**Session 1**

**Scripts allow you to**

1.automate tasks

2.save time

3.increase reliability

**Shell:**

* Shell is command-line interpreter that makes it easy for you to issue commands to your operating system
* You learned that the bash shell is a great shell to learn because it is fast, has a lot of feature, and is very commonly used
* Scripts are just files containing shell commands
* Scripts make automating sets of commands very convenient. Scripts save you time and makes you more productive

**Core Components of bash scripts**

#!/bin/bash - shebang line

=====================================================

First\_Shell\_Script.sh

#!/bin/bash

Echo "This is our first shell script"

exit 0

To run shell script - ./First\_shell\_Script.sh

Exit statements (0 = successful; 1-255 = unsuccessful)

**Structure of a Bash Script**

# is used to add comments in bash Scripts

# Author: Manjunath Gudur

# Date created: 21/08/2024

# Date modified: 21/08/2024

# Description about the Script

# Usage of the script

=====================================================

File permissions: https://chmod-calculator.com/

* Read and interpret file permissions
* Use chmod command to change file permission
* The recommended file permissions for scripts (744)

=====================================================

**First bash script to backup my folder**

vi backup\_script

#!/bin/bash

# Author: Manjunath Gudur

# Date Created: 21-08-2024

# Date Modified: 21-08-2024

# This Script will take the backup of my folder

tar -cvf ~/bash\_course/my\_backup\_"$(date +%d-%m-%y\_%H-%M-%s)".tar ~/\* 2>/dev/null

exit 0

**Making your scripts accessible from any folder:**

\* Edit your ~/.profile file to add a custom folder to your PATH

echo $PATH --- This command will provide the executible folder path

export PATH="$PATH:$/path/to/script/directory

\* Reload the ~/.profile file

source ~/.profile

\* Now you can run your scripts from anywhere in the system

**Session 2**

**Three types of shell parameters:**

1. Variables
2. Positional Parameters
3. Special Parameters

**Variables:**

There are two different types of Variables – 1. user defined variable 2. shell variable

**User defined variable:**

Parameter whose values you can manually change and Variables are most commonly used

Ex: a=2, student=”Manju”, b=100

To print variable in o/p --- echo “Hello ${student}”

**Shell Variables (Environment Variables):**

1. **PATH:** The PATH variable contains the list of folders that the shell will search for executable files to run as command names.

**echo $PATH** to get the all the paths

1. **HOME**: The HOME variable is used to store the absolute path to the current user’s home directory.

**echo $HOME**

1. **USER:** The USER variable contains the username of the current user

**echo $USER**

1. **HOSTNAME / HOSTTYPE:** HOSTNAME will provide the computer name and HOSTTYPE will give type of processor architecture the computer is running.
2. **PS1 – PS1** variable contains the prompt string shown in the terminal before each command

Parameter Expansion Tricks:

1. **Modify Parameter (Variables) Case:**

* To lower the case of a letter then use , after the parameter name

Name=ManJU

Echo ${name,} 🡺 manJU

Echo ${name,,} 🡺 manju

* To upper the case of a variable value then use ^ after the parameter name

Name=manju

Echo ${name^} 🡺 Manju

Echo ${name^^} 🡺 MANJU

1. **Determine the length of a parameter:**

Echo ${#name} 🡺 5

1. **Slice Parameter:**

Name=ManjunathGudur

Syntax: $[parameter:offset:length}

Echo ${name:0:9} 🡺 Manjunath

1. **Brace Expansion:**

There are two types of brace expansion:

* String lists: {1,2,3,manju,a,b,c} 🡺 1 2 3 manju a b c
* Range lists: {1..10} 🡺 1 2 3 4 5 6 7 8 9 10

How to expand incrementally using an additional “..” followed by the required increment

{1..100:3} 🡺 1 5 9 13 17 21 24 etc

How to use the leading zeros to make sure that all values you expand to have the same number of digits. {01..10} 🡺 01 02 03 04 05 06 07 08 09 10

SESSION 3

**How Bash Processes Command Line -**

1. Tokenisation:
2. Command Identification: simple commands(word commands and compound commands (loops commands)
3. Shell Expansions:
4. Quote Removal:
5. Redirection:

**Quote Removal:**

Three Types of Quoting:

1. **Backslash (\)** : Removes special meaning from next character

Echo jon \& jane 🡺 here \ is used before & (special character in bash)

1. **Single Quotes (‘’):** Removes special meaning from all characters inside.

Filepath=c:/users/manju/documents 🡺 this will remove all the / i.e c:usersmanjudocuments

Filepath=’c:/users/manju/documents’ 🡺 This will o/p filepath=c:/users/manju/document

1. **Double Quotes (“”):** Removes special from all except dollar sign ($) and backtick (`)

**Tokenisation:**

* The shell identifies unquoted metacharacters and uses them to divide up the command line into tockens.

**Metacharacters** are:

| & ; () <> space tab and newline

* It then characters the tokens into words and operators
* **Words**: are tokens that do not contains unquoted metacharacters
* **Operators**: are tokens that contains at least one unquoted metacharacters

**There are two types of Operators**:

* Control Operators:

Newline

| ||

& &&

; ;;

;& ;;&

|& ( )

* Redirectional Operators:

< >

<< >>

<& >&

>| <<-

<>

**Command Identification:**

1. **Simple Commands:**

A simple command is just a list of words separated by spaces and terminated by either a newline or one of the other control operators available in bash.

The first word of the simple command is interpreted as the command name, and the following words are interpreted as arguments to that command.

Ex: echo manju 🡺 echo is command name and manju is argument

1. **Compound Command:**

Loops are used and it will be in multiple lines

**Expansions:**

There are 4 stages of expansion:

1. Stage1: Brace expansion
2. Stage2: Includes- Parameter expansion, arithmetic expansion, command substitution and tild expansion
3. Stage3: Word splitting – Word splitting will perform based on the IFS(Internal Field Shell) echo “${IFS@Q}” – to check the default variable (newline space or tab)

IFS={.} – to set different field separator variable

1. Stage4: Globbings – File expansion( \* ? and [ )

**Globbing:**

* Globbing, is also known as filename expansion, is used to generate an alphabetically-sorted list of file names that match a certain pattern exactly
* Any word on the command line that contains an unquoted \*, ?, [] will be interpreted as a pattern.
* \* will match any string, regardless of its length or what characters it contains \* will even match an empty string.
* The ? character will, like the asterisk, match any character, but it will only match exactly 1 character
* The [] matches exactly 1 character, but only if it is one listed within the square brackets

**SESSION 4**

**Positional Parameters:**

* Command line arguments are information you give to your script from your command line. Each argument is separated by a space
* Positional Parameters are parameters set by the shell to store the value of each of these command line arguments
* You can’t save data in positional parameters manually
* However, you can refer to values help by positional parameters using parameter expansion.
* You need to use the advanced “dollar-curlybrace” syntax for positional parameters 10 and above.
* ${parameter:-word} to check if parameter is empty if it is then fill it with default value
* :-0 to add zero to empty positional parameter

**Example:**

**Positional\_script.sh**

My name is a

My home directory is b

My 10th argument is “${10}”

Running script by giving arguments

./Positional\_script.sh Manju $HOME 3 4 5 6 7 8 9 10

O/P:

My name is Manju

My home directory is /home/manju

My 10th argument is 10

**Special Parameter:**

**$0 and $#**

1. **$0**

$0 holds the script name if it’s used inside the script, it holds the value of bash if its used outside

1. **$#**

This holds the value of number of arguments passed while running the script

1. **$@**

This holds the all the command line arguments passed

./script.sh 1 2 3 4

o/p – 4 (total 4 arguments passed to the command line)

if we pass arguments in double quotes “” then $@ will split the words based word splitting ( space tab or new line)

ex: we have script named script.sh

#!/bin/bash

touch $@

Exit 0

If we pass “my name” “your place” as arguments then instead of creating 2 files it will create 4 files namely my name your and place.

To avoid this situation, we can wrap the $@ in double quotes

1. **“$@”**

To avoid the word splitting of arguments passed to command line

#!/bin/bash

Touch “$@”

Exit 0

./script “my file1” “file2 file3”

o/p – two files will be created i.e ‘my file1’ and ‘file2 file3’

1. **$\***

**Similar to $@**

1. **“$\*”**

Expands to all positional parameters as one word separated by first lettwr of the IFS variable without subsequent word splitting.

**Read Command:**

**Read: help read to get more info about read command**

read variable\_names

Input – user input to store in variables

**Methods of reading input from user:**

1. read -p “My name is: “ name

This will print “My name is: “ before taking input from the input

1. read -t -p “My name is: “ name

This will have 5 sec timeout and it will move to next command after 5 sec

1. read -s -p “type password: “ Password

Your input will be invisible in the terminal(This is used to hide input given by user)

1. read -N 4 🡺 this will read 4 digits from input

**SELECT:**

**PS3=** To add user message for select command

**Syntax:**

PS3=What is the day of the week

select day in mon tue wed thu fri sat sun;

do

echo “The day of the week is $day”

break #to break the command

done

results:

1. mon
2. tue
3. wed
4. thu
5. fri
6. sat
7. sun

What is the day of the week?: 1

The day of the week is mon

**SESSION 5**

**List:** When you put one or more commands on a given line

**List Operator:**

& 🡺 commands will execute asynchronously

; 🡺 commands will run one by one

&& 🡺 second command only runs if the first one was successful

|| 🡺 second command only runs if the first one failed

**Test Commands and Conditional Operator:**

**Test Command:** a command that can be used in bash to compare different pieces of information. Result of this command is TRUE or FALSE.

If test is evaluated to be true then test will return an exit status of 0(zero)

If test is evaluated to be false then test will return an exit status 0f 1.

Syntax:

[ test ]

e.g [ 2 -eq 2 ] ; echo $? 🡺 0 ( $? Will give exit code of last command and -eq checks if arguments are equal )

-eq 🡺 checks integer for **equal** to condition

-ne 🡺 checks integer is **not equal** or not

-gt 🡺 checks integer for **greater than** condition

-lt 🡺 checks integer for **less than** condition

-geq 🡺 checks integer for **greater than or equal** to condition

-leq 🡺 checks integer for **less than or equal** to condition

These commands are only for integer not for decimal numbers

**Conditional Operators for Strings:**

= 🡺 checks strings are equal

!= 🡺 checks strings for not equal

-z 🡺 checks for empty string

-n 🡺 matches non empty string

**File Test Operators:** File test operators compare files

**-**e 🡺 to check file exist or not

**-**f 🡺 checks for regular file (text file)

**-**d 🡺 checks for directory

**-**x 🡺 checks if file exist and has execute command

**-**r 🡺 file is readable or not

**-**w 🡺 file is writable or not

**IF Statements:**

**Syntax:**

If [ test cond1 ] ; then

Cond..

Elif [ test Cond2] ‘ then

Cond..

Else

Cond..

fi

e.g: if [ 2 -gt 1 ] ; then

echo “2 is greater”

else

echo “2 is not greater”

fi

**CASE Statements:**

Syntax:

Case “$variable” in [comma separated arguments]

Pattern1)

Commands

;;

Pattern2)

Commands

;;

PatternN)

Commands

;;

\*) –default command

esac

Ex: case “$number” in

[0-9]) echo “You have entered the single digit number”;;

[0-9][0-9] echo “you have entered a double digit number”;;

\*) echo “this is default command”

Esac

**WHILE LOOP:**

**Syntax:**

While condition;do

Commands

break

done

Example Script:

Read -p “Enter your number: ” num

While [ “$num” -gt 10 ]; do

Echo “$num”

Num=$(( “$num”-1))

done

**SESSION 7**

Array: Array can store multiple different values at the same time

Numbers=(1 2 3 4 5 6) 🡺 Array called Numbers with length of 6

Accessing 1st element of the array 🡺 $Numbers -- this will store first index value

Accessing any element in Array 🡺 ${Numbers[2]} --- 2(index you want to access)

Accessing all elements in array 🡺 ${Numbers[@]}

Slicing Array:

${Numbers[]:start\_index:length}

Add elements to array:

Numbers+=(6) -- ^ is the element you want to add

Add elements to specific index:

Numbers[1]=a 🡺 This will add a at the starting of the array i.e 1st index

Delete elements from an array:

Unset Numbers[2] 🡺 [2] – index you want to delete from an array