Ultimate Bash Scripting Handbook

~Sagar Biswas

PREFACE

Welcome to the Ultimate Bash Scripting Handbook, your essential guide to automating tasks and managing Unix-based systems with Bash. Whether you're a beginner or an experienced developer, this handbook will help you streamline workflows, manage servers, and enhance your command-line expertise.

You'll start with the basics and progress to advanced topics, including loops, functions, and file manipulation. Each chapter is packed with practical examples, giving you the skills and hands-on experience needed to optimize your workflow and fully leverage the power of the shell environment.

☑ WHAT IS BASH SCRIPTING?

Bash scripting is the process of writing a series of commands in a script that the Bash shell (Bourne Again SHell) can execute. These scripts automate tasks, helping save time and minimize errors compared to doing things manually. Just like any other programming language, Bash scripts provide the system with instructions, allowing users to perform complex operations quickly and reliably.

☑ FEATURES OF BASH SCRIPTING

- **Easy to learn and use:** Ideal for beginners who are familiar with command-line interfaces.
- **Automates tasks:** Saves time by automating repetitive tasks, boosting productivity.
- **Free and open-source:** Available by default on most Unix-based systems, no extra installation required.
- **Portable:** Scripts run smoothly across various Linux/Unix distributions and macOS with minimal changes.
- **Integration with other tools:** Easily works with other command-line utilities and programming languages.
- **Script Scheduling:** Automate tasks by scheduling scripts to run at specific times using tools like cron.
- **Lightweight:** No need for extra libraries or frameworks, ensuring fast execution.

☑ WHY BASH SCRIPTING?

Bash scripting is an incredibly powerful tool for automating tasks on Unix-like operating systems. It helps streamline repetitive tasks, manage system configurations, and execute complex workflows with ease. System administrators, developers, and anyone working with Linux/Unix systems rely on Bash scripting to manage servers, process files, and perform other administrative duties. It's an essential skill for anyone looking to work efficiently in a Unix environment.

☑ **Bash Scripting -- high-level or low-level language?**

Bash scripting is generally considered a high-level language, but it can also be used to perform some low-level tasks, making it a versatile tool. Here's a breakdown:

1. Bash as a High-Level Language: Bash scripting is mainly a high-level language but can also handle low-level tasks. Here is how:

1. Bash as a High-Level Language

- **Abstracts low-level details:** Bash simplifies system interactions, so you don't need to manage memory or hardware directly.
- **Automates tasks:** Automates operations like file handling, process management, and system configuration.
- **Easy to read:** Commands like ls, cp, mv, and grep make Bash scripts simple to understand.

2. Bash for Low-Level Tasks

While primarily high-level, Bash can manage low-level operations:

- **Process control:** Commands like kill, ps, and `top interact directly with the OS.
- **System calls:** Execute low-level system calls using tools like strace or lsof.
- **Hardware and network management:** Use commands like ifconfig and ip for device and network tasks.
- **File permissions:** Control file permissions and ownership with commands like chmod and chown.

Summary

Bash is a high-level language that simplifies automation but can also handle low-level tasks when needed, offering flexibility for a wide range of scripting needs.

Part 1: (Hello World) Introduction

The **shebang line** (#!) at the start of a Bash script specifies the interpreter that should execute the script. In our example, we use /bin/bash as the interpreter.

Code Example

```
#!/bin/bash

# Author: Sagar Biswas
# Description: Bash Variables
# Date: 09/11/2024

# The -p option with the read command displays a prompt message ("Enter your name: ")
before waiting for the user to input their name.

read -p "Enter Hello World: " texts
```

```
echo "..:: $texts!"
```

Output

```
Enter Hello World: Hello World
..:: Hello World!
```

Part 2: Comments

Overview

Comments are essential for explaining the code. Bash supports two types of comments:

1. **Single-line comments** start with #. Everything after the `#` on that line is ignored by the shell.
2. **Multi-line comments** can use specific syntax (example: '..' or <<comment ... comment).

Code Example

```
#!/bin/bash

# Author: Sagar Biswas
# Description: Single and Multi-line Comments
# Date: 09/11/2024

# Single-line comment example
x=200  # Variable to demonstrate comments
echo "This is a single-line comment example."

# Method 1: Using the : ' ... ' syntax # Best Practices for Comments
: '
This is a multi-line comment.
You can write multiple lines here.
'
# Method 2: Using the <<comment ... comment syntax
<<comment
This is another way to write
multi-line comments in Bash.
comment
echo "Multi-line comments are shown above."
```

Part 3: Quotation

Overview

Quotations in Bash are used to handle strings and text. They determine how special characters, variables, and escape sequences are interpreted.

Type	Behavior	Example
Double Quotes	Expands variables and interprets escape sequences.	"Hello $name"
Single Quotes	Treats the string literally; no variable expansion or escape sequence interpretation.	'Hello $name'
Unquoted	Works but can cause issues with spaces or special characters; not recommended.	Hello World
Here Document	Useful for printing complex multi-line strings with preserved formatting.	cat <<EOF ... EOF

Code Example

```
#!/bin/bash

# Author: Sagar Biswas
# Description: Using Quotations in Strings
# Date: 09/11/2024

# ----------------------------------------
# Double Quotes: Expands variables and escape sequences
# ----------------------------------------
x=42
echo "Hello World"           # Double quotes for standard strings
echo "The value of x is: $x" # Variables will be expanded
echo -e "Newline Example:\nHello\nWorld"  # Enables escape sequences with -e

# ----------------------------------------
# Single Quotes: Literal strings
# ----------------------------------------
echo 'Hello World'           # Single quotes treat the string literally
echo 'The value of x is: $x' # Variables are not expanded inside single quotes

# ----------------------------------------
# Unquoted Strings: Not recommended
# ----------------------------------------
echo Hello World             # Works, but avoid for special characters/spaces
echo Hello "World"           # May require manual quoting for clarity

# ----------------------------------------
# Multi-line Strings
# ----------------------------------------
# Using double quotes
echo -e "\nPrinting a multi-line string using double quotes:\nHELLO\nWORLD"

# Using a Here Document
cat <<EOF
This is a multi-line string printed using a Here Document.
It is helpful for preserving formatting.
    HELLO
    WORLD
EOF
```

Part 4: Variables

Overview

Variables are temporary storage units for data like integers, strings, and floats.

- **Valid variable names**: Start with letters or underscores and can include digits.
- **Invalid variable names**: Start with digits, or include special characters (example: @, !, -).

Types of Variables

1. **Local Variables**: Accessible only within the script or function.
2. **Environment Variables**: Shared across scripts or programs using export.

Code Example

```
#!/bin/bash

# Author: Sagar Biswas
# Description: Bash Variables
# Date: 09/11/2024
```

```
# ----------------------------------------
# Local Variables
# ----------------------------------------
name="Sagar Biswas"  # Assigning a value to the name variable
age=22               # Assigning a value to the age variable

# Displaying the variables
echo "My name is $name and my age is $age"

# Unset command: Deleting variables
unset name  # Removing the 'name' variable
unset age   # Removing the 'age' variable

# Trying to print unset variables (will be empty)
echo "My name is $name and my age is $age"

# ----------------------------------------
# Environment Variables
# ----------------------------------------
# Setting an environment variable with export
export The_Path="/mnt/m/Programming --Learning/Bash --Learning$"  # Path with special
characters and spaces

# Displaying the environment variable
echo "The path is: $The_Path"

# Unsetting the environment variable
unset The_Path

# Trying to print unset environment variable (will be empty)
echo "The path is: $The_Path"
```

Output

```
My name is Sagar Biswas and my age is 22
My name is  and my age is
The path is: /mnt/m/Programming --Learning/Bash --Learning$
The path is:
```

Part 5: System-defined Variables

Overview

Bash provides pre-defined variables to access system information.

Variable	Description
$HOME	Home directory
$PWD	Current working directory
$BASH	Path to the Bash shell
$BASH_VERSION	Version of the Bash shell
$OSTYPE	Type of operating system

Code Example

```
#!/bin/bash

echo $HOME          # Home directory
echo $PWD           # Current working directory
echo $BASH          # Path to Bash
echo $BASH_VERSION  # Bash version
```

```
echo $OSTYPE        # Operating system type
```

Output:

```
/home/sagar_biswas
/mnt/m/Programming --Learning/Bash --Learning/1_Basic
/bin/bash
5.2.21(1)-release
linux-gnu
```

Part 6: Arrays

Overview

Arrays in Bash store multiple values in a single variable.

- **Initialization**: array_name=(value1 value2 value3 ...)
- **Accessing values**: ${array_name[index]}
- **Iterating**: Use for loops.

Code Example

```
#!/bin/bash

# Author: Sagar Biswas
# Description: Using Arrays in Bash
# Date: 09/11/2024

# Array Initialization
Name[0]="Sagar"
Name[1]="Asif"
Name[2]="Sakib"
Name[3]="Shrabon"

# Accessing Array Elements
echo "First Name: ${Name[0]}"   # Access the first element
echo "Second Name: ${Name[1]}"  # Access the second element

# Print all elements of the array
echo "All Names: ${Name[@]}"    # Using [@] to get all elements

# Iterate through the array
echo "Iterating through the array:"
for i in "${Name[@]}"; do
    echo "$i"                   # Print each element
done

# Array Initialization (Alternative)
gang=("Sagar" "Asif" "Sakib" "Shrabon")  # Alternative array initialization
echo "Gang Members: ${gang[*]}"  # Using [*] to print all members

# Array Length
echo "Number of elements in the 'Name' array: ${#Name[@]}"  # Get length of the array
echo "Number of elements in the 'gang' array: ${#gang[@]}"  # Get length of the array
```

Output

```
First Name: Sagar

Second Name: Asif

All Names: Sagar Asif Sakib Shrabon

Iterating through the array:
```

```
Sagar
Asif
Sakib
Shrabon
Gang Members: Sagar Asif Sakib Shrabon
Number of elements in the 'Name' array: 4
Number of elements in the 'gang' array: 4
```

Part 7: Operators

Overview

Operators in Bash are used to perform operations on variables and values.

Category	Operators	Example
Arithmetic Operators	+, -, *, /, %	a=10; b=20; echo $((a + b))
Relational Operators	-eq, -ne, -lt, -gt, -le, -ge	[$a -lt $b]
Logical Operators	&&, `	
String Operators	==, !=, -z, -n	[-z "$str"]
File Test Operators	-e, -f, -d, -r, -w, -x, -s	[-f "$file"]

Code Example

```bash
#!/bin/bash

# Author: Sagar Biswas
# Description: Demonstrating Operators in Bash
# Date: 09/11/2024

echo -e "\n--- Arithmetic Operations ---"
a=10
b=20
echo "The sum of $a and $b is: $((a + b))"
echo "The modulus of $a and $b is: $((a % b))"

echo -e "\n--- Relational Operators ---"
if [ $a -lt $b ]; then
    echo "$a is less than $b"
fi

echo -e "\n--- Logical Operators ---"
if [ $a -lt 15 ] && [ $b -gt 15 ]; then
    echo "Condition met: $a < 15 and $b > 15"
fi

echo -e "\n--- String Operators ---"
str1="hello"
str2="world"
if [ "$str1" != "$str2" ]; then
    echo "$str1 is not equal to $str2"
fi

echo -e "\n--- File Test Operators ---"
file="/path/to/your/file.txt"
if [ -e "$file" ]; then
```

```
Output:

--- Arithmetic Operations ---
The sum of 10 and 20 is: 30
The modulus of 10 and 20 is: 10

--- Relational Operators ---
10 is less than 20

--- Logical Operators ---
Condition met: 10 < 15 and 20 > 15

--- String Operators ---
hello is not equal to world

--- File Test Operators ---
/path/to/your/file.txt does not exist
```

```
    echo "$file exists"
else
    echo "$file does not exist"
fi
```

Part 8: If-Else-If Statements

Overview

Bash supports conditional logic using **if-else-if** statements to execute specific code blocks based on conditions.

Operator	Description
-eq, -ne	Equal, Not equal
-lt, -gt	Less than, Greater than
-le, -ge	Less than or equal, Greater than or equal

Code Example

```
#!/bin/bash

# Author: Sagar Biswas
# Description: Demonstrating If-Else-If Statements
# Date: 10/11/2024

a=10
b=20

if [ $a -eq $b ]; then
    echo "$a is equal to $b"
elif [ $a -lt $b ]; then
    echo "$a is less than $b"
else
    echo "$a is greater than $b"
fi
```

Output:

```
10 is less than 20
```

Part 9: While Loops

Overview

While loops execute a block of code repeatedly as long as the condition evaluates to true.

Key Features	Description
Syntax	while [condition]; do ... done
Infinite Loop	Ensure the condition eventually becomes false

Code Example

```
#!/bin/bash

# Author: Sagar Biswas
# Description: Using While Loops
# Date: 09/11/2024
```

```
a=5
while [ $a -lt 10 ]; do
    echo -n "$a "  # Print numbers on the same line
    a=$((a + 1))
done

echo  # Print a newline
```

Output:

```
5 6 7 8 9
```

Part 10: Until Loops

Overview

Until loops execute a block of code repeatedly until the condition becomes true.

Key Differences (While vs Until)
While : Executes while the condition is true.
Until : Executes until the condition is true.

Code Example

```
#!/bin/bash

# Author: Sagar Biswas
# Description: Using Until Loops
# Date: 09/11/2024

i=0
until [ $i -ge 10 ]; do
    echo "Welcome $i times"
    i=$((i + 2))
done
```

Output:

```
Welcome 0 times
Welcome 2 times
Welcome 4 times
Welcome 6 times
Welcome 8 times
```

Part 11: For Loops

Overview

The **for loop** in Bash provides various ways to iterate over items, ranges, and outputs. Below is an overview of its syntaxes and behaviors:

Syntax	Description
for var in item1 item2 ...	Iterates through a list of items.
for var in {start..end}	Iterates through a numeric range.
for var in {start..end..step}	Iterates through a numeric range with steps.

for var in $(command)	Iterates through command output.
for ((init; condition; inc))	C-style for loop for numeric iteration.

Code Example

```bash
#!/bin/bash

# Author: Sagar Biswas
# Description: Demonstrates different uses of 'for' loops in Bash
# Date: 09/11/2024

# Traditional for loop with C-style syntax
echo -e "\n========= Traditional For Loop ========="
for ((i = 1; i <= 10; i++)); do
    echo "Welcome $i times"
done

# Simple for loop iterating through specified values
echo -e "\n========= Simple For Loop ========="
for i in 1 2 9 10; do
    echo "Welcome $i times"
done

# List all files in the current directory
echo -e "\n========= List Files in Current Directory ========="
for file in *; do
    echo "$file"
done

# Range-based for loop
echo -e "\n========= Range-based For Loop (1 to 5) ========="
for i in {1..5}; do
    echo "Welcome $i times"
done

# Step-based for loop
echo -e "\n========= Step-based For Loop (step by 2) ========="
for i in {1..10..2}; do
    echo "Welcome $i times"
done

# Using `seq` for step-based loops
echo -e "\n========= For Loop Using Seq ========="
for i in $(seq 1 2 10); do
    echo "Welcome $i times"
done
# Infinite loop with continue and break
echo -e "\n========= Infinite Loop with Continue and Break ========="
i=0
for ((; ;)); do
    if [ $i -eq 5 ]; then
        ((i++))
        continue
    fi
    if [ $i -eq 10 ]; then
        break
    fi
    echo -n "$i "
    ((i++))
done
echo
```

Output

```
========= Traditional For Loop =========
Welcome 1 times
Welcome 2 times
...
Welcome 10 times

========= Simple For Loop =========
Welcome 1 times
Welcome 2 times
Welcome 9 times
Welcome 10 times

========= List Files in Current Directory =========
file1.sh
file2.sh
...

========= Range-based For Loop (1 to 5) =========
Welcome 1 times
Welcome 2 times
...
Welcome 5 times

========= Step-based For Loop (step by 2) =========
Welcome 1 times
Welcome 3 times
...
Welcome 9 times

========= For Loop Using Seq =========
Welcome 1 times
Welcome 3 times
...
Welcome 9 times

========= Infinite Loop with Continue and Break =========
0 1 2 3 4 6 7 8 9
```

Part 12: Command Line Arguments

Overview

Bash provides special variables to handle arguments passed to a script:

Variable	Description
$0	Name of the script.
$1, $2, ...	Positional arguments.
$#	Number of arguments.
$*	All arguments as a single string.
$@	All arguments as an array.

Code Example

```
#!/bin/bash

# Author: Sagar Biswas
# Description: Command Line Arguments in Bash
# Date: 09/11/2024
```

```
echo -e "\nHow to use command line arguments?"
echo "================================="

echo "Script Name: $0"
echo "First Argument: $1"
echo "Second Argument: $2"
echo "Quoted \$@: \"$@\""
echo "Quoted \$*: \"$*\""
echo "Total Arguments: $#"

# Looping through $*
echo -e "\nUsing \$*: "
count=0
for arg in $*; do
    echo "Argument $count: $arg"
    count=$((count + 1))
done

# Looping through $@
echo -e "\nUsing \$@: "
count=0
for arg in "$@"; do
    echo "Argument $count: $arg"
    count=$((count + 1))
done
```

Input

```
./commandLineArgs.sh 10 20 30 40
```

Output

```
How to use command line arguments?
=================================
Script Name: ./commandLineArgs.sh
First Argument: 10
Second Argument: 20
Quoted $@: "10 20 30 40"
Quoted $*: "10 20 30 40"
Total Arguments: 4

Using $*:
Argument 0: 10
Argument 1: 20
Argument 2: 30
Argument 3: 40

Using $@:
Argument 0: 10
Argument 1: 20
Argument 2: 30
Argument 3: 40
```

Part 13: Functions

Overview

Functions in Bash allow grouping reusable logic into callable blocks. Parameters are passed as positional arguments ($1, $2, etc.), and functions can optionally return values.

Feature	Description
function name {}	Defines a function using the function keyword.
name() {}	Defines a function without the function keyword.

$1, $2, ...	Positional arguments passed to the function.
return	Explicitly returns a value (optional).
local	Declares a variable local to the function.

Code Example

```bash
#!/bin/bash

# Author: Sagar Biswas
# Description: Functions in Bash
# Date: 09/11/2024

function myFun1() {
    echo "Hello World 01: $1 $2"
}

myFun2() {
    echo "Hello World 02: $1"
}

myFun3() {
    echo "Sum of $1 and $2 is $(( $1 + $2 ))"
}

# Calling functions
myFun1 "Arg1" "Arg2"
myFun2 "Arg3"
myFun3 5 10
```

Output

```
Hello World 01: Arg1 Arg2
Hello World 02: Arg3
Sum of 5 and 10 is 15
```

Part 14: File Operations with while Loops

Overview

This section demonstrates how to use while loops for reading and writing to files in Bash.

Operation	Description
Reading files (line by line)	Uses while loop with or without IFS for line-by-line processing.
Writing to a file	Overwrites file contents using > operator.
Appending to a file	Appends content using >> operator.

Code Example

```bash
#!/bin/bash

# Author: Sagar Biswas
# Description: Demonstrates the use of while loops to read and write files in Bash
# Date: 09/11/2024

# Reading a file line by line using IFS
echo "Reading a file line by line using IFS:"
input="./text1.txt"
```

```bash
while IFS= read -r line; do
    echo "$line"
done < "$input"

# Reading a file line by line without using IFS
echo -e "\nReading a file line by line without using IFS:"
while read -r line; do
    echo "$line"
done < "$input"

# Writing to a file
output="./text.txt"
echo -e "\nWriting 'Hello World' to $output."
echo "Hello World" > "$output"

# Appending to a file
echo "Appending 'Hello Sagar' to $output."
echo "Hello Sagar" >> "$output"

# Appending the output of a command to a file
echo "Appending the output of 'ls' to $output."
ls >> "$output"
```

Output

Assuming the contents of text1.txt are:

```
Line 1: Hello Bash
Line 2: Welcome to scripting


Reading a file line by line using IFS:
Line 1: Hello Bash
Line 2: Welcome to scripting

Reading a file line by line without using IFS:
Line 1: Hello Bash
Line 2: Welcome to scripting

Writing 'Hello World' to ./text.txt.
Appending 'Hello Sagar' to ./text.txt.
Appending the output of 'ls' to ./text.txt.
```

Part 15: Command History

Overview

Command history in Bash allows you to re-run, edit, or reference past commands using **history expansion**.

Expansion	Description
!!	Runs the most recent command again.
!$	References the last argument of the previous command.
!^	References the first argument of the previous command.
!:n	References the n-th word of the previous command.
!:n-m	References words n through m of the previous command.

Code Example

```bash
#!/bin/bash
```

```
# Author: Sagar Biswas
# Description: Command History
# Date: 09/11/2024

<<comment
Another way of accessing the history is given by the so-called history expansion. Example:

!! run the most recent command again
!$ the last argument of the previous command line
!^ the first argument of the previous command line
!: the n-th word of the previous command line
!:n-m words n till m of the previous command line
comment


# Example commands to demonstrate history expansion (For the best understanding copy one by
one and run in terminal)

echo "1. This is the first test"
# The 'set' command assigns values to positional parameters ($1, $2, $3, etc.)
# In this case, we are setting $1 to 12, $2 to 14, and $3 to 16
set -- 12 14 16
echo "2. Second command with: $1 $2 $3"

# Test for !! (last command of the previous commands)
echo "The last command was: !!" # Output: The last command was: echo "2. Second command with:
$1 $2 $3" 12 14 16

# Test for !$ (last argument of the previous command)
echo "Last argument was: !$" # Output: Last argument was: 16

# Test for !^ (first argument of the previous command)
echo "First argument was: !^" # Output: First argument was: 2. Second command with:

# Test for !: (n-th argument of the previous command)
echo "Second argument of the last command was: !:2" # Output: Second argument of the last
command was: 12

# Test for !:n-m (Arguments n to m from the previous command)
echo "Arguments 2 to 3 of the last command: !:2-3" # Output: Arguments 2 to 3 of the last
command: 12 14
```

Output: `(For the best understanding copy one by one and run in terminal)`

Part 16: Redirection Commands

Overview

File redirection in Bash allows you to direct standard input/output/error to or from files.

Operator	Description
>	Redirects standard output, overwriting the file.
>>	Redirects standard output, appending to the file.
2>	Redirects standard error to a file.
2>>	Appends standard error to a file.
&>>	Redirects both standard output and error to a file.

Code Example

```
#!/bin/bash
```

```
# Author: Sagar Biswas
# Description: Uses of file redirection
# Date: 09/11/2024

# Write to the file, overwriting its contents
echo "Hi! I am Sagar Biswas." > text.txt  # Overwrites text.txt with this message

# Append to the file
echo "Hello! Bangladesh." >> text.txt  # Appends to text.txt

# Redirect error output (stderr) to the file (won't do anything here unless there's an error)
echo "Hello World" 2>> text.txt  # No error occurred, so nothing is appended for this line # Print "Hello World" in the terminal after executing the .sh file
echo "Hello World" &>> text.txt  # Appends both stdout and stderr to text.txt (includes both success and error messages)
nonexistent_command &>> text.txt  # Appends both error message and any output (if there is any) to text.txt

# Execute command and redirect error output (stderr) to the file
ls /3_ConditionalStatements 2>> text.txt  # Will append an error if this directory doesn't exist

# Generate an error and append the error message to the file
datex 2>> text.txt  # This command will cause an error and append it to text.txt

# Redirect both stdout and stderr to the file
date &>> text.txt  # Appends both the output of the date command and any errors (if they occur) to text.txt

# What are the differences between 2>> and &>>?

# 2>>:
# - Redirects only error output (stderr) to a file.
# - Used when you want to capture error messages without affecting normal output.
# - Example: capturing only errors, like `ls nonexistent_folder 2>> error_log.txt`.

# &>>:
# - Redirects both standard output (stdout) and error output (stderr) to the same file.
# - This is useful when you want to capture both successful outputs and error messages in a single file.
# - Example: capturing both normal output and errors together, like `command &>> output_and_error.txt`.
```

Output (Contents of text.txt)

```
Hi! I am Sagar Biswas.
Hello! Bangladesh.
Hello World
./function.sh: line 16: nonexistent_command: command not found
ls: cannot access '/3_ConditionalStatements': No such file or directory
./function.sh: line 22: datex: command not found
Tue Nov 26 15:18:46 +06 2024
```

Part 16: What is grep?

grep stands for **Global Regular Expression Print**. It's used to search for specific patterns in text files or input streams. It prints the lines that match the given pattern.

1. Basic Syntax of grep:

```
grep [options] pattern [file...]
```

- **pattern**: The string or regular expression to search for.

- **file**: One or more files to search through. If no file is specified, grep reads from the standard input (stdin).

2. Basic Usage Examples:

- **Search for a word in a file**:

```
grep "hello" text.txt
```

This searches for the word "hello" in text.txt and prints lines containing the word "hello".

- **Search for a word in multiple files**:

```
grep "hello" file1.txt file2.txt
```

This searches for "hello" in both file1.txt and file2.txt.

- **Case-sensitive search** (default):

```
grep "hello" text.txt
```

- **Case-insensitive search (-i option)**:

```
grep -i "hello" text.txt
```

This matches "hello", "Hello", "HELLO", etc.

- **Invert match (-v option)**:

```
grep -v "hello" text.txt
```

This prints lines **not** containing "hello".

- **Count the number of matching lines (-c option)**:

```
grep -c "hello" text.txt
```

This prints the number of lines that contain "hello".

- **Show line numbers of matches (-n option)**:

```
grep -n "hello" text.txt
```

This displays the line numbers along with the matching lines.

- **Show only the matched part of the line (-o option)**:

```
grep -o "hello" text.txt
```

This prints only the word "hello" wherever it occurs, instead of the entire line.

3. Advanced Options:

- **Recursive search (-r or -R option):**

```
grep -r "hello" /path/to/directory
```

This searches for "hello" in all files within the specified directory and its subdirectories.

- **Search for whole words only (-w option)**:

```
grep -w "hello" text.txt
```

This matches "hello" as a whole word and not as part of other words like "hellooo" or "hell".

- **Display file names only (-l option)**:

```
grep -l "hello" *.txt
```

This lists only the names of files that contain "hello".

- **Display lines that don't match (-L option)**:

```
grep -L "hello" *.txt
```

This lists files that **don't** contain "hello".

- **Search for patterns with extended regular expressions (-E option)**:

```
grep -E "hel+o" text.txt
```

This allows using extended regular expressions, so you can use +, ?, and {n,m} without escaping them.

4. Regular Expressions (Regex) in grep:

grep uses regular expressions (regex) to define patterns. A **regular expression** is a sequence of characters that defines a search pattern.

- **Special characters in regular expressions**:
  - . : Matches any single character.
  - * : Matches zero or more of the preceding character.
  - ^ : Anchors the match to the beginning of the line.
  - $: Anchors the match to the end of the line.
  - [] : Matches any single character in a set.
  - | : OR operator, to match one pattern or another.
  - \b : Word boundary.
- **Examples of regex patterns**:
  - ^hello : Matches lines starting with "hello".
  - hello$: Matches lines ending with "hello".
  - h.llo : Matches "hello", "hxllo", "hallo", etc. (any character between "h" and "llo").
  - [a-z] : Matches any lowercase letter.
  - \d : Matches any digit (if using -P for Perl regex).
  - hel|world : Matches either "hel" or "world".

5. Combining Options:

You can combine multiple options for more powerful searches.

- **Case-insensitive and count matches**:

```
grep -ic "hello" text.txt
```

This counts the number of case-insensitive matches for "hello".

- **Display line numbers, exclude matches, and use regular expressions**:

```
grep -v -n -E "hel+o" text.txt
```

This excludes lines that match "hel+o" and displays line numbers for the rest.

6. Example Use Cases:

- **Find all Python files in a directory**:

```
grep -r -l ".py" /path/to/directory
```

This will list all Python files by searching for .py in the specified directory and subdirectories.

- **Search for specific error messages in logs**:

```
grep -i "error" /var/log/syslog
```

This searches for any occurrence of "error" in the syslog file (case-insensitive).

- **Find all lines containing numbers**:

```
grep "[0-9]" text.txt
```

This matches all lines that contain any digit.

7. Summary of Common grep Options:

Option	Description
-i	Ignore case sensitivity
-v	Exclude lines that match the pattern
-c	Count the number of matching lines
-n	Show line numbers along with the lines
-l	List filenames that contain the pattern
-L	List filenames that do **not** contain the pattern
-r or -R	Recursive search in directories
-w	Match whole words only
-o	Show only the matched parts of the line
-E	Use extended regular expressions
-P	Use Perl-compatible regular expressions
-x	Match the whole line exactly

Conclusion:

grep is an extremely versatile tool for searching text using patterns. It supports simple string searches, as well as more advanced regular expressions for complex pattern matching. It is often used for log analysis, text processing, and automation tasks.

Bash Script: Demonstrating grep Usage

```bash
#!/bin/bash

# Author: Sagar Biswas
# Description: Demonstrating the uses of grep command
# Date: 26/11/2024

# Create a sample file
echo "Creating a sample file 'sample.txt'..."
cat <<EOL > sample.txt
Hello World
HELLO Linux
hello shell scripting
This is a test file
12345 is a number
Special symbols: @#!$
Another line with hello
EOL
echo "File 'sample.txt' created!"

# 1. Search for a simple pattern (case-sensitive)
echo -e "\n1. Searching for 'hello' (case-sensitive):"
grep "hello" sample.txt

# 2. Search for a pattern (case-insensitive)
echo -e "\n2. Searching for 'hello' (case-insensitive):"
grep -i "hello" sample.txt

# 3. Exclude lines that match a pattern
echo -e "\n3. Excluding lines with 'hello':"
grep -v "hello" sample.txt

# 4. Count the number of lines matching a pattern
echo -e "\n4. Counting lines with 'hello':"
grep -c "hello" sample.txt

# 5. Show line numbers of matches
echo -e "\n5. Showing line numbers for 'hello':"
grep -n "hello" sample.txt

# 6. Match whole words only
echo -e "\n6. Matching whole word 'hello':"
grep -w "hello" sample.txt

# 7. Use regular expressions: Lines starting with "hello"
echo -e "\n7. Matching lines starting with 'hello':"
grep "^hello" sample.txt

# 8. Match lines ending with "hello"
echo -e "\n8. Matching lines ending with 'hello':"
grep "hello$" sample.txt

# 9. Search for lines with digits
echo -e "\n9. Searching for lines with digits:"
grep "[0-9]" sample.txt

# 10. Search recursively in a directory (if needed)
# Uncomment and modify the path below for demonstration
# echo -e "\n10. Searching for 'hello' recursively in the current directory:"
# grep -r "hello" .

# Cleanup
echo -e "\nCleaning up..."
rm sample.txt
echo "Done!"
```

Part 16: What is awk?

awk is a powerful programming language and text-processing tool designed for tasks such as pattern scanning, data extraction, and report generation. It is widely used for manipulating and analyzing structured data in text files.

1. What is awk?

- **awk** is a text-processing language named after its creators **Aho, Weinberger, and Kernighan**.
- It processes input line by line, splitting each line into **fields** based on a delimiter (default: spaces or tabs).
- It allows:
    - Searching for patterns.
    - Performing operations on matched lines.
    - Generating formatted output.

2. Basic Syntax

```
awk 'pattern {action}' file
```

- **pattern**: A condition to match specific lines.
- **action**: Commands to execute on matching lines. Actions are enclosed in {}.
- **file**: The input file(s) to process. If omitted, awk reads from stdin.

3. Basic Usage Examples

a) Print all lines in a file:

```
awk '{print}' file.txt
```

b) Print specific fields (columns):

```
awk '{print $1, $3}' file.txt
```

$1: First field.

$3: Third field.

c) Filter lines by a pattern:

```
awk '/pattern/' file.txt
```

Prints lines containing pattern.

d) Perform arithmetic operations:

```
awk '{sum = $2 + $3; print sum}' file.txt
```

Calculates the sum of fields 2 and 3 for each line.

4. Common awk Options

Option	Description
-F	Specifies the field delimiter.
-v	Assigns a variable to use in the script.
-f	Reads awk script from a file.

--help	Displays help for awk.

5. Field and Record Variables

- **Field Variables**: $n (example: $1, $2, etc.)
  - **$0:** Entire line.
  - **$1:** First field.
  - **$2:** Second field, and so on.
- **Record Variables**:
  - **NR:** Current record number (line number).
  - **NF:** Number of fields in the current record.
  - **FS:** Field separator (default: space/tab).
  - **OFS:** Output field separator (default: space).
  - **RS:** Record separator (default: newline).
  - **ORS:** Output record separator (default: newline).

6. Pattern Matching

a) Match a string:

```
awk '/hello/' file.txt
```

b) Match lines by line number:

```
awk 'NR == 2' file.txt
```

Prints the second line.

c) Match using relational operators:

```
awk '$3 > 50' file.txt
```

Prints lines where the third field is greater than 50.

d) Match using logical operators:

```
awk '$3 > 50 && $2 == "Linux"' file.txt
```

Matches lines where field 3 is greater than 50 **and** field 2 equals "Linux".

7. Built-in Functions

String Functions:

Function	Description
length(s)	Returns the length of string s.
substr(s, p, n)	Extracts n characters from string s starting at position p.
tolower(s)	Converts string s to lowercase.
toupper(s)	Converts string s to uppercase.

index(s, t)	Returns the position of string t in string s.
split(s, a, d)	Splits string s into array a using delimiter d.

Mathematical Functions:

Function	Description
int(x)	Returns the integer part of x.
sqrt(x)	Returns the square root of x.
rand()	Generates a random number between 0 and 1.
srand(x)	Seeds the random number generator with x.

8. Advanced Examples

1. **Print lines with a specific field value**:

   ```
   awk '$2 == "Linux"' file.txt
   ```

2. **Print line numbers with each line**:

   ```
   awk '{print NR ": " $0}' file.txt
   ```

3. **Print the last field of each line**:

   ```
   awk '{print $NF}' file.txt
   ```

4. **Print lines where a field matches a regular expression**:

   ```
   awk '$1 ~ /^[A-Z]/' file.txt
   ```

   Matches lines where the first field starts with an uppercase letter.

5. **Calculate the sum of a column**:

   ```
   awk '{sum += $3} END {print "Total:", sum}' file.txt
   ```

6. **Custom field delimiter (-F)**:

   ```
   awk -F "," '{print $1, $3}' file.csv
   ```

   Uses a comma as the field delimiter.

7. **Generate a report with headers**:

```
awk 'BEGIN {print "Name\tScore"} {print $1, "\t", $2} END {print "Report End"}' file.txt
```

8. **Extract unique values**:

   ```
   awk '!seen[$1]++' file.txt
   ```

   Prints unique lines based on the first field.

9. Writing Scripts with awk

You can save complex awk commands in a script file.

Example: Script File script.awk

```
BEGIN {
    print "Name\tScore"
```

```
}

$3 > 50 {
    print $1, "\t", $3
}

END {
    print "Processing complete."
}
```

Run the Script:

```
awk -f script.awk file.txt
```

10. Summary of awk

Feature	Description
Pattern Matching	Search for lines using conditions or regular expressions.
Field Manipulation	Process specific fields in each line.
Arithmetic Operations	Perform mathematical calculations.
String Processing	Modify or analyze text using built-in string functions.
Input/Output Formatting	Format and control input/output data.

Conclusion

awk is an essential tool for anyone working with text processing, data analysis, or scripting. It is highly versatile, making it suitable for simple tasks like filtering data as well as complex report generation.

Bash Script: Demonstrating awk

```
#!/bin/bash

# Author: Your Name
# Description: Demonstrating the uses of awk command
# Date: 26/11/2024

# Step 1: Create a sample file
echo "Creating a sample file 'data.txt'..."
cat <<EOL > data.txt
Name,Age,Score,Country
Alice,25,88,USA
Bob,30,75,UK
Charlie,22,92,Canada
David,35,65,Australia
Eve,28,90,India
Frank,40,70,USA
EOL
echo "File 'data.txt' created!"

# Step 2: Demonstrate various awk operations

# 1. Print the entire file
echo -e "\n1. Printing the entire file:"
awk '{print}' data.txt

# 2. Print specific columns (fields)
echo -e "\n2. Printing Name and Country columns:"
awk -F "," '{print $1, $4}' data.txt
```

```
# 3. Count the number of lines (excluding the header)
echo -e "\n3. Counting the number of data rows:"
awk 'NR > 1 {count++} END {print "Number of rows:", count}' data.txt

# 4. Calculate the average score
echo -e "\n4. Calculating the average score:"
awk -F "," 'NR > 1 {sum += $3; count++} END {print "Average Score:", sum / count}' data.txt

# 5. Filter rows based on conditions (Score > 80)
echo -e "\n5. Rows where Score > 80:"
awk -F "," 'NR > 1 && $3 > 80 {print $0}' data.txt

# 6. Print lines with a specific pattern (Country = USA)
echo -e "\n6. Rows where Country is USA:"
awk -F "," 'NR > 1 && $4 == "USA" {print $1, $2, $3}' data.txt

# 7. Use regex to match patterns (Names starting with A)
echo -e "\n7. Rows where Name starts with 'A':"
awk -F "," 'NR > 1 && $1 ~ /^A/ {print $0}' data.txt

# 8. Add a header and format output
echo -e "\n8. Adding a header and formatting output:"
awk -F "," 'BEGIN {printf "%-10s %-5s %-10s\n", "Name", "Age", "Score"}
            NR > 1 {printf "%-10s %-5s %-10s\n", $1, $2, $3}' data.txt

# 9. Find the maximum score
echo -e "\n9. Finding the maximum score:"
awk -F "," 'NR > 1 {if ($3 > max) max = $3} END {print "Maximum Score:", max}' data.txt

# 10. Find unique countries
echo -e "\n10. Finding unique countries:"
awk -F "," 'NR > 1 {countries[$4]++} END {for (country in countries) print country}' data.txt

# Cleanup
echo -e "\nCleaning up..."
rm data.txt
echo "Done!"
```

How to Run

1. Save the script as awk_demo.sh.
2. Make it executable:

```
chmod +x awk_demo.sh
```

Run the script:

```
./awk_demo.sh
```

Output

```
Creating a sample file 'data.txt'...
File 'data.txt' created!

1. Printing the entire file:
Name,Age,Score,Country
Sagar,25,88,USA
Sarah,30,75,UK
Shisher,22,92,Canada
Asif,35,65,Australia
Eva,28,90,India
Kamal,40,70,USA

2. Printing Name and Country columns:
Name Country
Sagar USA
```

```
Sarah UK
Shisher Canada
Asif Australia
Eva India
Kamal USA

3. Counting the number of data rows:
Number of rows: 6

4. Calculating the average score:
Average Score: 80

5. Rows where Score > 80:
Sagar,25,88,USA
Shisher,22,92,Canada
Eva,28,90,India

6. Rows where Country is USA:
Sagar 25 88
Kamal 40 70

7. Rows where Name starts with 'A':
Asif,35,65,Australia

8. Adding a header and formatting output:
Name        Age    Score
Sagar       25     88
Sarah       30     75
Shisher     22     92
Asif        35     65
Eva         28     90
Kamal       40     70

9. Finding the maximum score:
Maximum Score: 92

10. Finding unique countries:
USA
UK
Canada
India
Australia

Cleaning up...
Done!
```

Part 17: Error Handling in Bash

Error handling ensures that your Bash scripts can manage unexpected issues gracefully, avoiding crashes or incomplete tasks. Bash offers several tools to manage errors effectively, making your scripts more robust and reliable.

1. Using trap

The trap command allows you to specify a command or function to execute when the script exits or when it receives a specific signal. This is especially useful for cleaning up temporary files, closing network connections, or rolling back changes.

Syntax:

```
trap 'commands' SIGNAL
```

Example: Cleaning up temporary files on exit

```
#!/bin/bash
trap 'rm -f /tmp/tempfile' EXIT
```

```
# Create a temporary file with some content
echo "This is a temporary file." > /tmp/tempfile
echo "Temporary file created and populated."

# Pause for manual inspection
sleep 10

# The file will be deleted automatically when the script exits
echo "Script completed."
```

Steps to Check:

1. Run the script.
2. Open a separate terminal and type:

   ```
   cat /tmp/tempfile
   ```

   If the file exists, you will see its contents in the output.
3. After 10 seconds (or when the script ends), repeat the ls command. The file should no longer exist.

Explanation:

- trap 'rm -f /tmp/tempfile' EXIT: Ensures the temporary file is deleted when the script exits, either normally or due to an error.

2. Exiting on Error (set -e)

The set -e command instructs the script to exit immediately if any command returns a non-zero (error) status. This is helpful to avoid continuing execution when something goes wrong.

Example: Exiting on error

```
#!/bin/bash
set -e  # Exit immediately if any command fails

echo "Starting the script."

# The cp command copies the file 'nonexistent_file.txt' to the directory '/tmp/'.
# If 'nonexistent_file.txt' does not exist, the command will fail with an error:
# "cp: cannot stat 'nonexistent_file.txt': No such file or directory".

# Simulating a command that fails
cp nonexistent_file.txt /tmp/

echo "This line will not be executed if the previous command fails."

<<comment

    Output:

    Starting the script.
    cp: cannot stat 'nonexistent_file.txt': No such file or directory

comment
```

Explanation:

- If cp nonexistent_file.txt /tmp/ fails, the script will exit immediately, and the message "This line will not be executed if the previous command fails." will not be printed.

3. Custom Error Messages Using ||

You can handle errors for specific commands by using || to execute a custom error-handling command when a failure occurs. This allows you to provide meaningful feedback to the user.

Example: Custom error message

```
#!/bin/bash

cp source.txt destination.txt || echo "Copy failed: Check if the files exist."

<<comment

    Output:

  cp: cannot stat 'source.txt': No such file or directory
  Copy failed: Check if the files exist.

comment
```

Explanation:

- If cp source.txt destination.txt fails, the custom error message "Copy failed: Check if the files exist." will be displayed.

Improved Example: Logging errors to a file

```
#!/bin/bash

log_file="error.log"

cp source.txt destination.txt 2>> $log_file || {
  echo "Error: Failed to copy source.txt to destination.txt."
  echo "Check $log_file for more details."
}

<<comment

    Output:

    Error: Failed to copy source.txt to destination.txt.
    Check error.log for more details.

Comment
```

4. Combining Techniques for Robust Error Handling

You can combine these techniques to create a more robust script.

Example: Comprehensive error handling

```
#!/bin/bash
set -e  # Exit on error
trap 'cleanup' EXIT  # Ensure cleanup on exit

cleanup() {
  echo "Cleaning up..."
  sleep 50
  rm -f /tmp/tempfile
  echo "Cleanup completed."
}
```

```
# Create a temporary file
touch /tmp/tempfile

# Simulating commands
echo "Starting the script."

# Intentional error for demonstration
cp error.log /tmp/ || echo "Error: File copy failed."

echo "Script completed successfully."
```

Steps to Check:

1. Run the script.
2. Open a separate terminal and type:

   ```
   echo "File content:"; cat /tmp/error.log
   ```

   If the file exists, you will see its contents in the output.
3. After 10 seconds (or when the script ends), repeat the ls command. The file should no longer exist.

5. Quick Tip: Using set -o Options

Bash provides additional set options for better error handling:

- set -e: Exit on any command failure.
- set -u: Treat unset variables as an error.
- set -o pipefail: Return an error if any command in a pipeline fails.

Example: Using multiple set options

```
#!/bin/bash
set -euo pipefail

echo "This script will exit on any error, unset variable, or pipeline failure."
```

Example: set -u

```
#!/bin/bash
set -u  # Treat unset variables as an error

echo "The value of UNSET_VAR is: $UNSET_VAR"
echo "This line will not be executed."

# Output: ./set-u.sh: line 4: UNSET_VAR: unbound variable
```

Example: Pipeline

Pipeline --

```
echo "apple orange banana apple" | tr ' ' '\n' | sort | uniq
# sort: Sorts the words alphabetically.
# uniq: Removes duplicates, leaving only unique words.
```

Example: set -eo pipefail

```
#!/bin/bash
set -eo pipefail  # Exit on error, and on pipeline failure
```

```
# Simulate a pipeline where an earlier command fails
cat nonexistent_file.txt | grep "text" | sort

echo "This line will not execute if any part of the pipeline fails."# because of set -e
```

Summary

1. **trap**: Execute cleanup or recovery commands on script exit or specific signals.
2. **set -e**: Exit immediately if a command fails.
3. **Custom error messages**: Use || to provide feedback or perform alternative actions.
4. **Combine techniques**: For robust and reliable scripts, mix and match error-handling tools as needed.

By implementing these techniques, your Bash scripts will be more reliable and maintainable, handling unexpected situations gracefully.

Part 18: Debugging Scripts in Bash

Debugging helps identify and fix issues in your Bash scripts by tracing the flow of execution and inspecting variable values. Bash provides several built-in tools and techniques to assist in debugging, making it easier to pinpoint errors and unexpected behavior.

1. Debugging Options

a. Enable Debugging with -x

The -x option displays each command along with its expanded arguments as it is executed. This helps track the flow of the script and see how variables are being evaluated.

Syntax:

```
bash -x script.sh
```

Example:

```
#!/bin/bash
x=5
y=10
result=$((x + y))
echo "Result is $result"
```

Run the script with debugging enabled:

```
bash -x script.sh
```

Output:

```
+ x=5
+ y=10
+ result=15
+ echo 'Result is 15'
Result is 15
```

b. Enable Debugging with -v

The -v option displays the script lines as they are read, which is useful for understanding the script's structure and flow.

Syntax:

```
bash -v script.sh
```

Example:

```
#!/bin/bash
echo "This is a test script."
echo "The script will now end."
```

Run the script with debugging enabled:

```
bash -v script.sh
```

Output:

```
#!/bin/bash
echo "This is a test script."
This is a test script.
echo "The script will now end."
The script will now end.
```

2. Inline Debugging

Sometimes, you might not want to debug the entire script. Instead, you can enable debugging for specific sections using set -x to start debugging and set +x to stop it.

Syntax:

```
set -x  # Enable debugging
# Commands to debug
set +x  # Disable debugging
```

Example: Debugging specific sections

```
#!/bin/bash

echo "Starting script."

set -x
x=10
y=5
result=$((x * y))
echo "Multiplication result: $result"
set +x

echo "Script finished."
```

Run in terminal by **./script.sh**

Output:

```
Starting script.
+ x=10
+ y=5
+ result=50
+ echo 'Multiplication result: 50'
Multiplication result: 50
Script finished.
```

3. Print Variable Values

Printing variable values at different points in the script is a simple yet effective way to debug.

Example: Printing variable values

```
#!/bin/bash

name="Alice"
age=25

echo "Name is $name"
echo "Age is $age"
```

Output:

Name is Alice

Age is 25

You can also use printf for more formatted output:

```
printf "Name: %s\nAge: %d\n" "$name" "$age"
```

4. Debugging Tips

a. Check for Syntax Errors

Use the -n option to check for syntax errors without executing the script.

Syntax:

```
bash -n script.sh
```

Example:

```
#!/bin/bash
echo "Missing closing quote
```

Run the script with:

```
bash -n script.sh
```

Output:

```
script.sh: line 2: unexpected EOF while looking for matching `"'
script.sh: line 3: syntax error: unexpected end of file
```

5. Advanced Debugging Techniques

a. Use trap to Debug Errors

You can set a trap to capture errors and display useful information before exiting.

Example:

```
#!/bin/bash

trap 'echo "Error occurred at line $LINENO"; exit 1' ERR

# Intentional error for demonstration
ls nonexistent_file.txt

echo "This will not be printed if an error occurs."
```

Output:

```
ls: cannot access 'nonexistent_file.txt': No such file or directory
Error occurred at line 6
```

b. Debug Pipelines with set -o pipefail

By default, Bash does not detect errors in pipelines. Use set -o pipefail to ensure errors in any part of a pipeline are detected.

Example:

```
#!/bin/bash
set -o pipefail
grep "text" nonexistent_file.txt | sort
```

Summary

1. **-x**: Displays each command and its expanded arguments as they execute.
2. **-v**: Displays script lines as they are read.
3. **Inline Debugging**: Use set -x and set +x to debug specific sections.
4. **Print Variables**: Use echo or printf to inspect variable values.
5. **Syntax Checking**: Use bash -n to check for syntax errors.
6. **Test in Chunks**: Isolate sections of the script for easier debugging.

By combining these techniques, you can efficiently debug and troubleshoot your Bash scripts, making them more reliable and error-free.

Patr 19: Security Best Practices in Bash

Bash scripts often handle sensitive tasks, making security a critical consideration. Following security best practices ensures your scripts are safe from vulnerabilities like code injection, privilege escalation, and data leaks.

1. Input Validation

Always validate user inputs to prevent malicious or unexpected data from being processed by your script. Use conditional checks or regular expressions (regex) to validate inputs.

Example: Validating Numeric Input

```
#!/bin/bash

read -p "Enter a number: " num
if [[ ! $num =~ ^[0-9]+$ ]]; then
    echo "Invalid input: Please enter a valid number."
    exit 1
else
    echo "You entered a valid number: $num"
fi
```

Explanation:

- [[! $num =~ ^[0-9]+$]]: Checks if the input contains only digits (0-9). ex: 637290
- If the input is invalid, the script exits with an error message.

2. Avoid Hardcoding Sensitive Data

Hardcoding sensitive information like passwords or API keys in your script can lead to security breaches. Instead, use environment variables or configuration files to store such data securely.

Example: Using Environment Variables

```
#!/bin/bash

# The `export` command makes the variable available to the current shell
# and any child processes (subshells or scripts) spawned from it.
# Here, DB_PASSWORD is set as an environment variable that can be accessed by this script
# and other programs or scripts executed from the same shell session.

      export DB_PASSWORD="my_secure_password"

      # Use the variable in the script
      if [[ -z "$DB_PASSWORD" ]]; then
          echo "Error: DB_PASSWORD is not set."
          exit 1
      else
          echo "Connecting to the database with the password."
      Fi
```

Tip:

- Store sensitive information in a .env file and load it using source .env at the beginning of the script.

3. Use Absolute Paths

Relying on the system's PATH variable to locate commands can be risky, as attackers might manipulate it to execute malicious scripts. Always use absolute paths to commands.

Example: Using Absolute Paths

```
      #!/bin/bash

      /usr/bin/cp /path/to/source.txt /path/to/destination.txt
```

Explanation:

- /usr/bin/cp: Specifies the full path to the cp command, ensuring the correct binary is executed.

4. Set Script Permissions

Ensure that your script has the minimum required permissions. Grant execution rights only to the script owner unless others need access.

Example: Setting Script Permissions

```
      chmod 700 script.sh
```

Explanation:

- 700: Grants read, write, and execute permissions to the owner, but no permissions to others.

5. Principle of Least Privilege

Run scripts with the lowest necessary privileges. Avoid running scripts as the root user unless absolutely required.

Example: Using sudo Only When Necessary

Instead of:

```
sudo cp /path/to/source.txt /path/to/destination.txt
```

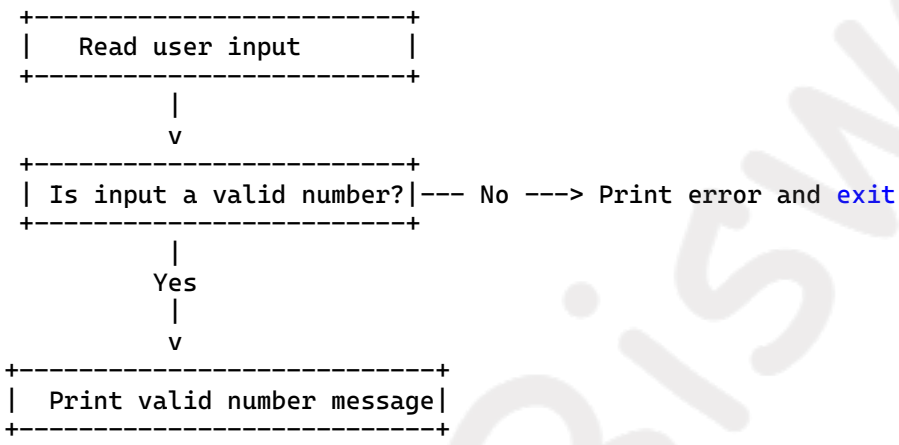
Use:

```
cp /path/to/source.txt /path/to/destination.txt
```

- **Visual Enhancements**

Flowchart Example: Conditional Execution

Here's a flowchart explaining the input validation example:



- **Table: Common Bash Operators**

Operator	Description	Example
-eq	Equal to	[$a -eq $b]
-ne	Not equal to	[$a -ne $b]
-gt	Greater than	[$a -gt $b]
-lt	Less than	[$a -lt $b]
-z	String is null/empty	[-z "$string"]
-n	String is not null	[-n "$string"]

- **Interactive Exercises**

To reinforce learning, add hands-on tasks at the end of each section.

Task 1: Input Validation

Objective:
Write a script that takes a username as input and checks if it only contains lowercase letters.

Solution:

```
#!/bin/bash

read -p "Enter a username: " username
if [[ ! $username =~ ^[a-z]+$ ]]; then
    echo "Invalid username: Only lowercase letters are allowed."
```

```
else
    echo "Valid username."
fi
```

Task 2: Avoid Hardcoding Sensitive Data

Objective:
Create a script that reads a password from an environment variable and prints a success message if it is set.

Solution:

```
#!/bin/bash

if [[ -z "$MY_PASSWORD" ]]; then
    echo "Error: MY_PASSWORD is not set."
    exit 1 # Exits the script with a non-zero status (1), indicating an error condition.
else
    echo "Password is set."
fi
```

Task 3: Print Numbers Using a Loop

Objective:
Create a script to print even numbers from 1 to 20.

Solution:

```
#!/bin/bash

for i in {1..20}; do
    if (( i % 2 == 0 )); then
        echo $i
    fi
done
```

Summary of Security Best Practices

1. **Validate Inputs**: Always validate user inputs to prevent unexpected behavior.
2. **Avoid Hardcoding Sensitive Data**: Use environment variables for credentials.
3. **Use Absolute Paths**: Ensure commands execute from their expected locations.
4. **Set Appropriate Permissions**: Minimize script permissions.
5. **Follow Principle of Least Privilege**: Avoid running scripts with unnecessary privileges.

By following these security practices, you can write secure and reliable Bash scripts.

Part 20: Interactive Exercises

Section 1: Variables

Task 1: Reverse User Input

Objective:
Write a script that takes user input and prints it in reverse.

Solution:

```bash
#!/bin/bash

read -p "Enter text: " text
echo "Reversed text: $(echo "$text" | rev)"
```

Explanation:

- read -p "Enter text: " text: Prompts the user for input and stores it in the variable text.
- echo "$text" | rev: Pipes the input to the rev command, which reverses the text.

Task 2: Convert Text to Uppercase

Objective:
Write a script that takes user input and converts it to uppercase.

Solution:

```bash
#!/bin/bash

read -p "Enter text: " text
echo "Uppercase: $(echo "$text" | tr '[:lower:]' '[:upper:]')"
# This line:
# 1. Uses `tr '[:lower:]' '[:upper:]'` to convert all lowercase letters in the input text to uppercase.
# 2. The `$(...)` syntax captures the output of the `tr` command and substitutes it into the `echo` command.
# 3. `"$text"` is the user-provided input passed through the pipeline to `tr`.
# The result is displayed as "Uppercase: [converted text]".
```

Section 2: Loops

Task 1: Print Multiplication Table

Objective:
Write a script that prints the multiplication table for a given number.

Solution:

```bash
#!/bin/bash

# Prompt the user to enter a number
read -p "Enter a number: " num

# Loop to print the multiplication table for the entered number
for i in {1..10}; do
    echo -e "$num x $i \t= $(( num * i ))"
done
```

```
Output:

Enter a number: 6
6 x 1   = 6
6 x 2   = 12
6 x 3   = 18
6 x 4   = 24
6 x 5   = 30
6 x 6   = 36
6 x 7   = 42
6 x 8   = 48
6 x 9   = 54
6 x 10  = 60
```

Part: 20. Advanced Topics

1. Subshells

A **subshell** is a child process created to run commands in isolation. Changes made within a subshell do not affect the parent shell.

Syntax:

```
result=$(command1 && command2)
```

Example: Using Subshell to Isolate Directory Change

```
#!/bin/bash

# This command runs in a subshell
result=$(cd /tmp && ls)

echo "Contents of /tmp:"
echo "$result"

# Current working directory remains unchanged
echo "Current directory: $(pwd)"
```

Explanation:

- cd /tmp && ls: Changes the directory to /tmp and lists its contents, but this change only affects the subshell.
- pwd: Displays the current directory, which remains unchanged in the parent shell.

Task: List Files in a Different Directory Without Changing the Current Directory

Objective:
Write a script that lists the contents of the /etc directory using a subshell, but keeps the current directory unchanged.

Solution:

```
#!/bin/bash

result=$(cd /etc && ls)
echo "Contents of /etc:"
echo "$result"
echo "Current directory remains: $(pwd)"
```

2. Signal Handling

Bash scripts can **trap** signals to handle or ignore specific events, such as a user pressing Ctrl+C (SIGINT).

Syntax:

```
trap 'commands' SIGNAL
```

Example: Handling SIGINT (Ctrl+C)

```
#!/bin/bash

# Trap SIGINT and define a handler
trap 'echo "Caught SIGINT (Ctrl+C). Custom handling in action."; exit 1' SIGINT

# Debug active traps
trap -p

# Simulate a long-running process
echo "Press Ctrl+C to test signal handling."
while true; do
    sleep 1
done
```

Explanation:

- trap 'echo "Ctrl+C is disabled"; exit 1' SIGINT: Captures the SIGINT signal and prevents the script from terminating immediately.

Task: Handle SIGTERM Signal

Objective:
Write a script that traps the SIGTERM signal and prints a message before exiting.

Solution:

```
#!/bin/bash

trap 'echo "SIGTERM received. Exiting gracefully."; exit 0' SIGTERM

echo "Waiting for SIGTERM signal..."
while true; do
    sleep 1
done
```

3. Combining Subshells and Signal Handling

Example: Script to Monitor Directory Changes

```
#!/bin/bash

trap 'echo "Terminating script..."; exit 0' SIGINT SIGTERM

echo "Monitoring /tmp directory for changes. Press Ctrl+C to stop."
while true; do
    result=$(ls /tmp)
    echo "Current contents of /tmp:"
    echo "$result"
    sleep 5
done
```

Explanation:

- The script monitors the /tmp directory every 5 seconds.
- It handles both SIGINT and SIGTERM signals to terminate gracefully.

Summary of Advanced Topics

1. **Subshells**: Run commands in isolation to avoid affecting the parent shell.
2. **Signal Handling**: Trap and handle signals like SIGINT and SIGTERM for graceful exits.

By mastering these advanced techniques, you can write more robust and flexible Bash scripts.

Part 21: Reference Section: Frequently Used Bash Commands and Techniques

This section provides a quick reference for commonly used Bash commands and variable manipulation techniques to assist with script writing.

1. Variable Manipulation

Bash provides powerful variable manipulation features, including string transformations and arithmetic operations.

Operation	Description	Example	Output
${var,,}	Convert string to lowercase	var1="HELLO"; echo "${var1,,}"	hello
${var^^}	Convert string to uppercase	var1="hello"; echo "${var1^^}"	HELLO
${#var}	Get the length of a string	var1="Hello"; echo ${#var1}	5
${var:position:length}	Extract a substring	var1="Hello"; echo ${var1:1:3}	ell
$((expression))	Perform arithmetic operations	x=5; y=3; echo $((x + y))	8

2. Commonly Used Commands

Command	Description	Example
ls	List files in a directory	ls /home/user
cat	Display the contents of a file	cat file.txt
grep	Search for patterns in files	grep "error" log.txt
echo	Print text to the terminal	echo "Hello, World!"
pwd	Print the current directory	pwd
cd	Change the current directory	cd /path/to/directory
cp	Copy files	cp source.txt destination.txt
mv	Move or rename files	mv oldname.txt newname.txt
rm	Remove files	rm file.txt
chmod	Change file permissions	chmod 700 script.sh
chown	Change file ownership	chown user:group file.txt
find	Find files in a directory	find /home -name "*.txt"
sort	Sort lines in a file	sort file.txt
uniq	Remove duplicate lines	uniq file.txt
head	Display the first lines of a file	head -n 10 file.txt
tail	Display the last lines of a file	tail -n 10 file.txt
wc	Count lines, words, and characters	wc -l file.txt
awk	Process and analyze text files	awk '{print $1}' file.txt
sed	Stream editor for text files	sed 's/old/new/g' file.txt

3. File Permissions

Command	Description	Example
chmod 700 file.sh	Grant read, write, and execute to the owner only	Secure the script file
chmod 644 file.txt	Grant read/write to owner, read to others	Public text file

4. Conditional Operators

Operator	Description	Example	Result
-e	File exists	[-e file.txt]	True/False
-d	Directory exists	[-d /path/to/dir]	True/False
-z	String is empty	[-z "$var"]	True/False
-n	String is not empty	[-n "$var"]	True/False
-eq	Equal to	[$x -eq $y]	True/False
-ne	Not equal to	[$x -ne $y]	True/False
-gt	Greater than	[$x -gt $y]	True/False

6. Process Management

Command	Description	Example
ps	List running processes	ps aux
top	Display real-time system stats	top
kill	Terminate a process by PID	kill 1234
killall	Terminate all processes by name	killall firefox

6. Networking Commands

Command	Description	Example
ping	Test connectivity to a host	ping google.com
curl	Transfer data from a URL	curl http://example.com
wget	Download files from the web	wget http://example.com/file.zip
netstat	Display network connections	netstat -an

7. Useful Shortcuts and Aliases

Shortcut	Description	Example
alias ll='ls -l'	Create an alias for ls -l	ll
Ctrl+C	Interrupt/Stop a running command	Stops the current process
Ctrl+Z	Suspend a running command	Puts the process in the background

Summary

This reference section covers essential Bash commands, variable manipulation techniques, file handling, and networking basics. Keep it handy while scripting to quickly recall command usage and syntax.

PROJECTS:

1. File is exists or not
2. How many file exists on home directory
3. Get the ip address of host
4. Check host is accessible or not
5. Detect disk usages info
6. Get RAM and CPU info
7. Delete file which is older than 7 days and input file name.
8. Status on total number of files