****** GUIDE TO SHELL SCRIPT *******

********** SET USER PERMISSIONS TO MAKE THE SHELL SCRIPTS RUN *********
> values of read, write, execute : r:4 (100 in binary), w:2(010 in binary), x:1(001)
> chmod a+x sample.sh --> 'a' as in grants permissions to all 3 and '+x' grants execute permissions
> chmod a-x sample.sh --> -x removes all the currently held permissions on that script file

Also: to only change permissions on specific user (ex: User or Public or Others), DO the following:

> chmod u+x / chmod u-x : accepts/removes permissions on the User
> chmod g+x / chmod g-x : accepts/removes permissions on the Groups
> chmod o+x / chmod o-x : accepts/removes permissions on the other users (non Users & Non Public)

---------- end of user permissions ----------

* To run shell script, doing ./sript.sh is what we know, however ./ implies we are in the current directory.
* So to, make it run universally, all we have to do is to append PATH name to it in which this shell script resides.

> So to avoid all these hassles, just append the current PWD's path to the streamline PATH environment variable.
> can be acheived by doing

export PATH = $PATH:$(pwd) and then we can normally run the file by just script.sh

* #!/bin/sh or #!/bin/bash mean the same
* # - sharp and ! - bang and #! - shabang

********** VARIABLES **********

> Explicit Definition : VAR = value
> Read Command: read VAR
> command substitution: VAR = $(pwd)

Explicit: VAR=value (no spaces between equal to on either sides) -> Reason for not having spaces in on either sides is
---
due to the fact that the shell treats the arguments surrounding the first one as subsequent arguments.

Example : DOGS_NUM = 8 will be considered as arg[0]: DOGS_NUM arg[1]: = and ard[2]: 8 which is not we want to happen.

******* READ CMD *****
> read VAR --> reads the input from the user from std input
read Age echo $Age

> if read -p is used, then everything will be displayed in one line
if read -sp is used, then what we are inputting is not visible, very useful while typing passwords

> For viewing hostname on Mac, do nano/cat /etc/hosts unlike on windows (/etc/hostname

******** IMPORTANT ******
echo -n "..." implies the input will be fead in the same line as echo if at all a read follows echo , However, this -n responds only in the bash and not sh, so include #!/bin/bash and not #!/bin/sh
****

********** ARGUMENTS **********
> Discusses about arguments passed along with the script name while compilation occurs
> $0 -> script name itself, $1, $2,$3 .... ->subsequent args
> $@ -> includes all arguments $* -> includes all args, however can expand as long as one string $1c$2c$3 .. so not as useful as $@ which does not have any such limitation
> $# -> returns the arg count, total number of arguments.

****************** REDIRECTION AND PIPING **************************
> cat sample.txt > output.txt does transfer output of sample to output text file,

> However as we know stdin - 0 , stdout - 1, stderr - 2

> If we want to output errors of sample.txt program to some errors.txt file, this is what we do:
	cat sample.txt 2 > errors.txt (Since 2 is the stderr number)

> Similarly, if we want to transfer all (0,1,2): then
	cat sample.txt & > output.txt

> And if we want to specifically output to output.txt and errors to error.txt, this is what we do:
	cat sample.txt 1>output.txt 2>error.txt

> Also, if we want to output file to output.txt and then also ouput errors to output.txt, then, do:
	cat sample.txt 1>output.txt 2>&1

> '>' can be used when we want to ouput to a new file, however, if we want to append to an existing file, then
	use '>>' sample.txt >> ouput.txt

********* EXIT STATUS ************
> echo $? -> returns the value of previous command, the return value is called exit status, If cmd is success- $? = 0,
  otherwise, it is non-zero val.

> Exit status repeats cyclically after 255, so valid exit status range (0-255) and 256 ==0, 257==1 ...so on..

********** IF STATEMENTS ************

> String comparisons: ["$STR1" = "$STR2"]

. If string is empty: [-z "$STR1"] -> returns TRUE if the string is empty.
> [-n "$STR1"] -> returns TRUE if the STR1 holds a non-empty string

> [[]] - double brackets notation is specific only to the Bash Shell and not others
> Alphabetical Comparison: [[$STR1 > $STR2]]

***** USER IDs *****
> If one wants to see if a user is typing the commands, we can validate it by UID (user ID) by doing a "echo $UID"
> cat /etc/passwd | grep root gives us the root user ID i.e., 0
> whoami - gives us the USER Name

********** WILDCARDS ********
> ? - A single character Example: hel? -> so matches can be help,hell,hel1 ...so on...
> * - Any number of Characters Example: ca* -> so matches car,carpet,carpenter,car112 ...so on...
> [] - Single Character from range Example: file[0-2],[hd]ello -> so matches can be file0,file1,file2, hello/dello
> {} - Comma seperated terms Example: {*.txt,*.pdf} -> so matches can hello.txt,doc.txt,source.pdf,book.pdf
> [!] - Any character not listed in the brackets Exmple: file[!1] -> so matches can be file2, file3 ...so on..

/* Common Examples (Globbing Patterns): [:upper:] - Uppercase Char, [:lower:] - Lowercase Char, [:alpha:] - Alphabetic Char, [:digit:]- digit char, [:alnum:]- AlphaNumeric Char, [:space:]- WhiteSpace Char */

**** WILDCARDS IN StrING COMPArISON ****
[[$STRING == file[0-9].txt]] or [[$STRING == rich*]]

* NOTE: To create n number of files txt or img or so on.. simple and effective terminal command would be to use for.. like this:

	for i in {1..30}; do touch file$i.txt; done

* To remove or list specific range of files , we can do the following:
	For example: If we want to remove already existing file13-file21.img of images, shell script can be:

		rm file{[1][3-9],[2][0-1]}.img -> this implies, 13-19 and 20-21 can be removed

* --------- A little complex case with wildcards -----
	Example: If we want to list all the files, which has 3 characters long extension starting with t.
		we therefore do: ls *.t?? --> here each ? serves as a predictor and therefore lists all the entities starting with .t
* ---

********** REGULAR EXPRESSIONS **********
> Pattern Matching Ex: Email, ID Ip Addr

> . - Any single character Example: bo.k -> so matches can be book,bo1k,bolk ...so on...

> * - Preceding items must match 0/more items Ex: co*l -> so matches can be cl,cool,col ...so on..
> ^ - Start of the line marker Ex: ^hello -> Line starting with hello
> $ - End of the line marker Ex: hello$ -> line ending with hello
> [] - Any one of the characters enclosed in [] Ex: coo[kl] -> cook or cool
> [-] - Any character within the range Ex: file[1-3] -> file1, file2, file3 ..
> [^] - Any character enclosed within the brackets Ex: file[^0123456789] -> file8,file9
> ? - Preceding items must match one or zero times Ex: colou?r -> color,colour & NOT colouur
> + - Preceding item must match one or more times Ex: file1+ -> file1,file11,file111 & NOT file
> {n} - Preceding item must match n times Ex: [0-9]{3} -> Any 3 digit number: 111,097,255,787..so on...
> {n, } - Min number of times preceding items must match Ex: [0-9]{3, }-> Any 3/more digit number 111,1258,14589..so.
> {n,m} - Min and Max number of times preceding item has to match Ex:[0-9]{1,3}->1,158,26,.. NOT 1258,1111,...
> \ - Escape Character- Esc any of the special char,ignoring meaning of them Ex: hell\.o-> hell.o NOT:helllo,hell10..

/* Common Examples (Globbing Patterns): [:upper:] - Uppercase Char, [:lower:] - Lowercase Char, [:alpha:] - Alphabetic Char, [:digit:]- digit char, [:alnum:]- AlphaNumeric Char, [:space:]- WhiteSpace Char */

********** FILE SYSTEM RELATED TESTS **********

File Test Operators	Description, What it returns
[-e $VAR]	True, if variable holds an existing file (file can be a file or directory)
[-f $VAR]	True, if variable holds an existing regular file (not a directory)
[-d $VAR]	True, if variable holds an existing directory
[-x $VAR]	True, if variable is an executable file
[-L $VAR]	True, if variable holds the path of a symbolic link
[-r $VAR]	True, if variable holding file is a readable one
[-w $VAR]	True, if variable holds the file that is writable

********** FOR LOOPS **********

```
> Usage : for arg in [list]    | for((i=1;i<=10;i++))
                        do                                  |       do
                                command(s)    |                 echo $i
                        done            |           done


**************** WHILE LOOPS *****************

   --- C STYLE WHILE LOOPING -----
> while(( condition ))

Example:
--------

A=1
LIMIT=10

while(( A < LIMIT )) # No $ should appear within this C style while loop
do
        touch $A
        let A++;
done

---------------- READING FILES WITH WHILE LOOPS -------------

#---- METHOD 1 ------

while read line
do
        echo $line
        ....
done < "$FILENAME" # Filename fed as std input to the file

#----- METHOD 2 ------

cat "$FILENAME" |
while read line
do
        echo $line
        ...
done
-----------------------------------------------------------------

************* CASES ****************

Example Case:
------------

case "$(whoami)" in
  "root" )
                        echo "It is the root user"
                        ;;
  "pavankumarpaluri" )
                        echo "It is the user Pavan"
```

```
                        ;;
 * )                      # * represents rest others(! above cases)
                        echo "unknown User"
                        ;;
esac
```

--- Keyword SHIFT ---
shift ; throws away the parameter that is being taken from the user, So for example:

if the user enters… ./files.sh -file temp.txt -h
On the first run, after processing $1 -file temp.txt , first shift throws away "-file" and next shift following it throws away "temp.txt" which essentially means we are left with "-h" and on the next shift it also throws -h as well. then we are left with no arguments.

********** ARRAYS **********

Declarations : ARRAY=(one,two,three)

Calling These Arrays : ${ARRAY[0]} #one | ${ARRAY[1]} #two | ${ARRAY[2]} #three

---

${ARRAY[@]} - Includes all items in the array

${ARRAY[*]} - All items in the array, delimited by first char of IFS

${!ARRAY[@]} - All indexes in the array (@/*) (or) ${!ARRAY[*]}

${#ARRAY[@]} - #number of items in the array (@/*) (or) ${#ARRAY[*]}

${#ARRAY[0]} - Length of item zero

********** FUNCTIONS **********

2 ways of creation -

```
->    function_name()
                {
                        <commands>
                        return -- with this we can only exit the function and not the entire script
                }

->    function function_name
                {
                   local name=$1 -- with this the name is only limited within the scope of the
function
                        <commands>
                }
```

Example using a "local"

```
        function hello{
        local name=$1           -- this local only saves the name locally (example: function.sh)
        echo "Hello $name"
        }
 ------------------------------ SHELL SCRIPT ENDS HERE
------------------------------------------------------------
```

********** AWK SCRIPTS **********

awk 'BEGIN{print "Hello"}' - Beginning Body only used

awk '{print "Hello"}' - keeps printing hello until manually killed

In awk, we only use $ to represent fields as opposed to what we do in shell scripting. In shell scripting we use $ to access variables

For example in AWK: echo "one two three four" | awk '{print $0}' -- prints all the four statements

echo "one two three four" | awk '{print $1}' -- prints only one

echo "one two three four" | awk '{print $2}' -- prints only two and so on....

---------- Tables in AWK ----------

To print only first field using AWK in a table.txt -- do the following

awk '{print $1}'

TO create spaces between fields -- do the following
awk '{print $1 " ", $3, " hello"}' (commas used only when we want to separate the columns with spaces)

---------- Search Patterns using AWK ----------

If we want to display a row info containing 23 in a file, this is how we do it:

cat file.txt | awk '/23/ {print $0}' --- (prints everything belonging to that row having 23)

NOTE: In case , we search with a keyword (IT) and the header (UNIT) has IT in it as well, so to avoid it , we use \t to separate only IT occurances , as shown below:

cat table.txt | awk '/\tIT/' {print $0}'

cat table.txt | awk '!/^Age/ {print $0}' -- Prints without the Headers column (excludes line containing Age | Name and UNit)

---------- String comparison using AWK ----------

cat table.txt | awk ' $3=="Pavan"' -- (to print only those entries that have 3rd field name as "Pavan")

cat table.txt | awk ' $3!="Pavan"' -- (to print only those fields that have their 3rd field as Pavan)

---------- Number of fields using AWK ----------

echo table.txt | awk '{print NF}' -- prints Number of fields in the text
echo table.txt | awk '{print NR}' -- prints number of records or entries or rows in the text

If we only want to see the final entry i.e., last row number that sums up the number of entries. Then. do the following:
echo table.txt | awk 'END{print NR}'

same for number of fields

---------- FIELD SEPARATOR ----------
If we want to separate the fileds using filed separator, this is how we do it and then extract the columns of our choice:

For example for /etc/passwd file it looks like this:
_reportmemoryexception:*:269:269:ReportMemoryException:/var/db/reportmemoryexception:/usr/bin/false

We can use ":" as field separator to extract columns without : and we do it as follows:
cat /etc/passwd | awk 'BEGIN{FS=":"} {print $1 " " $7}'

RESULT: _reportmemoryexception /usr/bin/false

easier command line option to exercise the same is:
cat /etc/passwd | awk -F ":" '{print $1, $3}'

---------- RECORD SEPARATOR ----------
use record separator to separate out the number of records in the /etc/passwd file for example

cat /etc/passwd | awk 'BEGIN{RS=":"} END{print NR}' - prints the count of total number of records with a : separator