

Week1

- Following permissions are required for file/directory level operations.

Permissions File type	Read	Write	Execute
File	Read file contents (cat) Execute/run file(./file)	Write/overwrite (touch, cat>, edit)	Execute/run file (./file) Change permissions(chmod)
Directory	List files/sub-folders (ls)	Create files/subfolders (mkdir)	Change directory(cd) Delete (rmdir, rm -r)

- In order to execute a file , *read* and *execute* permissions are both needed.
- `ln file1 file1_hard` creates a hard-link of *file1*, and calls it *file1_hard*.
- `ln -s file1 file1_soft` creates a soft-link of *file1*, and calls it *file1_soft*.
- To see the *inode* numbers of files, use `ls -li`

```
anand@DESKTOP-GG7GDNK:~/dir_2$ ls -li
total 8
22254 -rw-r--r-- 1 anand anand 6 Sep 26 11:40 file1
13958 lrwxrwxrwx 1 anand anand 5 Sep 26 11:38 file1_soft -> file1
11199 -rw-r--r-- 1 anand anand 6 Sep 26 11:31 file2
19577 lrwxrwxrwx 1 anand anand 5 Sep 26 11:38 file2_soft -> file2
```
- Hard links do not work across different file systems or devices, but soft links do.
- Hard links to the same file have same inode. Modifying one affects the other, but deleting one doesn't delete the other.
- Soft links to the same file have different inodes. Modifying one affects the other, and deleting one renders the others dead.
- Once created, moving the location of a soft-link renders it dead, unless the soft-link was created with an absolute path. Thus, in order to create a soft link *file_soft* to the absolute path to *file*, use `ln -s `pwd`/file file_soft`
- Copies of the same file have different inodes, and are entirely independent of each other. Modifying one doesn't affect the other, and deleting one doesn't delete the other.
- It's not possible to create hard links for directories, since it can lead to self-reference and thus infinite recursion.
- To display the type of the file (file/directory/link), use *file* command as follows

```
anand@DESKTOP-GG7GDNK:~/dir_2$ file *
file1:      ASCII text
file1_soft: symbolic link to file1
file2:      ASCII text
file2_soft: symbolic link to file2
```
- To see the file properties, use `stat <file_name>`
- Number of hard links to a directory is 2 (due to . and ..) more than number of its sub-directories (including hidden directories)
- `touch file` command changes the modification date/time of the file.

Week2

- **echo** command accepts multiple parameters separated by a single space, and prints them.
- **echo** accepts multiline input by using quotes (single/double) around the input.
- To display shell variables, use command **echo <variable>**. \$HOME, \$USER, \$HOSTNAME, \$PWD, \$PATH are a few among the built-in shell variables. You may use double quotes around the variables, but not single-quotes, in which case the variable is treated as a literal string.

```
gphani@icme:~$ echo "$USERNAME"
gphani
gphani@icme:~$ echo '$USERNAME'
$USERNAME
```

NOTE: Using double-quotes with a shell variable interpolates it inside a larger string, and escaping it using '\ ' treats it as a literal. It's a good idea to use braces around the variable while interpolating.

- **echo** prints the string/variable content, and terminates with a newline. **echo -n** doesn't print a newline at the end.
- **whoami** shows the current user of the terminal.
- **whatis <command>** displays a brief help on the command.
- **man <command>** shows the detailed help on the command. So does **<command> --help** and **info <command>**
- **whereis <command>** shows the location of the command executable.
- **which <command>** also shows the location of command executable, but has restricted scope of search in comparison to *whereis*.
- **echo \$HOSTNAME** displays the hostname of the local system, as stored in */etc/hostname*
- **echo \$SHELL** displays the path of the current shell.
- Assuming that the above command displays */bin/bash*, **echo \${SHELL:5:4}** displays *bash* (4 characters starting with the 5th character).
- **printenv, env, set** are commands that list the shell variables in the shell.
- Following are special shell variables used by the Linux bash shell, and their meanings.
 - **\$0** – name of the shell
 - **\$\$** - process ID of the shell (same as what gets displayed in *ps* command)
 - **\$?** – return code of the previous run program
 - **\$-** - flags set in bash shell
- Process control commands
 - **ps** – list of current processes.
 - **&** - run a process in the background
 - **fg** - bring the background process to foreground
 - **coproc** – run processes in the simultaneously.
 - **jobs** – list jobs in shell
 - **top** – list processes that hog memory
 - **kill** – kill processes.
- Program [exit codes](#) (between 0 and 255)

- 0 – Success (evaluated as true by shell script)
- 1 – Failure
- 2 – Misuse
- 126 – Command cannot be executed
- 127 – Command not found
- 130 – Process killed using Ctrl + C
- 137 – Process killed using kill -9 <pid>
- **date -R** prints the current system datetime in a format useful in email communications.
- **alias date="date -R"** creates an alias for *date* command and executes **date -R** in turn. If you want to NOT use the alias, use escaping as in `\date`. To unset the alias, use **alias date=date**
- **ps -ef** lists all processes, along with details about parent pid (ppid) and other details.
- **echo \$BASH_SUBSHELL** gives a number indicating the #subshell where execution is happening now. Consider the following execution, where inside each pair of parenthesis the \$BASH_SUBSHELL increments.


```
anand@DESKTOP-GG7GDNK:~$ echo $BASH_SUBSHELL;(echo $BASH_SUBSHELL;(echo $BASH_SUBSHELL;(echo $BASH_SUBSHELL)))
0
1
2
3
```
- **NOTE:** To run multiple commands in a sequence, separate each by semi-colon.
- When **&&** is used in between two commands, both commands are executed one after the other. However, if the first one fails, the second one is ignored.
- When **||** is used in between two commands, the second one is executed, only if the first one fails and is ignored, when the first one passes.


```
anand@DESKTOP-GG7GDNK:~$ ls this_folder_does_not_exist;ls
ls: cannot access 'this_folder_does_not_exist': No such file or directory
check_sum.py client client.code-workspace dir_2 go push server
anand@DESKTOP-GG7GDNK:~$ ls this_folder_does_not_exist && ls
ls: cannot access 'this_folder_does_not_exist': No such file or directory
anand@DESKTOP-GG7GDNK:~$ ls this_folder_does_not_exist || ls
ls: cannot access 'this_folder_does_not_exist': No such file or directory
check_sum.py client client.code-workspace dir_2 go push server
anand@DESKTOP-GG7GDNK:~$ ls || ls
check_sum.py client client.code-workspace dir_2 go push server
```
- Linux supports 3 file descriptors – stdin (0), stdout(1) and stderr(2)
- To redirect *stdout* to *out.txt*, use **>out.txt**. In this case, the output will not be displayed on the screen. If you want the output to also appear on the screen (In addition to redirecting to *out.txt*, use **| tee out.txt**. For example, **ls \$HOME | tee out.txt** will create *out.txt* with the contents of the \$HOME directory, as well display it on the screen. Note that *tee* command can be used to redirect output to multiple files by providing file names separated by space.
- To extract the parent directory, use **dirname "\$MYFILE"**. To extract the grandparent, use **dirname \$(dirname "\$MYFILE")**
- To redirect *stderr* to *err.txt*, use **2>err.txt**. It's common to redirect *stderr* output to */dev/null* – **2>/dev/null**.
- To redirect *stderr* to the same file where *stdout* is redirected, use **2>&1**. For example, **ls \$HOME /blah > file1 2>&1** will redirect *stderr* to *stdout* and list the files in the home directory and redirect *stdout* to *file1*. Thus, *file1* will contain the error (about non-existing directory /blah) as well as the listing of files in \$HOME. Alternatively, it can be rewritten as **ls \$HOME /blah &> file1**

- `less <file>` displays the contents of <file> and allows scrolling line by line.
- `more <file>` displays the contents of <file> and allows scrolling page by page.
- `Cat > file1` will accept the text typed by the user and stores in file1. Stop entering text by pressing Ctrl + D on keyboard.
- Variables can be assigned with a numeric or a string value. It can also be assigned with the result of a Linux command, by enclosing the command in backquotes (``). For example, to assign the number of files in the current directory to a variable, run `num_files=`ls | wc -l``. Now, `echo $num_files` will print the number of files in the directory.
- Use `read var` to read the user input into variable `var`. Use `read -a arr` reads user input split by spaces into an array `arr`.
- Variable name can have alphanumeric characters and `_`. Assigning values to variables can't have space around `=`.
- In order to make a variable available on the shell and its sub-shells, use `export` command. Note that `export` will not affect the parent shell variables.
- To assign `v` to `v1`, use `v1=$v`.
- To unassign `$v`, use `unset v` command or `v=<empty>` (leave RHS of `=` empty)
- `[[-v v]]` returns 0, if `$v` is set, else returns 1.
- To echo a default value 'default' if the variable `$v` is not set, say `echo ${v:-default}`
- To set a default value 'default' if the variable `$v` is not set, say `echo ${v:=default}`
- To display a message 'msg' if the variable `$v` is not set, say `echo ${v:?msg}`.
- To echo a default value 'default' if the variable `$v` is set, say `echo ${v:+default}`
- `echo ${!H*}` gives the names of all shell variables that start with H
- `echo ${#v}` gives the number of characters in the shell variable `v`.
- `echo ${v: <start>:2}` returns the 2 characters from `$v` starting from <start>. <start> can be a negative number, in which case, it'll start counting from the end of `$v`. Note the space before `start`. Thus,

```
anand@DESKTOP-GG7GDNK:~/my_scripts/Mock1$ echo $a
17:24
anand@DESKTOP-GG7GDNK:~/my_scripts/Mock1$ echo ${a: -4:2}
7:
anand@DESKTOP-GG7GDNK:~/my_scripts/Mock1$ echo ${a: 1:2}
7:
```

- Indirect referencing of variables: In order to reference to a variable whose name is the value of another variable, use `${!var}`.

```
a="indirect"
c="a"
echo $a #indirect
echo $c #a
echo ${!c} #indirect
```

- In order to drop part of a string that occurs at the start, use `#` after the variable. For example, in the last-but-one command below, characters before the first occurrence of `:` is dropped. In the last command, characters before the last occurrence of `:` is dropped.

```

anand@DESKTOP-GG7GDNK:~$ mydate=`date`
anand@DESKTOP-GG7GDNK:~$ echo $mydate
Fri, 29 Sep 2023 20:17:54 +0530
anand@DESKTOP-GG7GDNK:~$ echo ${mydate#*.}
17:54 +0530
anand@DESKTOP-GG7GDNK:~$ echo ${mydate##*.}
54 +0530

```

- In order to drop part of a string that occurs at the end, use % after the variable. For example, in the last-but-one command below, characters after the last occurrence of ':' is dropped. In the last command, characters after the first occurrence of ':' is dropped.

```

anand@DESKTOP-GG7GDNK:~$ mydate=`date`
anand@DESKTOP-GG7GDNK:~$ echo $mydate
Fri, 29 Sep 2023 20:29:04 +0530
anand@DESKTOP-GG7GDNK:~$ echo ${mydate%:*}
Fri, 29 Sep 2023 20:29
anand@DESKTOP-GG7GDNK:~$ echo ${mydate%%:*}
Fri, 29 Sep 2023 20

```

NOTE: Essentially, a single # performs non-greedy matching and ## performs greedy matching. Similarly, for % and %%

- Some additional notes on this is here - <https://discourse.onlinedegree.iitm.ac.in/t/pattern-matching-doubt/107003/7>
- In order to perform replacement of characters once, use `echo ${v/<find>/<replace>}`
- In order to perform replacement of characters globally, use `echo ${v//<find>/<replace>}`
- To convert first character in the string to lowercase, use `${string,}`
- To convert all characters in the string to lowercase, use `${string,,}`

```

string="Hello World"
echo ${string,,} # Output: hello world
echo ${string,} # Output: hello World

```

- To convert first character in the string to uppercase, use `${string^}`
- To convert all characters in the string to uppercase, use `${string^^}`

```

string="hello world"
echo ${string^^} # Output: HELLO WORLD
echo ${string^} # Output: Hello world

```

- To store only numbers in shell variables, declare it as an integer using `declare -i <variable>`. When a non-numeric value is stored in this variable, it gets initialized to 0.
- To store only small case characters in shell variables, declare it so using `declare -l <variable>`. When a non-lowercase value is stored in this variable, it gets initialized to all lower-case.
- To store only upper case characters in shell variables, declare it so using `declare -u <variable>`. When a non-uppercase value is stored in this variable, it gets initialized to all upper-case.
- To declare an array `arr`, use `declare -a arr`. Now, array elements can be accessed as `arr[0]`, `arr[1]` etc. For example, `declare -a arr=(1 2 3 4 5)` declares array `arr` with 5 integer values 1..5
- To append value1 to the array, use `arr+=(value1)`
- To insert value2 to the array at location 100, use `arr[100]=value2`
- In order to find the number of array elements, use `echo ${#arr[@]}`
- In order to list all the elements (values) in the array, use `echo ${arr[@]}`
- In order to list elements starting with 1st index (slicing), use `echo ${arr[@]:1}`

- In order to list all the indices in the array, use `echo ${!arr[@]}`
- To remove array element use unset. For instance, to remove the second index of arr, use `unset arr[2]`
- To declare a hashed array *hash*, use `declare -A hash`. Now, array elements can be accessed as `hash[<index>]`. Note that the index could be a numeric value or a string.
- To automatically create an array *myfiles* (without explicit declaration) consisting of a list of files in the current folder, use `myfiles=(`ls`)`. It could also be stored as `myfiles=($(ls))`.
- To print the values stored in *myfiles*, use `declare -p myfiles`
- To declare variables as read-only, use `declare -r <variable>`. Once the variable has been declared as read-only, it's not possible to turn off the attribute.
- In the above declarations, To turn off the restriction, declare it using +, instead of -
- `sleep <seconds>` sleeps for a specified number of seconds.
- `sleep 2 &` runs in the background (with only output pipe open), and `coproc sleep 2` runs asynchronously (and leaves input and output pipes open for communication)
- `Ctrl+C` kills the command, and `Ctrl+Z` suspends the command (and returns to the prompt)
- `jobs` lists background processes started using either methods
- To kill a job with process id `<p_id>`, use `kill -9 <p_id>`
- `top` will display all jobs active on the local system, ordered by decreasing order or CPU utilization.
- `type <command>` will show the type of `<command>`. Commands that don't have man pages available are typically *shell built-ins*. Help for such commands can be displayed using `<command> --help`.
- To start a new bash sub-shell and launch a set of commands, use `bash -c <commands>`.

```
gphani@icme:~$ bash -c "echo \$$; echo \$-; ps --forest"
11213
hBc
  PID TTY          TIME CMD
  3423 pts/0    00:00:00 bash
  11213 pts/0    00:00:00 \_  bash
  11214 pts/0    00:00:00 \_  ps
```
- To export variable *myvar*, use `declare -x myvar="abc"` or `export myvar="abc"`

Week3

- `lsb_release -a` shows the Linux distribution used in the system.
- `uname -a` shows the kernel information
- *RPM* and *DEBIAN* are two most widely used Linux packaging systems. *RPM* uses *yum*, *dnf* or *rpm* package managers. *DEB* uses *apt*, which internally uses *dpkg*.
- `apt-cache search <keyword>` is used to search for packages that contain the specified keyword.
- To get all the packages installed, use `apt-cache pkgnames`. Pipe it to `sort` command to see the list in alphanumeric order.
- To get a list of packages that start with *wget*, use `apt-cache pkgnames wget`
- To get more information of *wget*, use `apt-cache show wget`. Filename in the output shows the actual file name from where the package is installed. In the filename `wget_1.21.2-2ubuntu1_amd64.deb`, 1.21.2 is the version number, 2ubuntu1 is the revision number and amd64 is the architecture of the package.
- Packages have priorities as listed below - required, important, standard, optional, extra.
- Only sudoers (super users) are allowed to install, update or remove packages. Thus, if a user is available on `/etc/sudoers`, then these operations are allowed for the user.
- `tail -10 /var/logs/auth.log` displays the last 10 lines of the `auth.log`, which displays the access information of protected files.
- `/etc/apt/sources.list` contains the web URLs from where packages can be downloaded.
- In order to upgrade the packages to the latest version/release, run the following commands.
 - `sudo apt-get update`
 - `sudo apt-get upgrade`

While the first command fetches the latest updates from the web locations as per `sources.list` to the cache, the second command applies these updates to the system.

- Use `sudo apt-get install <package>` to install the package into the system, and `sudo apt-get remove <package>` to remove the package from the system

Week4

- `grep` command can use regex, in order to create the pattern. Syntax is `grep <pattern> <file>`. Alternatively, `grep` can be used on the output of a command. Thus, in order to `grep` a `<pattern>` on the output of `ls -l`, use `ls -l | grep <pattern>`
- For extended `grep`, use `egrep` or `grep -e`
- Print only the matched part of the line using `grep -o`. For example, `ls -l | grep -eo '^[-[rwx]-]{9}'` will extract the permission bits only for files in the current folder.
- Ignore case sensitivity using `grep -i`
- In order to invert the matching by pattern, use `grep -v`. For example, `ls -l | egrep -o '\.{3}$'` will get a list of all 3-letter extensions of all files in the current folder. In order to get all files/folders that

doesn't have 3-letter extensions use it as `ls | egrep -v '\.{3}$'`. Note that none of the entries in this output has a 3-letter extension.

- In order to print all the fields following field #5 of every line, where each field is separated by a comma, use `cut -d ',' -f 5-`. Note that the field number starts from 1 (not 0)
- In order to cut characters use `cut -c`. For example, to cut the permission bits of files/folders in the current folder, use `ls -l | cut -c 2-10`
- In order to show line numbers in the grep results, use `grep -n`
- In grep, use of | (pipe symbol) is treated literally. If it needs to be interpreted as an *or* operation, then escape it. Else, use `egrep` or `grep -e`
- To detect empty lines, use `grep '^$'`. Thus, if you need to find the number of empty lines in a file called poem.txt, use `cat poem.txt | egrep '^$' | wc -l`
- Use of backreference in regex needs the original pattern to be parenthesized.
- To find the pattern in a given string *s*, use `$(echo $s | grep <pattern>)`

Week5

- `source <script>` runs the script from the login shell. Thus, when the script is *sourced*, all environment variables created in the script will be available after the script has finished running.
- `./<script>` runs it from within a child process. However, if the script is run as a child process, none of the changes made is available after the script finishes running.
- `printf` supports format specifiers like in C. For example, `printf "My home is %s\n" $HOME`.
- `read var` reads input from the user, and stores in *var*; now `echo $var` prints the stored value.
- `$0` contains name of the shell program (including the invocation path), `$1..$n` contains the command line arguments. `$#` contains the number of such arguments. `$*` (or `"$*"`) contains all arguments as one string. `"$@"` contains all arguments as separate strings.
- To store output of a command into *var*, use `var=$(<command>)` or `var=`<command>``
- To store the *var* into array *my_arr*, use `my_arr=($var)`.
- Thus, the following are equivalent. Both create arrays containing the files in the current directory.

```
anand@DESKTOP-GG7GDNK:~$ ls
check_sum.py  client  go  output.txt  poem.txt  push  server  test.sh
anand@DESKTOP-GG7GDNK:~$ ls_arr=( $(ls) )
anand@DESKTOP-GG7GDNK:~$ declare -p ls_arr
declare -a ls_arr=([0]="check_sum.py" [1]="client" [2]="go" [3]="output.txt" [4]="poem.txt" [5]="push" [6]="server" [7]="test.sh")
```

Is equivalent to

```
anand@DESKTOP-GG7GDNK:~$ ls_var=$(ls)
anand@DESKTOP-GG7GDNK:~$ echo $ls_var
check_sum.py client go output.txt poem.txt push server test.sh
anand@DESKTOP-GG7GDNK:~$ ls_arr1=( $ls_var )
anand@DESKTOP-GG7GDNK:~$ declare -p ls_arr1
declare -a ls_arr1=([0]="check_sum.py" [1]="client" [2]="go" [3]="output.txt" [4]="poem.txt" [5]="push" [6]="server" [7]="test.sh")
```


- Use `if (($expression$))` to compare two integer values. Note that double-parenthesis can be used only to check math expressions, for checking string expressions double-square-brackets must be used.
- You can alternatively use `if test $a -eq $b` or `if [[$a -eq $b]]` to compare variables a and b for equality, if both have numeric values.
- If a and b contain text (non-numeric), use `if [$a = $b]` or `if [$a == $b]` to check a and b for equality. Similarly, use `if [$a = $b]` (not `if [$a -gt $b]`) to check if a is greater than b.
- Use `[[$string =~ $pattern]]` to check if the pattern is found in string.
- To perform arithmetic operations and assign the result to a variable, use `variable=$((math expression))`
- C-style loop uses two pairs of parentheses as in `for ((i=1; i<10; i++))`
- *for-each* loop (similar to Python) can also be used, as in `for i in 1 2 3 4 5`. Similarly, `for i in $@` loops through all command-line arguments to the script.
- `find . -maxdepth 1 -name '*.sh'` prints the names of all files in the current folder. If the *maxdepth* parameter is 2, then it prints the file names in the sub-folder also, but doesn't go further.
- Following are different ways to increment the variable c

- `let c=c+1`
- `c=$((expr $c + 1))`
- `c=$((c+1))` or `((c=c+1))`
- `((c++))`

- Following are different ways to multiply the variable c with 2.

- `let c=c*2`
- `c=$((expr $c \ * 2))`
- `c=$((c*2))` or `((c=c*2))`

NOTE: `c=$((expr $c * 2))` wouldn't work, because `*` has special meaning and hence needs to be escaped.

- Following are different ways to add two numbers from the terminal (without using a variable)

- `expr 1 + 2`
- `let c=1+2; echo $c`
- `echo $((1 + 2))`

NOTE: `expr 1 + 2` cannot be rewritten as `expr 1+2` (without spaces)

- `expr` doesn't support exponentiation, so use a `let` command like `let c=c**2`
- To work with bench calculator, use `bc -l`
- Following are some interesting options available with `if` statement:
 - `[-z STRING]` - checks if *STRING* is empty.
 - `[-n STRING]` - checks if *STRING* is not empty (same as `[! -z STRING]`)
 - `[-d FILE]` - Checks if *FILE* exists and is a directory.
 - `[-e FILE]` - Checks if *FILE* exists.
 - `[-f FILE]` - Checks if *FILE* exists and is a regular file.

- `[-x FILE]` - Checks if *FILE* exists and execute permission is granted.
- `[-s FILE]` - Checks if *FILE* exists and its size is greater than zero (i.e., it's not empty).
- To check if a pattern is found in a string, use `echo $string | grep $pattern`. Use `grep -E` if pattern check needs extended regex.
- Given a string `<str>`, you can replace a substring `<sub>` with `<rep>` using `str=${str/<sub>/<rep>}`. However, regex patterns cannot be used here. For this, use `sed`. Thus, if `str="123Anand345"`, `echo $str | sed -e "s/[[[:digit:]]/_/g"` will replace all digits with `"_"` and result in `"__Anand__"`
- In order to read from *file*, use `cat file | while read line; do echo $line; done`. Alternatively, use `while read line; do echo $line; done < file`
- To read different parts of each line (in a *file*) into separate variables, use `while read part1 part2; do echo $part1 $part2; done < file`
- To find the unique lines in *file*, use `cat file | sort | uniq`. Note file contents need to be sorted using `sort` before `uniq` can be applied.

Week6

- `PS1="\u@|h:\w|$"` sets the prompt string to `"<user>@<host>:<working directory>$"`
- To get back to the default prompt string, run `source ~/.bashrc`
- `PS2` is used to denote continuation of a command across multiple lines.
- `PS3` is used to denote the selection from a set of options.

```
anand@DESKTOP-GG7GDNK:~$ PS3="Choose one of the options:"
anand@DESKTOP-GG7GDNK:~$ select x in alpha beta gamma; do echo $x; done
1) alpha
2) beta
3) gamma
Choose one of the options:1
alpha
```

- `PS4` is used to denote the repetition of the command, before it runs.

```
anand@DESKTOP-GG7GDNK:~$ set -x
anand@DESKTOP-GG7GDNK:~$ PS4='Running command...'
+ PS4='Running command...'
anand@DESKTOP-GG7GDNK:~$ ls
Running command...ls --color=auto
check_sum.py  go          poem.txt    shell-script-
client        hwinfo.txt  push        shell-script-
my_scripts    server      sum.sh
```

NOTE: This behavior is available in the terminal only after command `set -x` is issued. To revert to the old behavior, use `set +x`

- Following options could be used to create a prompt string.
 - `\t` – time
 - `\d` – date
 - `\#` - command number
- `crontab` is a service in Linux that helps to schedule tasks and automatically execute scripts at pre-determined time. To add a new entry into the `crontab` list, use `crontab -e`
- `crontab` sequence given below will execute `/home/scripts/backup/mkbackup.sh` script at 2.05 AM everyday of every month.

```
5 2 * * 1-5 root cd /home/scripts/backup && ./mkbackup.sh
```

NOTE: `5 2 10 12 1-5 cd /home/scripts/backup && ./mkbackup.sh` will execute `/home/scripts/backup/mkbackup.sh` script at 2.05 am on 10th December, provided the date is one of the week days. If 10th December falls on a week-end, it'll not run.

- `/etc/init` and `/etc/init.d` contains scripts that needs to be run during startup of the system. Note that WSL don't have these folders. Instead, add them to `~/.bashrc` file.
- `-e` option passes inline script to `sed`. It's not mandatory to use `-e` switch, unless executing multiple inline commands.
- Following are some of the important options with `sed`.

- `sed ''` prints the input stream as is
- `sed '='` prints line numbers before every line in the input.txt. `-e '$='` prints the line number of the last line, thus effectively printing the number of lines in the file.

In the following cases, use `sed` with `-n` option (whenever option `p` is used)

- `sed '5p'` prints only the 5th line.
- `sed '5!p'` prints all except 5th line.
- `sed '5,8p'` prints only lines 5 to 8.
- `sed '5,8{=;p}'` prints line numbers in addition to printing the lines 5 to 8.
- `sed '1~2p'` prints first, third, fifth... lines.
- `sed '1~2!p'` prints all except first, third, fifth... lines.
- `sed '/<pattern>/p'` prints all lines that match pattern. If the pattern is stored in `$str`, use it as `sed "/$str/p"`. Note that you must use double-quotes so that `$str` is expanded.
- `sed '/<pattern>/!p'` prints all lines except those that match pattern. If the pattern is stored in `$str`, use it as `sed ""/$str"/!p'`

Here are some examples of substituting using regex patterns.

- `sed 's/<pattern>/<replace>/g'` replace all matches on all lines.
- `sed 's/<pattern>/<replace>/2'` replace only second match on each line. If the line has only one match, nothing will get replaced.
- `sed 's/<pattern>/<replace>/2g'` replace all matches starting with the second one, on each line. If the line has only one match, nothing will get replaced.
- `sed '2s/<pattern>/<replace>/g'` replace all matches only on line #2.
- `sed '9,$s/<pattern>/<replace>/g'` replace all matches from 9th till the last line.
- `sed -E 3,9s/^L[[:digit:]] //g'` replaces all matches of L followed by one or more digits, in 3rd to 9th line.
- `sed -E '1~2s/^L[[:digit:]] //g'` replaces all matches of L followed by one or more digits, in every alternate line starting with the first line.
- `sed -E '3,/shore/s/^L[[:digit:]] //g'` replaces all matches of L followed by one or more digits, in 3rd to the line that contains word 'shore'.

Here are some additional options with the `sed`

- `sed '1i _____'` inserts a line before the 1st line. Use `1a` instead of `1i` to append a line after the 1st line.
- `sed '/[bB]reak$/a #####'` appends a line `#####` after every line that ends with the words `break` or `Break`.

- `sed '1~3c _____ removed _____'` changes every 4th line starting with the 1st line to "_____ removed _____"
- `-n` switch is used to NOT perform default *action*. Thus, `sed -n -e "5p" input.txt` prints only the 5th line in *input.txt*. Without `-n`, the command would print all lines in *input.txt*, and 5th line twice.
- `-i` switch is used to write the changes back to the input file. Thus, `sed -i 's/<pattern>/<replace>/g' input.txt` will replace all matches on all lines of *input.txt*, and will overwrite it with new contents.
- `sed -f script.sed input.txt` will run all commands contained by *script.sed* on *input.txt*.
- `echo "hello world" | sed 's/\b\(. \)/\U\1/g'` converts the first character of each word in the input string to upper case, and results in "Hello World"

Week7

Week8

- 1024 – 49151 are registered ports.
- *ifconfig* displays and manipulates route and network interfaces.
- *ping* checks connectivity between two nodes.
- *dig* is used to query DNS related information
- *traceroute* is a network troubleshooting utility.
- *wget* downloads a file from internet.
- *netstat* gives information on open network ports on the machine.
- IP address is a 32-bit non-negative integer; Port number is a 16-bit non-negative integer.
- *nslookup* retrieves information about a domain name such as its server and IP address.
- RAID configurations:
 - RAID-0: Also known as “disk striping,” it optimizes the speed of hard drives by writing data on multiple drives simultaneously or sequentially, improving read and write speeds
 - RAID-1: Known as “disk mirroring,” it duplicates data and stores a copy on each drive, ensuring data won’t be lost if a drive fails. It can handle one disk failure without losing data.
 - RAID-4: It uses block-level data striping with a dedicated disk for storing parity bits, providing good performance of random reads. It can handle one disk failure without losing data.
 - RAID-5: A combination of striping and parity, it boosts system reliability and read speed by distributing files and parity information across all integrated hard drives
 - RAID-6: An enhancement of RAID-5, it uses block-level striping and distributes two parity blocks on each disk within the array, increasing data protection. It can be deployed to achieve higher read speeds.

[OPPE Reference](#). Also see [this discourse post](#) for some context.