/files_module2_l1$ sed -i 's/fruits/oranges/g' fruit.txt
sayak@Ubuntu:~/harry/files_module2_l1$ cat fruit.txt
This is a sample file with some oranges.
I like oranges. Who does not like oranges?
oranges are delicious.
```

SA

sed -i 's/apples/oranges/g' fruit.txt &
sed 's/fruits/apples/gi' fruit.txt : Difference??

The two **sed** commands you provided perform different tasks on the same input file, "fruit.txt," but there is a crucial difference in how they operate:

1. **sed -i 's/apples/oranges/g' fruit.txt:** This command performs an in-place substitution using the **-i** flag. It replaces all occurrences of "apples" with "oranges" in the "fruit.txt" file directly, without creating a new file. This means that **the original "fruit.txt" file is modified**, and its content will now have "oranges" in place of "apples."

2. **sed 's/fruits/apples/gi' fruit.txt:** This command performs a case-insensitive global substitution of "fruits" with "apples" in the "fruit.txt" file. However, **it does not modify the**

original "fruit.txt" file. Instead, it outputs the modified content to the standard output (your terminal). If you want to save the changes, you would need to redirect the output to a new file or use the `-i` flag as in the first command.

In summary, the key difference is that the first command directly modifies the original file, while the second command only outputs the modified content to the terminal (or a new file if you redirect the output) but leaves the original file unchanged.

```
sed 's|fruits|apples|gi' fruit.txt
```

pipe symbol “|” Just used as an alternate delimiter in place of “/”. promotes case-insensitive substitution of “fruits” with “apples”.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed 's|fruits|apples|gi' fruit.txt
This is a sample file with some apples.
I like apples. Who does not like apples?
apples are delicious.
```

```
sed '1,2s/fruits/apples/gi' fruit.txt
```

1, 2: This range specifies that the substitution should only be applied to lines 1 and 2 of the input file.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '1,2s/fruits/apples/gi'
fruit.txt
This is a sample file with some apples.
I like apples. Who does not like apples?
Fruits are delicious.
```

```
sed "s/fruits/$(echo apples | tr a-z A-Z)/gi" fruit.tx
```

This command is not working??!!

-E for Extended regular expressions

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat more_fruits.txt
This is a sample file with some fruits, such as apples, bananas,
oranges, mangoes, etc.
I do not like apple pie.
I really like apple juice, orange juice, but not banana juice.
Yesterday I got 3 bananas and 4 oranges.
```

sed -E 's/(apple|banana)/fruit/g' file

It replaces multiple pattern (here, “apple” and “banana”) with a single pattern (“fruit” here).

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed -E 's/(apple|banana)/
fruit/g' more_fruits.txt
This is a sample file with some fruits, such as fruits, fruits,
oranges, mangoes, etc.
I do not like fruit pie.
I really like fruit juice, orange juice, but not fruit juice.
Yesterday I got 3 fruits and 4 oranges.
```

sed -E 's/(apple) pie/\1 strudel/g' file

It uses extended regular expressions (-E flag) to perform a substitution in the "file" file, replacing "apple pie" with "apple strudel".

\1: This is a backreference to the first capture group, which is "apple." It is used in the replacement text to preserve "apple" while replacing "apple pie" with "apple strudel."

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed -E 's/(apple) pie/\1
strudel/g' more_fruits.txt
This is a sample file with some fruits, such as apples, bananas,
oranges, mangoes, etc.
I do not like apple strudel.
I really like apple juice, orange juice, but not banana juice.
Yesterday I got 3 bananas and 4 oranges.
```

```
sed -E 's/(apple|orange) juice/\1 smoothie/g' file
```

It replaces “apple juice” and “orange juice” with “apple smoothie” and “orange smoothie” respectively.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed -E 's/(apple|orange) juice/\1 smoothie/g' more_fruits.txt
This is a sample file with some fruits, such as apples, bananas,
oranges, mangoes, etc.
I do not like apple pie.
I really like apple smoothie, orange smoothie, but not banana juice.
Yesterday I got 3 bananas and 4 oranges.
```

```
sed -E 's/[0-9]+/NUM/g' file
```

It replaces any single or multi-digit numbers with string “NUM”. If we execute **sed -E 's/[0-9]/NUM/g' file** then for each digit it will give 1 time ‘NUM’. See the difference below:

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat NUM.txt  
Yesterday I got 3 bananas and 14 oranges, 123 mangoes  
sayak@Ubuntu:~/harry/files_module2_l1$ sed -E 's/[0-9]+/NUM/g' NUM.txt  
Yesterday I got NUM bananas and NUM oranges, NUM mangoes  
sayak@Ubuntu:~/harry/files_module2_l1$ sed -E 's/[0-9]/NUM/g' NUM.txt  
Yesterday I got NUM bananas and NUMNUM oranges, NUMNUMNUM mangoes
```

Delete “d” with sed:

We will work with more_fruits.txt as shown below:

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat more_fruits.txt  
This is a sample file with some fruits, such as apples, bananas,  
oranges, mangoes, etc.  
I do not like apple pie.  
I really like apple juice, orange juice, but not banana juice.  
Yesterday I got 3 bananas and 4 oranges.
```

```
sed '/banana/d' file
```

delete lines containing the word "banana" from the "file" file.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '/banana/d' more_fruits.txt  
I do not like apple pie.
```

```
Sed '/^I/d' file
```

used to delete lines that start with "I" from the "file" file.

/^I/: This part specifies a pattern to search for lines that start with "I" (uppercase "I" at the beginning of the line).

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '/^I/d' more_fruits.txt
This is a sample file with some fruits, such as apples, bananas, o
ranges, mangoes, etc.
Yesterday I got 3 bananas and 4 oranges.
```

sed '/^\$/d' file

used to delete **empty lines** (lines that contain no characters) from the "file" file.

/^\$/: This part specifies a pattern to search for lines that consist of **nothing between the start and end of the line**, effectively matching empty lines.

```
sayak@Ubuntu:~/harry/files_module2_l1$ vim empty.txt
sayak@Ubuntu:~/harry/files_module2_l1$ cat empty.txt
I don't like bananas.

Banana is a fruit.

Oranges are delicious.

sayak@Ubuntu:~/harry/files_module2_l1$ sed '/^$/d' empty.txt
I don't like bananas.
Banana is a fruit.
Oranges are delicious.
```

sed '/[aeiou]/d' file

used to delete lines in the "file" file that contain any vowel (a, e, i, o, or u).

/[aeiou]/: This part specifies a pattern to search for lines that contain any of the characters 'a', 'e', 'i', 'o', or 'u'.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '/[aeiou]/d' more_fruit  
s.txt
```

```
sayak@Ubuntu:~/harry/files_module2_l1$
```

sed '/[0-9]/d' file

Deletes the lines containing digits.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '/[0-9]/d' more_fruits.  
txt
```

```
This is a sample file with some fruits, such as apples, bananas, o
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat more_fruits.txt  
apple This is a sample file with some fruits, such as apples, bananas, oranges,  
mangoes, etc.  
I do not like apple pie.  
I really like apple juice, orange juice, but not banana juice.  
Yesterday I got 3 bananas and 4 oranges.
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed -n '/banana/p' more_fruits.txt  
apple This is a sample file with some fruits, such as apples, bananas, oranges,  
mangoes, etc.  
I really like apple juice, orange juice, but not banana juice.  
Yesterday I got 3 bananas and 4 oranges.
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '/banana/p' more_fruits.txt  
apple This is a sample file with some fruits, such as apples, bananas, oranges,  
mangoes, etc.  
apple This is a sample file with some fruits, such as apples, bananas, oranges,  
mangoes, etc.  
I do not like apple pie.  
I really like apple juice, orange juice, but not banana juice.  
I really like apple juice, orange juice, but not banana juice.  
Yesterday I got 3 bananas and 4 oranges.  
Yesterday I got 3 bananas and 4 oranges.
```

sed '2,3d' file

used to delete lines 2 to 3 (inclusive) from the "file" file. 2 , 3: This range specifies that lines 2 to 3 (inclusive) should be deleted.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '2,3d' more_fruits.txt  
This is a sample file with some fruits, such as apples, bananas, o  
ranges, mangoes, etc.  
Yesterday I got 3 bananas and 4 oranges.
```

Print p

sed -n '/banana/p' file

The `sed` command you provided is used to search for lines containing the word "banana" in a file and then print those lines. Here's what each part of the command does:

- `sed` is a stream editor used for text manipulation.
- `-n` tells `sed` to suppress automatic printing of pattern space (lines).
- `'/banana/p'` is a pattern. It consists of two parts:
 - `/banana/` is a regular expression pattern that matches lines containing the word "banana."
 - `p` is a command that tells `sed` to print the matched lines.

What happens if I do not give -n ?

Without the `-n` option, `sed` will perform the default behavior, which is to automatically print every line of input after processing it. In the context of your command, this means that not only will `sed` print the lines that match the pattern `/banana/` but it will also print all other lines in the input file, whether they match the pattern or not.

```
sed -n '/banana/!p' file
```

#print line NOT containing banana

```
sed -n '/^apple/p' file
```

#print line starting with apple

```
sed -n '/^$/p' file
```

#print empty lines (non-empty?)

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '/banana/i This is a fruit:' more_fruits.txt
This is a fruit:
apple This is a sample file with some fruits, such as apples, bananas, oranges,
mangoes, etc.
I do not like apple pie.
This is a fruit:
I really like apple juice, orange juice, but not banana juice.
This is a fruit:
Yesterday I got 3 bananas and 4 oranges.
```

sed -n '2,4p' file

#print line 2 to 4.

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '/^apple/i This starts with "apple".'
some_more_fruits
This starts with "apple".
apple

banana
cherry
date
elderberry
fig
```

sed -n '/[aeiou]/p' file

#print line containing letters aeiou

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '/banana/a This is a fruit:' some_more_fruits
apple

banana
This is a fruit:

cherry
date
elderberry
fig
```

sed -n '/[0-9]/p' file

```
#print line containing numbers
```

sed -n 's/apple/date/gp' file

s/ indicates that you want to perform a substitution.

g stands for "global," meaning it will replace all occurrences of the pattern on each line. If you omit g, it will only replace the first occurrence on each line.

Append/ Insert:

sed '/banana/i This is a fruit:' file

```
sayak@Ubuntu:~/harry/files_module2_11$ sed '/banana/s/^The fruit is:/' some_more_fruits
apple

The fruit is:banana

cherry

date

elderberry

fig
```

This command insert a line "This is a fruit:" before each line that contains the word "banana" in the file. i is the insert command in sed.

sed '/^apple/i This starts with "apple".' file

The sed command you provided is used to insert the line "This starts with "apple"." before each line in the file that starts with the word "apple."

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed '/banana/s/$/ is a fruit/' some_more_fruits
apple

banana is a fruit

cherry

date

elderberry

fig
```

sed '/banana/a This is a fruit:' file

appends the line "This is a fruit:" immediately after each line in the file that contains the word "banana." a is the append command in sed.

sed '/^apple/a This starts with "apple".' file

```
sayak@Ubuntu:~/harry/files_module2_l1$ sed 's/\b./\U&/g' more_fruits.txt
Apple This Is A Sample File With Some Fruits, Such As Apples, Bananas, Oranges,
Mangoes, Etc.
I Do Not Like Apple Pie.
I Really Like Apple Juice, Orange Juice, But Not Banana Juice.
Yesterday I Got 3 Bananas And 4 Oranges.
```

appends the line "This starts with "apple"." immediately after each line in the file that starts with the word "apple."

sed '/banana/s/^/The fruit is:/' file

The sed command you provided, sed '/banana/s/^/The fruit is:/' file, will indeed add "The fruit is:" to the beginning of lines in the file that contain the word "banana."

'/banana/ ' is a regular expression pattern that matches lines containing the word "banana."

`s/^The fruit is:/` is the substitution command that performs the modification. It replaces the start of each matching line (^ represents the start of a line) with "The fruit is:".

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat 20.txt
I like apples.
I have a banana.
Cashews are delicious.
Almonds are great too.
sayak@Ubuntu:~/harry/files_module2_l1$ sed -E '/(apple|banana)/s/$/ is a fruit/';
 /(cashew)/s/$/ is a nut/' 20.txt
I like apples. is a fruit
I have a banana. is a fruit
Cashews are delicious.
Almonds are great too.
```

ed '/banana/s/\$/ is a fruit/' file

This will add " is a fruit" to the end of lines in the file that contain the word "banana."

- `'/banana/'` is a regular expression pattern that matches lines containing the word "banana."
- `s/$/ is a fruit/` is the substitution command that performs the modification. It replaces the end of each matching line (\$) represents the end of a line) with " is a fruit."

Multiple Sed

```
sed 's/\b./\U&/g' file | sed 's/$/- Fruit/' | sed '/banana/d'
```

Capitalize the first letter of each word. It replaces each word boundary (\b) followed by any character (.) with the uppercase version of that character (\U&).

Here's how the command works:

- sed is a stream editor used for text manipulation.
- s/\b ./\U&/g is the substitution command:
 - \b represents a word boundary.
 - . represents any character.
 - \U& transforms the matched character into uppercase.
 - g at the end means to perform this substitution globally on each line.

Append " - Fruit" to the end of each line.

Delete lines containing the word "banana".

```
sed -E '/(apple|banana)/s/$/- fruit/; /cashew/s/$/- nut/;
```

The sed command you provided uses extended regular expressions (-E option) to perform two different substitutions on each line in the input file. Append 'is a fruit' to apple or banana; 'is a nut' to cashew..

AWK:

Developed in the 1970s, AWK's name is derived from its creators' initials: Alfred Aho, Peter Weinberger, and Brian Kernighan. It is a text-processing tool used in Unix-like environments to perform pattern scanning and text/data manipulation tasks. Key Features and Use Cases:

Pattern Matching:

Data Extraction: (extracts specific columns or fields from structured data)

Data Transformation: Transforms data using mathematical operations, string concatenation, and conditionals.

Conditional Processing: You can selectively apply actions to lines that meet specific criteria.

Text Formatting and Reporting: Enables you to create custom reports and summaries from data. By performing calculations and combining text, you can generate formatted reports for analysis and presentation.

Automating Tasks: Can be used in shell scripts or one-liners to automate text processing.

Basic AWK Syntax

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

Note the single quotes and { }

- pattern is a condition that, if true, triggers the execution of the associated action.
- action is the code block executed when the pattern is satisfied.
- input_file is the text file that AWK processes.

Simple AWK usage

```
awk '{ print $1 }' names.txt
```

#no pattern, only action. Prints the first column of a file.

```
sayak@Ubuntu:~/harry/files_module2_l1$ vim biodata.csv
sayak@Ubuntu:~/harry/files_module2_l1$ cat biodata.csv
Name,Age,Email,Phone
John Doe,30,johndoe@example.com,555-123-4567
Jane Smith,25,janesmith@example.com,555-987-6543
Bob Johnson,40,bobjohnson@example.com,555-555-5555
Alice Brown,35,alicebrown@example.com,555-111-2222
sayak@Ubuntu:~/harry/files_module2_l1$ awk '{ print $1}' biodata.csv
Name,Age,Email,Phone
John
Jane
Bob
Alice
```

default field separator is space. Thats why the result comes like that.

```
awk -F',' '{ print $2 }' contacts.csv
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ awk -F ',' '{ print $1 }'
biodata.csv
Name
John Doe
Jane Smith
Bob Johnson
Alice Brown
```

Here, we specified the field separator as ','. Note the difference between this and the previous one where the field separator was space.

```
awk -F',' '{ print $1 , $4 }' numbers.csv
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ vim numbers.csv
sayak@Ubuntu:~/harry/files_module2_l1$ cat numbers.csv
10,20,30,40,50
15,25,35,45,55
5,15,25,35,45
8,18,28,38,48
22,33,44,55,66
sayak@Ubuntu:~/harry/files_module2_l1$ awk -F',' '{ print $1 ,
$4 }' numbers.csv
10 40
15 45
5 35
8 38
22 55
```

```
# displays multiple columns with field separator as ','.
```

```
awk -F',' '{ print $1 + $4 }' numbers.csv
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ awk -F',' '{ print $1 + $4 }' numbers.csv
50
60
40
46
77
```

```
# Do some mathematical operations.
```

Simple AWK usage: (with pattern and action)

```
awk '$2 > 80 { print }' age.txt
```

command you provided will print all lines in the "scores.txt" file where the second field (column) is greater than 80. Default field separator is space here.

```
sayak@Ubuntu:~/harry/files_module2_l1$ vim age.txt
sayak@Ubuntu:~/harry/files_module2_l1$ cat age.txt
John 90
Alice 88
Bob 77
Eve 89
sayak@Ubuntu:~/harry/files_module2_l1$ awk '$2 > 80 { print }' age.txt
John 90
Alice 88
Eve 89
sayak@Ubuntu:~/harry/files_module2_l1$ awk '/error/ { print }' log.txt
```

The `awk` command you provided is used to print all lines in the "log.txt" file that contain the word "error." Here's what this command does:

- awk: This is the AWK text processing utility.
- '/error/ { print }': This is an AWK program.
 - /error/: This part specifies a pattern to search for lines that contain the word "error."
 - { print }: This action tells AWK to print the entire line for lines that match the specified pattern (contain the word "error").
- log.txt: This is the input file from which lines containing the word "error" will be printed.

```

sayak@Ubuntu:~/harry/files_module2_l1$ cat log.txt
2023-09-01 10:15: Error: Connection timeout
2023-09-01 10:30: Warning: Low disk space
2023-09-01 10:45: Error: Database query failed
2023-09-01 11:00: Information: Task completed successfully

sayak@Ubuntu:~/harry/files_module2_l1$ awk '/Error/ {print}' log.txt
2023-09-01 10:15: Error: Connection timeout
2023-09-01 10:45: Error: Database query failed

awk 'NR >= 10 { print }' book.txt
awk 'NR <= 10 { print }' book.txt

awk 'NR >= 10 && NR <= 20 { print }' book.txt

```

NR >= 10 && NR <= 20: This condition checks if the current line number (NR) is greater than or equal to 10 and less than or equal to 20. In other words, it selects lines 10 to 20.

- { print }: This action tells AWK to print the entire line for lines that meet the condition.

- book.txt: This is the input file from which lines 10 to 20 will be printed.

```
sayak@Ubuntu:~/harry/files_module2_l1$ awk 'NR >= 2 && NR <= 3
{print}' log.txt
2023-09-01 10:30: Warning: Low disk space
2023-09-01 10:45: Error: Database query failed
sayak@Ubuntu:~/harry/files_module2_l1$ awk 'NF > 3 { print }' scores.txt
```

awk: This is the AWK text processing utility.

- 'NF > 3 { print }': This is an AWK program.
- NF > 3: This condition checks if the number of fields (NF) in the current line is greater than 3.
- { print }: This action tells AWK to print the entire line for lines that meet the condition (have more than 3 fields).

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat field.txt
John 90 85 78
Alice 88 92 76 95
Bob 77
sayak@Ubuntu:~/harry/files_module2_l1$ awk 'NF > 3 { print }'
field.txt
John 90 85 78
Alice 88 92 76 95
sayak@Ubuntu:~/harry/files_module2_l1$
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ awk -f avg.awk numbers.txt
475.04
sayak@Ubuntu:~/harry/files_module2_l1$ awk -f avg.awk numbers.sh
61.6923
```

Simple AWK usage: (with mathematical operation)

Calculating the Sum of Numbers

```
awk '{ sum += $1 } END { print "Sum:", sum }' numbers.txt
```

```
awk '{ sum += $1 } END { average = sum / NR; print "Average:", average }' grades.txt
```

Calculate the average of Numbers

Perform Arithmetic Operations

```
awk '{ result = $1 + $2 + $3; print $1, $2, $3, "=", result }' operations.txt
```

AWK SCRIPTS: Examples

- Given a file named numbers.txt containing a list of numbers, one number per line, write an AWK script to calculate and print the average of these numbers.

```
BEGIN {
    # Initialize the sum and count variables.
    sum = 0
    count = 0
}
{
    # Check if the line is empty.
    if ($0) {
        # Add the number on the line to the sum.
        sum += $1
        # Increment the count.
        count++
    }
}
END {
    # Print the average.
    print sum / count
}
```

To use the script, save it as a file with the .awk extension, for example average.awk. Then, run the following command:

```
awk -f average.awk numbers.txt
```

`awk` is a powerful text processing tool commonly used for data manipulation.

- The `-f` option in the `awk` command is used to specify a file that contains an AWK script. When you use `-f` followed by a filename, AWK reads the script from that file and executes it.
- `numbers.txt` is the input file that the AWK script will process.

What is the difference between scripts with extension `.awk` and with `.sh`

The difference between scripts with the file extensions `.awk` and `.sh` lies in their intended use, scripting languages, and functionality:

1. `.awk` Scripts:

- **AWK** is a text processing and data manipulation language that excels at working with structured data, such as columns and fields.
- AWK scripts are primarily used for **text processing tasks** like extracting specific data from files, reformatting text, and performing calculations on structured data.
- AWK scripts are especially useful for tasks involving tabular data, logs, and reports.
- AWK scripts are typically more concise and tailored for text manipulation tasks.
- Example `.awk` script:
`awk`

- # This is a simple AWK script to print the second column of a CSV file

```
BEGIN { FS = "," }  
{ print $2 }
```

- **.sh Scripts (Bash Scripts):**

- **.sh scripts** are often written in the **Bash** scripting language, which is a command-line shell for Unix-like operating systems.
- Bash scripts are versatile and can handle a wide range of tasks, including system administration, process automation, file manipulation, and more.
- Bash scripts can execute system commands, control the flow of programs, and handle user input.
- Bash scripts are not specifically designed for text processing but can perform such tasks when needed.
- Example .sh script:

bash

```

• # This is a simple Bash script to count the
  number of files in a directory

  count=$(ls | wc -l)

  echo "There are $count files in the directory."

```

In summary, .awk scripts are specialized for text processing and data

```

sayak@Ubuntu:~/harry/files_module2_l1$ vi numbers.txt
sayak@Ubuntu:~/harry/files_module2_l1$ awk -f maxmin.awk numbers.txt
Max: 3634
Min: -90

```

manipulation, particularly with structured data, while .sh (Bash) scripts are more general-purpose and versatile, often used for system administration and automation tasks. The choice between .awk and .sh scripts depends on the specific requirements of the task at hand.

2. Given a file named values.txt with a list of numbers, one number per line, write an AWK script to find and print the maximum and minimum values from the file.

```

BEGIN {
    # Initialize the maximum and minimum values with very low and high values
    # respectively.
    max = -99999999
    min = 99999999
}

{
    # Check if the line is empty.
    if ($0) {
        # Check if the number on the line is greater than the maximum value.
        if ($1 > max) {
            max = $1
        }

        # Check if the number on the line is less than the minimum value.
        if ($1 < min) {
            min = $1
        }
    }
}

END {
    # Print the maximum and minimum values.
    print "Max:", max
    print "Min:", min
}

```

```

sayak@Ubuntu:~/harry/files_module2_li$ cat sales.csv
Product Name,Quantity Sold,Price Per Unit
Product A,10,15.00
Product B,5,25.50
Product C,8,12.75
Product D,15,10.00

sayak@Ubuntu:~/harry/files_module2_li$ awk -f calculate_sales.awk sales.csv
Product Name,0
Product A,150
Product B,127.5
Product C,102
Product D,150
,0
Overall Total Sales Revenue:,529.5

```

Checking if the line is not empty (`if ($0)`): This part of the script ensures that empty lines in the input file are not considered when determining the maximum and minimum values. The condition `if ($0)` checks if the entire line (`$0`) contains any characters. If it's empty,

meaning there are no characters on that line, the line is skipped. This is important to avoid incorrect calculations if the file contains empty lines.

Comparing the number on the line (\$1) with max and min: The script reads each line of the input file and extracts the number from the line using \$1. It then compares this number to the current maximum and minimum values (max and min). If the number is greater than the current maximum value ($\$1 > \text{max}$), it updates max. Similarly, if the number is less than the current minimum value ($\$1 < \text{min}$), it updates min. This way, the script keeps track of the maximum and minimum values encountered while processing the file.

3. Given a file named grades.csv with student names and their respective scores, write an AWK script to calculate and print the letter grades for each student based on the following scale:

EX: 90-100; A: 80-89 ...

```
#!/usr/bin/awk -f
# Define the score-to-grade mappings
BEGIN {
    # Define the score ranges and corresponding letter
grades
    grade_scale[90] = "A"
    grade_scale[80] = "B"
    grade_scale[70] = "C"
    grade_scale[60] = "D"
    grade_scale[0] = "F"
}

# Process each line in the CSV file
{
    # Extract the student name and score from the CSV
line
    student_name = $1
    score = $2

    # Initialize the letter grade to be empty
    letter_grade = ""

    # Determine the letter grade based on the score
    for (score_limit in grade_scale) {
```

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat sales_data.csv
Month,Sales
January,1000
February,1250
March,1350
April,1100
May,1400
sayak@Ubuntu:~/harry/files_module2_l1$ awk -f increase.awk sales_data.csv
Month: ,January
Percentage Increase: ,inf%

Month: ,February
Percentage Increase: ,25%

Month: ,March
Percentage Increase: ,8%

Month: ,April
Percentage Increase: ,-18.5185%

Month: ,May
Percentage Increase: ,27.2727%
```

```
if (score >= score_limit) {
    letter_grade = grade_scale[score_limit]
}
}

# Print the student's name and letter grade
print student_name ":" letter_grade
}
```

4. Given a file named sales.csv with product names and their quantities sold and prices per unit, write an AWK script to calculate and print the total sales revenue for each product (quantity * price) and the overall total sales revenue.

```
#!/usr/bin/awk -f

# Initialize variables
BEGIN {
    FS = "," # Set the field separator to comma
    OFS = "," # Set the output field separator to comma
    total_sales = 0
}

# Process each line in the CSV file
{
    # Extract fields from the CSV line
    product_name = $1
    quantity_sold = $2
    price_per_unit = $3

    # Calculate the total sales revenue for the product (quantity * price)
    product_sales = quantity_sold * price_per_unit

    # Add the product sales to the overall total sales
    total_sales += product_sales

    # Print the product name and its sales revenue
    print product_name, product_sales
}

# Print the overall total sales revenue
END {
    print "Overall Total Sales Revenue:", total_sales
}
```

5. Given a file named sales_data.csv containing monthly sales data for a product, write an AWK script to compute the percentage increase in sales from one month to the next and print the results.

```
#!/usr/bin/awk -f

# Initialize variables
BEGIN {
    FS = "," # Set the field separator to comma
    OFS = "," # Set the output field separator to comma
    prev_sales = 0
}

# Process each line in the CSV file
{
    # Extract fields from the CSV line
    month = $1
    sales = $2

    # Calculate the percentage increase from the previous month
    if (NR > 1) { # Skip the header line
        percentage_increase = ((sales - prev_sales) / prev_sales) * 100

        # Print the result
        print "Month:", month
        print "Percentage Increase:", percentage_increase "%"
        print "" # Print an empty line for readability
    }

    # Store the current month's sales as the previous month's sales for the next iteration
    prev_sales = sales
}
```

In the **BEGIN** block:It sets the output field separator (OFS) to a comma for consistent CSV output formatting.Initializes the **prev_sales** variable to keep track of the previous month's sales.The script skips the header line (assumed to be the first line in the CSV file) by checking **NR > 1** (record number greater than 1).

ETSC CLASS ASSIGNMENT

Essential Tools for Scientific Computing Module – II Class Project

Team Members:

1. Tarif Hasan 22CE92R06
2. Abhinav Raj 23CE91R02
3. Jhantu Pal 22MA92F05
4. Sayak Karmakar 22CE92R05

Title: “**Understanding the global population trends**”

Summary:

When working with population data, we can create various types of plots to visualize different aspects of population trends.

1. Population Over Time - Line Plot: Plot a line chart to show how the population has changed over time. This is useful for tracking population growth or decline.
2. Population Distribution - Histogram: Plot a histogram to visualize how the population is distributed across different population ranges. This can reveal population concentration patterns.
3. Population by Region - Bar or Stacked Bar Plot: Create a bar chart to compare the populations of different regions or countries.
4. Population Growth Rate - Bar Plot or Line Plot: Plot a bar chart or line chart to visualize the annual or periodic population growth rate. This can help identify periods of rapid or slow population growth.
5. Urban vs. Rural Population - Grouped Bar Plot: Compare the population counts between urban and rural areas using a grouped bar plot.

1. Population Over Time -

We displayed the world population changed over the period 2018-2021 in a line plot of xmgrace. Firstly, we downloaded from kaggle.com. There was a lot of data of year-wise population for all the countries within this time period. We sorted these data year-wise using the following command:

```
awk '/year/ { print }' file_name > file_2
```

Therefore we got 5 different files. Then from those files we extracted the countrywise population data by using the following command as it was in column 3:

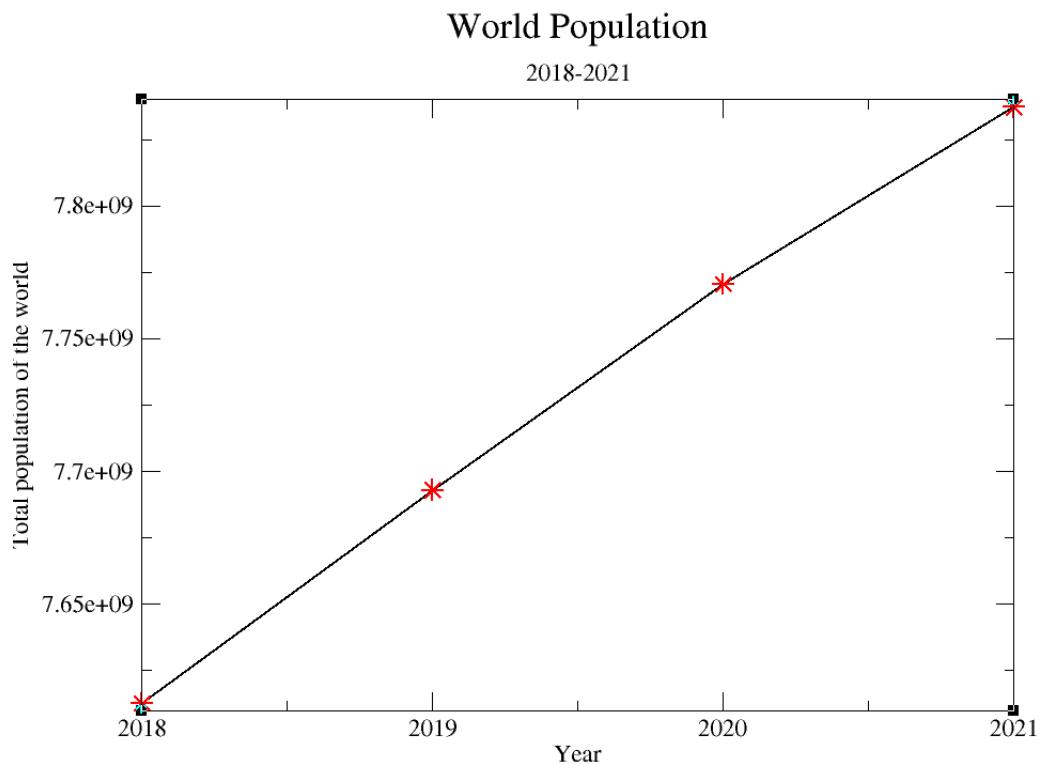
```
awk -F',' '{ print $3 }' file_2 > file_3
```

Then we summed up the population of all the countries for a particular year using:

```
awk { sum += $column_number; print sum} file_3 > file_4
```

Then using “xmgrace file_4.txt” command , we plotted that graph in xmgrace.

2.



Population Distribution:

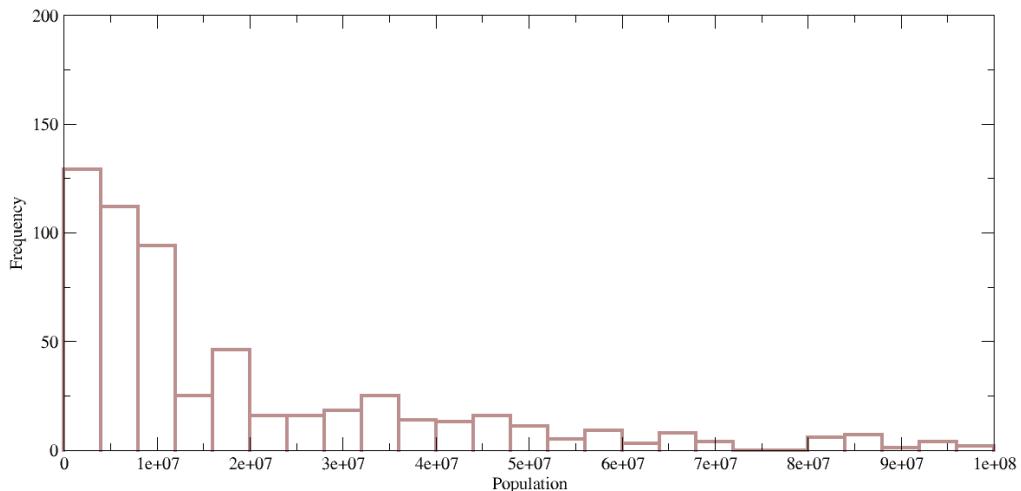
We plotted a histogram with population bins of countries for the year 2021. The population of most of the countries was less than 1e7 but few countries have huge population even grater than up to a range of 2e8. The following commands were used:

The plot is shown as follows.

```
awk '/2021/ { print }' GP.csv > gp.txt
```

```
awk -F"" '{ print $2 }' gp.txt > gp1.txt
```

```
cat gp1.txt|tr -d ','>gp2.txt
```



3. Population by Region -

We selected first 10 countries and extracted the populations of them by the following commands:

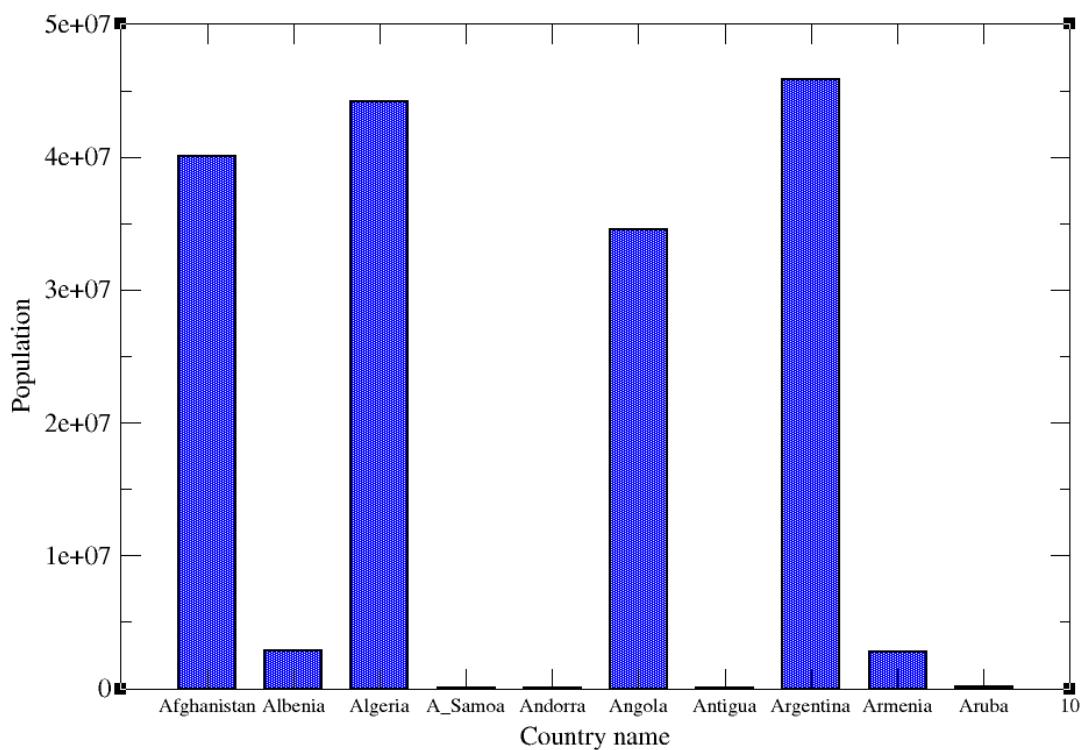
```
awk '/2021/ { print }' file_name > file_2 [ to get 2021 dataset]
```

```
awk 'NR <= 10 { print $3 }' file_2 > file_3 [ Because, column 3 contains total populations]
```

```
awk 'NR <= 10 { print $1 }' file_2 > file_4 [coulumn 1 contains the names of the countries]
```

paste -d' ' file_3 file_4 [Now we got file which contains country name and total population only]. Then we plotted graph in xmgrace and changed it to a bar graph.

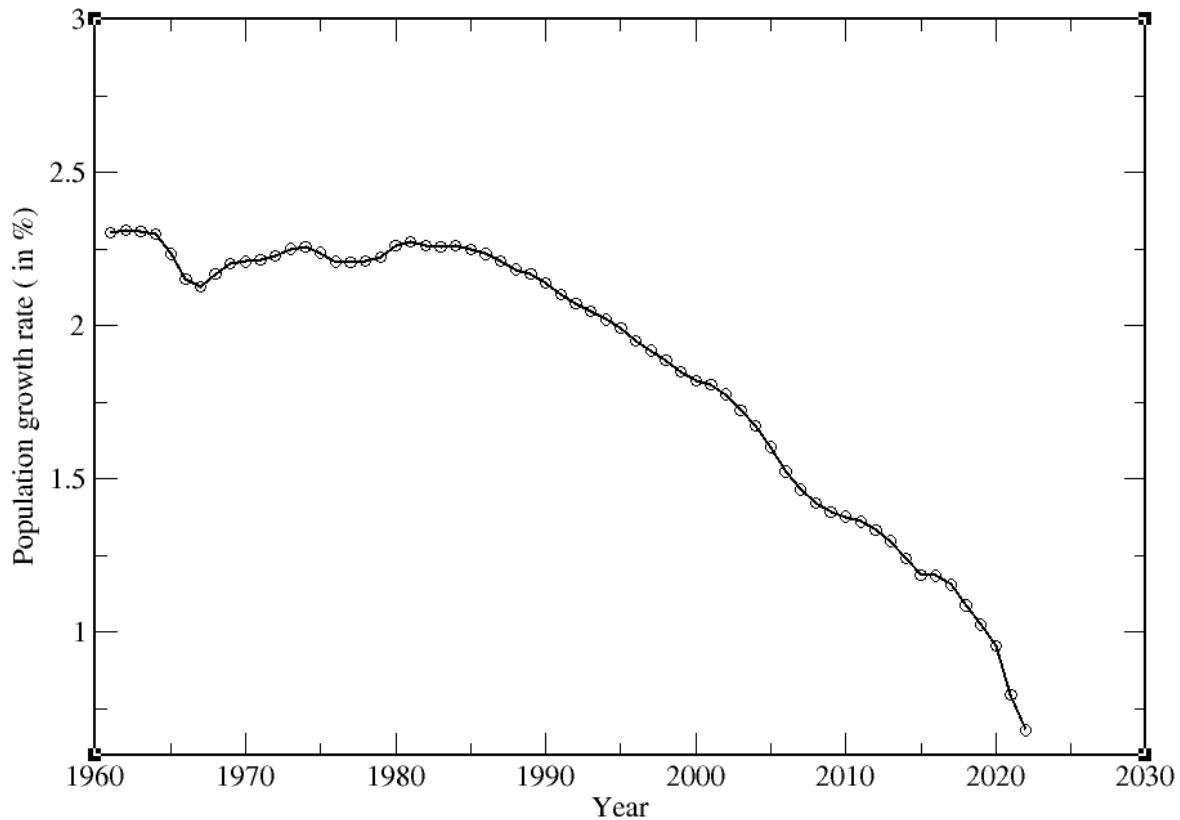
4.



Population Growth Rate

We have downloaded data for population growth rate of India for the timespan of 1961-2022. We sorted the data in the way similar to problem 1. After that, we plotted the data in the xmGrace environment.

Population Growth Rate of India (in %)



5.1 Urban vs. Rural Population

We first stored country name, total population, rural population and urban populations of first 10 countries in the year 2021 in three different files (let their names are file_1, file_2, file_3 and file_4). Then we pasted those files to get a single file named as file_5. So now we have four different columns in that file. Then we calculated percentage rural and urban populations using

```
awk -F',' '{ print $3 / $2 * 100 }' file_5 > rural.txt [It contains rural poputations of those countries]
```

```
awk -F',' '{ print $4 / $2 * 100 }' file_5 > urban.txt [It contains urban poputations of those countries]
```

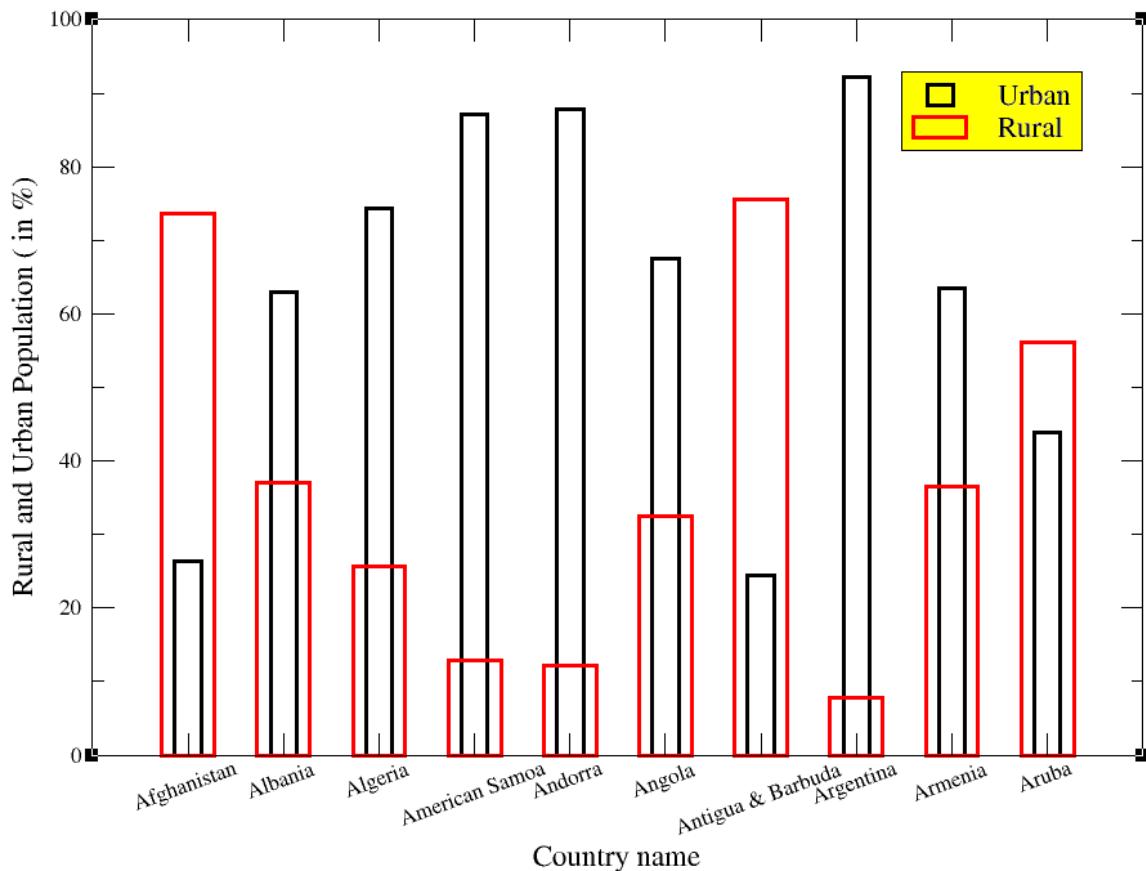
Then we use paste file_1 urban.txt rural.txt > final.txt

So, we plotted a multi-graph using the following command,

```
xmgrace -nxy final.txt
```

Rural and Urban Population of Different Countries

(in percentage)



5.2 Urban vs Rural population:

The year, total population, urban population and rural population is extracted from the “Global Population Trends (2016-2022)” in a separate new2.txt file. All the commands are written in a bash script file q6.sh. For each year that is from 2018 to 2021, percentage of rural and urban population is calculated and plotted as bar plots. The commands in the bash file is given below.

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

```
awk -F"" '{print $1,$2,$4,$6}' GP.csv|cut -d ',' -f 2-30|awk -F',' '{if ($1==2018 || $1==2019 || $1==2020 || $1==2021) {print} }'|awk '{print $2,$3,$4}'|tr -d ',' > new1.txt
```

```
awk -F',' '{ if ($2!=2017) {print $2} }' GP.csv|sed '1d' > new.txt
```

```
paste -d '' new.txt new1.txt|tr ' ' ' > new2.txt
```

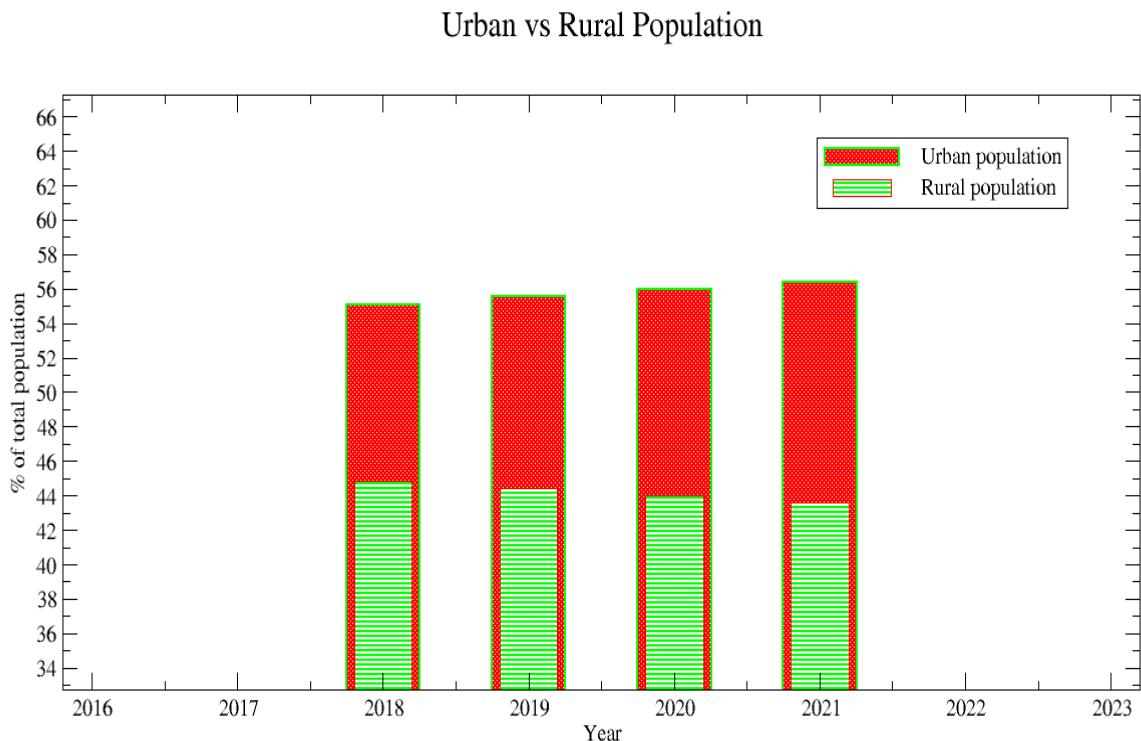
```
awk -F',' 'BEGIN {sum1=""}; sum2=""'; sum3=""'; percentage=""' {if ($1==2018) {sum1+=$2; sum2+=$3; sum3+=$4} } END {print "2018 "sum2*100/sum1, sum3*100/sum1}' new2.txt> grph.txt
```

```
awk -F',' 'BEGIN {sum1=""; sum2=""; sum3=""; percentage=""} {if ($1==2019) {sum1+=$2; sum2+=$3; sum3+=$4} } END {print "2019 "sum2*100/sum1, sum3*100/sum1}' new2.txt>> grph.txt
```

```
awk -F',' 'BEGIN {sum1=""; sum2=""; sum3=""; percentage=""} {if ($1==2020) {sum1+=$2; sum2+=$3; sum3+=$4} } END {print "2020 "sum2*100/sum1, sum3*100/sum1}' new2.txt>> grph.txt
```

```
awk -F',' 'BEGIN {sum1=""; sum2=""; sum3=""; percentage=""} {if ($1==2021) {sum1+=$2; sum2+=$3; sum3+=$4} } END {print "2021 "sum2*100/sum1, sum3*100/sum1}' new2.txt>> grph.txt
```

The plot is plotted by the commands xmgrace -free -nxy grph.txt



Xm Grace Assignment

Hoek-Brown strength criterion

The Hoek-Brown strength criterion is an empirical stress-strain relationship that is used to predict the failure of rock masses. It is a nonlinear criterion that takes into account the effects of the intact rock strength, the stress state, and the degree of fracturing of the rock mass.

The Hoek-Brown strength criterion is defined by the following equation:

$$\sigma_1 = \sigma_3 + \sigma_c \cdot m \cdot (\sigma_3/\sigma_c + 1/m)^n$$

where:

- σ_1 is the major principal stress at failure
- σ_3 is the minor principal stress at failure
- σ_c is the uniaxial compressive strength of the intact rock
- m is the Hoek-Brown constant
- n is the Hoek-Brown exponent

The Hoek-Brown constant, m , is a measure of the degree of fracturing of the rock mass. The higher the value of m , the more fractured the rock mass is. The Hoek-Brown exponent, n , is a measure of the nonlinearity of the failure criterion. The higher the value of n , the more nonlinear the failure criterion is.

The Hoek-Brown strength criterion is widely used in geotechnical engineering for the design of tunnels, slopes, and foundations. It is also used to assess the stability of rock masses.

Here are some of the advantages of using the Hoek-Brown strength criterion:

- It is simple to use and understand.
- It can be used to predict the failure of rock masses under a wide range of stress conditions.
- It has been validated by experimental data.

Here are some of the disadvantages of using the Hoek-Brown strength criterion:

- It is an empirical criterion, so it is not always accurate.
- It requires the knowledge of the uniaxial compressive strength of the intact rock, which may not always be available.
- It is not always applicable to all types of rock masses.

Overall, the Hoek-Brown strength criterion is a useful tool for the design and analysis of rock masses. It is important to be aware of its limitations, but it can be a valuable asset for geotechnical engineers.

Parameters for assignments

UCS(σ_c)= Unified Compressive strength.

GSI = Geological strength Index.

D= Rock Damage factor

m_i = Hoek Brown Constant.

σ_s =Support pressure.

Plot σ_c vs σ_s curve for different m_i values.

some commonly used symbols in xmgrace:

Total list

\a: Greek Alpha (α)

\b: Greek Beta (β)

\c: Circle (\bullet)

\d: Delta (Δ)

\e: Epsilon (ε)

\f{}: Text Formatting

\g: Gamma (γ)

\h: Theta (θ)

\i: Infinity (∞)

\j: Integral (\int)

\l: Lambda (λ)

\m: Mu (μ)

\n: Nu (ν)

\o: Omega (Ω)

\p: Pi (π)

\r: Rho (ρ)

\s: **Superscript.** "x\s{2}" can be used to display "x²" with the "2" as a superscript.

\S: **Subscript.** "H\S{2}O" can be used to display "H₂O" with the "2" as a subscript.

\t: Tau (τ)

\u: Upsilon (υ)

\v: Phi (ϕ)

\x: Special Characters

\y: Psi (ψ)

\z: Zeta (ζ)

\A: Degree ($^\circ$)

\B: Bullet (\bullet)

\C: Copyright (\circledcirc)

\D: Dagger (\dagger)

\E: Euro (\euro)

\F: Fraction ($\frac{1}{4}, \frac{1}{2}, \frac{3}{4}$)

\G: Greater Than (>)

\H: Less Than (<)

\I: Intersection (\cap)

\J: Union (\cup)

\K: Subset (\subseteq)

\L: Superset (\supseteq)

\M: Micro (μ)

\N: **Line Break or Newline.** "Line 1\nLine 2" would display "Line 1" on one line and "Line 2" on the next line within the same label.

\O: Ohm (Ω)

\P: Paragraph (\P)

\Q: Square Root ($\sqrt{}$)

\R: Registered (\circledR)

\T: Trademark ($^{\text{TM}}$)

\U: Integral with Limits (\oint)

\V: Integral with Circled Plus ($\oint\!\!\!\circ$)

\W: Integral with Circled Minus ($\oint\!\!\!\circ\!\!$)

\X: Checkmark (\checkmark)

\Y: Ballot X (\times)

\Z: Circled X ($\circ\!\!\!x$)

\leq : Less Than or Equal To (\leq)

\geq : Greater Than or Equal To (\geq)

\f{}: Text Formatting

In xmgrace, " $\f{}$ " is a formatting code used within text labels and annotations to apply different font styles and formatting options to the enclosed text. Here's how it works:

- **\f{Formatting Code}**: This format code is enclosed in curly braces, and you place the text you want to format within these braces. The "Formatting Code" specifies the formatting you want to apply to the enclosed text.

Here are some examples of how to use "\f{}" for text formatting:

1. **Bold Text**: To make text bold, you can use the "\f{B}" formatting code. For example, "\f{B}This is bold text" will display "This is bold text" in a bold font.
2. **Italic Text**: To make text italicized, you can use the "\f{I}" formatting code. For example, "\f{I}This is italic text" will display "This is italic text" in an italic font.
3. **Underlined Text**: To underline text, you can use the "\f{U}" formatting code. For example, "\f{U}This is underlined text" will display "This is underlined text" with an underline.
4. **Superscript**: To make text appear as a superscript (raised above the baseline), you can use the "\f{+}" formatting code. For example, "x\f{+2}" will display "x²" with the "2" as a superscript.
5. **Subscript**: To make text appear as a subscript (lowered below the baseline), you can use the "\f{-}" formatting code. For example, "H\f{-2}O" will display "H₂O" with the "2" as a subscript.
6. **Font Color**: You can also specify font color using "\f{color=ColorName}" or "\f{color=#RRGGBB}" where "ColorName" is the name of a color (e.g., "red," "blue") or a hexadecimal color code (e.g., "#FF0000" for red).
 - Example: "\f{color=red}Red Text" or "\f{color=#0000FF}Blue Text" will change the font color accordingly.
7. **Font Size**: You can adjust the font size using "\f{size=SizeValue}" where "SizeValue" is the desired font size in points (e.g., "\f{size=12}" for a font size of 12 points).
 - Example: "\f{size=14}Larger Text" will increase the font size.

\x: Special Characters

The `\x` escape sequence in xmgrace is used to enter the symbol font. This is a special font that contains a variety of mathematical and scientific symbols, such as Greek letters, mathematical operators, and special characters.

To use the `\x` escape sequence, you need to type the `\x` character, followed by the name of the symbol you want to use. For example, to enter the Greek letter alpha, you would type `\xalpha`.

Here is a table of some of the symbols that can be entered using the `\x` escape sequence:

Symbol	Escape sequence
Alpha	<code>\xalpha</code>
Beta	<code>\xbeta</code>
Gamma	<code>\xgamma</code>
Delta	<code>\xdelta</code>
Epsilon	<code>\xepsilon</code>
Zeta	<code>\xzeta</code>
Eta	<code>\xeta</code>
Theta	<code>\xtheta</code>
Iota	<code>\xiota</code>
Kappa	<code>\xkappa</code>
Lambda	<code>\xlambd</code>

Assignment Procedure

```
sayak@ubuntu:~/Desktop$ awk -F',' '{print $1 "\t" $4 "\t" $5}'  
Tdata.csv > output.dat
```

```
sayak@ubuntu:~/Desktop$ vim data1.dat
```

```
sayak@ubuntu:~/Desktop$ vim data2.dat
```

```
sayak@ubuntu:~/Desktop$ vim data3.dat
```

```
sayak@ubuntu:~/Desktop$ vim data4.dat
sayak@ubuntu:~/Desktop$ paste data1.dat data2.dat data3.dat
data4.dat > data_combined.dat
sayak@ubuntu:~/Desktop$ awk -F'\t|\s' '{print $1 "\t" $3 "\t" $6 "\t"
$9 "\t" $12}' data_combined.dat > final.dat
sayak@ubuntu:~/Desktop$ xmgrace -nxy final.dat
**From plot--> graph appearance, you can chnge the size and the display
of the legend box. For adding the names of the legend, go plot --> set
appearance.
** Files should be saved in .agr format. Go file-->Save as--> Selection-->
the path of the file (Eg. /home /sayak/ Desktop / ETSC.agr).
```

Sample Question paper

diff < cat 1.txt > cat 2.txt

The command you've provided uses the `diff` command to compare the contents of two files, `1.txt` and `2.txt`, and then it redirects the output of the `diff` command to `2.txt`. However, there is a small issue with your command. The `<` symbol is used for input redirection, not for specifying the files to be compared. To correctly compare the contents of `1.txt` and `2.txt` and store the differences in `2.txt`, you should use the `diff` command like this:

bash

```
diff 1.txt 2.txt > diff_output.txt
```

This command will compare `1.txt` and `2.txt` and save the differences in a file called `diff_output.txt`. If `diff_output.txt` already exists, it will be overwritten. If you want to append the differences to an existing file instead of overwriting it, you can use `>>` instead of `>`:

```
bash
```

```
diff 1.txt 2.txt >> diff_output.txt
```

This will append the differences to the end of the `diff_output.txt` file if it exists, or create it if it doesn't.



chown :hpc test.dat

The `chown` command in Unix-like operating systems is used to change the ownership of a file or directory. In the command you've provided:

```
bash
```

```
chown :hpc test.dat
```

You are attempting to change the group ownership of the file `test.dat` to the group `hpc`. However, you haven't specified the user ownership, which is typically represented before the colon (`:`).

If you want to change both the user and group ownership of `test.dat`, you should specify both the user and group like this:

```
bash
```

```
chown username:hpc test.dat
```

Replace `username` with the actual username you want to set as the owner. This command will change the user ownership to `username` and the group ownership to `hpc` for the `test.dat` file.

If you only want to change the group ownership and leave the user ownership unchanged, you can use the following command:

```
bash
```

```
chown :hpc test.dat
```

This will change only the group ownership of `test.dat` to `hpc`, leaving the user ownership unchanged.



sed '/start/,/end/ s/old/new/g' file.txt

The `sed` command you've provided is used to perform a search and replace operation on lines of a file located between lines containing "start" and "end." Here's a breakdown of the command:

- `/start/,/end/` specifies a range of lines starting with a line containing "start" and ending with a line containing "end."
- `s/old/new/g` is the substitution part of the `sed` command, which replaces all occurrences of "old" with "new" on the lines within the specified range.

So, if you have a file named `file.txt` that looks like this:

arduino

This is a test file.

start

Replace old with new in this line.

Replace old with new in this line too.

end

This line should not be changed.

Running the `sed` command you provided will result in:

arduino

This is a test file.

start

Replace new with new in this line.

Replace new with new in this line too.

end

This line should not be changed.

As you can see, the "old" has been replaced with "new" within the range of lines between "start" and "end."



Write the command to calculate the sum of 12, 18, and 22?

You can use the `expr` command in Unix-like operating systems to calculate the sum of numbers. Here's the command to calculate the sum of 12, 18, and 22:

bash

```
expr 12 + 18 + 22
```

When you run this command, it will output the result, which in this case is:
52

So, the sum of 12, 18, and 22 is 52.



Using awk, print lines from a file where the first column matches a string HPC.

You can use the `awk` command to print lines from a file where the first column (field) matches the string "HPC." Here's the command:

bash

```
awk '$1 == "HPC" {print}' your_file.txt
```

Replace `your_file.txt` with the name of your input file. This command will search for lines where the first field is equal to "HPC" and then print those lines.

For example, if your input file `your_file.txt` looks like this:

yaml

```
HPC 1234 Data1
```

```
ABC 5678 Data2
```

```
HPC 9876 Data3
```

```
XYZ 4321 Data4
```

Running the above `awk` command will output:

```
yaml
```

```
HPC 1234 Data1
```

```
HPC 9876 Data3
```

It prints the lines where the first column matches "HPC."

Default (GPT-3.5)

SA

```
sed -n '/banana/p' file
```

The `sed` command you provided is used to search for lines containing the word "banana" in a file and then print those lines. Here's what each part of the command does:

- `sed` is a stream editor used for text manipulation.
- `-n` tells `sed` to suppress automatic printing of pattern space (lines).
- '`/banana/p'` is a pattern. It consists of two parts:
 - `/banana/` is a regular expression pattern that matches lines containing the word "banana."
 - `p` is a command that tells `sed` to print the matched lines.

So, when you run the command `sed -n '/banana/p file`, it will search for lines containing "banana" in the file named "file" and print those lines to the standard output.

For example, if "file" contains the following lines:

```
css
```

```
I like bananas.
```

```
Banana is a delicious fruit.
```

```
Apples and oranges are also good.
```

Running the `sed` command you provided will result in the following output:

```
css
I like bananas.
Banana is a delicious fruit.
```

SA

What happens if I do not give `-n` ?

If you do not provide the `-n` option when using the `sed` command as follows:

```
bash
sed '/banana/p' file
```

Without the `-n` option, `sed` will perform the default behavior, which is to automatically print every line of input after processing it. In the context of your command, this means that not only will `sed` print the lines that match the pattern `/banana/` but it will also print all other lines in the input file, whether they match the pattern or not.

So, using the same example input file as before:

```
css
I like bananas.
Banana is a delicious fruit.
```

Apples and oranges are also good.

Running the command `sed '/banana/p' file` without the `-n` option will produce the following output:

```
css
I like bananas.
I like bananas.

Banana is a delicious fruit.

Banana is a delicious fruit.

Apples and oranges are also good.

Apples and oranges are also good.
```

As you can see, the lines containing "banana" are printed twice, and all other lines are also printed. This is because `sed` automatically prints every line after processing it when the `-n` option is not used.

SA

```
sed -n 's/apple/date/gp' file
```

The `sed` command you provided is used to search for the word "apple" in a file and replace it with the word "date," and then print the modified lines. Here's what each part of the command does:

- `sed` is a stream editor used for text manipulation.
- `-n` tells `sed` to suppress automatic printing of pattern space (lines).
- '`s/apple/date/g'` is a substitution command. It consists of the following parts:
 - `s/` indicates that you want to perform a substitution.
 - `apple` is the pattern you want to find and replace.
 - `date` is the replacement text.
 - `g` stands for "global," meaning it will replace all occurrences of the pattern on each line. If you omit `g`, it will only replace the first occurrence on each line.
- `p` is a command that tells `sed` to print the modified lines after performing the substitution.

So, when you run the command `sed -n 's/apple/date/gp' file`, it will search for the word "apple" in the file named "file," replace it with "date," and print the modified lines to the standard output.

For example, if "file" contains the following lines:

```
css
I like apples.
Apples are delicious.

I have an apple.
```

Running the `sed` command you provided will result in the following output:

```
css
```

```
I like dates.  
Dates are delicious.
```

```
I have an date.
```

As you can see, "apple" has been replaced with "date," and the modified lines are printed.

```
SA
```

```
sed '/banana/i This is a fruit:' file
```

The `sed` command you provided is used to insert a line "This is a fruit:" before each line that contains the word "banana" in the file. Here's what each part of the command does:

- `sed` is a stream editor used for text manipulation.
- `'/banana/'` is a regular expression pattern that matches lines containing the word "banana."
- `i` is the insert command in `sed`.
- `"This is a fruit:"` is the text you want to insert before each line that matches the pattern.
- `file` is the name of the file you want to perform this operation on.

So, when you run the command `sed '/banana/i This is a fruit:' file`, it will search for lines containing "banana" in the file named "file" and insert the line "This is a fruit:" before each of those lines.

For example, if "file" contains the following lines:

```
css  
I like apples.  
Banana is a delicious fruit.
```

```
I have an apple and a banana.
```

Running the `sed` command you provided will result in the following output:

```
less  
I like apples.  
This is a fruit:
```

Banana is a delicious fruit.

This is a fruit:

I have an apple and a banana.

As you can see, the line "This is a fruit:" has been inserted before each line containing "banana."

SA

`sed '/^apple/i This starts with "apple".' file`

The `sed` command you provided is used to insert the line "This starts with "apple"." before each line in the file that starts with the word "apple."

Here's what each part of the command does:

- `sed` is a stream editor used for text manipulation.
- `'/^apple/'` is a regular expression pattern that matches lines starting with the word "apple." The `^` symbol in the pattern denotes the start of a line.
- `i` is the insert command in `sed`.
- `"This starts with "apple"."` is the text you want to insert before each line that matches the pattern.
- `file` is the name of the file you want to perform this operation on.

So, when you run the command `sed '/^apple/i This starts with "apple".' file`, it will search for lines in the file named "file" that start with "apple" and insert the line "This starts with "apple"." before each of those lines.

For example, if "file" contains the following lines:

```
css
I like apples.
Apple is a fruit.
```

I have an apple and a banana.

Running the `sed` command you provided will result in the following output:

```
vbnet
I like apples.
This starts with "apple".
```

Apple is a fruit.

This starts with "apple".

I have an apple and a banana.

As you can see, the line "This starts with "apple"." has been inserted before each line starting with "apple."

SA

sed '/^apple/i This starts with "apple".' file sed '/banana/i This is a fruit:'
file Difference?

The two `sed` commands you provided have different purposes and effects:

1. `sed '/^apple/i This starts with "apple".' file`
inserts the line "This starts with "apple"." before each line in the file
that starts with the word "apple." This means it targets lines starting
with "apple" and inserts the specified text before those lines.

Example: If the "file" contains these lines:

```
css
I like apples.
Apple is a fruit.
```

I have an apple and a banana.

Running this `sed` command will result in:

```
vbnet
I like apples.
This starts with "apple".
```

Apple is a fruit.

This starts with "apple".

I have an apple and a banana.

2. `sed '/banana/i This is a fruit:'` file inserts the line "This is a fruit:" before each line in the file that contains the word "banana." This means it targets lines containing "banana" and inserts the specified text before those lines.

Example: If the "file" contains these lines:

```
css
I like bananas.
Banana is a delicious fruit.

I have an apple and a banana.
```

Running this `sed` command will result in:

```
less
I like bananas.
This is a fruit:

Banana is a delicious fruit.

This is a fruit:

I have an apple and a banana.
```

In summary, the first `sed` command targets lines starting with "apple" and inserts a specific line before them, while the second `sed` command targets lines containing "banana" and inserts a different specific line before them.

SA

`sed '/banana/a This is a fruit:'` file

The `sed` command you provided, `sed '/banana/a This is a fruit:'` file, appends the line "This is a fruit:" immediately after each line in the file that contains the word "banana." Here's how it works:

- `sed` is a stream editor used for text manipulation.
- `'/banana/'` is a regular expression pattern that matches lines containing the word "banana."
- `a` is the append command in `sed`.

- "This is a fruit:" is the text you want to append after each line that matches the pattern.
- **file** is the name of the file you want to perform this operation on.

For example, if your "file" contains these lines:

```
css
I like bananas.
Banana is a delicious fruit.
```

I have an apple and a banana.

Running the **sed** command you provided will result in the following output:

```
less
I like bananas.
This is a fruit:
Banana is a delicious fruit.
```

This is a fruit:

I have an apple and a banana.

This is a fruit:

As you can see, the line "This is a fruit:" has been appended immediately after each line containing "banana."

SA

`sed '/^apple/a This starts with "apple".' file`

The **sed** command you provided, `sed '/^apple/a This starts with "apple".' file`, appends the line "This starts with "apple"." immediately after each line in the file that starts with the word "apple."

Here's how it works:

- **sed** is a stream editor used for text manipulation.
- `'/^apple/'` is a regular expression pattern that matches lines starting with the word "apple." The `^` symbol in the pattern denotes the start of a line.

- `a` is the append command in `sed`.
- "This starts with "apple". " is the text you want to append after each line that matches the pattern.
- `file` is the name of the file you want to perform this operation on.

For example, if your "file" contains these lines:

```
css
I like apples.
Apple is a fruit.
```

I have an apple and a banana.

Running the `sed` command you provided will result in the following output:

```
vbnnet
I like apples.
This starts with "apple".
```

Apple is a fruit.

This starts with "apple".

I have an apple and a banana.

As you can see, the line "This starts with "apple"." has been appended immediately after each line starting with "apple."

SA

```
sed '/banana/s/^/The fruit is:/' file
```

The `sed` command you provided, `sed '/banana/s/^/The fruit is:/' file`, modifies lines in the file that contain the word "banana" by adding "The fruit is:" to the beginning of those lines. Here's how it works:

- `sed` is a stream editor used for text manipulation.
- `'/banana/'` is a regular expression pattern that matches lines containing the word "banana."

- `s/^/The fruit is:/` is the substitution command that performs the modification. It replaces the start of each matching line (^ represents the start of a line) with "The fruit is:".
- `file` is the name of the file you want to perform this operation on.

For example, if your "file" contains these lines:

```
css
I like bananas.
Banana is a delicious fruit.
```

`I have an apple and a banana.`

Running the `sed` command you provided will result in the following output:

```
less
I like bananas.
The fruit is:Banana is a delicious fruit.

I have an apple and a banana.
```

As you can see, "The fruit is:" has been added to the beginning of lines that contain "banana." If you want to insert a space between "The fruit is:" and the rest of the line, you can modify the command as follows:

```
bash
sed '/banana/s/^/The fruit is: /' file
```

With this modification, you will get the following output:

```
less
I like bananas.
The fruit is: Banana is a delicious fruit.

I have an apple and a banana.
```

Now there's a space between "The fruit is:" and the rest of the line.

SA

```
sed '/banana/s/^/The fruit is:/' file
```

Apologies for any confusion in my previous response. The `sed` command you provided, `sed '/banana/s/^/The fruit is:/' file`, will

indeed add "The fruit is:" to the beginning of lines in the file that contain the word "banana." Here's how it works:

- `sed` is a stream editor used for text manipulation.
- `'/banana/'` is a regular expression pattern that matches lines containing the word "banana."
- `s/^/The fruit is:/` is the substitution command that performs the modification. It replaces the start of each matching line (^ represents the start of a line) with "The fruit is:".
- `file` is the name of the file you want to perform this operation on.

For example, if your "file" contains these lines:

```
css
I like bananas.
Banana is a delicious fruit.

I have an apple and a banana.
```

Running the `sed` command you provided will result in the following output:

```
less
I like bananas.
The fruit is:Banana is a delicious fruit.

I have an apple and a banana.
```

As you can see, "The fruit is:" has been added to the beginning of lines that contain "banana." If you want to insert a space between "The fruit is:" and the rest of the line, you can modify the command as follows:

```
bash
sed '/banana/s/^/The fruit is: /' file
```

With this modification, you will get the following output:

```
less
I like bananas.
The fruit is: Banana is a delicious fruit.

I have an apple and a banana.
```

Now there's a space between "The fruit is:" and the rest of the line.

SA

sed '/banana/s/\$/ is a fruit/' file

The `sed` command you provided, `sed '/banana/s/$/ is a fruit/' file`, will add " is a fruit" to the end of lines in the file that contain the word "banana." Here's how it works:

- `sed` is a stream editor used for text manipulation.
- `'/banana/'` is a regular expression pattern that matches lines containing the word "banana."
- `s/$/ is a fruit/` is the substitution command that performs the modification. It replaces the end of each matching line (\$) represents the end of a line) with " is a fruit."
- `file` is the name of the file you want to perform this operation on.

For example, if your "file" contains these lines:

```
css
I like bananas.
Banana is a delicious.
```

I have an apple and a banana.

Running the `sed` command you provided will result in the following output:

```
css
I like bananas is a fruit
Banana is a delicious is a fruit

I have an apple and a banana is a fruit
```

As you can see, " is a fruit" has been added to the end of lines that contain "banana."

SA

sed 's/b./\U&/g' file

The `sed` command you provided, `sed 's/\b./\U&/g' file`, performs a substitution on each line in the file named "file." It replaces each word boundary (`\b`) followed by any character (.) with the uppercase version of that character (`\U&`).

Here's how the command works:

- `sed` is a stream editor used for text manipulation.
- `s/\b./\U&/g` is the substitution command:
 - `\b` represents a word boundary.
 - `.` represents any character.
 - `\U&` transforms the matched character into uppercase.
 - `g` at the end means to perform this substitution globally on each line.

For example, if your "file" contains these lines:

`hello world`

`goodbye earth`

Running the `sed` command you provided will result in the following output:

`Hello World`

`Goodbye Earth`

As you can see, it capitalized the first letter of each word in the file while leaving the rest of the text unchanged.

SA

`sed -E '/(apple|banana)/s/$/ is a fruit/; /(cashew)/s/$/ is a nut/;`

The `sed` command you provided uses extended regular expressions (-E option) to perform two different substitutions on each line in the input file. Here's how it works:

- `/ (apple|banana) /` is a regular expression pattern that matches lines containing either "apple" or "banana."

- `/cashew/` is a regular expression pattern that matches lines containing "cashew."
- `s/$/ is a fruit/` is the substitution command for lines containing "apple" or "banana." It appends "is a fruit" to the end of those lines.
- `s/$/ is a nut/` is the substitution command for lines containing "cashew." It appends "is a nut" to the end of those lines.

For example, if your input file contains the following lines:

```
css
I like apples.
I have a banana.
```

Cashews are delicious.

Almonds are great too.

Running the `sed` command you provided will result in the following output:

```
css
I like apples is a fruit.
I have a banana is a fruit.
```

Cashews are delicious is a nut.

Almonds are great too.

As you can see, it applied the appropriate substitution based on the presence of "apple," "banana," or "cashew" in each line. Lines containing "apple" or "banana" have "is a fruit" appended, and lines containing "cashew" have "is a nut" appended. Lines without any of these words remain unchanged.

SA

Is there any error in this command?

My apologies, but there is indeed an issue with the command you provided. It seems to be missing the closing single quote and the input file name. Here's the corrected version of the `sed` command:

```
bash
sed -E '/(apple|banana)/s/$/ is a fruit/; /(cashew)/s/$/
is a nut/' inputFile
```

I added the single quote to close the `sed` command and replaced `inputfile` with the actual name of the file you want to operate on. Please replace `inputfile` with the name of your input file, and the command should work correctly.

SA

Given a file named `numbers.txt` containing a list of numbers, one number per line, write an AWK script to calculate and print the average of these numbers

You can use the following AWK script to calculate and print the average of numbers from a file named "numbers.txt," where each number is on a separate line:

```
awk
#!/usr/bin/awk -f

# Initialize variables
{
    sum += $1      # Accumulate the sum of all numbers
    count++        # Count the number of lines (numbers)
}

# END block is executed after processing all lines
END {
    if (count > 0) {
        average = sum / count    # Calculate the average
        printf "Average: %.2f\n", average    # Print the
        average with two decimal places
    }
}
```

```
    } else {  
        print "No numbers found in the file."  
    }  
}  
}
```

To use this AWK script, follow these steps:

1. Open a text editor and paste the script into a new file. Save the file with a .awk extension, e.g., "calculate_average.awk."
2. Make the script executable (if it's not already) using the following command:

```
bash  
chmod +x calculate_average.awk
```

3. Run the script with your "numbers.txt" file as input like this:

```
bash  
../calculate_average.awk numbers.txt
```

Replace "numbers.txt" with the name of your input file. The script will read the numbers from the file and calculate the average, then print it to the console with two decimal places.

3 / 3

SA

```
awk -f average.awk numbers.txt
```

It seems like you're trying to use an AWK script named "average.awk" with the input file "numbers.txt" to calculate the average of the numbers. Assuming you have the "average.awk" script set up correctly, you can run it with the following command:

```
bash  
awk -f average.awk numbers.txt
```

Make sure that both the "average.awk" script and the "numbers.txt" file are in the same directory where you execute the command, or provide the full path to the script and file if they are in different locations.

The script will read the numbers from "numbers.txt" and calculate the average, then print it to the console.

SA

explain the command

Certainly, let me explain the command:

- `awk` is a powerful text processing tool commonly used for data manipulation.
- `-f average.awk` specifies that you want to use the AWK script stored in the file named "average.awk" for processing.
- `numbers.txt` is the input file that the AWK script will process.

Here's a breakdown of what the AWK script in "average.awk" does:

1. Initialize Variables:

- When the AWK script starts executing, it processes each line of the input file "numbers.txt."
- For each line, it adds the number on that line to the `sum` variable using `sum += $1`, where `$1` represents the first field (in this case, it's the number on the line).
- It also increments the `count` variable by 1 to keep track of how many numbers are processed.

2. END Block:

- The `END` block is executed after processing all lines in the input file.
- Inside the `END` block, it checks if `count` (the number of lines/numbers) is greater than zero.
 - If there are numbers in the file, it calculates the average by dividing `sum` (the sum of all numbers) by `count`.
 - It then uses `printf` to print the average with two decimal places: `printf "Average: %.2f\n", average`.

- The `% .2f` format specifier is used to display the average as a floating-point number with two decimal places.
- If there are no numbers in the file (i.e., `count` is zero), it prints "No numbers found in the file."

So, when you run the command `awk -f average.awk numbers.txt`, it executes the "average.awk" script, which calculates the average of the numbers in "numbers.txt" and prints it to the console.

SA

what -f do?

The `-f` option in the `awk` command is used to specify a file that contains an AWK script. When you use `-f` followed by a filename, AWK reads the script from that file and executes it. Here's how it works:

1. You create an AWK script and save it in a file. For example, let's say you have an AWK script in a file named "myscript.awk."
2. You can then use the `-f` option to tell AWK to read and execute the script from that file. The command looks like this:

```
bash  
awk -f myscript.awk inputFile
```

- `-f myscript.awk`: Specifies the AWK script file ("myscript.awk" in this case).
- `inputfile`: Specifies the input file that the AWK script will process.

Using the `-f` option is convenient when you have a complex or lengthy AWK script because it allows you to keep the script code in a separate file rather than including it directly in the command line. This can make your commands more readable and maintainable, especially for larger scripts.

SA

What is the difference between scripts with extension `.awk` and with `.sh`

The difference between scripts with the file extensions .awk and .sh lies in their intended use, scripting languages, and functionality:

1. .awk Scripts:

- **AWK** is a text processing and data manipulation language that excels at working with structured data, such as columns and fields.
- AWK scripts are primarily used for text processing tasks like extracting specific data from files, reformatting text, and performing calculations on structured data.
- AWK scripts are especially useful for tasks involving tabular data, logs, and reports.
- AWK scripts are typically more concise and tailored for text manipulation tasks.
- Example .awk script:

```
awk
# This is a simple AWK script to print the
# second column of a CSV file
BEGIN { FS = "," }

{ print $2 }
```

2. .sh Scripts (Bash Scripts):

- **.sh scripts** are often written in the **Bash** scripting language, which is a command-line shell for Unix-like operating systems.
- Bash scripts are versatile and can handle a wide range of tasks, including system administration, process automation, file manipulation, and more.
- Bash scripts can execute system commands, control the flow of programs, and handle user input.
- Bash scripts are not specifically designed for text processing but can perform such tasks when needed.
- Example .sh script:

```
bash
# This is a simple Bash script to count the
# number of files in a directory
count=$(ls | wc -l)

echo "There are $count files in the directory."
```

In summary, .awk scripts are specialized for text processing and data manipulation, particularly with structured data, while .sh (Bash) scripts are more general-purpose and versatile, often used for system administration and automation tasks. The choice between .awk and .sh scripts depends on the specific requirements of the task at hand.

SA

Given a file named values.txt with a list of numbers, one number per line, write an AWK script to find and print the maximum and minimum values from the file

You can use the following AWK script to find and print the maximum and minimum values from a file named "values.txt," where each number is on a separate line:

```
awk
#!/usr/bin/awk -f

# Initialize variables
BEGIN {
    max = min = 0
}

# Process each line of the input
{
    if (NR == 1) {
        max = min = $1
    }
    else {
        if ($1 > max)
            max = $1
        if ($1 < min)
            min = $1
    }
}
END {
    print "Max value is " max
    print "Min value is " min
}
```

```

} else {

    if ($1 > max) {

        max = $1

    }

    if ($1 < min) {

        min = $1

    }

}

# END block is executed after processing all lines

END {

    if (NR > 0) {

        printf "Maximum value: %s\n", max
        printf "Minimum value: %s\n", min

    } else {

        print "No numbers found in the file."
    }

}

```

To use this AWK script, follow these steps:

1. Open a text editor and paste the script into a new file. Save the file with a .awk extension, e.g., "find_max_min.awk."

2. Make the script executable (if it's not already) using the following command:

```

bash
chmod +x find_max_min.awk

```

3. Run the script with your "values.txt" file as input like this:

```
bash  
./find_max_min.awk values.txt
```

Replace "values.txt" with the name of your input file. The script will read the numbers from the file, find and print the maximum and minimum values, and display them on the console.

3 / 3

SA

```
BEGIN { # Initialize the maximum and minimum values. max = 0 min = 0  
} { # Check if the line is empty. if ($0) { # Check if the number on the line  
is greater than the maximum value. if ($1 > max) { max = $1 } # Check if  
the number on the line is less than the minimum value. if ($1 < min) { min  
= $1 } } } END { # Print the maximum and minimum values. print  
"Max:", max print "Min:", min }
```

Your AWK script looks good, and it will effectively find and print the maximum and minimum values from the "values.txt" file. Here's a breakdown of how your script works:

- In the **BEGIN** block, you initialize the **max** and **min** variables to 0.
- For each line in the input file:
 - You check if the line is not empty (**if (\$0)**), which ensures that empty lines are not considered.
 - You compare the number on the line (**\$1**) with the current maximum and minimum values (**max** and **min**).
 - If the number on the line is greater than the current maximum (**\$1 > max**), you update the maximum value (**max = \$1**).
 - If the number on the line is less than the current minimum (**\$1 < min**), you update the minimum value (**min = \$1**).
- In the **END** block, you print the maximum and minimum values with appropriate labels.

Your script will correctly identify the maximum and minimum values from the "values.txt" file and print them to the console.

SA

You check if the line is not empty (`if ($0)`), which ensures that empty lines are not considered. You compare the number on the line (`$1`) with the current maximum and minimum values (`max` and `min`).

You are correct in your understanding. Let me clarify the two points you mentioned:

- 1. Checking if the line is not empty (`if ($0)`):** This part of the script ensures that empty lines in the input file are not considered when determining the maximum and minimum values. The condition `if ($0)` checks if the entire line (`$0`) contains any characters. If it's empty, meaning there are no characters on that line, the line is skipped. This is important to avoid incorrect calculations if the file contains empty lines.
- 2. Comparing the number on the line (`$1`) with `max` and `min`:** The script reads each line of the input file and extracts the number from the line using `$1`. It then compares this number to the current maximum and minimum values (`max` and `min`). If the number is greater than the current maximum value (`$1 > max`), it updates `max`. Similarly, if the number is less than the current minimum value (`$1 < min`), it updates `min`. This way, the script keeps track of the maximum and minimum values encountered while processing the file.

Your explanation is accurate, and your AWK script effectively implements these checks to find the maximum and minimum values in the input file.

SA

Given a file named `grades.csv` with student names and their respective scores, write an AWK script to calculate and print the letter grades for each student based on the following scale: EX: 90-100; A: 80-89 ...

To calculate and print letter grades for each student based on a score scale in an AWK script, you can use the following example script. In this script, we'll assume the score scale ranges from 0 to 100:

```
awk
#!/usr/bin/awk -f

# Define the score-to-grade mappings

BEGIN {
    # Define the score ranges and corresponding letter
    # grades

    grade_scale[90] = "A"
    grade_scale[80] = "B"
    grade_scale[70] = "C"
    grade_scale[60] = "D"
    grade_scale[0] = "F"
}

# Process each line in the CSV file
{
    # Extract the student name and score from the CSV
    # line

    student_name = $1
    score = $2

    # Determine the letter grade based on the score
    for (score_limit in grade_scale) {
        if (score >= score_limit) {
```

```

        letter_grade = grade_scale[score_limit]
    break
}
}

# Print the student's name and letter grade
print student_name ":" letter_grade
}

```

Here's how the script works:

1. In the **BEGIN** block, we define the score-to-grade mappings. You can customize this part to match your specific grading scale.
2. For each line in the CSV file, the script extracts the student name and score.
3. It then iterates through the score ranges defined in the **grade_scale** array and checks which range the student's score falls into.
4. Once it finds the correct range, it assigns the corresponding letter grade to the **letter_grade** variable.
5. Finally, it prints the student's name and their calculated letter grade.

To use this script, save it in a file (e.g., "calculate_grades.awk") and make it executable with `chmod +x calculate_grades.awk`. Then, you can run it with your "grades.csv" file as follows:

```

bash
./calculate_grades.awk grades.csv

```

Replace "grades.csv" with the name of your input CSV file containing student names and scores. The script will calculate and print the letter grades for each student based on the specified scale.

SA

It is giving wrong output

I apologize for any confusion. Let's correct the script to ensure it calculates the letter grades based on the specified score ranges. We'll use a revised approach where we check each score against the grading scale. Here's the corrected AWK script:

```
awk
#!/usr/bin/awk -f

# Define the score-to-grade mappings
BEGIN {
    # Define the score ranges and corresponding letter
    # grades
    grade_scale[90] = "A"
    grade_scale[80] = "B"
    grade_scale[70] = "C"
    grade_scale[60] = "D"
    grade_scale[0] = "F"
}

# Process each line in the CSV file
{
    # Extract the student name and score from the CSV
    # line
    student_name = $1
    score = $2
```

```

# Initialize the letter grade to be empty
letter_grade = ""

# Determine the letter grade based on the score
for (score_limit in grade_scale) {
    if (score >= score_limit) {
        letter_grade = grade_scale[score_limit]
    }
}

# Print the student's name and letter grade
print student_name ": " letter_grade
}

```

With this script, it will correctly assign the letter grades based on the specified score ranges. Make sure you have saved the script, made it executable, and then run it with your "grades.csv" file as previously described. If you encounter any issues or need further assistance, please provide more details about the problem you are facing.

SA

Given a file named sales.csv with product names and their quantities sold and prices per unit, write an AWK script to calculate and print the total sales revenue for each product (quantity * price) and the overall total sales revenue

To calculate and print the total sales revenue for each product and the overall total sales revenue based on a CSV file named "sales.csv," you can use the following AWK script:

```

awk
#!/usr/bin/awk -f

```

```
# Initialize variables

BEGIN {

    FS = "," # Set the field separator to comma
    OFS = "," # Set the output field separator to comma
    total_sales = 0

}

# Process each line in the CSV file

{

    # Extract fields from the CSV line
    product_name = $1
    quantity_sold = $2
    price_per_unit = $3

    # Calculate the total sales revenue for the product
    # (quantity * price)
    product_sales = quantity_sold * price_per_unit

    # Add the product sales to the overall total sales
    total_sales += product_sales

    # Print the product name and its sales revenue
    print product_name, product_sales

}
```

```

# Print the overall total sales revenue

END {
    print "Overall Total Sales Revenue:", total_sales
}

```

Here's how the script works:

1. In the **BEGIN** block:

- It sets the field separator (**FS**) to a comma, assuming that the CSV file uses commas to separate fields.
- It sets the output field separator (**OFS**) to a comma for consistent CSV output formatting.
- Initializes the **total_sales** variable to keep track of the overall total sales revenue.

2. For each line in the CSV file:

- It extracts the product name, quantity sold, and price per unit from the CSV line.
- Calculates the total sales revenue for the product by multiplying the quantity sold by the price per unit.
- Adds the product sales revenue to the overall total sales (**total_sales**).

3. In the **END** block:

- It prints the overall total sales revenue, along with a label.

To use this script, save it in a file (e.g., "calculate_sales.awk") and make it executable with **chmod +x calculate_sales.awk**. Then, run it with your "sales.csv" file as follows:

```

bash
./calculate_sales.awk sales.csv

```

Replace "sales.csv" with the name of your input CSV file. The script will calculate and print the total sales revenue for each product and the overall total sales revenue.

SA

generate sales.csv file also

Certainly, here's an example of a "sales.csv" file with some sample data for demonstration purposes:

csv

Product Name	Quantity Sold	Price Per Unit
Product A	10	15.00

Product B, 5, 25.50

Product C, 8, 12.75

Product D, 15, 10.00

You can create this "sales.csv" file by opening a text editor and copying the content above into a new file. Save the file with the name "sales.csv" and ensure that it uses commas (,) as field separators to match the AWK script provided earlier.

You can replace the sample data with your actual sales data when you use the script to calculate sales revenue.

SA

Given a file named sales_data.csv containing monthly sales data for a product, write an AWK script to compute the percentage increase in sales from one month to the next and print the results.

To compute the percentage increase in sales from one month to the next in a CSV file named "sales_data.csv" and print the results, you can use the following AWK script:

```
awk  
#!/usr/bin/awk -f
```

```
# Initialize variables
```

```

BEGIN {

    FS = "," # Set the field separator to comma
    OFS = "," # Set the output field separator to comma
    prev_sales = 0
}

# Process each line in the CSV file
{
    # Extract fields from the CSV line
    month = $1
    sales = $2

    # Calculate the percentage increase from the previous
    # month
    if (NR > 1) { # Skip the header line
        percentage_increase = ((sales - prev_sales) /
prev_sales) * 100
    }

    # Print the result
    print "Month:", month
    print "Percentage Increase:", percentage_increase
    "%" 
}

```

sayak@Ubuntu:~/harry/files_module2_l1\$ cat text.txt

```

chown :hst.dat
C) chorx test.dat HPC
4. What is the function o
June,160 HPC year hpc

```

sayak@Ubuntu:~/harry/files_module2_l1\$ bash count_hpc.sh text.txt

```

The number of occurrences of the word HPC in the file text.txt is 2.

```

sayak@Ubuntu:~/harry/files_module2_l1\$

```

        print "" # Print an empty line for readability
    }

# Store the current month's sales as the previous
month's sales for the next iteration

prev_sales = sales
}

```

Here's how the script works:

1. In the **BEGIN** block:

- It sets the field separator (**FS**) to a comma, assuming that the CSV file uses commas to separate fields.
- It sets the output field separator (**OFS**) to a comma for consistent CSV output formatting.
- Initializes the **prev_sales** variable to keep track of the previous month's sales.

2. For each line in the CSV file:

- It extracts the month and sales data from the CSV line.
- Calculates the percentage increase from the previous month's

```

sayak@Ubuntu:~/harry/files_module2_l1$ bash file_exist.sh calculate_grades.awk
File calculate_grades.awk exists and is executable by the current user.
sayak@Ubuntu:~/harry/files_module2_l1$ bash file_exist.sh count_hpc.sh
File count_hpc.sh is not executable by the current user.
sayak@Ubuntu:~/harry/files_module2_l1$ bash file_exist.sh my.txt
File my.txt does not exist.
sayak@Ubuntu:~/harry/files_module2_l1$ 

```

sales to the current month's sales.

- Prints the month and its corresponding percentage increase.
- An empty line is printed for readability between results.

3. The script skips the header line (assumed to be the first line in the CSV file) by checking **NR > 1** (record number greater than 1).

To use this script, save it in a file (e.g., "calculate_percentage_increase.awk") and make it executable with **chmod**

+x calculate_percentage_increase.awk. Then, run it with your "sales_data.csv" file as follows:

```
bash  
./calculate_percentage_increase.awk sales_data.csv
```

Replace "sales_data.csv" with the name of your input CSV file containing monthly sales data for a product. The script will calculate and print the percentage increase in sales from one month to the next and display the results for each month.

SA

generate sale_data.csv file

Certainly, here's an example of a "sales_data.csv" file with some sample monthly sales data for demonstration purposes:

```
csv  
Month,Sales  
January,1000
```

February,1250

```
sayak@Ubuntu:~/harry/files_module2_l1$ cat text.txt  
chown :hst.dat  
C) chorx test.dat HPC  
4. What is the function o  
June,160 HPC year hpc  
sayak@Ubuntu:~/harry/files_module2_l1$ bash word_counter.sh text.txt  
The number of words in the file text.txt is 16 text.txt
```

March,1350

April,1100

May,1400

June,1600

July,1550

August,1650

September,1800

October,1900

November , 2100

December , 2200

You can create this "sales_data.csv" file by opening a text editor, copying the content above into a new file, and then saving the file with the name "sales_data.csv."

Each line represents a month, and the "Sales" column contains the corresponding sales data. You can replace this sample data with your actual monthly sales data when you use the AWK script to calculate and print the percentage increase in sales.

SA

Write a Bash script that accepts a filename as an argument and counts the number of occurrences of the word HPC in the file? The output should print the number of occurrences.

```
#!/bin/bash

# Check if the script was invoked with an argument. This part of the script checks if exactly one command-line argument (the filename) is provided when invoking the script. If not, it displays a usage message and exits with a non-zero status code.
if [ $# -ne 1 ]
then
    echo "Usage: $0 <filename>"
    exit 1
fi

# Get the filename from the argument. It assigns the first command-line argument (the filename) to the filename variable.
Filename=$1

# Count the number of occurrences of the word HPC in the file.
num_occurrences=$(grep -c "HPC" "$filename")

# Print the number of occurrences to the standard output.
echo "The number of occurrences of the word HPC in the file $filename is $num_occurrences."
```

2. Write a Bash script that accepts a filename as input, checks if the file exists, and determines if it is executable by the current user. Print the results.

```
#!/bin/bash

# Get the filename from the argument.
filename=$1

# Check if the file exists.
if [ ! -f "$filename" ]
then
    echo "File $filename does not exist."
    exit 1
fi

# Check if the file is executable by the current user.
if [ ! -x "$filename" ]
then
    echo "File $filename is not executable by the current user."
    exit 1
fi

# The file exists and is executable by the current user.
echo "File $filename exists and is executable by the current user."
```

output went to page 413

3. Write a script that counts the number of words in a text file and prints the count. The script should accept a filename as input.

```
#!/bin/bash

# Get the filename from the argument.
filename=$1

# Check if the file exists.
if [ ! -f "$filename" ]
then
    echo "File $filename does not exist."
    exit 1
fi

# Count the number of words in the file.
num_words=$(wc -w "$filename")

# Print the number of words to the standard output.
echo "The number of words in the file $filename is $num_words."
```

output went to page 414