The shell is the part of the operating system that interprets commands and translates them to machine code that is executed by the kernel. The kernel is the heart of the operating system.

OS can be used either as CLI or GUI. but GUI offers limited set of functionality compare to CLI.

The STDIN is where commands are typed and STOUT where commands sends their result by default. STDERR where commands sends their error messages by default. ALL pf these can be manipulated by redirecting them .

ALL the files in linux OS are managed by file descriptor. everything is afile in linux,file descriptor are also used for input and output (including stdin,stdout,stderr),pipes and sockets. stdin=file descriptor 1, stdout=2, stderr=2. file descriptors are visible in /proc/PID/fd.

**STDOUT -> grep root /etc/\* 2>/dev/null > grepout.txt**

**ps aux** -> to get list all of our processes.

**cd /proc/** **-> ls ->** lists all process id's

**cd /814 -> cd fd -> ls -l** ( lists all the file descriptors with numbers and all the outputs and errors are written to systemd journal )

**PIPING : ps aux | grep 8**

**\*\*\*\*\* UNDERSTANDING THE INTERNAL COMMANDS \*\*\*\*\***

External commands are useful and functional but need to fetch from the disk. but the BASH uses internal commands which are part of BASH that provide core functionality. Internal commands are part of the binary and dont have to be loaded from disk which males this as fast access.

HELP : use help ( $ help ) commands to list all internal commands . $help command which gives the short description of the command. sometimes internal commands exists as external command , then $which command(test).

**$ man bash** -> to see all the internal commands.

**\*\*\*\*VARIABLES \*\*\*\*\***

using variables allows an OS to keep program code generic and seperated from site-specific information.

**$env** -> print a list of these all env variables that are part of the operating system.

key=value ( define variable)

BY default variables will be available in the current shell only(local). to define varialbles for the

current and all subshells we use **"export key=value".** (the variable will be available even after exiting from the current shell )

$myvar=green

To avoid ambiguity between $key1, use echo {key}1

To make sure a variable is automatically set, put it in one of the bash startup files.

**ALIAS :**

Alias is a bash internal command that allows you to define your own commands. allows us to re-write commands which are longer to shorter. DO NOT use alias in shell scripts as alias settings are not universal and might not exists in other computers where the shell script is used.

**$alias ->** to see the list of all current aliases

**$alias user='man useradd' -> $user ->** gets info about the manual of useradd command.

**$unalias user -> t**o cancel the alias.

**UNDERSTANDING THE BASH STARTUP FILES :**

Bash startup files are used to provide default settings for the OS environment. There are 2 files by default an we can create our own in user directory as well. The startup files are shell script themselves.

/etc/profile is a generic startup file that starts for every login shell.where login shell is a shell is opened when a user logging in. Always keep the /ect/profile as it is by not changing it to /etc/profile.d.

/etc/bashrc is a generic startup file that is started when opening a subshell. A subshell is opened when commands are executed, because every command that is executed on linux starts its own subshell. In this subshell the command is working in its own environment. It allows your commands to work in ur environment that slightly diff from ur main env.

User specific file : 1) /.bash\_profile 2) /.bashrc

bashrc is an optional file that users can create and define what should be happening when user is opening the subshell.

**LOGIN SHELL :**

login shell is a shell that initialize the user env upon login.all commands that are excuted, are

executed in subshell.login shells themselves should include the env for a subshell but the subshell doesnt always include all the env for login shell.

-->> **Vi /etc/profile** -> its a systemwide file that is used for the bourne shell but also for bourne comapatible shells

-->> **cd /etc/profile.d/** -->> shows all the packages and files we installed

-->> **vi /etc/bash.bashrc** -->> we have settings for the subshell ( settings required for the login shell are different from settings required for subshell.)

-->> ls -a -->> shows all hidden files. “.profile” is a user specific profile containing settings that should be executed when user is logging in .

“.bashrc is this user specific bashrc which conatins ALIAS.

“.basg\_logout” is a file which can be used to clean up things when users are logging out.

**UNDERSTANDING THE EXIT CODES :**

every command generates the exit codes after execution. Meaning of the exit codes is to tell the parent shell how successful the command was. “echo $?”

The developer of a programe can decide to code other exit codes as well. In shell scripts this is done by using “exit n” in case an error condition occurs. For a specific error we can can set the exit codes.

--> echo $ -> 0 success --> echo $ -> 1 failure.

--> man ls -> to check all the exit codes for that perticular command

**SHELL SCRIPTS IN DEVOPS ENV :**

What is a shell script ?

A shell script is a computer program that is designed to be run in a shell and shell is a main command interpretor that is used on the computer operating system.

Scripts can be written in different scripting languages, and typical functions u will find in shell scripts. Its a way of automating the tasks that we do in command line we do manually.

Shell scripts are a part of default working environment. Shell scripts are strong in manipulating the data in text files and many manipulations.

**WHAT IS DEVOPS :**

DevOps is a set of practices that combines software development and IT Operations. Pupose is to shorten the development lifecycle.

DevOps is a generic approch which can be implemeted in different ways.

1. Tool Chains 2) CI/CD pipelines 3) 12-factor application development 4) Deployment strategies.

**1 : ToolChain :-**

In devops toolchains are typically used to bring an application frm source code to full operation. And purpose of that toolchain is to automate that process.

Coding -> building (build in to functional application ) -> Testing -> Packaging ( makes distribution of app easier ) -> Releasing ( app releases in live env ) -> configuring ( make sure app is working properly ) -> Monitering ( to ensure the app is indeed operational )

**2: CI/CD PIPELINES :-**

automating the process of building, testing packaging in stage by stage.

**3: 12-FACTOR :-**

the 12-factor app is a methadology for building software-as-a-service apps that defines different factors which shloud be used in the apps. 12-factor app is commanly seen in containeraised env such as kubernetes env. In 12-factor app we find diff aspects such as

1 CODEBASE : one code base, tracked by revision control

2 DEPENDENCIES : explicit and isolated dependencies

3 config : configuration as code stored in the environment

4 backing services : treated as attached resources.

5 BUILD,RELASE,RUN : seperate build and run stages.

6 PROCESSES : execute the app as stateless process

7 PORT Binding : to expose services.

8 CONCURRENCY : each instance can be replaced

9 DISPOSABILITY : each instance can be replaced

10 DEV/PROD parity : keep all stages as similar as possible.

11 LOGS : treat logs as seperate event streams

12 ADMIN : treated seperatley

no matter which flow u r using in devops it all comes down to processing files through diff stages.

While using the shell scripts in devops , its important to develop them in an idempotent way, which means if u run same script the second time it will lead to exact same result.

**SHELL SCRIPTING VS AUTOMATION:-**

The aim of automation tool is configuration management ( management of configuration files, of states, installing software ). In automation tools like ANSIBLE,PUPPET,CHEF are to get managed systems in a desired state.

To bring it to the desired state, should described in a file written in YAML. YAML is a descriptive language that allows users to easily define what exactly the desired state is. To do that the automation tool comapares the current state of managed systems to the desired state and takes action if needed.

Running the automation tool multiple times, should not lead to anything different than implemetation of the desired state which is call idempotecy.

The advantage of automation tools is they can address a wide range of managed assets which includes not only linux machines and unix but also windows machines, containers, virtual mahcines, cloud instances and network devices which makes automation tools more practical in heterogeneous env. In order to manage wide range of assets sometimes agents are used.

BASH SCRIPT is action oriented and it defines actions to be accomplished. Should use “ test “ commands to make sure that based on the result of a test command with loops. BASH is not an automation tool, it is much more than automation tool. Its a programming laguage that helps in processing data, dealing with files, running specific tasks.

Automation does not replace the need for bash sripting and vice-versa.

**SHELL SCRIPTING vs PYTHON :-**

python has more advanced debugging tools and error handling features, which makes it better solution for writing bigger programs. Its richer than bash.

Bash scripts are using bash shell and core parts of the linux are written in bash and it uses shell features in more efficient way.

BASH is for OPS role in DevOps role and python is for DEV role.

**RUNNING A BASH SHELL IN ZSH :-**

$zsh

%bsh ( for changing the env)

$ cat /etc/shells ( to see available shells in ur system )

$ chsh -s {shell-name}

$ sudo chsh -s {shell-name} {username} (root user can change login shell for any account )

**LEARNING OBJECTIVES:-**

**USING Echo :-**

echo is a bash internal command. To use the formatting options, use “echo -e and put string btw double quotes.

**\b --> backspace --> echo -e “b\bc” --> prints c**

**\n --> newline --> echo -e “b\nc” --> prints b anf c in next line**

**\t --> tab --> echo -e “b\tc\td” --> prints b** **c** **d**

**USING BASH OPTIONS:-**

bash shell script can be configured with additional options and these options tell bash how to do its job.

Use “set” in a terminal to make it standard behavior from that terminal. Can be used in script directly in shebang line .( #!/bin/bash -x ).

**set -x --> ls --> set +x**

set -x prints the exact command that executed.

**USING shopt :-**

additional options can be configured from a script using SHOPT.

Shopt +s extglob --> enables extended globbing patterns. Using these options we can add features to the shell and ur scripts.

$ shopt --> lists the current options

$ shopt -s checkjobs --> lists all the jobs which ON and OFF

sleep 3600 & -> exit -> bg -> fg -> ctrl+c

**SHELL PATTERNS:-**

these patterns allow you to refer the text in flexible way. There are 2 sets of them.

1. **Regular**  **Expression :** which are patterns that are used by specific tools.
2. **Globbing**  **:** is for file patterns in bash and basic globbing applies to standard features.

\*--> matches zero or more characters , ? --> matches a single character

[..] --> matches any of characters listed , @ --> matches one occurence of pattern

Extended globbing provides additional options, but we need to enable it ( shopt +s extglob ).

**shopt +s extglob**

touch .txt e.txt ee.txt eee.txt

**ls \*.txt , ls ?(e).txt , ls \*(e).txt , ls+(e).txt , ls @(e).txt**

**GREP :-**

when using grep it is recommended to keep text pattern in single quotes to avoid interpretation by the shell because shell does the command line parsing where shell analyzes what u typed and if there are any special characters that need to be interpreted.we dont want this reg exp which are used by the grep to be interpreted by the shell.

**$ grep 'root' \* , $ ps aux | grep 'ssh'**

the results of the grep command should be clear and to avoid false positives in the output anf to exclude them we use “ $ ps aux | grep 'cron' | grep -v 'grep' “

$ **grep 'root' \* 2>/dev/null** --> redirects all the error messages to null. And to display the files of that text pattern we mentioneed “ **$ grep -l 'root' \* 2>/dev/null** “

**$ grep -A 1 'root' \* 2>/dev/null , $ grep -B 3 – A 3 'student' \* 2>/dev/null -->** prints the lines before and after the text pattern we mentioned.

$ grep -r 'root' \* 2>/dev/null --> prints all the sub-directories of that text which we mentioned in the command .

**Regular Expressions:-**

Globbing is about the shell patterns and regular expressions is about text patterns. Should use single quotes while using reg exp because the characters we mentiones should not be interpreted by shell but by the reg exp. Reg exp can be used by few utilities ( grep,awk,sed,vim).

**Elements of regular expressions :-**

1. Reg exp are built around atoms where atom is a core part which specifies what text should be matched.
2. Second element is the repitition operator, which specifies how many times a character should occur.
3. Indicating where to find the next character.

All these 3 may occur in reg exp but not always.

**$ grep '^l' text.txt , $grep '^lea$' text.txt** --> $ is line sould end with.

$ grep '^lea\b' text.txt --> to see the words that ends with.

$ grep 'b.\*t text.txt --> . prints any character from zero or more times which prints all in between b and t .

$ grep -E 'bo+t' text --> gives one or more times. And this is extended reg exp.

$ grep -E 'bo?t' text --> ? gives zero or one time.

$ grep 'bo\{2\}t' text --> prints the string which has exactly 2 o's .

**USING CUT and SORT :-**

cut allows you to filter out feilds based on field seperator. **cut -d : -f1 /etc/passwd.**

**:** is the feild seperator. “ -d = to tell cut that it should use the colon as feild seperator. “

**SORT :-** sort by default sorts in the order of ASCII table. With -n it sorts from low to high value.

sort -n : for numeric sort , -d : for dictionary order.

Utilities like cut uses IFS ( internal feild seperator ). IFS for most utilities is SPACE and many many commands do have options to use IFS that should be used. If u have feilds that arenot distinguished by spaces make sure u specify IFS to make sure ur utility work.

**cut -d : -f 3 /etc/passwd | sort -n**

**TAIL and HEAD :-**

allow us to print lines of top and bottom of the files. “ **tail /etc/passwd “** if we dont specify any number we get last 10 lines by default.

tail -n 5 /etc/passwd | head -1

**Using SED :-**

its a stream editor and it allows to edit files even if no full screen is visible and to edit files in a line oriented way.

**sed -n 5p /etc/passwd** -> p=print the 5th line

**sed -i s/old/new/g~/myfile** -> s = substitute , g = global substitute( changes the text in multiple occurences)

**sed -i -e '2d' ~/myfile** -> d = delete

**for i in \*txt; do sed -i 's/hello/bye/g' $i; done**

sed is powerful external command which needs to be fecthed from the disk which makes the shell scripts slow.

**Using AWK :-**

awk is filtering utility, it has a little bit id grep,cut.

**awk -F : '{ print $4 }' /etc/passwd** → prints 4th column of that file. And -F is field seperator. We can do this with cut as well.

**awk -F : '/user/ { print $4 }' /etc/passwd** → searches the specific text with matching pattern. Can do with grep and cut command together by piping.

**awk -F : '$3 < 1000 { print $1 }' /etc/paswd →** looking for the 3rd line of the file less than value 1000 and print the 1st line which are having less than value 1000.

This is useful because in linux u have system id's and u have ordinary users. And system id's normally uses smaller than 1000

**Considering External Tools and Performance :-**

best practice is to avoid external tools because these tools and all libraries require needed to be fetched from disk which makes the slow. And external tools are OS specific sometimes even distribution specific which makes the script less portable.

Use **type** to find out if a tool is external or not. To measure performance use **time myscript** to measure the amount of time it takes to run a script with or without external tools.

$help shows the utilities that are part of bash shell. Internal utilties are prefered to use

compare to external utilities.

**ls -l $(which bash) $(which sh) →** to print the complete paths of the utilities.

**LAB 1:-**

create a list of all users on your linus system that have UID higher than 1000. 1) full lines from /etc/passwd are printed 2) list is alphabetically sorted by username 3) occurrence of /bin/bash is replaced with /bin/zsh 3) changes are written to /tmp/myusers.

**Solution:-**

**awk -F : '$3 > 1000 { print $0 }' /etc/passwd | sort | sed 's/\/bin\/bash/\/bin\/zsh/'**

in the sed substitution the slashes has a special meaning. If we are going to substitute that anything that has a slash (/) we need to escape these characters by putting forward slash infront of them.

Changes will be made only if it has '-i' in the sed command.

**MODULE 2:-**

**( variables and arguments, Transforming input)**

Core bash scripting Ingredients:-

1) #!/bin/bash 2) comment to explain what script is doing

3) white lines to increase readability 4) Diff blocks of code to all parts of the script.

→ scripts cannot be executed from the current directory. Consider using **/bin** to store scripts for personal use. Consider /usr/local/bin for scripts that should be available for all users ( to put script in /usr/local/bin u need sudo previliges).

**bash ./script.sh or chmod +x script.sh** ( adding the execute persmission andd u need to be owner of that file)

if the directory containing the script is not in $PATH, run it using **./myscript**

→ script can also be started as an argument to the shell, in which case no execute permission and $PATH is needed. : **bash myscript**

**Finding help about scripting components:-**

help provides an overview of bash internal commands which are core components of any scripting.

Use man pages for checking info about internal commands.

**First Script:-**

keep the script in $PATH to be executable by all the members. Copy the script in to the path. **Sudo cp myscript.sh /usr/bin/bash**

**$id →** to check weather u are in sudo previleges are not.

**Working with Variables and Arguments:-**

An Argument is anything that can be put behind the name of a command or script.

**$ ls -l /etc** → -l and /etc arguments. --l is an option which is also can be an argument that changes the behavior of the command or script. And its functionality is programmed in to the command.

Options are always an arguments but arguments are not always options. Positional parameter is another word for an argument.

A Variable is a key with a name that can refer to a specific value.

While treating arguments, arguments becoming the variables with in the script.In further all options or all arguments are treated as variables.

**Quoting:-**

Escaping :- escaping is the solution to take away special meaning from characters. To apply escaping ,use quotes. Double quotes are used to avoid interpretation of spaces ( eg: a string will be having spaces in between words. To avoid bash treating them as separate things use double quotes.)

→ single quotes are strong which are used to avoid interpretation of anything.

Echo '$SHELL' prints the line and echo $SHELL will interpreted and shows the current shell.

→ ( \ ) back slash is used to avoid interpretation of next character. ( echo \$SHELL )

echo the value of '$SHELL' is $SHELL → the value of $shell is /bin/bash

echo 'the value of '$SHELL' is $SHELL' → the value of /bin/bash is $SHELL

**Defining and Using Variables :-**

Local variable works only in current shell. Environment variable is an operating system setting that is set while booting. bash variables dont use data types..

**declare -r ANSWER=yes** → sets $ANSWER as read-only variable.

declare -p is to find out which type of variable something is.

**declare -p GROUPS** → shows the value of the groups variable.

declare -p PATH →

**declare** is not for declaring variables but used to declare specific variable attributes. Declare is not to set a variable but may required to set an array.

Defining variable is key=value. Variables are not case sensitive. Env variables by default written in upper case. Local variables are written in any case. After defining variables are available only in current shell and to make them available in subshell use export key=value.

→ to wipe the value of the variable without deleting variable itself. **Key= , unset var** → to remove the variable.

→ export color=red → $color → bash → $color ( variable is available in subshell as well)

a subshell is a shell that is defined as a child of the current shell. Export is only for subshells and subshells only. If we want variable to be available in all shells we should put it in some startup files like .bashrc.

USING Variables :-

color=red → echo $colour → echo ${color}

color=red-> echo ${color}1 → red1

curly braces are best to use with variables when we are using regular expressions with value or if we want to append anything to the value.

Special Variables:-

$RANDOM-> a random number

$SECONDS → the number of seconds that this shell has been running.

$LINENO → the number of lines of current script.

$GROPS → an array that holds the names of groups that the current user is member of.

→ compare echo $GROUPS vs echo “${GROUPS[@]}”

**Define Variables with READ :-**

→ when read is used shell script execution will stop to read user input. The user input is stored in the variable that is provided as an argument to **read.**

**echo enter any value**

**read value**

**echo u have entered $value** → here value is a variable defined with read which stores the value given by the user.

→ Read can also be used without further arguments.

**echo press enter to continue**

**read**

**echo continuing...**

→ read can be used to define more than one variable at same time

**#!/bin/bash**

**echo enter firstname, lastname and city**

**read firstname lastname city**

**echo nice to meet u MR. $firstname $lastname from $city**

**#!/bin/bash**

**echo “ which directory u want to switch “**

**read DIR**

**cd ${DIR}**

**pwd**

**ls →** after running the script it list files from the directory we mentioned but it wont switch to the directory in the terminal. Because if u start a script, the script starts subshell, but once script is done will get out of subshell.

**Separating Your Dynamic Code with Site Specific Data :-**

→ **you are going to separate code from variables. Because code, and that includes shell scripts as well, should be portable. Portable meaning that it is possible to use your shell script in another environment. And there are two ways in general to make your shell script portable. To start with,**

→ **you use shebang to make sure that it works in other shells.**

→ **And secondly, if you work with any site-specific data, you should put that in variables and you should not include those variables in the shell script itself.**

**That is what we refer to as decoupling. So decoupling means that you have a strict separation between your static code and your dynamic site-specific information. And to implement decoupling in shell scripts, variables should be defined in separate files that are used by sourcing or including them in the main script .And when a file is sourced, the contents of that file is included in the current shell without starting a subshell.**

**Using Source**

when sourcing variables the files to be sourced should be stored separately. To source a script 2 approaches exists:

1 . **source myvars** will source the file

2 **. myvars** is an alternative way to do the same

**LAB:-**

**sudo yum install network-scripts**

**vim /etc/init.d/network →** init was using to start the services in the linux in previous versions and network is one of those script.

If a script is being sourced then u do not have shebang line in the beginning because if we have it will start as a subshell. We want that souce script to contain variables, structures that we want to import from another script.

**mkdir sourcefile → vim sourcefile/colors → COLOR=red → save it → vim main →**

**#!/binbash**

**. /sourcefile/colors**

**echo the color is $COLOR**

**Viewing Variables :-**

**set** shows all the current variables including functions and the values of the variables

**compgen -v** will show variables only and not values.

**$color=green → set | grep '^color'**

**$compgen -v →** shows all the variables available in current shell

**Handling Script Arguments :-**

The first argument of the script is stored in $1 and the second argument is stored in $2 and so on. The script name itself is also stored in a dollar value, that's $0. And the values of these variables cannot be changed from within a script. By default, a maximum of nine arguments can be defined in this way. And when you are using curly braces, more than nine arguments can be provided.

To refer the script arguments we can use **$@** and **$\*.** with out the quotes both are identical. $@ is the rule and $\* is the exception.

#!/bin/bash

echo “ Hi $1 how are u “

echo “ Hi $2 how are u “

echo “ Hi $10 how are u “

echo “ Hello ${10} “

echo “ Hi ${12}”

shift

echo “hi $1”

echo “ \$0 is $0”

Script 4 :( positional parameters )

#!/bin/bash

echo “\$# gives $#” → prints number of arguments provided.

echo “\$@ gives $@”

echo “\$\* gives $\*”

echo “\$0 is $0”

echo showing the implementation of \$\*

for i in “$\*”

do

echo $i

done

echo showing the implementation of \$@

for i “$@”

do

echo $i

done

**Shift :**

**shift** is used to shift the positional parameters to left. It can take a number as its arguments to shift the positional parameters to the left by that number.

#!/bin/bash

echo the script has $# arguments

echo print $1

shift

echo print $1

shift 3

echo print $1

**Command Substitution:-**

command substitution is used to work with the result of a command instead of a static value that is provided. We can use this to refer to values that are changing frequently.

Eg: **today=$(date +%d-%m-%y) , mykernel=$(uname -r)**

**#!/bin/bash**

**if [ -z $1 ]**

**then**

**echo argument required**

**exit 9**

**fi**

**sudo tar -cvf $(date +%d-%m-%y).back $@ →** writes backup file to current directory and backs up everything provided as argument.

**Here Document: -**

A here document is used as I/O redirection to feed a command list to an interactive program or command. We can use it in 2 ways which is input to a command from a script and other way is from output.

→ with use of here doc all the code that needs to be processed is a part of the script and there is no need to process the external commands.

**Lesson 7-> Working with parameter substitution: -**

-> This is a solution that allows u to deal with missing parameters.use it to set a default or to display a message in case missing parameters.

**#!/bin/bash**

**echo enter username or press enter to use default value**

**read username**

**echo ${username:-$(whoami)} ->** :- is setting a default value for the parameter in case if not set.

--> If parameter is not set it uses as **echo ${username:-$(whoami)}** uses as default but does not set it. **Filename=${1:-$DEFAULT\_FILENAME}**

--> if parameter is not set it sets as default.**echo ${username:=$(whoami)} ,**

**Filename=${1:=$DEFAULT\_FILENAME}**

**Using Pattern Matching Operators:-**

The purpose of pattern matching operator is to clean up the string where string contains too much info and we can delete all the info from string that we dont need.

**${1#}** prints string length of $1.

**${1#pattern}** removes the shortest match of pattern from the front end of $1.

**${1##pattern}** removes the longest match of pattern from the end of $1.

**${1%%pattern}** removes the shortest pattern from the back end.

**${1%%pattern}** removes the longest pattern from the back end.

**Understanding Variable String Replacement :-**

**${var/pattern/replacement}** -> is used to replace a pattern inside a variable

**${var//pattern/replacement}** -> is used to replace the pattern globally

**${var/#pattern/replacement}** -> will only replace if the variable starts with the pattern.

**${var%pattern/replacement}** -> will only replace if variable ends with pattern.

**#!/bin/bash**

**VAR=monkey**

**echo $VAR**

**${VAR/monkey/horse}**

**echo $VAR**

--> instead of **sed ,** using **pattern matching operators** is more efficient because these are bash internals and **sed** needs to be loaded from disk.

**Using patterns and Extended Globbing: -**

extended globbing can be used to analyse file patterns in smart way.

**#!/bin/bash**

**shopt -s extglob** -> enabing shell option extended globbing

**for i in \***

**do**

**case $i in** --> evaluating spcific case statements

**!(\*.doc|\*.txt|\*.pdf))**

**echo $i is not a document**

**;;**

**\*)** ---> no opening braces in case syntax

**echo $i is a document**

**;;**

**esac**

**done**

**Remove pattern with extended globbing: -**

**#!/bin/bash**

**shopt -s extglob**

**for i in \***

**do**

**echo ${i%%\*(.doc|.txt|.pdf)}**

**done**

**#!/bin/bash**

**shopt -s extglob**

**for i in \***

**do**

**mv $i ${i%%\*(.doc|.txt|.pdf)}** --> changes the name of files from x.doc.y.txt,z.pdf to x y z

**done**

**Bash Calculations: -**

Bash offers different solutions for calculating with integers.

-> **let a=1+2 -> echo $a**

-> **echo $((2 \* 3))**

-> **echo $(( $(sudo fdisk -l | grep '/dev/sda:' | awk '{ print $5 }') / $(( 1024 \* 1024 ))))**

**bc** is an advanced calculation tool that allows to work with decimals.

**Echo “12/5” | bc -l** -> to print decimals in result

**echo “sqrt(1000)” | bc -l**

**factor** decomposes an integer in to prime factors -> **factor 399**

**Using tr: -**

**tr** is an external utility that allows for translation for characters. **echo hello | tr [a-z][A-Z] →** HELLO

**echo hello | tr [:lower:][:upper:]** → HELLO

**echo how are u | tr [:space:] '\t'** → how are you.

**Echo hello | tr [a-m] [n-z]**

Changing case in Variables :-

**color=red; echo ${color^^}**

**color=BLUE; echo ${color,,}**

**LAB 7 :-**

**#!/bin/bash**

**for i in \***

**do**

**mv $1 ${i%.\*}**

**done**

**TEST :-**

**test** is foundation of many **if** statements. It is typically used in conditional statements.

**help test**

**test -f /etc/hosts → echo $? or [ -f /etc/hosts ] → echo $?**

**if statement :-**

if is used to verify a statement.

**If [ -f /etc/hosts ]; then echo “ file exists”; fi**

**if [ -f /etc/hosts ] && echo file exists**

**if [ -f /etc/hos ]; || echo file exists**

**Using if...then‚Ä¶else with elif :**

it allows us to do additional testing if the condition is not true.

**If [ -d $1 ]; then**

**echo $1 is a directory**

**elif [ -d $1 ]; then**

**echo $1 is a file**

**else**

**echo $1 is an unknown entity**

**fi**

**Applying Conditions and Loops: -**

**for loop** → for is to iterate through a range of items.

**for i in {15..51}; do echo $i; done**

**for i in {125..145}; do ping -c 1 192.168.64.$i; done** → pings all the ip addresses in that range.

**Break and Contunie: -**

break is used to leave a loop straight away.

Continue is used to stop running through this iteration and begin the next iteration. It is useful if a situation was encountered that makes it impossible to proceed.

**Advance Bash Scripting Options: -**

An OPTION is an argument that changes the script behavior.

→ use **while getops “ab:” opts** to evaluate options a and b, and while evaluating them, put them in a temp variable. If we want to handle options we need to use **while** and the builtin **getops. Getops** evaluating the string that is provided and its doing a **while** iteration over it so it will consider one by one and put them in temp variable opts(here).

**b: →** is an option that can work with an option argument

→ use **case $opts** to define what should happen if a specific option was encountered.

**Eg:**

**#!/bin/bash**

**while getops “hs:” arg; do**

**case $arg in**

**h)**

**echo “usage”**

**;;**

**s)**

**strenght=$OPTARG**

**echo $strength**

**;;**

**esac**

**done**

If we dont provide option argument in the command input with the option it will throw an error in the output as “ **option requires an argument --s”**

→ If we give diff option in the command input which is not defined in the script it throws an error message as “ illegal option “

**GETOPS: -**

The getopts command in Bash scripting is specifically designed to handle command-line options within a script. It allows the script to parse and process the options provided by the user when executing the script. By using the getopts command, the script can define and specify the available options, handle options with or without arguments, and perform appropriate actions based on the options selected by the user. It is a powerful tool for adding flexibility and functionality to Bash scripts when dealing with command-line inputs

**Variables in Functions: -**

→ Variables always have global scope no matter where they defined. Even if tghey defined inside the function.

→ Use the **local** keyword to define variables with a local scope inside of a function.

\*\*\* **Understanding BASH Menu \*\*\***

→ The **select** statement is used to select a menu.

→ use **ps3** to define a prompt

→ The **select** statement embeds the case statement.

→ After executing the menu option u will get back to the menu .

→ **$REPLY** is a default variable that contains the string that was entered at the prompt.

( eg script is in other doc )

**Signals && Traps : -**

**Signal → In Linux, Signals are the interrupts that are sent to the program to specify that an important event has occurred.** Signals are specific software interrupts that can be sent to a command.

**ctrl+c → used to exit from the script**

**sig kill -9 pid → to kill a command. ( -9 is symbol of SIG KILL signal)**

→ while our script is doing something or running and suddenly when something interrupts in the middle in the form signal which stops our script or interrupt the execution of our script. Here to avoid this we use **TRAP** command which provides the script to capture the signal and interrupt and clean it with in the script.

→ **man 7 signal →** to see all the signal options in man page

→ TRAP cannot catch the SIGKILL and SIGSTOP command.

Eg :

**#!/bin/bash**

**trap “echo Exit signal is detected” 2 →** 2 or SIGINT is value of ctrl+c

#**trap “echo exit signal is detected” SIGKILL →** $kill -9 pid (which kills the process)

**echo “ pid is $$ “**

**while ((count < 10 ))**

**do**

**sleep 5**

**(( count ++ ))**

**echo $count**

**done**

**exit 0**

**ARRAYS: -**

→ An ARRAY is a parameter which can holds multiple values, stored as key/value pairs where each value can be addressed individually.

→ Use arrays to ensure that list of things that need to be processed always work.

**Files=(\*.txt); cp “${files[@]}” /tmp** → assigning array of files to an variable and copy them in to other directory.

Array Types :

1. **Index**  **arrays :-** it address a value using an index number and “ declare “ command does not have to used in index array.
2. **Associative**  **Arrays : -** it address a value by using a name. These arrays provide the benefit of using meaningful keys.

**$ declare -A assArray2=( [HDD]=Samsung [Monitor]=Dell [Keyboard]=A4Tech )**

**Reading command output in to arrays: -**

→ **readarray** is a new solution that allows you to put the result of a command in an array.

**$ readarray -t my\_array < <(seq 5) → declare -p my\_array**

**$ readarray -t my\_files < <(ls) → declare -p my\_files**

The mapfile command allows you to fill an array with the result of a command**.** Youcan use the syntax **mapfile -t my\_array < <(my\_command)** to assign the command output to the array **my\_array"**. The **-t** option is used to remove trailing newline characters from each line of the command output. This provides a convenient way to store the command output in an array for further processing in your script.

**Script Methodology : -**

1. Start writing bigger structure if the scripts by commenting the lines.
2. Write and test each part of the script separately.
3. While doing try not to use commands that change everything, instead use **echo.**
4. Use white lines to make script readable.

**Using set options : -**

Bash **set** can be used in a script to enable / disable specific functionality. Set options can be specified in 2 ways **1) set option 2) set -o option-name**

**eg : - set -e & set -o errexit** are same

set -x → to enable an option.

Set +x → to disable an option

-e → exit the script when command fails

-i → runs the script in interactive mode

-v → runs a script in verbose mode

-x → runs a script in verbose mode while expanding commands.

**Writing Debug Info in to a file : -**

→ when using **bash -x** , debug info is sent to **STDERR.** Since bash 4.1, the **BASH\_XTRACEFD** variable can be set to write debug info to a file. This variable defines a file descriptor that is next redirected to a specific file.

→ when using **BASH\_XTRACEFD** you will need a custom file descriptor. To define a custom file descriptor use **exec .**

→ **exec 15 > myfile** is sending all that is sent to FD 15 the file myfile. Next **BASH\_XTRACEFD** is assigned to use this file descriptor.

**Running bash -x : -**

→ instead of using **set** option u can use **bash -x** from the command line. Its an elegant option as it debugs every single line of the script which we need to focus on one specific section. The benefit is that u don't have to modify the script code while using **bash -x**

**Using Functions: -**

A function is a small block of reusable code that can be called from the script by referring to its name. It is useful when blocks of code are needed repeatedly.

Functions are local to script when they are executed. Bash wide functions are initialized from the bash startup scripts .

Functions can work with arguments but it has only local scope. Functional arguments are not affected by passing positional parameters to a script while executing it.

**#!/bin/bash**

**COUNTER=$1**

**COUNTER=$(( COUNTER \* 60 ))**

**minusone()({**

**COUNNTER=$(( COUNTER - 1 ))**

**sleep 1**

**}**

**while [ $COUNTER -gt 0 ]**

**do**

**echo you still have $COUNTER seconds left**

**minusonne**

**done**

**[ $COUNTER = 0 ] && echo time is up && minusone**

**[ $COUNTER = "-1" ] && echo you now are one second late && minusone**

**while true**

**do**

**echo you now are ${COUNTER#-} seconds late**

**minusone**

**done**

**Script 10 :-**

**#!/bin/bash**

**COUNTER=$1**

**COUNTER=$(( COUNTER \* 60 ))**

**minusone(){**

**COUNTER=$(( COUNTER - 1 ))**

**sleep 1**

**}**

**while [ $COUNTER -gt 0 ]**

**do**

**echo you still have $COUNTER seconds left**

**minusone**

**done**

**[ $COUNTER = 0 ] && echo time is up && minusone**

**[ $COUNTER = "-1" ] && echo you now are one second late && minusone**

**while true**

**do**

**echo you now are ${COUNTER#-} seconds late**

**minusone**

**done**