**BASH Scripting**

**shebang (#!)**

Shebang is the first line of script it will tell what type of script is this and explain where it will be executed.

Used to tell the operating system kernel which interpreter to use to parse the rest of the file.

#! interpreter [arguments]

* The directive must be the first line in the script.
* The directive must start with shebang #!
* White space after the shebang characters is optional.
* Interpreter is the full path to a binary file (ex: /bin/sh, /bin/bash).
* Interpreter arguments are optional.

Examples:

* #!/bin/bash - Uses bash to parse the file.
* #!/usr/bin/env perl - Uses the env command to find the path to the Perl executable.
* #!/usr/bin/python Executes the file using the python binary.

**Defining and Using Variables**

Variables are the containers which store data or a useful piece of information as the value inside.

variable\_name=value

**Defining Variables**

* No spaces around the equal sign (=)
* Variable names should be alphanumeric and can include underscores (\_)
* Variable names should not contain any operators
* # Correct
* variable=value
* variable\_name=value
* # Incorrect
* variable\_name = value
* variable-name = value
* variable$name = value

variable.name = value

**Accessing Variables**Use the dollar sign ($) before the variable name to access its value.

echo $variable\_name

**Escape character (\)**

* A non-quoted backslash ‘\’ is the Bash escape character.
* It preserves the literal value of the next character that follows and makes it a string only.  
  Below only the character $ will be escaped and treated as a string

>$ echo "\$var"

output: $var

* If a \n pair appears, and the backslash itself is not quoted, a new line will be added.

echo -e "\nThis is to add newline before this echo"

Multiple newlines

echo -e "\nThis is \n to add newline \n before this echo\n"

**env:**

* Pre-defined variables are environmental variables.
* Environment variables control the behaviour of the software packages installed in Linux
* The path where the packages have been installed will be specified in environment.

**Single quote ('') vs Double quote ("")**

var=10

('')

* Considers all inside single quotes as String
* The $var variable output does not get replaced with the variable value but displays the variable name within the quotes, i.e., its literal form.

>$ echo '$var'

output: $var

("")

* Looks for assigned values to variables and executes the content.
* Running the echo command with double quotes expands the $var variable and outputs the assigned value instead of printing the characters within the quotes.

>$ echo "$var"

output: 10

**Special Variables**

These are special shell variables which are set internally by the shell and which are available to the user:

|  |  |
| --- | --- |
| **VARIABLE** | **DESCRIPTION** |
| **$0** | The filename of the current script. |
| **$n** | These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on). User flower braces for double or more digits like ${10}, ${100} |
| **$$** | The process ID of the current shell. For shell scripts, this is the process ID under which they are executing. |
| **$#** | The number of arguments supplied to a script. |
| **$@** | All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.It will print all arguments in a list |
| **$\*** | All the arguments are double quoted. If a script receives two arguments, $\* is equivalent to $1 $2. It will print all arguments in a single line |
| **$?** | The exit status of the last command executed. |
| **$!** | The process ID of the recent process went into background |
| **$\_** | The last argument of the previous command. |

**Integer Comparison (int1 -operator int2)**

|  |  |
| --- | --- |
| **Operator** | **Purpose** |
| **-eq** | Integer equality |
| **-ne** | Integer inequality |
| **-lt** | Integer less than |
| **-le** | Integer less than or equal to |
| **-gt** | Integer greater than |
| **-ge** | Integer greater than or equal to |

**Inode:**

* Every file in a system has a Inode (index node)
* It contains all information of file except file contents and name (just like a person id or passport without a name)

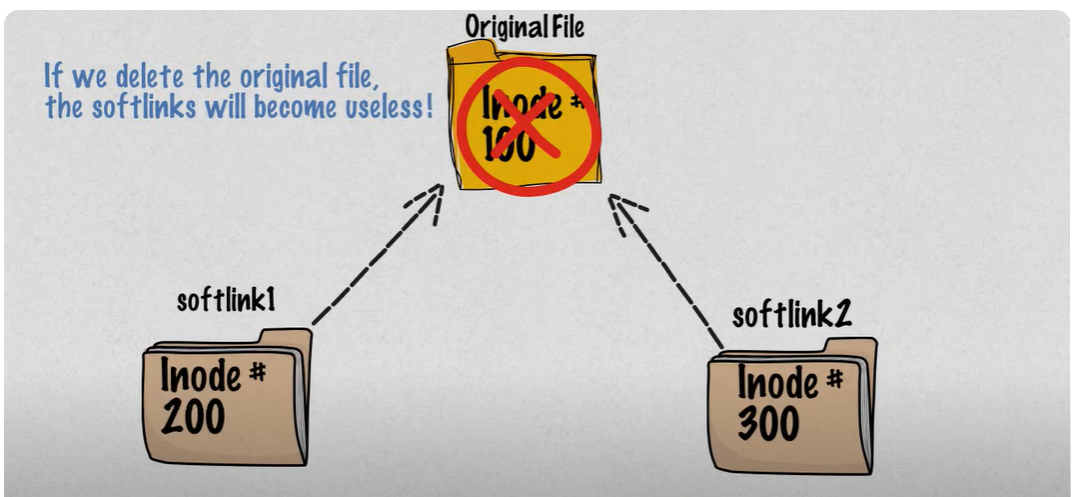
They contain:

* Inode number
* owner information
* number of links
* file type
* file size
* permissions
* etc

**LINKS**

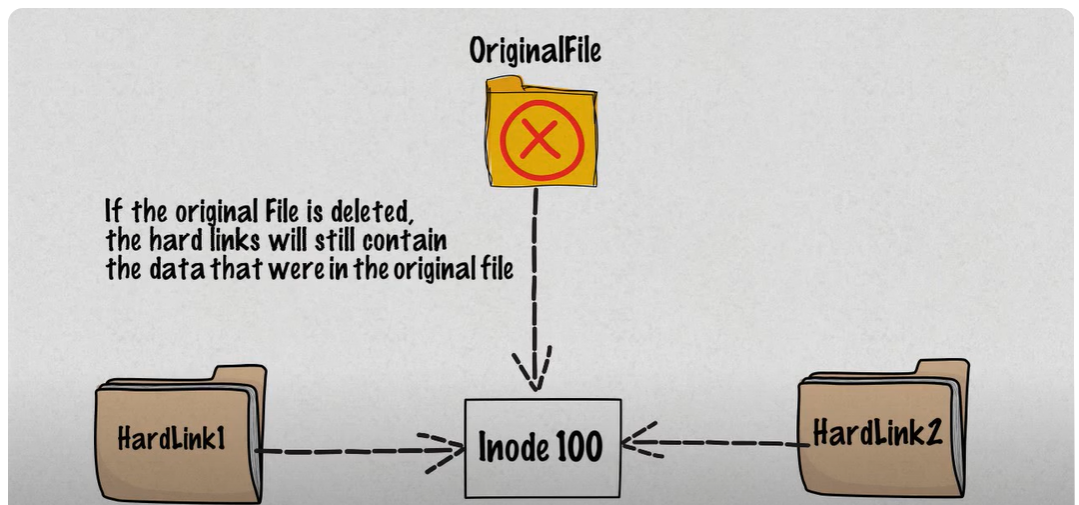
**soft link:**

* Soft link is also called as symbolic links
* It is a pointer to an original file (just link a shortcut in windows)
* But it has a different Inode number
* Soft link files are smaller in size compared to original



**Hard link:**

* Different name of the same file
* Same file size
* Same Inode number



**CREATING A SOFT LINK (SYMBOLIC LINK)**

To create a symbolic link, you use the ln command with the -s option

**ln -s <target> <link\_name>**

* <target>: This is the file or directory you want to link to. It can be specified with either an absolute or relative path.
* <link\_name>: This is the name of the symbolic link you want to create.

**Example:**

1. Create a symbolic link named link\_to\_file that points to original\_file in the current directory:

**ln -s original\_file link\_to\_file**

1. Create a symbolic link named link\_to\_directory that points to original\_directory:

**ln -s /path/to/original\_directory link\_to\_directory**

### **CREATING A HARD LINK**

To create a hard link, you use the ln command without any options:

**ln <target> <link\_name>**

* <target>: This is the existing file you want to link to. It must be specified using its relative or absolute path.
* <link\_name>: This is the name of the hard link you want to create.

**Example:**

1. Create a hard link named hard\_link\_to\_file that points to original\_file in the current directory:

**ln original\_file hard\_link\_to\_file**

1. Create a hard link named hard\_link\_to\_file in another directory:

**ln /path/to/original\_file /path/to/hard\_link\_to\_file**

**Important Notes:**

* Hard links cannot cross filesystem boundaries; both the target and the link must be on the same filesystem.
* Symbolic links can point to directories or files, while hard links can only point to files.

We can use -h and -L operators of the test command to check whether a link is hard or soft (symbolic link).

**-h file //true if the file is a symbolic link**

**-L file //true if the file is a symbolic link**

One can also use:

**readlink FILE; echo $?** // This returns 1 if it's a hard link and 0 if it's a symbolic link.

**Shell scripts can be written for a variety of reasons:**

* Keeping repetitive tasks to a minimum.
* Can be used by system administrators for routine backups.
* Monitoring the system.
* Adding new functions to the shell.
* Shell scripting allows you to create your own tools.
* System admin can automate daily tasks.

**Shell scripting offers the following benefits:**

* An interactive debugging tool, as well as a quick start.
* Programmers need not change their syntax since both command and syntax are identical to those entered directly into the command line.
* Shell scripts are easy to use and quicker to write.
* It helps automate administrative tasks, so it is time-saving.
* As shell scripts are written in an interpreted language, they can be run without any additional effort on almost any modern operating system, including UNIX, Linux, BSD, and Mac OS X.
* They can be utilized for bulk execution rather than single instructions.
* Using it, you can develop your own custom operating system with relevant features.
* Software applications can be developed according to their respective platforms with this tool.

**Shell scripting has the following disadvantages:**

* Errors are frequent and costly, and a single error can alter the command.
* The execution speed is slow.
* Bugs or inadequacies in the language's syntax or implementation.
* Large, complex tasks aren't well suited to it.
* Contrary to other scripting languages, etc., it provides a minimal data structure.
* Every time a shell command is executed, a new process is launched

A file called sh stores shell programs. Sh files contain commands written in a scripting language that is run by Unix shells.

1. **Name the alternative command for echo.**

The tput command is an alternative command for echo.

**If Condition Single Brackets [ ] vs Double Brackets [[ ]]**

* The single brackets [ ] are a symbolic link to the test command, which is a built-in command in most Unix-like operating systems including Bash.
* Single brackets have fewer features compared to double brackets.
* Single brackets are more portable across different shells, while [[ ]] is a Bash-specific construct.
* It’s possible to use the comparison operators with the double brackets in newer versions of BASH-like (>, <, >=, <=, !=) with integers also.

**BASH String Operations  
# Replace first occurrence of 'World' with 'Universe'**  
new\_string=${string//World/Universe}

**# Replace all occurrences of 'o' with '0'**  
new\_string=${string//o/0}

**# Convert to lowercase**  
lowercase=${string,,}

**# Convert to uppercase**  
uppercase=${string^^}

**#number of characters**  
number\_of\_characters=${#string}

**# Extract substring from index 3 and extract 5 characters**  
substring=${string:3:5}

**# Extract the substring from the end**  
substring=${string: -5}

**Comparison Operators**

* **Equal (==) or (=):** This operator tests if two strings are equal.
* **Not Equal (!=):** It checks if two strings are not equal.
* **Greater Than (>):** This operator compares two strings alphabetically and checks if the first string is greater than the second.
* **Less Than (<):** It compares two strings alphabetically and checks if the first string is less than the second.
* **Greater Than or Equal To (>=):** This operator checks if the first string is greater than or equal to the second.
* **Less Than or Equal To (<=):** It checks if the first string is less than or equal to the second.
* **Pattern matching (=~):** This operator checks if a string matches a specific pattern using regular expressions.

#!/bin/bash

string="$1"

#To check if string variable is equal to 'hi'

if [[ ${string,,} == 'hi' ]]; then

echo "string are equal"

fi

#To check if string variable contains/substring 'hi'

if [[ ${string,,} == \*"hi"\* ]]; then

echo "$string contains hi"

fi

#To check if string variable contains/substring 'hi'

if [[ ${string,,} =~ "hi" ]]; then

echo "$string contains hi"

fi

**To execute a command and store its output in a variable using command substitution.**

$(...) syntax

#!/bin/bash

# Execute the command and store its output in a variable

output1=$(ls -lrt)

echo "$output1"

`...` syntax (backticks)

#!/bin/bash

# Execute the command and store its output in a variable

output=`ls -lrt`

echo "$output"

* The $() syntax is generally considered more readable than backticks.
* $() syntax is easier to understand at a glance and would be less error-prone if you were to add more commands or nesting.

**comments  
Single-Line Comments:**

* You can add single-line comments using the # character.
* Everything after # on the same line is considered a comment.

#!/bin/bash

# This is a single-line comment

echo "Hello, world" # This is also a comment

#!/bin/bash

# This is a first-line comment

# Here's the second line of the comment

# And the third line of comment

echo "Hello, world"

**Multi-Line Comments:**

#!/bin/bash

: '

This is a long explation

multi comment

in bash

'

**IF OPTIONS**

* **Empty string (-z):**This operator checks if a string is empty.
* **Non-empty string (-n):**This operator checks if a string is non-empty.
* **Directory exists (-d):**To check if Directory exists and it is a directory**.**
* **File exists (-f):**To check if FILE exists and it is a directory.
* **Read permission (-r):**To check if FILE exists and the read permission is granted**.**
* **File size (-s):**To check if FILE exists and its size is greater than zero (which means that it is not empty).
* **Read permission (-w):**To check if FILE exists and the write permission is granted.
* **Read permission (-x):**To check if FILE exists and the execute permission is granted.

**For loop**

* **for:** This keyword initiates the for loop.
* v**ariable:** This is a variable that will hold each value from the list as the loop iterates. You can choose any variable name you like.
* **in:** This keyword separates the variable from the list of values.
* **list:** This is a list of values that the loop will iterate over. It can be an array, a sequence of numbers, or a list of strings separated by spaces/lines.
* **do:** This keyword marks the beginning of the loop body.
* **commands to execute:** These are the commands that will be executed in each iteration of the loop. You can put any valid bash commands here.
* **done:**This keyword marks the end of the loop.

**for loop BASH syntax**

for variable in list

do

# commands to execute

done

**for loop using an space seperated of string;**

#!/bin/bash

for n in a b c; do

echo $n

done

**for loop using an Array of string**

#!/bin/bash

declare -A fruits

fruits=(

[0]="Apple"

[1]="Mango"

[2]="Banana"

[3]="Orange"

)

for i in ${fruits[@]}; do

echo "fruits $i"

done

**Range-based for loop**

#!/bin/bash

for n in {1..10}; do

echo $n

done

#!/bin/bash

# Print odd number series

for n in {1..15..2}; do

echo $n

done

**C Style for loop**

#!/bin/bash

for (( i=1 ; i<=10 ; i++ )); do

echo $i

done

**Infinite for loop**

#!/bin/bash

n=0

for (( ; ; )); do

n=`expr $n + 1`

[ $n -eq 99 ] && exit

echo $n

done

**for loop on a command output**

#!/bin/bash

for i in $(find $1 -type f -iname "\*.sh"); do

if [[ -x "$i" ]]; then

echo "EXECUTABLE $(basename $i)"

else

echo "NOT EXECUTABLE $i"

sudo chmod 777 $i

if [[ -x "$i" ]]; then

echo "Changed the permission to EXECUTABLE: $i"

fi

fi

done

**while loop**

A **while loop** is a statement that iterates over a block of code till the condition specified is evaluated to false.

A while loop will run untill the condition is true and once condition is false it will stop.

Syntax

while [ condition ];

do

# statements

# commands

done

example

#!/bin/bash

var=7

while [[ $var -gt 0 ]]

do

echo $var

((var--))

done

while loop to read a file line by line

#!/bin/bash

while read line; do

echo $line

done < log.txt

**PROGRAMS**

**IF CONDITION [input comparison using if, else if and else]**

#!/bin/bash

if [ $# -eq 2 ];then

    echo "entered input is $1 and $2"

if [ $1 -gt $2 ];then

    echo "bigger:$1"

    echo "smaller:$2"

elif [ $1 -eq $2 ];then

    echo "two numbers are equal"

elif [ $1 -lt $2 ];then

    echo "bigger:$2"

    echo "smaller:$1"

fi

else

    echo "enter two inputs"

fi

**o**utput🡪./ifcondition.sh 5 2

bigger:5

smaller:2

**String Comparison**

**Search string (string = “hi” then print output)**

#!/bin/bash

string=$1

    echo "this is the input string: $string"

if [[ ${string,,} == "hi"]];then

    echo "Hi how are you"

fi

**output🡪 ./string.sh hi**

**hi how are you**

**Replace string pattern**

#!/bin/bash

string="hi how are you"

echo "${string}"

variable=$(echo ${string} | sed "s/o/l/g")

echo "$variable"

**output🡪 hi hlw are ylu**

**substring comparison**

#!/bin/bash

string="hi how are you"

if [[ ${string,,} == \*"hi"\* ]];then

    echo "string is have hi"

fi

**Arithmetic operations**

#!/bin/bash

if [[ $# -eq 2 ]];then

    if[[ $1 =~ ^[0-9]+$ ]];then

    if[[ $2 =~ ^[0-9]+$ ]];then

        add=$(echo "$1 + $2" | bc)

            echo "addition: $add"

        sub=$(expr $1 -$2)

            echo "substraction: $sub"

        mul=$(echo "$1 \* $2" | bc)

            echo "multiplication: $mul"

        div=$(expr $1 / $2)

            echo "division: $div"

        mod=$(expr "$1 % $2" | bc)

            echo "modulas: $mod"

    fi

    else

        echo "enter only numbers"

    fi

else

    echo "enter only 2 input"

fi

output🡪 ./calculation.sh 6 2

addition:8

subtraction: 4

multiplication:12

division: 3

modules: 0

**simple calculator program**

#!/bin/bash

echo "Simple Calculator"

echo "-----------------"

echo -n "Enter first number: "

read num1

echo -n "Enter second number: "

read num2

echo "Choose operation:"

echo "1. Addition"

echo "2. Subtraction"

echo "3. Multiplication"

echo "4. Division"

echo "5. Exit"

echo -n "Enter your choice: "

read choice

if [ $choice -eq 1 ]; then

    result=$((num1 + num2))

    echo "Result: $result"

elif [ $choice -eq 2 ]; then

    result=$((num1 - num2))

    echo "Result: $result"

elif [ $choice -eq 3 ]; then

    result=$((num1 \* num2))

    echo "Result: $result"

elif [ $choice -eq 4 ]; then

    if [ $num2 -eq 0 ]; then

        echo "Error: Division by zero"

    else

        result=$(echo "scale=2; $num1 / $num2" | bc)

        echo "Result: $result"

    fi

elif [ $choice -eq 5 ]; then

    echo "Exiting..."

else

    echo "Invalid choice. Please enter a valid option."

fi

output:

Simple Calculator

-----------------

Enter first number: 5

Enter second number: 5

Choose operation:

1. Addition

2. Subtraction

3. Multiplication

4. Division

5. Exit

Enter your choice: 3

Result: 25

**To check file or directory**

#!/bin/bash

input=$1

if [ -z $input ];then

echo "pass input"

else

if [[ -e $1 ]];then

if [[ -f $1 ]];then

echo "input is file "

fi

if [[ -d $1 ]];then

echo "input is directory"

fi

else

echo "enter correct path"

fi

fi

**for loop**

#!/bin/bash

for i in {1..5}

do

    echo "$i"

done

**output🡪 1**

**2**

**3**

**4**

**5**

**Sum of 5 natural numbers:**

#!/bin/bash

sum=o

for i in {1..5}

do

sum=$(echo "$i + $sum" | bc)

done

echo "total:$sum"

**sum of n natural numbers**

#!/bin/bash

sum=0

for i in $\*

do

sum=$(expr $i + $sum)

done

echo " total:$sum

**Copy all the file ending with .txt to directory by checking weather it exits or not**

#!/bin/bash

if [[ -d ./backup]];then

for i in \*.txt

do

cp $i ./backup

done

**To check file or directory**

#!/bin/bash

for i in $(ls)

do

echo "$i"

if [[ -d $i ]];then

echo "$i is a directory"

elif [[ -f $i ]];then

echo "$i is a file"

fi

done

**prime numbers between range**

#!/bin/bash

for num in {11..20}

do

        is\_prime=true

        for(( i=2; i<=num/2; i++ ))

        do

                if [ $((num % i)) -eq 0 ];then

                       is\_prime=false

               break

                fi

        done

if $is\_prime;then

        echo "$num"

fi

**Question: Write a Bash script that counts the number of files (excluding directories) in the current directory.**

#!/bin/bash

count=0

for file in \*

do

    if [ -f "$file" ]; then

        (( count++ ))

    fi

done

echo "Total files in the directory: $count"

### 2. Sum of Even Numbers

**Question**: Write a Bash script that calculates the sum of all even numbers from 1 to a given number n.

#!/bin/bash

echo -n "Enter a number (n): "

read n

sum=0

# Loop through numbers from 1 to n

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

do

    # Check if the number is even

    if [ $((i % 2)) -eq 0 ]; then

        (( sum += i ))  # Add the even number to sum

    fi

done

echo "Sum of even numbers from 1 to $n is: $sum"

### 3. Fibonacci Sequence

**Question**: Write a Bash script that prints the Fibonacci sequence up to a specified number of terms.

#!/bin/bash

echo -n "read fibo seriers "

read num

a=0

b=1

for(( i=1; i<=num; i++))

do

echo -n "$a"

next=$(a+b)

a=$b

b=$next

done

echo "fibonacci series are :$next

### 4. Factorial Calculation

**Question**: Write a Bash script that calculates the factorial of a given number n.

#!/bin/bash

echo -n "enter a number"

read n

factorial=1

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

do

factorial \*= i

done

echo "factorial is: $factorial"

### 5. File Backup Script

**Question**: Write a Bash script that copies all files from a specified directory to a backup directory, preserving the directory structure.

#!/bin/bash

echo -n "Enter source directory: "

read source\_dir

echo -n "Enter backup directory: "

read backup\_dir

# Create backup directory if it doesn't exist

mkdir -p $backup\_dir

# Copy files preserving directory structure

cp -r $source\_dir/\* $backup\_dir/

echo "Backup completed."

 **Question: Write a Bash script to list all files in a directory and its subdirectories recursively.**

Answer:

#!/bin/bash

# Function to list files recursively

list\_files() {

    local dir="$1"

    local indent="$2"

    for file in "$dir"/\*; do

        if [ -f "$file" ]; then

            echo "${indent}${file}"

        elif [ -d "$file" ]; then

            echo "${indent}${file}/"

            list\_files "$file" "${indent}  "

        fi

    done

}

# Starting point: current directory

list\_files "." ""

Explanation: This script defines a function list\_files that recursively lists all files and directories in the specified directory ($dir). It uses indentation ($indent) to show the directory structure visually.

 **Question: Explain how you would process command-line arguments in a Bash script.**

Answer:

#!/bin/bash

# Check if no arguments were provided

if [ $# -eq 0 ]; then

    echo "Usage: $0 <arg1> <arg2> ..."

    exit 1

fi

# Loop through each argument

while [ $# -gt 0 ]; do

    echo "Argument: $1"

    shift

done

Explanation: $# represents the number of arguments passed to the script. shift is used to iterate through each argument ($1, $2, etc.) until no arguments are left ($# becomes 0).

 **Question: Explain how you would read a file line by line in Bash and perform an operation on each line.**

Answer:

#!/bin/bash

filename="example.txt"

# Check if file exists and is readable

if [! -r "$filename”]; then

    echo "Error: Cannot read $filename"

    exit 1

fi

# Read file line by line

while IFS= read -r line; do

    echo "Processing line: $line"

    # Perform operation on each line (example: echo the line)

done < "$filename"

Explanation: The while loop reads each line from the file specified by $filename using read -r line. Inside the loop, you can perform operations on each line as needed.

 **Question: Write a Bash script that counts the number of occurrences of each word in a text file and prints the results sorted by frequency.**

#!/bin/bash

filename="textfile.txt"

# Check if file exists and is readable

if [ ! -r "$filename" ]; then

    echo "Error: Cannot read $filename"

    exit 1

fi

# Count occurrences of each word and store in associative array

declare -A word\_count

while IFS= read -r line; do

    # Convert line to lowercase and split into words

    words=($line)

    for word in "${words[@]}"; do

        word="${word,,}"  # Convert to lowercase

        ((word\_count[$word]++))

    done

done < "$filename"

# Print results sorted by frequency

for word in "${!word\_count[@]}"; do

    echo "$word: ${word\_count[$word]}"

done | sort -rn -k2

### **Find the largest number in an array of integers**

#!/bin/bash

# Array of integers

numbers=(10 5 20 15 8)

# Initialize max with the first element of the array

max=${numbers[0]}

# Loop through the array to find the maximum number

for num in "${numbers[@]}"

do

    if (( num > max )); then

        max=$num

    fi

done

echo "The largest number in the array is: $max"

**Explanation:**

* Define an array numbers with integers.
* Initialize max with the first element of the array.
* Iterate through the array using a for loop.
* Compare each element with max and update max if the current element is greater.
* Print the largest number after the loop completes.

**Count the number of lines in a file specified by a filename argument**

#!/bin/bash

filename="$1"

# Check if filename argument is provided

if [ -z "$filename" ]; then

    echo "Usage: $0 <filename>"

    exit 1

fi

# Check if the file exists and is readable

if [ ! -f "$filename" ]; then

    echo "Error: $filename does not exist or is not a regular file"

    exit 1

fi

# Count the number of lines in the file

num\_lines=$(wc -l < "$filename")

echo "Number of lines in $filename: $num\_lines"

**Explanation:**

* Accept the filename as the first argument ($1).
* Check if the filename argument is provided.
* Check if the file exists and is a regular file (-f).
* Use wc -l < "$filename" to count lines in the file.
* Print the number of lines.

### **Function to check if a string is a palindrome**

#!/bin/bash

# Function to check if a string is a palindrome

is\_palindrome() {

    local input=$1

    local reversed=$(echo "$input" | rev)

    if [ "$input" = "$reversed" ]; then

        echo "Palindrome"

    else

        echo "Not a palindrome"

    fi

}

# Usage example

input\_string="madam"

is\_palindrome "$input\_string"

**Explanation:**

* Define a function is\_palindrome that takes a string ($1) as input.
* Use rev command to reverse the string ($(echo "$input" | rev)).
* Compare the original string with the reversed string to determine if it's a palindrome.
* Print "Palindrome" or "Not a palindrome" based on the comparison.

### **Script to read a directory and print all file names with a .txt extension**

#!/bin/bash

directory="$1"

# Check if directory argument is provided

if [ -z "$directory" ]; then

    echo "Usage: $0 <directory>"

    exit 1

fi

# Check if the directory exists

if [ ! -d "$directory" ]; then

    echo "Error: $directory is not a directory or does not exist"

    exit 1

fi

# List all .txt files in the directory

txt\_files=$(find "$directory" -type f -name "\*.txt")

# Print the list of .txt files

echo "Text files in $directory:"

echo "$txt\_files"

**Explanation:**

* Accept the directory path as the first argument ($1).
* Check if the directory argument is provided.
* Check if the directory exists (-d).
* Use find command to search for all .txt files (-type f -name "\*.txt").
* Print the list of .txt files found in the directory.

### 5. Handling Errors and Exceptions in Bash Script

Error handling in Bash scripts involves checking conditions that might lead to errors, such as missing files or invalid user input. Here are some common techniques:

* **Checking Command Exit Status:** Use $? to check the exit status of the last command. Typically, 0 indicates success, and non-zero values indicate failure.
* **Conditional Statements (if):** Use if statements to check conditions and handle errors gracefully. For example:

bash

Copy code

if [ condition ]; then

# Handle success case

else

# Handle error case

fi

* **Error Messages and Exiting:** Print error messages to stderr (>&2) and exit with a non-zero status (exit 1):

bash

Copy code

echo "Error: File not found" >&2

exit 1

* **Parameter and Argument Validation:** Validate script arguments and parameters to ensure they meet expectations:

bash

Copy code

if [ -z "$1" ]; then

echo "Usage: $0 <filename>"

exit 1

fi

* **Using set -e for Immediate Exit on Error:** Add set -e at the beginning of the script to exit immediately if any command exits with a non-zero status:

bash

Copy code

set -e

* **Trap Command for Cleanup:** Use trap to set up a cleanup routine when a script exits due to an error or upon receiving a signal:

bash

Copy code

trap 'cleanup\_function' EXIT

### Method 1: Using rev command

The rev command in Bash reverses lines of a file or input from stdin. This can be used directly to reverse a string:

bash

Copy code

#!/bin/bash

# Function to reverse a string using `rev` command

reverse\_string() {

local input=$1

local reversed=$(echo "$input" | rev)

echo "$reversed"

}

# Example usage

input\_string="hello world"

reversed\_string=$(reverse\_string "$input\_string")

echo "Original string: $input\_string"

echo "Reversed string: $reversed\_string"

**Explanation:**

* rev is a command-line utility that reverses lines of text.
* In the reverse\_string function, echo "$input" | rev pipes the input string into rev to reverse it.
* The reversed string is stored in the reversed variable and then echoed back.

### Method 2: Iterative approach without rev

If you want to reverse a string without using rev, you can iterate through the string manually:

bash

Copy code

#!/bin/bash

# Function to reverse a string without using `rev`

reverse\_string() {

local input=$1

local len=${#input}

local reversed=""

for (( i=$len-1; i>=0; i-- )); do

reversed="$reversed${input:$i:1}"

done

echo "$reversed"

}

# Example usage

input\_string="hello world"

reversed\_string=$(reverse\_string "$input\_string")

echo "Original string: $input\_string"

echo "Reversed string: $reversed\_string"

**Explanation:**

* len=${#input} gets the length of the input string.
* The for loop iterates from the last character to the first ($len-1 to 0).
* ${input:$i:1} extracts the character at position $i from input.
* Characters are appended to the reversed variable to build the reversed string.

### Notes:

* Both methods are effective for reversing strings in Bash.
* The rev command is straightforward and concise but may not be available on all systems.
* The iterative approach provides more control and flexibility, especially in environments where rev is not available.

**Input will be 4 numbers u should check if number is even or odd if its even add and if its odd multiply and give the result as subtract of these two**

#!/bin/bash

 echo -n "enter inputs"

 read n

 add=0

 mul=1

 for i in $n

 do

         if [ $((i % 2)) -eq 0 ]

         then

                 echo "$i is even number"

                 add=$(( i + $add ))

         else

                 echo "$i is odd number"

                 mul=$((i \* $mul))

         fi

 done

 result=$(($add - $mul))

 echo "the result us $result"