Week 1 Notes

Terminal Emulators

- Terminal
- Konsole
- xterm
- guake

Command Prompt

- `username@hostname:~$`
  - `~$` is the path

Commands and Flags

- `uname`
  - prints the name, version and other details about the current machine and the operating system running on it
  - the -a displays hidden files that have a dot in front of them
- `pwd`
  - Present Working Directory
- `ls - a` : all . displays hidden files `- l` : use a long listing format `- i` : print index number of each file (inode) `- s` : shows blocks occupied by each file `- 1` : each file name on a separate line
  - output of `ls -l` : `drwxr-xr-x 5 ckg ckg 12288 Nov 25 10:00 Documents` (`d` is file type ; `rwxr-xr-x` owner,group,others permissions ; `5` no of hard links ; `ckg` is owner ; `ckg` is group ; last modified time stamp ; filename)
  - `ls F*` gives a list of all files starting with F
- `rm`
  - remove a file
  - `rm -i` prompts before every removal (it can be set using `alias rm="rm -i"`)
  - works only with write permission
  - use `-d` for removing directories
  - `rm -r mydirectory`
- `mv`
  - move , rename
  - `mv file1 ..` (moves file to parent dir)
  - `mv file1 file1a` (renames file1 to file1a)
- `ps`
  - currently running processes
- `clear`
  - or ctrl+l
- `exit`
  - or ctrl+d
- `man`
  - get help on any command in linux. eg : man ls
  - man sections (1 to 9) eg : man 1 ls

    - 1 - Executable programs or shell commands
    - 2 - System calls provided by Kernel
    - 3 - Library calls
    - 4 - Special files usually found in /dev
    - 5 - File formats and conversions
    - 6 - Games
    - 7 - Misc : macro packages and conventions
    - 8 - System admin commands
    - 9 - Kernel routines
- `cd`
    - change directory eg cd .. - goes to parent directory
    - `cd` without any arguments will take you to the home directory
    - `cd /` takes you to the root folder
    - `cd -` takes you to previous directory
    - `cd ~` takes you to home directory
- `cp`
    - copy command : cp file1 file2
- `date`
    - date and time
    - `date -R` gives in RFC 5322 standard (used for email communications)
- `cal`
    - calendar of a month
    - eg : `cal aug 1947`
    - `ncal` gives calendar in flipped orientation
- `free`
    - memory statistics
    - use `h` flag to make it human readable
- `groups`
    - groups to which a user belongs
- `file`
    - what type of file
    - `-f` allows you to pass a file in which file names are separated by lines (ls -1 > files.txt; file -f files.txt)
    - `file *` will give a list of file name and types directly
- `mkdir`
    - create a directory
    - default permissions (umask)
- `touch`
    - used to change the last modified timestamp of a file
    - also used to create empty files
- `chmod`
    - `chmod 777 file.txt`
    - `chmod g-w file.txt` (removes write permissions from the group)
    - `chmod o-x file.txt` (removes executable permission from others)
    - `chmod u-r file.txt` (removes read permission from owner)
- `whoami`

    - prints username
- `less`
    - allows you to read a file page by page
- `ln`
    - used to create a hard link or a symbolic link (symlink) to an existing file or directory
    - `s` flag is used to create a soft link
    - usage : `ln file1 file2` ; `ln -s file1 file2`
- `cat`
    - stands for concatinate
    - allows you to view the contents of a single file or multiple files (gets concatinated)

File types

- output of `ls -l` : `drwxrwxrwx or lr-x--x--x` (l indicates symbolic link and d indicates directory)
- `-` Regular file
- `d` Directory
- `l` Symbolic link
- `c` Character file (usually found in /dev ; typically the terminal)
- `b` Block file (usually found in /dev ; typically the hard disk)
- `s` Socket file
- `p` named pipe

Viewing and Adding to files

- `cat` - to view the contents of a file
- writing to a file : `>` eg : `echo "Hello world" > test.txt`
- appending to a file : `>>` eg : `echo "Helo world" >> test.txt`

Hard links and Soft links

- inode - An entry in the filesystem table about the location in the storage media
- hard link points to the same inode
- soft link points to a hard link
- hard link must be on the same partition while soft link can point to a file at a totally different geographical location.
- inode is metadata for the file . eg : size ,permissions,blocks etc.
- `ls -i <name>`
- `ln` and `ln -s` is used for creating hard links and soft links
- inode is unique for every file : if there are multiple entries of inode then it means that they are all hard links
    - if there is a dir level1 with inode = 18874686
    - when you cd into that dir . will also have inode = 18874686
    - if i make a dir level2 inside level1 and then cd into level2 .. will have inode = 18874686 (no of hard links will increase by 1)
    - as number of sub directories increases the number of hardlinks also keeps increasing
- users cannot create hard links for directories (level1 to level2 and level2 to level1 will create a back and forth)

Permissions

- Files and directories do not inherit the parent directory permissions
- `rwxrwxrwx` (777)
    - 7 rwx
    - 6 rw-
    - 5 r-x
    - 4 r--
    - 3 -wx
    - 2 -w-
    - 1 --x
- rwx rwx rwx : Owner Group Others
- only owners can change permissions of a file
- Execute permission is required on a directory to cd into it (Even ls and tocuh to a dir will not work)
- If you want to access a file, all its parent direcories should have **x** permission. This works even without r and w permissions if you know the path.
- r and w permissions along with x is required to ls a directory or touch a file into a directory
- Removing a file works only if it has write permission

Linux Virtual Machine

- **ISO**
    - image of Linux OS (Ubuntu 20.04 LTS for x86_64 platform)
- **Hypervisor**
    - (eg: Oracle VirtualBox or VMWare Workstation Player)
    - A Hypervisor creates and runs virtual machines
    - It allows running multiple operating systems while sharing hardware resources

Command Line Environments

- Cloud - replit and cocalc
- Phone - Termux by Fredrick Fornwall

File System of Linux OS

- Filesystem Hirearchy Standard FHS 3.0 (June 03, 2015) (refspecs.linuxfoundation.org/fhs.shtml)
- `/` is root directory and field separator or delimiter for sub-directories
- `.` references the current directory (. is a special file in every directory)
- `..` references the parent directory (.. is a special file in every directory)
- Path for traversal can be absolute or relative
- `boot` directory is where the kernel is located
- `/usr/bin` contains commands that we will use
- `/bin` - essential command binaries
- `/boot` static files of the bootloader

- `/dev` device files (different character in long format of file listing 'c' instead of 'l' or 'd'. 'c' indicates character file - means you can read from it character by character. if first character is 'b' they are block devices typically hdds - the block devices are made available as files.)
- `/etc` Host specific system configuration (.conf files)
- `/lib` Essential shared libraries and kernel modules (Typically contain files with version number at the end)
- `/media` mount points for removable devices
- `/mnt` mount points
- `/opt` add on application software packages
- `/run` Data relevant to running processes
- `/sbin` essential system binaries
- `/srv` data for services
- `/tmp` temporary files (normally flushed when system is rebooted)
- `/usr` secondary hierarchy
    - `/usr/bin` : user commands
    - `/usr/lib` : libraries
    - `/usr/local` : local hierarchy
    - `/usr/sbin` : non vital system binaries
    - `/usr/share` : architecture dependent data
    - `/usr/include` : header files included by c programs
    - `/usr/src` : source code
- `/var` variable data (`/var/log` contains logs for various services)
    - `/var/cache` : Application cache data
    - `/var/lib` : Variable state informtion
    - `/var/local` : variable data for /usr/local
    - `/var/lock` : lock files
    - `/var/log` : log files and directories
    - `/var/run` : data relevant to running processes
    - `/var/tmp` : temporary files preserved between reboots

	Shareable	**Unsharable**
static	`/usr and /opt`	`/etc` and `/boot`
variable	`/var/mail`	`/var/run` and `/var/lock`

Week 2 Notes

- Multiple uses of / is as good as one
  - ie : `cd usr//////bin` will take you to `usr/bin`
- The root folder / is its own parent
  - ie : if you do `cd ..` within the root directory you stay in the same directory.
- Options / Flags can be written in multiple combinations
  - `ls -l level1 -di`
  - `ls -d level1 -il`
  - `ls level1 -ldi`
  - `ls -ldi level1`
- long formats for options are also available
- `ls -a` is equivalent to `ls --all`

Commands

- `ls`
  - R flag lists all subdirectories recursively
  - Passing directory name to ls shows what is within that directory. ie : ls -l level1
  - d flag displays details of a folder without traversing inside it. it : ls -ld level1
  -
- `ll`
  - a shortcut for the ls -la command
- `which`
  - which *command* will show the location of the command
  - `which less` will show usr/bin/less
- `whatis`
  - gives a brief description of the command
- `alias`
  - give a nickname to a frequently used command
  - usage : `alias ll = 'ls -l'`
  - Just typing alias will show a list of aliases
  - `alias date = 'date -R'`
  - If the command is executed by typing the whole path eg : `/usr/bin/date` the alias is not invoked. (`cd /usr/bin` and `./date`)
  - An alias can be escaped by prefixing a \ ie: `\date`
- `unalias`
  - used to remove an alias
- `rmdir`
  - removes an empty directory
- `ps`
  - displays current processes
  - `ps --forest` - which process has launched which child process.
  - `ps -f` - displays parent process id
  - `ps -ef` - all the processes running in the operating system now
  - PID is the process ID , PPID is the parent process ID.

    - PID 1 is `/sbin/init`
- `bc` - bench calculator
    - exit using `Ctrl`+`D`

Commands to know contents of a text file

- `less`
    - displays the content in one screen
    - ls -l /usr/bin/less shows that the command takes 180KB
- `wc`
    - prints newline,word and byte counts for the file
    - the -l flag shows just the number of lines
- `head`
    - head profile displays the first ten lines
    - use -n flag to specify the number of lines
- `tail`
    - tail profile displays the last ten lines
    - use -n flag to specify number of lines to be displayed
- `cat`
    - in /etc , cat profile would just dump contents on the screen without any further prompts.
    - disadvantages : cant move back and forth to view page by page, can't come out half way through.
    - if the file is very long cat is not the best way to look at the content.
- `more`
    - similar to less. Allows page by page viewing
    - ls -l /usr/bin/more shows that the command takes 43KB

Knowing more commands

- `man`
- `which`
- `apropos`
    - For a keyword it shows you all the commands which have that keyword in the description
    - Used to discover new commands
    - If you type `ls -l /usr/bin/apropos` you see that it is a symbolic link to whatis, but the outputs are different : Why?
    - Reason : In Linux every executable will know in what name it has been invoked - can have different behaviour depending on the name that invoked it.
    - It also has the same output as `man -k` : Searching for a keyword
- `info`
    - Allows browsing through commands using the cursor
    - Can go back using < or 'shift'+','
- `whatis`
- `help`
    - displays keywords reserved for the shell being run
- `type`
    - displays what type of command it is

    - type type shows that it is a 'shell built in' being offered from the shell and not the os
    - type ls shows that it is aliased with some option. which ls shows that it is coming from os because there is an executable available.

Multiple Arguments

- **Recap : Arguments and Options**

- Options are enhanced features of the command
- Arguments are specific names of files or directories

- Second arrument behaviour and interpratation of last argument should be seen in the man pages
- Recursion is assumed for `mv` and not `cp`

- recursion is assumed for some commands and should be explicitly stated in others
- For copy command recurssion is not assumed
- `cp dir1 dir2` need not work. dir1 has 2 files in it.
- `cp -r dir1 dir2` works - recurssion is specified explicitly.
- `mv dir1 dir3` works - it just renames the directory.

- `touch file1 file2 file3` creates all 3 files in one go with identical timestamp.

Links (Hard Links and Soft Links)

- Can determine whether a link is HL or SL by looking at the Inode numbers
    - Hard links will have the same inode numbers
    - Soft Link will have different inode numbers
    - If you delete a certain file using the `rm` command (`rm` unlinks the file from the filesystem. the data is still at the memory location. `shred` for permanant deletion)
        - Its hard link will still give you access to the original file data.
        - Its soft link will not work
- `ln -s source destination` to create symbolic link. `ln -s file1 file2`
    - file2 is a separate inode entry but it is just a shortcut to file1
    - file2 has only 1 hardlink.
- `ln source destionation` to create a hard link . `ln file1 file3`
    - file1 and file 3 have the same inode number - They are basically the same file.
    - file1 and file3 have 2 hard links when we do `ls -li`
- You can create a Soft Link `ln -s ../dir/filex fileSL` but creating a hard link using `ln ../dir/filex fileHL` will not work.
    - the first/source-file parameter is interpreted in the case of hard link creation and not in soft link creation
    - In the above example, assume that `../dir/filex` does not exist.
    - soft links useful in version control systems

File Sizes

- `ls -s`
    - file size appears in the first column
- `stat`

    - in `/usr/bin` we look at `stat znew`
    - Gives information about the size, how many blocks are being occupied
    - Here the size is little more than 4kb
    - `stat zmore` shows that it takes less than one block
- `du`
    - in `/usr/bin` we look at `du znew` or `du -h znew`
    - Gives information about the size
    - Here the size is displayed as 8.0KB since there is a block overflow.
    - This means that files that are smaller than the block size will actually take up a whole block
    - `du -h zmore` shows that it occupies one block - around 4.0K
- Role of block size
    - explained in stat and du

In-Memory File Systems

- `/proc`
    - Is an older system
    - `ls -l` will display several zero-size files, even though we can read content from them.
    - These are only a representation and not real files on the HDD.
    - `less cpuinfo` - information about the cpu
    - `cat version` - information about the OS. Also accessible using `uname -a`
    - `cat meminfo` - information about the memory - also `free -h`
    - `cat partitions` - information about the partitions - also `df -h`
    - The `kcore` file appears to take huge space - Shows maximum virtual memory that the current linux os is able to handle. 2^47 or 140 TB
- `/sys`
    - Used from Kernel v2.6 onwards, however information about various processes that are running are still stored in the /proc directory itself.
    - Much more well organised than /proc
    - eg : `sys/bus/usb/devices/1-1` points to a specific usb device.
- These are directories that are visible in the root folder. They are not on the disk but only in the memory.
- Important system information can be viewed from these directories in a read-only manner.

Shell Variables

- Makes it possible to communicate between 2 processes very efficiently. Need not write and read the filesystem.
- Security Concern : Some information that you write to the filesystem may be visible to other processes.
- Shell variables are available only within the shell or its child processes.
- `echo` prints strings to screen
    - uses space as a delimiter so multiple spaces between words are ignored. For multiple spaces, enclose the string in quotes.
    - can print a multi-line string by using double quotes and not closing it
    - ** Difference between ' and " **
    - `echo $USERNAME` and `echo "$USERNAME"` give the same result but `echo '$USERNAME'` is not interpreted to give the value of the shell variable.
    - ** Escaping to prevent interpretation **

    - `echo "username is $USERNAME and host name is \$HOSTNAME"`
    - Escaping is usefule when you want to pass on the information to a child shell, without it being interpreted by the shell launching it.
- `echo $HOME` prints values of variables
    - By convention every shell variable starts with a Dollar
- **Commonly used shell variables**
    - `$USERNAME` eg : `echo "User logged into system now is : $USERNAME"`
    - `$HOME`
    - `$HOSTNAME`
    - `$PWD`
    - `$PATH` - variable contains a list of directories which will be searched when you type a command. When ever you type a command the system scans these paths from left to right to see if the command is in the directory.
- Commands like `printenv` , `env` , `set` to see variables that are already defined
    - `printenv` displays all the shell variables defined in the shell that you are running.
    - `env` gives the same output
    - `set` displays some functions defined to interpret what you are typing on the command line.
- **Special Shell Variables**
    - `$0` : name of the shell eg `bash` or `ksh`
    - `$$` : process ID of the shell
    - `$?` : return code of previously run program
    - `$-` : flags set in the bash shell . The man page for bash shows the meaning of the flags.
- **Process Control** `echo $$`
    - use of `&` to run a job in the background
    - `fg` - bring process to foreground
    - `coproc` - run a command while also being able to use the shell
    - `jobs` - list programs running in the background
    - `top` - See programs that are hogging the CPU or memory (refreshed every second)
    - `kill` - kill process owned by you
- **Program Exit Codes** `echo $?`
    - exit code always has a value between *0 and 255*
    - 0 : Success
    - 1 : Failure
    - 2 : Misuse (insufficient permissions)
    - 126 : command cannot be executed (usually due to insufficient permissions to execute a file)
    - 127 : command not found (usually due to command typos)
    - 130 : processes killed using control+c
    - 137 : processes killed using `kill -9 <pid>`
    - If the exit code is more than 256 then the exitcode%256 will be reported as the exit code
    - `exit 0` or `exit 1` or `exit <n>` exits with exit code n
    - Used when there are command dependencies (ie: run second command only if first command completes successfully)
- **Flags set in bash** `echo $-`
    - h : locate hash commands
    - B : braceexpansion enabled
    - i : interactive mode

    - m : job control enabled (can be taken to bg or fg)
    - H : !style history substitution enabled
    - s : commands are read from stdin
    - c : commands are read from arguments

Linux Process Management

- `sleep` command to create processes
    - usage : `sleep 3` for 3 seconds
- If you have a command running in the Foreground for a long time but you need to write something else on the command line :
    - kill the process
    - suspend the process
    - run it in the background `coproc sleep 10` - When complete it gives a message.
- `coproc` is a shell keyword. No manual entry for it.
    - To learn more about a shell key word use `help coproc`
    - a running background process can be killed by process id (use : `ps --forest` to find PID and `kill -9 <pid>`)
- A command followed by an `&` means that it is being assigned to the background
    - Executing the command `fg` will bring it back to foreground
- `jobs` is a shell builtin - it lists active jobs in the current shell
- `top` shows processes taking up maximum cpu and memory. Exit gracefully by pressing Q
- `Ctrl`+`z` suspends a process.
    - Suspended processes can be seen with `jobs`
    - Can be brought back to foreground using `fg` command
- `Ctrl`+`c` kills a process
- `fg` is a shell builtin
- `bash -c "echo \$-"` creates a child shell, gets the value of `echo $-`, gives the output to the parent shell
    - `bash -c "echo \$-; ps --forest;"` - multiple commands separated by ;
    - `bash -c "echo \$$ ; ps --forest ; exit 300"` : custom error code mod 256 = 44
- `history` displays a list of commands that have been run on that computer
    - `!n` executes command line no n displayed by `history`
    - useful for repeating long commands
    - The `H` flag in bash means the history is being recorded
- Brace expansion option `B`
    - if you type `echo {a..z}` character in the ASCII sequence will be expanded.
    - In combination `echo {a..d}{a..d}` will display all possible combinations of the 2 alphabets.
    - `*` exapnds to all the files in the current directory
    - `echo D*` lists all the files begining with D.
    - Examples :
        - `mkdir {1..12}{A..E}` or `rmdir {1..12}{A..E}` or `touch {1..12}{A..E}/{1..40}`
- `;` acts as a separator between individual commands eg : `echo hello ; ls`

REPLIT CODE WITH US

Link to Replit

- `date -d "2024-04-01" +%A` - Day of the week for given date
- `file --mime-type somefile` - mime type of a given file
- `mkdir {1..12}{A..E}`
- `rmdir {1..12}{A..E}`
- `touch {1..12}{A..E}/{1..40}`
- `lscpu | grep -i "model name"| cut -d ":" -f "2"`

Week 3 Notes

Combining Commands and Files

- Executing Multiple Commands
  - `command1; command2; command3;`
    - Each command will be executed one after the other.
  - `command1 && command2`
    - command2 will be executed only if command 1 succeeds
    - If the return code is 0 it is true and if it is greater than 0 it is false
    - `ls && date -Q && wc -l /etc/profile` will display the dir listing followed by error that -Q is invalid; wc is not executed.
  - `command1 || command2`
    - command2 will not be executed if command1 succeeds
    - `ls /blah || date` will display current date after "No such file or directory"
    - `ls || date` will display just the directory listing
    - command2 is like a Plan B if command1 doesn't succeed.
  - Example `ls /blah ; date ; wc -l /etc/profile ;`
  - If we use parenthesis ie `(ls /blah ; date ; wc -l /etc/profile ;)` the command gets executed in a subshell and is returned back to the shell we are using.
  - We can use `echo $BASH_SUBSHELL` to return an integer which tells us at what level of execution we are.
    - `(echo $BASH_SUBSHELL)` will report a value of 1
    - `(ls; (date; echo $BASH_SUBSHELL))` will report a value of 2
  - Launching too many subshells could be expensive computationally.
- File Descriptors
  - Every command in linux has 3 file descriptors - `stdin` (0) , `stdout` (1), `stderr` (2).
    - `stdin` is a pointer to a stream that is coming from the keyboard or use input
    - `stdout` or `stderr` usually points to the screen where the display or output is made.
    - the three pointers are looking at only the stream of characters.
    - they can be directed to a file or a command, or the default behaviour can be left as it is.
  - Combining a command and a file
    - `command > file1`
      - `stdout` is redirected to `file1`
      - `file1` will be created if it does not exist
      - if `file1` exists, its contents will be overwritten
      - example : `ls -1 /usr/bin > file1` - displays no output on the screen because there is no error
      - `ls -1 /blah > file1` - displays an error. file1 is overwritten and is now 0 Bytes.
      - `hwinfo > hwinfo.txt`
      - trying this command in a folder where there is no w permissions will generate an error
      - The `cat` command tries to read from the provided file name if not given it tries to read from stdin (keyboard)
        - `cat > file1` will allow you to type content. The feature could be used to create text files on the command line. You can come out using the `Ctrl`+`D`

option.
        - `cat file1` displays the content of `file1`
        - `cat` takes input from the keyboard and displays it on the screen (line by line; when you press enter) - Finish by pressing `Ctrl`+`D` to signify end of file.
    - `command >> file1`
        - contents will be appended to file1
        - new file1 will be created if it does not exist.
        - Example : `date >> file2 ; wc -l /etc/profile >> file2 ; file /usr/bin/znew >> file2 ;`
        - `cat >> file1` to append text to a file from command line. Come out using `Ctrl` + `D`

Redirections

- combining command and file (continued ..)
    - (contd..)
        - `command 2> file1`
            - redirects `stderr` to `file1`
            - `file1`, if it exists, will be overwritten.
            - `file1` will be created if it does not exist.
            - Example `ls $HOME /blah 2> error.txt`
        - `command > file1 2> file2`
            - `stdout` is redirected to `file1`
            - `stderr` is redirected to `file2`
            - Contents of file1 and file2 will be overwritten.
            - The output is in one file and the errors are in another file.
            - Example : `ls $HOME /blah > output.txt 2> error.txt`
            - `ls -R /etc > output.txt 2> error.txt` - permission related errors in error.txt
        - `command < file1`
            - `stdin` is redirected - a command expecting input from the keyboard could take the input from a file.
            - Example : `wc /etc/profile` behaves similar to `wc < /etc/profile`
        - `command > file1 2>&1`
            - command output will be redirected to `file1`
            - `2>` indicates `stderr` and that is being redirected to `&1` (first stream) which is `stdout`
            - contents of `file1` will be overwritten
            - Example : `ls $ HOME /blah > file1` output alone is sent to file1. Error on screen
            - Example : `ls $ HOME /blah > file1 2>&1` output and error is sent to file1.
        - `command1 | command2` Pipe
            - `stdout` output of command 1 is sent to `stdin` of command2 as input
            - Example `ls /usr/bin | wc -l`
        - `command1 | command2 > file1`
            - command1 and command2 are combined and the `stdout` of command2 is sent to `file1`. Errors are still shown on the screen.
            - Example `ls /usr/bin | wc -l > file1` - file1 has the number of lines counted by wc
        - `command > file1 2> /dev/null`

        - `/dev/null` file - A sink for output to be discarded. Like a "black hole"
        - We normally don't do anything with the `/dev` folder as there are sensitive system files there.
        - If you are confident that the script is running well and you do not want to display any error on the screen, you can redirect the `stderr` to `/dev/null`
        - `stderr` is redirected to `/dev/null`
        - Example : `ls $HOME /blah > file1 2> /dev/null`
        - Example : `ls -R /etc > file1 2> /dev/null` - file1 contains the output except errors
    - `command1 | tee file1`
        - Used in sitiations where you want to have a copy of the output in a file as well as on the screen.
        - The `tee` command reads from `stdin` and writes to `stdout` and file/s.
        - Example : `ls $HOME | tee file1` also `ls $HOME | tee file1 file2` for creating multiple copies
        - `diff file1 file2` comapares files line by line
            - no output if the files are identical
        - Example : `ls $HOME /blah | tee file1 file2 | wc -l` - Here `tee` keeps copy of output in a file and also sends output to `wc -l` for further processing.
        - Example : `ls $HOME /blah 2> /dev/null | tee file1 file2 | wc -l` to supress errors. Note location of `2>` is since the error is generated there.

Shell Variables - Part 1

- Creation, inspection, modification, lists
- Creating a variable
    - `myvar="value string"`
        - `myvar` can't start with a number, but you can mix alphanumeric and `_`
        - No space around the `=`
        - `"value string"` is the number,string or `command`. Output of a command can be assigned to myvar by enclosing the command in back-ticks.
- Exporting a variable
    - `export myvar="value string"` or
    - `myvar="value string" ; export myvar`
    - This makes the value of the variable available to a shell that is spawned by the current shell.
- Using variable values
    - `echo $myvar`
    - `echo ${myvar}`
        - can manipulate the value of the variable by inserting some commands within the braces.
    - `echo "${myvar}_something"`
- Removing a variable
    - `unset myvar`
    - Removing value of a variable `myvar=`
- Test is a variable is set
    - `[[ -v myvar ]] ; echo $?`
        - 0 : success (variable myvar is set)
        - 1 : failure (variable myvar is not set)

    - `[[ -z ${myvar+x} ]] ; echo $?` (the `x` can be any string)
        - 0 : success (variable myvar is not set)
        - 1 : failure (variable myvar is set)
- Substitute default value
    - If the variable `myvar` is not set, use "default" as its default value
    - `echo ${myvar:-"default"}`
        - if `myvar` is set display its value
        - else display "default"
- Set default value
    - If the variable `myvar` is not set then set "default" as its value
    - `echo ${myvar:="default"}`
        - if `myvar` is set display its value
        - else set "default" as its value and display its new value
- Reset value if variable is set
    - If the variable `myvar` is set, then set "default" as its value
    - `echo ${myvar:+"default"}`
        - if `myvar` is set, then set "default" as its value and display the new value
        - else display null
- List of variable names
    - `echo ${!H*}`
        - displays the list of names of shell variables that start with H
- Length of string value
    - `echo ${#myvar}`
        - Display length of the string value of the variable `myvar`
        - if `myvar` is not set then display 0
- Slice of a string value
    - `echo ${myvar:5:4}` (5 is the offset and 4 is the slice length)
        - Display 4 characters of the string value of the variable `myvar` after skipping first 5 characters.
    - if the slice length is larget than the length of the string then only what is available in the string will be displayed.
    - the offset can also be negative. However you need to provide a *space* after the *:* to avoid confusion with the earlier usage of the `:-` symbol. The offset would come from the right hand side of the string.
- Remove matching pattern
    - `echo ${myvar#pattern}` - matches once
    - `echo ${myvar##pattern}` - matches maximum possible
    - Whatever is matching the pattern will be removed and the rest of it will be displayed on the screen.
- Keep matching pattern
    - `echo ${myvar%pattern}` - matches once
    - `echo ${myvar%%pattern}` - matches maximum possible
- Replace matching pattern
    - `echo ${myvar/pattern/string}` - match once and replace with string
    - `echo ${myvar//pattern/string}` - match max possible and replace with string
- Replace matching pattern by location

    - `echo ${myvar/#pattern/string}` - match at begining and replace with string
    - `echo ${myvar/%pattern/string}` - match at the end and replace with string
- Changing case
    - `echo ${myvar,}` - Change the first character to lower case.
    - `echo ${myvar,,}` - Change all characters to lower case.
    - `echo ${myvar^}` - Change first character to uppercase
    - `echo ${myvar^^}` - Change all characters to upper case
    - The original value of the variable is not changed. Only the display will be modified as the trigger commands are within braces.
- Restricting value types
    - `declare -i myvar` - only integers assigned
    - `declare -l myvar` - Only lower case chars assigned
    - `declare -u myvar` - Only upper case chars assigned
    - `declare -r myvar` - Variable is read only
    - Once a variable is set as read only you may have to restart the bash to be able to set it
- Removing restrictions
    - `declare +i myvar` - integer restriction removed
    - `declare +l myvar` - lower case chars restriction removed
    - `declare +u myvar` - upper case chars restriction removed
    - `declare +r myvar` - *Can't do once it is read-only*
- Indexed arrays
    - `declare -a arr`
        - Declare `arr` as an indexed array
    - `$arr[0]=”value”`
        - Set value of element with index 0 in the array
    - `echo ${arr[0]}`
        - Value of element with index 0 in the array
    - `echo ${#arr[@]}`
        - Number of elements in the array. The `@` symbol is a wild character to run through all the elements in the array
    - `echo ${!arr[@]}`
        - Display all indices used
    - `echo ${arr[@]}`
        - Display values of all elements of the array
    - `unset ‘arr[2]’`
        - Delete element with index 2 in the array
    - `arr+=(“value”)`
        - Append an element with a value to the end of the array
- Associative arrays
    - `declare -A hash`
        - declare `hash` as an associative array
    - `$hash[“a”]=”value”`
        - set the value of element with index a in the array
    - `echo ${hash[“a”]}`
        - value of element with index a in the array
    - `echo ${#hash[@]}`

        - number of elements in the array
    - `echo ${!hash[@]}`
        - display all indices used
    - `echo ${hash[@]}`
        - display values of all elements of the array
    - `unset 'hash["a"]'`
        - delete an element with index a in the array
    - Can do everything in the indexed array except append because there is nothing called the end of the array as there is no sequence for the elements of a hash
- Examples
    - `true` always returns exit code 0
    - `false` always returns exit code 1 (Check with `echo $?`)
    - To check whether a variable is present
        - `[[ -v myvar ]] ; echo $?` returns 1 if the variable is not present in the memory
        - `[[ -z ${myvar+x} ]] ; echo $?` returns 0 if variable is not present and 1 if it is present. `x` is a string that will be used as a replacement if the variable was not present.
    - Use of Braces
        - `myvar=FileName`
        - `echo $myvar`
        - `echo "$myvar.txt"` prints FileName.txt
        - `echo "$myvar_txt"` does not print anything as the variable `myvar_txt` does not exist
        - `echo "${myvar}_txt"` prints Filename_txt
        - Braces are useful in stating clearly the name of the variable.
        - Can also be used outside quotes `echo ${myvar}`
    - Does the variable we have created get passed on to the shell or any other program created within the shell
        - `myvar=3.14 ; echo $myvar`
        - `bash` one more level of bash
        - `ps --forest` to show that we are one level below
        - `echo $myvar` not present
        - Use `export myvar=3.14` to ensure this variable is available to all spawned sub shells.
        - Change value of variable within the child shell
        - modification of value is not reflected in the value of the variable in the parent shell
        - even if you do export of the variable within the child shell it will not change the value within the parent shell.
    - Use of back-ticks
        - `mydate=`date`` value of mydate will be output of date.
        - `mydate=`echo Sunday that is today` ; echo $mydate`
    - Manipulations for variables within the shell environment
        - We would like to have echo display a default value if variable is not available
        - `echo ${myvar:-hello}` the `-` indicates if the value is not present what is the display value
        - `echo ${myvar:-"myvar is not set"}`
        - Set the value if it was not set already
        - `echo ${myvar:=hello}` if absent / not set then set it to the value after `=`
        - If it is present it will not change

        - `echo ${myvar:?"myvar is not set"}` displays a little more information and a debug message. `bash: myvar: myvar is not set`
        - Unset the value of a variable using `unset myvar`
        - `echo ${myvar:+HELLO}` displays the message if the variable is present
    - Inspecting all the variables in the shell environment
        - `printenv`
        - `env`
        - `echo ${!H*}` displays the names of variables begining with 'H' - `!` indicates names of the variables instead of value.
    - Counting characters
        - `mydate=`date`` stores the output of the `date` command in `mydate`
        - `echo ${#mydate}` prints the length of the value present in mydate.
        - length of a non-existing variable is zero
    - Features of using colon : within braces {}
        - Extracting part of a string from the value of a particular variable.
            - `echo ${mydate:6:10}`
            - `echo ${myvar:3:3}` will print `def` for `myvar=abcdefg` ie: 3 characters after the offset (position 3)
        - Using negative offset
            - `echo ${myvar: -3:3}` and `echo ${myvar: -3:4}` will print `efg` for `myvar=abcdefg`
                - note `-` is to be preceeded with a blank to avoid confusion
                - asking for more characters, will print just what is available
            - `echo ${myvar: -3:2}` will print `ef` for `myvar=abcdefg`
        - Extracting a portion of the date
            - Output of `date` is `Tuesday 25 January 2022 09:10:20 PM IST`
            - Output of `date +"%d %B %Y"` is `25 January 2022`
            - if `mydate=`date`` then `echo ${mydate:8:16}` will also print `25 January 2022`
        - Extracting patterns from a string
            - `myvar=filename.txt.jpg`
            - `echo ${myvar#*.}` minimal matching displays `txt.jpg`
            - `myvar=filename.somethingelse.jpeg`
            - `echo ${myvar##*.}` maximal matching displays `jpeg`
            - `echo ${myvar%*.}` displays filename.somethingelse
                - the `%` is used to indicate what has not been matched. (minimal)
            - `echo ${myvar%%*.}` displays filename
                - the `%` is used to indicate what has not been matched. (maximal)
            - Can be combined `echo ${myvar%%.*}.${myvar##*.}` to get `filename.jpeg`
        - Replacing what has been matched
        - Pattern matching in Linux usually goes with a pair of forward slashes.
        - Convert all `e` to `E` in a string
            - `echo ${myvar/e/E}` replaces only the first occurance of `e`
            - `echo ${myvar//e/E}` replaces all occurances of `e`
        - Replace characters at the begining of a string
            - `echo ${myvar/#f/F}` replaces the occurance of `f` in the begining of the string with `F`. The `#` indicates the begining of the string

- Replace characters at the end of a string
  - `echo ${myvar/%g/G}` replaces the occurance of `g` at the end of the string with `G`. The `%` indicates the end of the string.
- Replace jpeg with jpg, only if it is at the end of a string
  - `echo ${myvar/%jpeg/jpg}`
- Modifying and storing it in a variable
  - `myvar1=`echo ${myvar//jpeg/jpg}``
- Generic command to remove day from date
  - `echo ${mydate#*day}`
- Upper case to lower case and vice-versa
  - `echo ${mydate,}` changes first character to lowercase
  - `echo ${mydate,,}` converts all characters to lowercase
  - `echo ${mydate^}` changes first character to uppercase
  - `echo ${mydate^^}` changes all characters to uppercase
- Restricting values that can be assigned to shell variables using `declare`
  - `declare` is a shell builtin
  - `+` to **unset** a restriction and `-` to **set** it (Note : counterintutuve)
  - `-a` for indexed arrays (need not be ordered indexes)
  - `-A` for associative arrays (dictionaries)
  - `-i` for integers
  - `-u` for uppercase conversion on assignment
  - Integer restriction
    - `declare -i mynum`
    - `mynum=10` will assign 10 to mynum
    - `mynum=hello` will assign 0 to mynum
  - lowercase restriciton
    - `declare -l myvar`
    - `myvar=hello` assigns hello to myvar
    - `myvar=BELLOW` converts BELLOW to lowercase and assigns it to myvar.
  - removing a restriction
    - `declare +l myvar`
    - the value is still contained after removing the restriction but you can now store upper case characters as well
  - declaring a read-only variable
    - `declare -r myvar`
    - once a variable has been set as read only, you cannot change its value and you cannot remove the read-only restriction using `+r`
    - `declare +r myvar` gives the error `bash: declare: myvar: readonly variable`
- Arrays
  - `declare -a arr`
  - `arr[0]=Sunday`
  - `arr[1]=Monday`
  - `echo ${arr[0]}`
  - `echo ${arr[1]}`
  - `echo ${#arr[@]}` gives number of elements in the array

    - `echo ${arr[@]}` displays all values
    - `echo ${!arr[@]}` displays the indices`
    - You can have any index without filling up intermediate indices. Indices are not necessarily contiguous.
    - `arr[100]=Friday` is also valid
    - Removing an element from an array = `unset 'arr[100]'`
    - Appending to an array `arr+=(Tuesday)`
    - Populating an array in one go `arr=(Sunday Monday Tuesday)` . The indices are sequential
- Associative Arrays / Hashes
    - `declare -A hash`
    - `hash[0]="Amal"`
    - `hash["mm12b001"]="Charlie"`
    - `echo ${!hash[@]}` to get indices
    - `echo ${hash["mm12b001"]}`
- File names in a shell variable
    - `myfiles=(`ls`)`
    - `echo ${myfiles[@]}`

Week 4 Notes

Software Management

- Using Package Management Systems
    - Tools for installing, updating, removing and managing software
    - Install new / updated software across network
    - Package - File look up, both ways
        - Which files are given by a particular package and which package contains a given file
    - Database of packages on the system including versions (compatibility and requirements)
    - Dependency checking
    - Signature verification tools (to check authenticity of source of the software)
    - Tools for building packages (to build packages from soure code - particularly true for kernel modules)
- Package types
    - Package
        - RPM
            - Red Hat
                - CentOS
                - Fedora
                - Oracle Linux
            - SUSE Enterprise Linux
                - OpenSUSE
        - DEB
            - Debian
                - Ubuntu
                    - Mint
                - Knoppix
- Commands
    - `lsb_release -a` to find version of Operating System
    - When searching for packages for this version of the OS you can search by OS code name eg: `focal`
- Architectures
    - amd64 | x86_64
    - i386 | x86
    - arm (RISC5 Sakthi)
    - ppc64el | OpenPOWER
    - all | noarch |src (not tied to any architecture)
- Commands
    - `uname -a` gives the kernel version and the type of architecture.

- Tools
    - Package Type
        - RPM
            - Yellowdog Updater Modifier (yum)
                - Red Hat Package Manager (rpm)
                - Dandified YUM (dnf)
        - DEB
            - synaptic (GUI)
            - aptitude (Command Line)
                - Advanced Package Tool (apt)
                    - dpkg
                        - dpkg-deb
- Package managemet in Ubuntu using `apt`
    - Inquiring package db
        - Search packages for a keyword
            - `apt-cache search keyword`
        - List all packages
            - `apt-cache pkgnames`
            - `apt-cache pkgnames | sort | less` for page by page sorted display
            - `apt-cache pkgnames nm` for all packages starting with nm
        - Display package records of a package
            - `apt-cache show -a package`
- Package Names
    - Package
        - RPM
            - package-version-release.architecture.rpm
        - DEB
            - package_version-revision_architecture.deb
            - eg : `pool/universe/n/nmap/nmap_7.80+dfsg1-2build1_amd64.deb`
- Package Priorities
    - required : essential to proper functioning of the system
    - important : provides functionality that enables the system to run well
    - standard : included in a standard system installation
    - optional : can omit if you do not have enough storage
    - extra : could conflict with packages with higher priority, has specialized requirements, install only if needed.
    - Priority is displayed as `extra` in the output of `apt-cache show nmap` or `apt-cache show wget` for example.
- Package Sections
    - Package Sections for Ubuntu focal
    - `apt-cache show fortunes` shows `Section : universe/games`

- Checksums
    - For a small change in the original file the checksum is very different. This is useful to chack if the original file has been tampered or not.
    - Can be used to verify that nothing has gone wrong to the contents of the file while downloading.
    - md5sum
        - 128 bit string
        - `md5sum filename`
    - SHA1
        - 160 bit string
        - `sha1sum filename`
    - SHA256
        - 256 bit string
        - `sha256sum filename`

4.2

- Who can install packages in Linux OS ?
    - administrators
    - sudoers in the case of Ubuntu
    - Only sudoers can install/upgrade/remove packages
    - a sudo command can be executed by those who are listed in `/etc/sudoers`
    - Command `sudo cat /etc/sudoers` . If the current `$USER` is not in the sudoers file the incident will be reported.
    - In the file the users listed under `# User privilege specification` have sudo permission.
    - sudo attempts and authentication failures get recorded in `/var/log/auth.log`. View using `sudo tail -n 100 /var/log/auth.log`
- When installing a package the system knows the website/server from which the packages have to be downloaded
    - This information is stored in the folder `/etc/apt`
    - Uncommented lines in the file `sources.list` have the debian/ubuntu sources
    - A directory `sources.list.d` stores sources for third party software. Allows `apt update` to know new versions to download from repositories stored in these files
    - Synchronize package overview files - `sudo apt-get update` fetches updates and keeps them in cache
    - Upgrade all installed packages - `sudo apt-get upgrade` upgrades the packages. It lists how many updates are going to be affected and how much data is going to be downloaded.
    - `sudo apt autoremove` to remove unused packages that were earlier installed to satisfy a particular dependency but are not needed now.
    - Install a package - `sudo apt-get install packagename`
    - `sudo apt-get remove packagename` to remove a particular package
    - `sudo apt-get reinstall packagename` to fix problems caused by accedential file deletions.
    - Clean local repository of retreived package files - `apt-get clean`
    - Purge package files from the system - `apt-get purge package`
- Package management in Ubuntu using `dpkg`
    - Allows installation directly from a `.deb` file. Package management at a lower level.
    - `/var/lib/dpkg` has some information about the packages

        - Files - `arch`,`available`,`status`
            - `cat arch` displays the architectures for which packages have been installed on the system - `amd64`,`i386`
            - `less available` displays list of packages with info.
            - `less status` displays if a particular package is installed or not
        - Folder - `info`
            - contains a set of files for each of the packages that have been installed
            - `ls wget*` will give files with information about `wget`
                - `more wget.conffiles`gives location of configuration file
                - `more wget.list` displays list of files that would get installed on the system with the package
                - `more wget.md4sums` displays the listof md5sums of the installed files. (Used to catch tampering)
- Using `dpkg`
    - List all packages whose names match the pattern
        - `dpkg -l pattern`
    - List installed files that came from packages
        - `dpkg -L package`
    - Display/Report the status of packages
        - `dpkg -s package`
    - Search installed packages for a file
        - `dpkg -S pattern`
        - eg : `dpkg -S /usr/bin/perl` shows the package from which the executable has come. ie : `perl-base`
    - To query the dpkg database about all the packages - `dpkg-query`
        - Example `dpkg-query -W -f='${Section} ${binary:Package}\n' | sort | less`
        - Example where output is filtered `dpkg-query -W -f='${Section} ${binary:Package}\n' | grep shells`
- Installing a deb package
    - `dpkg -i package_version-revision_architecture.deb`
    - not a good idea since it may have some dependencies that will have to be taken care of manually
    - Do not download deb files from unknown sources and install it on the system
    - By default use package management pointing to a reliable repository
    - Uninstalling packages using `dpkg` is NOT recommended. You may be removing a package that is required by many other packages.
- When compatibility issues cannot be resloved one can use `snap` or `docker` as alternatives when you are unable to install a particular version of a package.

4.3

Pattern Matching

- Regular Expressions `regex` and `grep` commands
    - POSIX standard
        - IEEE 1003.1-2001 IEEE Standard for IEEE Information Technology – Portable Operating System Interface (POSIX(TM))

        - Refer
    - POSIX defines regular expressions to be of 2 different types - Basic and Extended.
- Regex
    - regex is a pattern template to filter text
    - BRE: POSIX Basic Regular Expression engine
    - ERE: POSIX Extended Regular Expression engine
- Why learn regex?
    - PRocess some input from the user or perform some string operations.
    - Languages: Java, Perl, Python, Ruby, ...
    - Tools: grep, sed, awk, ...
    - Applications: MySQL, PostgreSQL, ...
- Usage
    - `grep 'pattern' filename` - to operate on every line in the file
    - `command | grep 'pattern'`
        - the `grep` command operates line after line. A common feature in many utilities in linux.
        - enclose pattern in single quotes
    - Default engine: BRE
    - Switch to use ERE in 2 ways:
        - `egrep 'pattern' filename`
        - `grep -E 'pattern' filename`

- Special characters (BRE & ERE)

Character	Description
`.`	Any single character except null or newline
`*`	Zero or more of the preceding character / expression
`[]`	Any of the enclosed characters; hyphen (-) indicates character range
`^`	Anchor for beginning of line or negation of enclosed characters
`$`	Anchor for end of line
`\`	Escape special characters

- Special characters (BRE)

Character	Description
`\{n,m\}`	Range of occurances of preceding pattern at least n and utmost m times
`\( \)`	Grouping of regular expressions

- Special characters (ERE)

Character	Description
`{n,m}`	Range of occurances of preceding pattern at least n and utmost m times

Character	Description
`()`	Grouping of regular expressions
`+`	One or more of preceding character / expression
`?`	Zero or one of preceding character / expression
`|`	Logical OR over the patterns

- Character Classes

Class	Description
`[[:print:]]`	Printable
`[[:alnum:]]`	Alphanumeric
`[[:alpha:]]`	Alphabetic
`[[:lower:]]`	Lower case
`[[:upper:]]`	Upper case
`[[:digit:]]`	Decimal digits
`[[:blank:]]`	Space / Tab
`[[:space:]]`	Whitespace
`[[:punct:]]`	Punctuation
`[[:xdigit:]]`	Hexadecimal
`[[:graph:]]`	Non-space
`[[:cntrl:]]`	Control characters

- Backreferences
    - `\1` through `\9`
    - `\n` matches whatever was matched by nth earlier paranthesized subexpression
    - A line with two occurances of hello will be matched using: `\(hello\).*\1`
- BRE operator precedence

Highest to Lowest
[..] [==] [::] char collation
\metachar
☐ Bracket expansion
( ) \n subexpresions and backreferences
* { } Repetition of preceding single char regex

Highest to Lowest
Concatenation
^ $ anchors

- ERE operator precedence

Highest to Lowest
[..] [==] [::] char collation
\metachar
☐ Bracket expansion
( ) grouping
* + ? { } Repetition of preceding regex
Concatenation
^ $ anchors
| alternation

- Examples using grep
  - Example File names.txt (Containing Names/Roll-No)
  - Basic use
    - `grep 'Raman' names.txt` matches line with Raman Singh
    - `cat names.txt | grep 'ai'` matches line with Snail
  - Usage of `.`
    - `cat names.txt | grep 'S.n'` matches lines with Singh and Sankaran
  - Usage of `$`
    - `cat names.txt | grep '.am$'` matches lines that end with xam
  - Escaping a `.`
    - `cat names.txt | grep '\.'` matches lines that have a `.`
  - Using anchors at the begining
    - `cat names.txt | grep '^M'` matches lines begining with m
  - Case insensitive matching with the `i` flag
    - `cat names.txt | grep -i '^e'` matches lines begining with e or E.
  - Word boundaries `\b`
    - `cat names.txt | grep 'am\b'` matches lines with words that end with 'am'
  - Use of square brackets `[]` to give options
    - `cat names.txt | grep 'M[ME]'` matches lines containing 'MM' or 'ME'
    - `cat names.txt | grep '\bS.*[mn]'` matches lines containing words begining with S and ending with m or n.
    - `cat names.txt | grep '[aeiou][aeiou]'` matches lines that have 2 vowels side by side

        - `cat names.txt | grep 'B90[1-4]'` matches words begining with B90 and ending with range 1-4.
        - `cat names.txt | grep 'B90[^1-4]'` matches words begining with B90 and ending with characters other than the range 1-4. A hat inside square brackets implies negation
    - Specifying occurances using escaped braces
        - `cat names.txt | grep 'M\{2\}'` matches lines which have 'MM'
        - `cat names.txt | grep 'M\{1,2\}'` matches lines which have one or 2 'M's
    - Grouping patterns that are matched using parenthesis. Repeating whatever is matched by using `\1`
        - `cat names.txt | grep '\(ma\)'` matches lines containing 'ma'
        - `cat names.txt | grep '\(ma\).*\1'` matches a pattern begining with 'ma' and ending with 'ma' eg: U'mair Ahma'd. The `\1` back-references the first parenthesis.
        - `cat names.txt | grep '\(.a\).*\1'` matches a pattern like 'Mary Ma'nickam
        - `cat names.txt | grep '\(a.\)\{3\}'` matches a pattern like S'agayam'
    - Using Extended Regular Expression Engine
        - `cat names.txt | egrep 'M+'` will match lines where M occures one or more times.
        - `cat names.txt | egrep '^M+'` will match lines where M occures one or more times at the begining of a line.
        - `cat names.txt | egrep '^M*'`
            - `cat names.txt | egrep '^M*a'` matches lines where 'M' may or may not occur followed by 'a'
            - `cat names.txt | egrep '^M.*a'` matches lines where 'M' has to occur at the begining of a line followed by any number of characters and ending with 'a'
            - Watch out for the interpretation of `*`
        - `cat names.txt | egrep '(ma)+'` 'ma' could occur one or more times.
        - `cat names.txt | egrep '(ma)*'` 'ma' could occur zero or more times.
    - Use of pipe as an alternation between 2 patterns of strings to be matched
        - `cat names.txt | egrep '(ED|ME)'` matches lines containing 'ED' or 'ME'
        - `cat names.txt | egrep '(Anu|Raman)'` matches lines containing 'Anu' or 'Raman'. Length of string on both sides of pipe need not be the same.
        - `cat names.txt | egrep '(am|an)$'` matches lines containing 'am' or 'an' at the end.

4.4

- More Examples using grep and egrep
    - Get package names that are exactly 4 characters long
        - `dpkg-query -W -f'${Section} ${binary:Package}\n' | egrep ' .{4}$'`
    - Get package names that are from the math section
        - `dpkg-query -W -f'${Section} ${binary:Package}\n' | egrep '^math'`
    - Example File chartype.txt (Containing few lines with control character)
        - control character inserted using `echo $'\cc' >> chartype.txt`
    - get lines that have an alphanumeric character at the begining of the line
        - `cat chartype.txt | grep '^[[:alnum:]]'`
    - get lines that have digits at the end of the line
        - `cat chartype.txt | grep '[[:digit:]]$'`

- get lines that have a ctrl character
  - `cat chartype.txt | grep '[[:ctrl:]]'`
  - `cat chartype.txt | grep -v '[[:ctrl:]]'` will show the reverse including the empty lines
- get lines that do not have a ctrl character
  - `cat chartype.txt | grep '[^[:ctrl:]]'` (This does not work as intended)
- get lines that have printable characters (exclude blank lines)
  - `cat chartype.txt | grep '[[:print:]]'`
- get lines that have blank space characters (exclude blank lines)
  - `cat chartype.txt | grep '[[:blank:]]'`
- `[[:graph:]]` is used to match any non space character
- To skip blank lines
  - `cat chartypes.txt | egrep -v '^$'` Here `-v` excludes and `'^$'` captures empty lines
- Identify a line with a 12 digit number
  - `egrep '[[:digit:]]{12}' patterns.txt`
- Identify a line with a 6 digit number (Use word boundaries)
  - `egrep '\b[[:digit:]]{6}\b' patterns.txt`
- Match lines containing Roll Number of the form MM22B001
  - `egrep '\b[[:alpha:]]{2}[[:digit:]]{2}[[:alpha:]][[:digit:]]{3}\b' patterns.txt`
- Match urls without the http
  - `egrep '\b[[:alnum:]]+\.[[:alnum:]]+\b' patterns.txt`
- **Trimming text**
  - top to bottom using `head` and `tail`
  - sidways or horizontal trimming of lines using `cut`
    - `cut -c 1-4 fields.txt` displays only first 4 characters. Can also use `-4` for begining to 4th place or `2-` to cut from 2nd place to end.
    - `cat fields.txt | cut -d " " -f 1` - This uses " " as a delimiter `-d` and prints only the first field `-f 1`
    - `cat fields.txt | cut -d ' ' -f 1-2` - to get both fields
    - Capture `hello world` from `1234;hello world,line 1`
      - `cat fields.txt | cut -d ';' -f 2 | cut -d "," -f 1`
      - `egrep ';.*,' fields.txt` (To trim pass the output of grep to `sed`)
    - Combining this with top to bottom trimming
      - `cat fields.txt | cut -d ';' -f 2 | cut -d "," -f 1 | head -n 2 | tail -n 1`

- Own experiments using regex
  - Get strictly alphanumeric words
    - `cat test.txt | egrep '\b([a-z]+[0-9]+|[0-9]+[a-z]+)\b'`
- REPLIT Code with Us session
  - Getting files with a specific permission pattern from a file
    - `cat lsinfo.txt | grep 'rw-r--r--' ;`
  - Get all files excluding directories in lsinfo.txt whose last modified date is in January

        - `cat lsinfo.txt | grep '^[^d].*Jan'`
    - To count the number of lines that starts with a capital letter and contains the word it (case-sensitive)
        - `cat twocities.txt | grep -c '^[[:upper:]].*\bit\b'`
    - to display all the lines that does not contain the word "we" in it
        - `cat twocities.txt | egrep -v '\bwe\b'`
    - using cut to display only the countries and its capitals of file.txt in the format Country, Capital (eg in file.txt : India, New Delhi; Asia)
        - `cat file.txt | cut -d ';' -f 1`
    - all the countries in the file file.txt sorted alphabetically by name in reverse order
        - `cat file.txt | cut -d ',' -f 1 | sort -r`
    - cut command to extract the continents (including the one white space in the beginning) of the first 5 lines of file.txt and store it in another file named continent.txt
        - `head -n 5 file.txt | cut -d ';' -f 2 > continent.txt`
    - list the names of all the c++ files in the current directory which contains a line such that the line starts with the string void main() and ends with the character {. There should be one or more spaces/tabs between the characters { and).
        - `egrep '^void[[:space:]]main\(\)[[:space:]]+{$' *.cpp | cut -d '.' -f 1`
        - `grep '^void[[:space:]]main()[[:space:]][[:space:]]*{$' *.cpp | cut -d '.' -f 1`
    - print the count of these files in the following line
        - `egrep -l '^void[[:space:]]main\(\)[[:space:]]+{$' *.cpp |tee /dev/tty | wc -l`
        - `|tee /dev/tty` is used to print the output to terminal and also pipe the output to the next command.
        - `-l` flag for `grep` and `egrep` prints the name of each input file that matches
    - command to list all the packages installed on your machine and their versions in the format Package Version in a sorted manner
        - `dpkg-query -W -f='${Package} ${Version}\n' | sort`

Week 5 Notes

Command Line Editors

- Working with text files in the terminal
- Editors
  - Line Editors (Present in almost every flavour of UNIX / GNU Linux)
    - `ed`
    - `ex` (improved version of ed)
  - Terminal Editors
    - `pico` (Came along with the pine email application)
      - `nano` (Features added to pico)
    - `vi` (most popular and complex)
    - `emacs`
  - GUI Editors
    - KDE
      - `kate`
      - `kwrite`
    - GNOME
      - `gedit`
    - sublime
    - atom (popular among github users)
    - brackets (Popular for those writing html code)
  - IDE
    - eclipse
    - Bluefish
    - NetBeans
- Features of text editors
  - Scrolling , view modes, current position in file
  - Navigation (char,word,line,pattern)
  - Insert, Replace, Delete
  - Cut-Copy-Paste
  - Search-Replace
  - Language-aware syntax highlighting
  - Key-maps, init scripts, macros
  - Plugins
  - Both `vi` and `emacs` editors satisfy all the above requirements

ed commands

Action	Command
Show the Prompt	`P`
Command Format	`[addr[,addr]]cmd[params]`
Commands for location	`2 . $ % + - , ; /RE/`

Action	Command
Commands for editing	`f p a c d i j s m u`
Execute a Shell command	`!command`
edit a file	`e filename`
read file contents into buffer	`r filename`
read command output into buffer	`r !command`
write buffer to filename	`w filename`
quit	`q`

Using `ed`

- `man ed` doesn't give much info . Use `info ed`
- `ed test.txt` shows a number indicating number of bytes read into memory
- `1` displays the first line
- `$` displays the last line
- `,p` and `%p` shows the contents of the entire buffer
- `2,3p` range - 2nd to 3rd line
- `/hello/` matches and shows first occurance of the pattern
- `+` and `-` to scroll by line
- `;p` from current position to end of buffer
- `.` displays the current line
- `!date` running the date command within `ed`
- `r !date` read output of date command to buffer at current position
- `w` writes the file (saves it)
- `d` delete current line
- `a` to append after current line. Press `.` and `enter` when done
- `s/appended/Appended/` Substitute - Search and replace from current line.
- `f` shows the name of the file being edited
- `p` shows the contents of the current line
- `j` for joining lines . Usage `5,6j` to join line 5 and 6
- `m` to move a line to a particular position. Usage `m1` to move current line to just below line 1. `m0` to move it right to the top
- `u` to undo previous change
- To add something to every line `%s/\(.*\)/PREFIX \1/`
    - `\1` is the back substitution
    - `\(.*\)` indicates any character that can be matched
    - `PREFIX` is the replacement string
- `3,5s/PREFIX/prefix/` substitutes prefix for PREFIX from line 3 to 5

Commands for editing in ed / ex

Command	Action

Command	Action
f	show name of **f**ile being edited
p	**p**rint the current line
a	**a**ppend at the current line
c	**c**hange the line
d	**d**elete the current line
i	**i**nsert line at the current position
j	**j**oin lines
s	**s**earch for regex pattern
m	**m**ove current line to position
u	**u**ndo latest change

Using nano

- Link

Using vi

- Link

Using emacs

- Link

6.1

Shell programming

More features in bash scripts

- Debugging
  - Print the command before executing it
    - `set -x ./myscript.sh`
    - `bash -x ./myscript.sh`
  - Place the `set -x` inside the script
- Combining conditions
  - `[ $a -gt 3 ] && [ $a -gt 7 ]`
  - `[ $a -lt 3 ] || [ $a -gt 7 ]`
  - Example condition-examples.sh
- Shell arithmetic
  - Using `let`
    - `let a=$1+5`
    - `let "a= $1 + 5"`
  - Using `expr`
    - `expr $a +20`
    - `expr "$a + 20"`
    - `b=$( expr $a + 20 )`
  - Using `$[ expression ]`
    - `b=$[ $a + 10 ]`
  - Using `$(( expression ))`
    - `b=$(( $a + 10 ))`
    - `(( b++ ))`- Without $. Not intending to return. Useful for incrementing
  - Example arithmetic-example-1.sh
- expr command operators

Expression	Description
`a + b`	Return arithmetic sum of a and b
`a - b`	Return arithmetic difference of a and b
`a * b`	Return arithmetic product of a and b
`a / b`	Return arithmetic quotient of a divided by b
`a % b`	Return arithmetic remainder of a divided by b
`a > b`	Return 1 if a greater than b; else return 0

Expression	Description
`a >= b`	Return 1 if a greater than or equal to b; else return 0
`a < b`	Return 1 if a less than b; else return 0
`a <= b`	Return 1 if a less than or equal to b; else return 0
`a = b`	Return 1 if a equals b; else return 0
`a | b`	Return a if neither argument is null or 0; else return b
`a & b`	Return a if neither argument is null or 0; else return 0
`a != b`	Return 1 if a is not equal to b; else return 0
`str : reg`	Return the position upto anchored pattern match with BRE str
`match str reg`	Return the pattern match if reg matches pattern in str
`substr str n m`	Return the substring m chars in length starting at position n
`index str chars`	Return position in str where any one of chars is found else return 0
`length str`	Return numeric length of string str
`+ token`	Interpret token as string even if its a keyword
`(exprn)`	Return the value of expression exprn

- Example expr-examples.sh
- Bench Calculator
    - An arbitaty preciscion calculator language
    - `bc -l`
        - the `l` option loads the math library.
    - `12^6` or `12.6/3.6`
    - Can be used for floating point operations.
- heredoc feature
    - helps while passing long strings without having to worry about `\n` etc.
    - Example heredoc-example-1.sh
    - Example heredoc-example-2.sh
    - A hyphen tells bash to ignore leading tabs
- PATH variable
    - Example path-example.sh
        - IFS (Internal Field Seperator)

L6.3

- if-elif-else-fi loop

```
if condition1
then
    commandset1
else
    commandset2
fi
```

```
if condition1
then
    commandset1
elif condition2
then
    commandset2
elif condition3
then
    commandset3
else
    commandset4
fi
```

- case statement options
    - commandset4 is the default for values of $var not matching what are listed

```
case $var in
    op1)
        commandset1;;
    op2 | op3)
        commandset2;;
    op4 | op5 | op6)
        commandset3;;
    *)
        commandset4;;
esac
```

- c style for loop : one variable
    - extention of POSIX and maynot be available in all the shells
    - Adding `time` before a script command gives the amount of time taken to execute.

```
begin=1
finish=10
for (( a = $begin; a < $finish; a++ ))
do
    echo $a
done
```

- c style for loop : two variables
    - Note: Only one condition to close the for loop

```
begin1=1
begin2=10
finish=10
for (( a=$begin1, b=$begin2; a < $finish; a++, b-- ))
do
    echo $a $b
done
```

- processing output of a loop
    - Output of the loop is redirected to the tmp file

```
filename=tmp.$$
begin=1
finish=10
for (( a = $begin; a < $finish; a++ ))
do
    echo $a
done > $filename
```

- break
    - Break out of inner loop

```
n=10
i=0
while [ $i -lt $n ]
do
    echo $i
    (( i++ ))
    if [ $i -eq 5 ]
    then
        break
    fi
done
```

```
- Break out of outer loop
```

```
n=10
i=0
while [ $i -lt $n ]
do
```

```
    echo $i
    j=0
    while [ $j -le $i ]
    do
        printf "$j "
        (( j++ ))
        if [ $j -eq 7 ]
        then
            break 2
        fi
    done
    (( i++ ))
done
```

- continue
    - Continue will skip rest of the commands in the loop and goes to next iteration

```
n=9
i=0
while [ $i -lt $n ]
do
    printf "\n loop $i:"
    j=0
    (( i++ ))
    while [ $j -le $i ]
    do
        (( j++ ))
        if [ $j -gt 3 ] && [ $j -lt 6 ]
        then
            continue
        fi
        printf "$j "
    done
done
```

- shift
    - shift will shift the command line arguments by one to the left.
    - shift is destructive - after the arguments are shifted t the left they are gone. This is only helpful if you don't need the arguments later.
    - n checks if it is a non-zero argument
    - except $0, which is the name of the script, the rest of the arguments get popped

```
i=1
while [ -n "$1" ]
do
    echo argument $i is $1
    shift
```

```
    (( i++ ))
done
```

- exec
    - `exec ./my-executable --my-options --my-args`
    - To replace shell with a new program or to change i/o settings
    - If new program is launched successfully, it will not return control to the shell
    - If new program fails to launch, the shell continues
- eval
    - `eval my-arg`
    - Execute argument as a shell command
    - Combines arguments into a single string
    - Returns control to the shell with exit status
    - Example eval-example.sh
- function
    - Example function-example.sh
- getopts
    - This script can be invoked with only three options: `a`, `b`, `c`. The options `b` and `c` will take arguments.
    - Example getopts-example.sh

```
while getopts "ab:c:" options;
do
    case "${options}" in
        b)
            barg=${OPTARG}
            echo accepted: -b $barg
            ;;
        c)
            carg=${OPTARG}
            echo accepted: -c $carg
            ;;
        a)
            echo accepted: -a
            ;;
        *)
            echo Usage: -a -b barg -c carg
            ;;
    esac
done
```

- select loop
    - Text Menu

- Example select-example.sh

```
echo select a middle one
select i in {1..10}
do
    case $i in
        1 | 2 | 3)
            echo you picked a small one;;
        8 | 9 | 10)
            echo you picked a big one;;
        4 | 5 | 6 | 7)
            echo you picked the right one
            break;;
    esac
done
echo selection completed with $i
```

- Additional notes
  - Warning : Never `eval` a user supplied string on any command line
  - eval is sending the strings to the shell and printing them out.
  - Can source a file with functions to use it in a shell script
  - `source mylib.sh`
  - Do not give set uid permission to the scripts unless you know what you are doing

---

L6.4

awk

A language for processing fields and records

- Introduction
  - awk is a programming language, quick to code and fast in execution
  - awk is an abbreviation of the names of three people who developed it: **A**ho, **W**einberger & **K**ernighan
  - It is a part of POSIX, IEEE 1003.1-2008
  - Variants: nawk, gawk, mawk ...
  - gawk contains features that extend POSIX (normally seen on GNU Linux systems with a symbolic link from awk to gawk)
  - Though awk is viewed as a scripting language it has enough mathematical functions to use for routine calculations. IT can do things that spreadsheets cannot do.
- Execution model
  - Input stream is a set of records
  - Eg., using "\n" as record separator, lines are records
  - Each record is a sequence of fields
  - Eg., using " " as field separator, words are fields. Evan a regular expression can be used as an FS.

    - Splitting of records to fields is done automatically
    - Each code block executes on one record at a time, as matched by the pattern of that block

- Usage

    - Single line at the command line
        - `cat /etc/passwd | awk -F":" '{print $1}'`
    - Script interpreted by awk
        - `./myscript.awk /etc/passwd`
        - `myscript.awk`

```
#!/usr/bin/gawk -f
BEGIN {
    FS=":"
    }
{
    print $1
}
```

- Examples

    - block-ex-1.awk
        - `./block-ex-1.awk block-ex-1.input`
        - `cat block-ex-1.input | ./block-ex-1.awk`
        - For each line Default block will be processed once.
        - You can have as many begin and end blocks wherever required in the awk script. BEGIN will be processed before the default block and and END will be processed after the default block.
        - We don't need `;` at the end of every statement unless you need to write multiple statements on a single line.
        - `$0` represents the line(record) which is currently being processed.

- Built-in variables

Variable	Description
`ARGC`	Number of arguments supplied on the command line (except those that came with -f & -v options)
`ARGV`	Array of command line arguments supplied; indexed from 0 to ARGC-1
`ENVIRON`	Associative array of environment variables
`FILENAME`	Current filename being processed
`FNR`	Number of the current record, relative to the current file
`FS`	Field separator, can use regex
`NF`	Number of fields in the current record

Variable	Description
`NR`	Number of the current record
`OFMT`	Output format for numbers
`OFS`	Output fields separator
`ORS`	Output record separator
`RS`	Record separator
`RLENGTH`	Length of string matched by match() function
`RSTART`	First position in the string matched by match() function
`SUBSEP`	Separator character for array subscripts
`$0`	Entire input record
`$n`	nth field in the current record

- **awk scripts**
    - `pattern {procedure}`
    - pattern (optional. If not given the code block is called default block and it is applied to every line in the input stream.)
        - **BEGIN**
        - **END**
        - general expression
        - **regex**
        - Relational Expression
        - Pattern-matching expression
    - procedure (will be applied to all recors that match the pattern)
        - Variable assignment
        - Array assignment
        - Input / output commands
        - Build-in functions
        - User-defined functions
        - Control loops

- **Execution**
    - `BEGIN { commands; }`
        - Executed once, before files are read
        - Can appear anywhere in the script
        - Can appear multiple times
        - Can contain program code
    - `END { commands; }`
        - Executed once, after files are read
        - Can appear anywhere in the script

        - Can appear multiple times
        - Can contain program code
    - `pattern { commands; }`
        - Patterns can be combined with `&& || !`
        - Range of records can be specified using comma
        - Executed each record pattern evalutes to true
        - Script can have multiple such blocks
    - `{ commands; }`
        - Executed for all records
        - Can have multiple such blocks

- **operators**
    - Assignment
        - `= += -= *= /= %= ^= **=`
    - Logical
        - `|| &&`
    - Algebraic
        - `+ - * / % ^ **`
    - Relational
        - `> <= > >= != ==`

Operation	Description
`expr ? a : b`	Conditional expression
`a in array`	Array membership
`a ~ /regex/`	Regular expression match
`a !~ /regex/`	Negation of regular expression match
`++`	Increment, both prefix and postfix
`--`	decrement, both prefix and postfix
`$`	Field reference
	Blank is for concatenation

- Adding 0 to a string makes it get interpreted as a number.

- **Functions and commands**

Operation	Commands
Arithmetic	`atan2 cos exp int log rand sin sqrt srand`
String	`asort asorti gsub index length match split sprintf strtonum sub substr tolower toupper`
Control Flow	`break continue do while exit for if else return`

Operation	Commands
Input / Output	`close fflush getline next nextline print printf`
Programming	`extension delete function system`
bit-wise	`and compl lshift or rshift xor`

- Example
  - block-ex-2.awk
  - block-ex-3.awk
    - Blocks get executed based on whether the line has `alpha`,`alnum` or `digits`
  - block-ex-4.awk
    - Matching only the first field in the record with a pattern
  - block-ex-5.awk
    - Field Separator as regular expression
    - Number of fields as condition

L6.4

- **arrays**
  - Associative arrays
  - Sparse storage
  - Index need not be integer
  - `arr[index]=value`
  - `for (var in arr)`
  - `delete arr[index]`
- **Loops**

```
for (a in array)
{
    print a
}
```

```
if (a > b)
{
    print a
}
```

```
for (i=1;i<n;i++)
{
```

```
    print i
}
```

```
while (a < n)
{
    print a
}
```

```
do
{
    print a
} while (a<n)
```

- Example
  - block-ex-6.awk
- **Functions**
  - `cat infile |awk -f mylib -f myscript.awk`
  - `mylib`

```
function myfunc1()
{
    printf “%s\n”, $1
}
function myfunc2(a)
{
    return a*rand()
}
```

  - `myscript.awk`

```
BEGIN
{
    a=1
}
{
    myfunc1()
    b = myfunc2(a)
    print b
}
```

- Example
  - func-example.awk
  - func-lib.awk

- **Pretty printing**
  - `printf "format", a, b, c`
    - format - `%[modifier]control-letter`
      - modifier
        - `width`
        - `prec`
        - `-`
      - control-letter
        - `c` ascii char
        - `d` integer
        - `i` integer
        - `e` scientific notation
        - `f` floating notation
        - `g` shorter of scientific & float
        - `o` octal value
        - `s` string text
        - `x` hexadecimal value
        - `X` hexadecimal value in caps

- **bash + awk**
  - Including awk inside shell script
  - heredoc feature
  - Use with other shell scripts on command line using pipe

- Examples
  - Spreadsheet handling efficiency demo
  - rsheet-create.awk
  - `echo " " | ./rsheet-create.awk`
  - Time the process for 2 million records `time ./rsheet-create.awk emptyfile > rsheet-data.txt`
  - rsheet-process.awk
  - `time ./rsheet-process.awk rsheet-data.txt > rsheet-pdata.txt`
  - Analysing the server logs of https://semantic.iitm.ac.in/
  - access-full.log (80MB)
  - Making smaller files
  - `head access-full.log > access-head.log` and `tail access-full.log > access-tail.log`
  - To print ip addreses alone `awk BEGIN{FS=" "}{print $1} access-head.log`
  - To print only the date and ip `awk BEGIN{FS=" "}{d=substr($4,2,11);print d, $1} access-head.log`

- The date from 5 days ago - `date --date="5 days ago" +%d/%m/%Y`
- Use the above date string to extract details from the Apache log book
- apache-log-example-1.awk
- `./apache-log-example-1.awk access-head.log`
- Investigate a suspicious ip address using `dig`
- `dig -x 136.123.209.54`
- `dig +noall +answer -x 34.234.167.93` for a one line output
- apache-log-example-2.awk
- `time ./apache-log-example-2.awk access-full.log > nstats.txt`

sed

- Working with `sed`
    - `sed -e "" edit.txt` -The default action of sed is to just print out the contents of the file if nothing is specified.
    - `sed -n` - the default action of printing is not performed
    - `sed -e '=' sample.txt` the `=` prints the line number
    - `sed -n -e '5p' sample.txt` -`-n` says not to print anything else by default. `5p` says to print the 5th line. Without `-n` all the lines will be printed and the 5th line will be printed 2 times.The fifth line is the address space
    - `sed -n -e '5!p' sample.txt` The `!` means that all lines except the 5th line will be printed. `!` Exclamation mark negates an address.
    - `sed -n -e '$!p' sample.txt` prints all except the last line. Careful with using `'` instead of `"` here.
    - `sed -n -e '5,8p' sample.txt` prints the 5th to the 8th line both inclusive.
    - `sed -n -e '=; 5,8p' sample.txt` prints all line numbers and from 5 to 8 prints lines also.
    - `sed -n -e '5,8{=;p}' sample.txt` prints line numbers for the line 5 to 8 alone.
    - `sed -n -e '1~2p' sample.txt` prints lines 1,3,5,7 ... Number coming after `~` specifies step size.
    - `sed -n -e '1~2!p' sample.txt` prints the remaining lines due to the negation specified by `!`
    - `sed -n -e '/microsoft/p' sample.txt` Supplying a phrase and an action. The hrase is microsoft and the action is to print every line containing the phrase.
    - `sed -n -e '/in place of/!p' sample.txt` prints the lines that do not contain the phrase "in place of"
    - `sed -n -e '/adobe/,+2p' sample.txt` prints the line containing "adobe" and two more lines that come immediately after that.
    - `sed -n -e '5d' sample.txt` deletes the 5th line and prints the rest
    - `sed -e '5,8d' sample.txt` deletes from the 5th to the 8th line and prints the rest
    - `sed -e '1,$d' sample.txt` deletes from the 1st to the last line and prints nothing
    - `sed -e '/microsoft/d' sample.txt` deletes all the lines containing microsoft and prints the rest
    - Most popular usage of the `sed` command is to substitute one phrase with another.
    - `sed -e 's/microsoft/MICROSOFT/g' sample.txt` search and replace. `s` implies search and `g` implies global.
    - `sed -e '1s/linux/LINUX/g' sample.txt` replaces 'linux' with 'LINUX' on only the first line.
    - `sed -e '1,$s/in place of/in lieu of/g' sample.txt` replaces 'in place of' with 'in lieu of' from the first line to the last line.
    - Modifying the incoming stream using the extended regular espression engine.
    - `sed -E -e '3,6s/^L[[:digit:]]+ //g' sample.txt` performs a search and replace from the 3rd to the 6th line of capital L followed by number/s and then a space. `-E` indicates that the Extended regular expression set should be used.
    - `sed -E -e '3,/symbolic/s/^L[[:digit:]]+ //g' sample.txt` performs a search and replace from the 3rd to the line where the phrase 'symbolic' occurs, of capital L followed by number/s and then a space. `-E` indicates that the Extended regular expression set should be used.

    - `sed -E -e '1~3s/^L[[:digit:]]+ //g' sample.txt` performs a search and replace from the 1st line every third line, of capital L followed by number/s and then a space. `-E` indicates that the Extended regular expression set should be used.
    - `sed -E -e '1~3!s/^L[[:digit:]]+ //g' sample.txt` Negation of the address range performs the opposite of the previous command.
    - Address range as a regular expression
    - `sed -E -e '/text/,/video/s/^L[[:digit:]]+ //g' sample.txt` performs a search and replace from the line that contains 'text' to the line where the phrase 'video' occurs, of capital L followed by number/s and then a space. `-E` indicates that the Extended regular expression set should be used.
    - `sed -e '1i ----------header----------' -e '$a ----------footer----------' sample.txt` Here `i` inserts before the first line and `a` appends after the last line.
    - `sed -e '/microsoft/i ----------watchout----------' -e '/in place of/a ----------alternative---------' sample.txt` 'watchout' appears above every line that contains microsoft and alternative comes below every line that contains 'in place of'
    - `sed -e '1~5i ----------break----------' sample.txt` inserts 'break' after every 5 lines.
    - `sed -e '/microsoft/c ------censored--------' sample.txt` For every line that has 'microsoft' `c` or change command is executed
    - `sed -e '1~3c ------censored--------' sample.txt`

- An sed script file

    - `more hf.sed`

```
#!/usr/bin/sed -f
1i ------header--------
$a ------footer--------
1,5s/in place of/in lieu of/g
6i ------ simpler stuff here onward --------
6,$s/in place of.*//g
```

- First line mentions the interpreter
- last line removes all the characters whenever 'in place of' is encountered
- `sed -f hf.sed sample.txt` The `-f` implies that sed will use a file.
- `more clean.sed`

```
/[[:alpha:]]{2}[[:digit:]]{2}[[:alpha:]][[:digit:]]+/!d
s/[ ]+/ /g
s/ ([[:difit:]]+).*/ \1/g
```

- For the input file block-ex-6.input - File containing roll number and fees paid
- First line deletes all lines that dont contain roll number

- 2nd line replaces multiple spaces with single space
- 3rd line keeps number by back referencing
- `sed -E -f clean.sed block-ex-6.input`
- Joining lines
  - Example : joining lines which are ending with a `\`
  - `cat join.sed`

```
#!/usr/bin/sed -f
:x /\\$/N
/\\/s/\\\n//g
/\\$/bx
```

- The `:` indicates a label. Whenever there is a `\` the `N` causes it to read the next line in the buffer.
- 2nd line - On the lines which have `\`, if there is a new line character it will be replaced with null.
- 3rd line on those lines which contain `\` we branch to the first line.
- `sed --debug -f join.sed sample-split.txt`
- The debug option helps to debug infinite loops in sed

L7.2

Version Control

- Every Save is effectively a new version of the code
  - "Make" - Compile only those parts of code that has changed. You do not touch what has not been modified.
  - If a group of programmers are working on a project with lots of codes and lots of files, following a modlar approach (Each function as a separate file in C for example). There is a tacit understanding that programmers are not going to work on the same file.
  - Each programmer has multiple verions of the each file they worked on.
  - Why is version control necessary ? To trace back to a woring version of code.
  - Versions will depend on number of users, number of files and number of versions. This needs to be kept in a database.
  - Two major version control systems
    - SVN - Centrally hosted and managed version system
      - Allows for one master who keeps track of the version of code that is being officially supported.
      - Storage Systems - Not if it fails but when it fails - When it fails no one can access. RAID - Redundant Array of Inexpensive/Independent Disks.
    - GIT - Distributed version control system
      - Even if something happens to the master server disappears nothing significant is lost because every collaborator has a copy of everything.

- GIT system doesn't really require a server