Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

INTRODUCTION TO SHELL SCRIPTING (BASH, KORN, C SHELL)

CHAPTER 1: UNDERSTANDING SHELL SCRIPTING

What is Shell Scripting?

Shell scripting is a method of automating tasks in **UNIX and Linux** by writing a series of commands in a script file that the shell executes sequentially. The shell acts as an **interface between the user and the operating system**, allowing users to perform operations such as **file management, process automation, and system administration** efficiently.

Shell scripting is widely used for:

- **Automating repetitive tasks** – Instead of manually entering commands, scripts can execute predefined actions.
- **System administration** – Automating backups, monitoring processes, and configuring network settings.
- **Software development** – Managing build processes, running test scripts, and deploying applications.
- **Task scheduling** – Using cron jobs to run scripts at scheduled intervals.

Shell scripting is supported by several UNIX shells, each with its own syntax and features. The most commonly used shells for scripting are:

1. **Bash (Bourne Again Shell)** – The default shell in most Linux distributions, offering advanced scripting capabilities.
2. **Korn Shell (Ksh)** – An enhanced shell with additional features for scripting and performance improvements.
3. **C Shell (csh and tcsh)** – Designed for **C-like syntax**, making it more familiar to programmers with a C language background.

Example

A simple shell script (hello.sh) that prints "Hello, World!" on the terminal:

```
#!/bin/bash

echo "Hello, World!"
```

To run the script:

```
chmod +x hello.sh

./hello.sh
```

Exercise

1. Create a shell script that prints your name and the current date.
2. Modify the script to accept user input and display a custom message.

Case Study: Automating Daily Backups Using Shell Scripts

A system administrator at a company needs to back up important log files daily. Instead of manually copying files, they create a **shell script that compresses and stores logs in a backup directory**. This script is scheduled to run automatically using **cron jobs**, ensuring **consistent backups without human intervention**.

CHAPTER 2: BASH (BOURNE AGAIN SHELL) SCRIPTING

Introduction to Bash Scripting

Bash (Bourne Again Shell) is the most commonly used shell in Linux systems due to its **flexibility, scripting features, and compatibility** with older UNIX shells. Bash scripts are simple to write and support:

- **Variables and data manipulation**
- **Conditional statements (if-else, case statements)**
- **Loops (for, while, until)**
- **Functions for modular programming**

Writing a Simple Bash Script

To create and execute a basic Bash script:

1. Open a terminal and create a new script:
2. nano myscript.sh
3. Add the following code:
4. #!/bin/bash
5. echo "Welcome to Bash scripting!"
6. Save and exit (Ctrl + X, then Y to confirm).

7. Make the script executable:
8. chmod +x myscript.sh
9. Run the script:
10. ./myscript.sh

Example: Using Variables in Bash

```
#!/bin/bash

name="John Doe"

echo "Hello, $name! Today is $(date)."
```

This script stores a variable (name) and uses $(date) to print the current date.

Exercise

1. Write a Bash script that calculates the sum of two numbers entered by the user.
2. Modify the script to check if the entered numbers are valid before performing the addition.

Case Study: Using Bash Scripts for User Account Management

A company with multiple employees needs an efficient way to **create and manage user accounts**. The IT team writes a **Bash script that adds new users, assigns home directories, and sets passwords automatically**, reducing the time spent on manual account creation.

CHAPTER 3: KORN SHELL (KSH) SCRIPTING

Introduction to Korn Shell (Ksh)

The **Korn Shell (ksh)** is an advanced shell developed by **David Korn** at AT&T Bell Labs. It provides:

- **Faster execution speed** compared to Bash.
- **Better support for floating-point arithmetic and arrays.**
- **Enhanced job control and command-line editing.**

Ksh scripting syntax is similar to Bash but includes additional features for **performance and scripting enhancements**.

Writing a Simple Ksh Script

1. Create a new script:
2. nano mykshscript.ksh
3. Add the following code:
4. #!/bin/ksh
5. echo "This is a Korn shell script."
6. Save and execute it:
7. chmod +x mykshscript.ksh
8. ./mykshscript.ksh

Example: Using Loops in Korn Shell

```
#!/bin/ksh

for i in 1 2 3 4 5

do

 echo "Number: $i"
```

done

This script prints numbers from 1 to 5 using a for loop.

Exercise

1. Write a Korn shell script that **calculates the factorial of a given number**.
2. Modify the script to handle invalid input and prompt the user to enter a valid number.

Case Study: Korn Shell Scripts in Financial Applications

A bank uses Korn shell scripts for **processing large financial transactions**. These scripts handle **interest calculations, balance updates, and transaction logging**, ensuring efficient banking operations.

CHAPTER 4: C SHELL (CSH AND TCSH) SCRIPTING

Introduction to C Shell (csh and tcsh)

The **C Shell (csh)** was developed at **UC Berkeley** and is designed to resemble the **C programming language**, making it easier for C programmers to adapt. The **tcsh** (enhanced C Shell) provides additional features like **command-line editing and autocomplete**.

C Shell scripting syntax is different from Bash and Korn Shell. It uses:

- **set** to declare variables instead of =
- **@** for arithmetic operations
- **endif** to close conditional statements

Writing a Simple C Shell Script

1. Create a new script:
2. nano mycshscript.csh
3. Add the following code:
4. #!/bin/csh
5. echo "This is a C shell script."
6. Save and execute:
7. chmod +x mycshscript.csh
8. ./mycshscript.csh

Example: Using If Statements in C Shell

```
#!/bin/csh

set num = 10

if ($num > 5) then

 echo "The number is greater than 5"

endif
```

This script checks if a number is greater than 5 and prints a message accordingly.

Exercise

1. Write a C Shell script that **checks if a file exists** in the current directory.
2. Modify the script to prompt the user for a file name before checking.

Case Study: Using C Shell Scripts in Scientific Computing

A research institute uses C Shell scripts to automate **data processing in large-scale simulations**. Researchers run **batch jobs** using scripts, allowing them to focus on analysis instead of manual data handling.

CONCLUSION

Shell scripting is a **powerful tool** for automating UNIX tasks. Each shell—**Bash, Korn, and C Shell**—has **unique features** that cater to different scripting needs. Mastering shell scripting enables users to **automate processes, enhance productivity, and optimize system performance**.

WRITING AND EXECUTING BASIC SHELL SCRIPTS

CHAPTER 1: INTRODUCTION TO SHELL SCRIPTING

What is Shell Scripting?

Shell scripting is the process of writing a sequence of commands in a **script file** that can be executed by the shell to **automate tasks, manage system operations, and simplify repetitive tasks**. In UNIX and Linux environments, shell scripts are widely used for:

- **Automating administrative tasks**, such as backups and updates.
- **Managing files and processes** efficiently.
- **Scheduling jobs** to run at specific intervals.
- **Configuring system settings** dynamically.

A shell script is essentially a text file containing a series of **commands** that the shell interprets and executes sequentially. Shell scripts eliminate the need for manually typing commands repeatedly and enhance productivity.

Types of Shells Used for Scripting

1. **Bash (Bourne Again Shell)** – Most common and default shell in Linux.
2. **Korn Shell (Ksh)** – Faster execution and better scripting capabilities.
3. **C Shell (Csh and Tcsh)** – Uses C-like syntax, making it familiar for C programmers.

A **basic shell script** typically consists of:

- A **shebang (#!)** at the beginning to specify the shell interpreter.
- **Commands and logic,** such as loops and conditions.

Example

A simple Bash script (hello.sh) to print "Hello, World!"

```
#!/bin/bash

echo "Hello, World!"
```

Exercise

1. Create a shell script that prints your name and the current date.
2. Modify the script to accept user input and display a personalized message.

Case Study: Automating Software Updates Using Shell Scripts

A system administrator needs to update multiple servers daily. Instead of manually running update commands, they create a **shell script that automatically updates and reboots all servers,** saving time and reducing human errors.

CHAPTER 2: CREATING AND WRITING A SHELL SCRIPT

Step 1: Creating a New Shell Script

To create a shell script, follow these steps:

1. Open a terminal and create a new file using nano, vi, or gedit:

2. nano myscript.sh
3. Add the following script:
4. #!/bin/bash
5. echo "This is my first shell script!"
6. Save and exit (Ctrl + X, then Y to confirm in nano).

Step 2: Adding Comments to a Script

Comments improve script readability and are ignored during execution. Use # to write comments.

#!/bin/bash

This script prints a message

echo "Welcome to Shell Scripting!"

Step 3: Using Variables in Shell Scripts

Variables store values and can be used within the script.

#!/bin/bash

name="Alice"

echo "Hello, $name! Welcome to UNIX."

Example

A script that prints a personalized greeting:

#!/bin/bash

echo "Enter your name:"

read user_name

echo "Hello, $user_name! Have a great day!"

Exercise

1. Write a script that asks for your favorite programming language and prints a response.
2. Modify the script to store the input in a variable and display it dynamically.

Case Study: Using Shell Scripts for Automated Login Messages

A company wants to display a custom **welcome message** when users log in. A shell script is used to **retrieve usernames** and print a welcome message dynamically.

CHAPTER 3: EXECUTING A SHELL SCRIPT

Step 1: Making a Shell Script Executable

Before running a shell script, it must be given execute permissions:

chmod +x myscript.sh

Step 2: Running a Shell Script

- **Method 1: Execute using ./ (current directory)**
- ./myscript.sh
- **Method 2: Execute using bash**
- bash myscript.sh
- **Method 3: Execute using sh**
- sh myscript.sh

Step 3: Running Scripts Automatically on Boot

To run a script at startup, add it to the system's **crontab** or startup scripts directory (/etc/init.d/).

Example

Running a script in the background:

nohup ./myscript.sh &

Exercise

1. Create a script, grant execute permissions, and run it.
2. Modify the script to include comments and user input.

Case Study: Scheduling Daily Tasks with Shell Scripts

A university schedules **daily system maintenance** using cron jobs and shell scripts, ensuring system stability and automated backups without manual intervention.

CHAPTER 4: IMPLEMENTING LOGIC IN SHELL SCRIPTS

Using Conditional Statements (if-else)

Shell scripts can **make decisions** using if-else statements.

```
#!/bin/bash

echo "Enter a number:"

read num

if [ $num -gt 10 ]; then

 echo "The number is greater than 10"
```

```
else
  echo "The number is 10 or less"
fi
```

Using Loops (for, while, until)

Loops help execute commands multiple times.

For Loop Example

```
#!/bin/bash
for i in 1 2 3 4 5
do
  echo "Iteration $i"
done
```

While Loop Example

```
#!/bin/bash
counter=1
while [ $counter -le 5 ]
do
  echo "Count: $counter"
  ((counter++))
done
```

Example: Automating File Cleanup with Loops

A script that **deletes old log files** automatically:

```
#!/bin/bash

for file in /var/log/*.log

do

 rm $file

done

echo "Old logs deleted."
```

Exercise

1. Write a script that asks for a number and checks if it's even or odd using if-else.
2. Create a loop that prints numbers from 1 to 10.

Case Study: Automating System Cleanup Using Shell Scripts

A company schedules a **weekly script to clean temporary files and clear logs** using loops and conditionals, ensuring optimal system performance.

CHAPTER 5: DEBUGGING AND BEST PRACTICES IN SHELL SCRIPTING

Debugging Shell Scripts

To find errors in a script, use:

```
bash -x myscript.sh
```

Best Practices for Writing Shell Scripts

- **Use meaningful variable names** for clarity.
- **Include comments (#)** to explain each section.

- **Handle user input carefully** to prevent errors.
- **Use error handling (set -e)** to stop execution on failure.

Example: Using Error Handling in Scripts

```
#!/bin/bash

set -e

echo "Copying files..."

cp file1.txt /nonexistent_directory/  # This will cause an error and stop the script

echo "Done!"
```

Exercise

1. Modify a script to include error handling.
2. Add comments to explain each step in your script.

Case Study: Improving Script Reliability for Server Management

A tech company improves the **reliability of automation scripts** by **adding error handling, logging, and user-friendly prompts,** reducing system crashes due to misconfigured scripts.

CONCLUSION

Shell scripting is **an essential skill** for UNIX users**,** automating tasks and improving efficiency. This guide covered:

- **Writing basic scripts.**
- **Executing and debugging scripts.**

- **Implementing conditions and loops.**
- **Using best practices to write reliable scripts.**

USING VARIABLES, LOOPS, AND CONDITIONAL STATEMENTS IN SHELL SCRIPTING

CHAPTER 1: INTRODUCTION TO VARIABLES, LOOPS, AND CONDITIONAL STATEMENTS

Understanding Variables, Loops, and Conditional Statements in Shell Scripting

Shell scripting allows users to **automate repetitive tasks, manage system configurations, and perform complex operations** using a sequence of commands. Three fundamental concepts in shell scripting are:

1. **Variables** – Store and manipulate data dynamically.
2. **Conditional Statements (if, case)** – Allow decision-making based on conditions.
3. **Loops (for, while, until)** – Automate repetitive tasks efficiently.

Each of these concepts helps create **efficient and interactive shell scripts** that improve productivity and system administration.

Example

A script that **checks if a file exists** using a variable and conditional statement:

```
#!/bin/bash

filename="testfile.txt"
```

```
if [ -f "$filename" ]; then
 echo "$filename exists."
else
 echo "$filename does not exist."
fi
```

Exercise

1. Write a script that **stores a username in a variable** and prints a welcome message.
2. Modify the script to **check if the username is "admin"** and display a different message accordingly.

Case Study: Automating Server Monitoring Using Shell Scripts

A system administrator needs to **monitor server disk space** and send alerts if usage exceeds 80%. By using **variables, conditional statements, and loops**, a shell script checks disk usage and **notifies the admin** when storage reaches critical levels.

CHAPTER 2: USING VARIABLES IN SHELL SCRIPTING

Understanding Variables in Shell Scripting

Variables in shell scripts allow users to **store, retrieve, and manipulate data dynamically**. Variables make scripts **more flexible and reusable** by eliminating hardcoded values.

Types of Variables

1. **User-defined Variables** – Assigned by the user and used within the script.

2. **Environment Variables** – Predefined system variables that control shell behavior (e.g., $HOME, $PATH).
3. **Special Variables** – Used for **command-line arguments and script execution** (e.g., $0, $1, $?).

Declaring and Using Variables

- Assign a variable:
- name="Alice"
- Access a variable:
- echo "Hello, $name!"
- Use a system variable:
- echo "Current user: $USER"
- echo "Home directory: $HOME"

Example: Using User-Defined Variables

```
#!/bin/bash

user="John"

echo "Welcome, $user!"

current_date=$(date)

echo "Today's date is: $current_date"
```

Exercise

1. Write a script that **stores and prints the current system date and time**.
2. Modify the script to **store the user's home directory in a variable** and print it.

Case Study: Using Variables for User Account Management

A company creates a **shell script to add new users** automatically. The script **stores usernames and passwords in variables,** ensuring **efficient and secure** account creation.

CHAPTER 3: USING CONDITIONAL STATEMENTS IN SHELL SCRIPTING

Understanding Conditional Statements

Conditional statements allow a script to **make decisions** based on specific conditions. The two most common conditional statements in shell scripting are:

1. **if-else Statement** – Executes different code based on whether a condition is true or false.
2. **case Statement** – Simplifies multiple conditions in a structured format.

Using if-else Statements

Syntax:

```
if [ condition ]; then
  # Code block if condition is true
else
  # Code block if condition is false
fi
```

Example: Checking If a User Is Root

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

```
if [ "$USER" == "root" ]; then
  echo "You are logged in as root."
else
  echo "You are not root."
fi
```

Using case Statements

Syntax:

```
case $variable in
  value1) echo "Action for value1" ;;
  value2) echo "Action for value2" ;;
  *) echo "Default action" ;;
esac
```

Example: Checking User Input with case Statement

```
#!/bin/bash
echo "Enter a choice: start, stop, restart"
read choice

case $choice in
  start) echo "Starting service..." ;;
  stop) echo "Stopping service..." ;;
  restart) echo "Restarting service..." ;;
```

*) echo "Invalid choice!" ;;

esac

Exercise

1. Write a script that **checks if a number is positive, negative, or zero** using if-else.
2. Modify the script to **use a case statement for handling user input**.

Case Study: Automating Network Configuration Using Conditional Statements

A network administrator writes a **script that checks network connectivity** and **automatically restarts services** if a failure is detected.

CHAPTER 4: USING LOOPS IN SHELL SCRIPTING

Understanding Loops in Shell Scripting

Loops allow a script to **execute a set of commands repeatedly** until a specific condition is met. The three main types of loops are:

1. **for Loop** – Iterates over a list of values.
2. **while Loop** – Runs as long as a condition is true.
3. **until Loop** – Runs until a condition becomes true.

Using for Loop

Syntax:

for var in value1 value2 value3

```
do
  # Commands
done
```

Example: Printing Numbers from 1 to 5

```
#!/bin/bash
for i in 1 2 3 4 5
do
  echo "Iteration $i"
done
```

Using while Loop

Syntax:

```
while [ condition ]
do
  # Commands
done
```

Example: Counting Down from 5 to 1

```
#!/bin/bash
count=5
while [ $count -gt 0 ]
do
  echo "Countdown: $count"
```

```
 ((count--))
done
```

Using until Loop

Syntax:

```
until [ condition ]
do
 # Commands
done
```

Example: Printing Numbers Until a Condition Is Met

```
#!/bin/bash
num=1
until [ $num -ge 5 ]
do
 echo "Number: $num"
 ((num++))
done
```

Exercise

1. Write a script that **prints numbers from 1 to 10 using a for loop**.
2. Modify the script to **use a while loop to keep asking the user for input until they type "exit"**.

Case Study: Automating File Processing Using Loops

A company generates **hundreds of log files daily**. A shell script **uses loops to process each file automatically,** reducing **manual work** and ensuring **consistent file handling**.

CONCLUSION

This guide covered:

- **Variables** for storing and managing data.
- **Conditional statements (if, case)** for decision-making.
- **Loops (for, while, until)** for automating repetitive tasks.

FUNCTIONS IN SHELL SCRIPTS

CHAPTER 1: INTRODUCTION TO FUNCTIONS IN SHELL SCRIPTING

What are Functions in Shell Scripting?

Functions in shell scripting are **blocks of reusable code** that help in organizing scripts efficiently. They allow **modular programming,** making scripts easier to **maintain, debug, and reuse**. Instead of writing the same commands multiple times, we can define a function once and call it whenever needed.

Functions in shell scripts help in:

- **Reducing code duplication** by reusing a block of code multiple times.
- **Improving readability and organization** by separating logic into smaller units.
- **Enhancing maintainability** by making modifications easier without affecting the entire script.
- **Handling complex operations efficiently** by breaking them into multiple functions.

A function is defined using the following syntax:

```
function_name() {
  # Commands
}
```

To **call the function,** simply use its name:

```
function_name
```

Example: Basic Function to Print a Message

```
#!/bin/bash

greet() {

 echo "Hello, welcome to shell scripting!"

}

greet
```

Exercise

1. Create a function that **prints your name and the current date**.
2. Modify the function to accept **user input** and print a personalized message.

Case Study: Automating User Account Creation Using Functions

A system administrator needs to **create multiple user accounts** efficiently. Instead of manually adding users, they write a function that **creates users, assigns home directories, and sets default permissions**, improving efficiency and reducing errors.

CHAPTER 2: DEFINING AND CALLING FUNCTIONS IN SHELL SCRIPTS

How to Define and Call a Function

Functions in shell scripts can be defined using two different methods:

1. **Using the function keyword**
2. function my_function {
3. echo "This is a shell function"

4. }
5. **Without the function keyword** (most commonly used)
6. my_function() {
7. echo "This is a shell function"
8. }

To **call the function,** simply use its name in the script:

my_function

Example: Calling a Function Multiple Times

```
#!/bin/bash
print_message() {
 echo "This message is printed from a function"
}
print_message
print_message
```

Returning Values from Functions

Functions do not return values like traditional programming languages. Instead, they **use echo or return** to pass values.

Example: Returning Values Using return

```
#!/bin/bash
sum() {
 return $(($1 + $2))
```

```
}

sum 5 3

echo "Sum is: $?"
```

$? captures the return value of the function.

Exercise

1. Write a function that **calculates and prints the square of a number**.
2. Modify the function to **return the result using return** and display it using $?.

Case Study: Using Functions to Manage Log Files

A company maintains **log files** that must be archived weekly. A shell script function is created to **compress and move logs automatically,** reducing manual effort and improving system maintenance.

CHAPTER 3: PASSING ARGUMENTS TO FUNCTIONS

How to Pass Arguments to Functions

Functions can accept **arguments (parameters),** making them more dynamic and reusable. Arguments are passed when calling the function and accessed inside the function using $1, $2, etc.

Example: Function with Arguments

```
#!/bin/bash

greet_user() {
```

```
 echo "Hello, $1! Welcome to $2."
}
greet_user "Alice" "UNIX"
```

Output:

```
Hello, Alice! Welcome to UNIX.
```

Handling Multiple Arguments

We can use $@ or $* to handle multiple arguments:

```
#!/bin/bash
print_all() {
 echo "All arguments: $@"
}
print_all "One" "Two" "Three"
```

Example: Function to Add Two Numbers

```
#!/bin/bash
add_numbers() {
 sum=$(($1 + $2))
 echo "The sum is: $sum"
}
add_numbers 10 20
```

Exercise

1. Write a function that **accepts a user's first and last name** and prints a greeting.
2. Modify the function to **handle any number of arguments dynamically**.

Case Study: Using Functions to Automate System Monitoring

A shell script function monitors **CPU usage, memory consumption, and disk space**. It accepts **threshold values as arguments** and **sends alerts** when limits are exceeded, helping administrators **manage resources proactively**.

CHAPTER 4: USING LOOPS INSIDE FUNCTIONS

Why Use Loops in Functions?

Using loops inside functions makes it easier to **iterate over files, process multiple items, or repeat actions dynamically**. We can use **for, while, or until loops** inside functions to automate repetitive tasks.

Example: Function with a for Loop

```
#!/bin/bash

print_numbers() {

 for i in {1..5}

 do

  echo "Number: $i"

 done

}
```

print_numbers

Example: Function with a while Loop

```
#!/bin/bash

count_down() {
 count=$1
 while [ $count -gt 0 ]
 do
  echo "Countdown: $count"
  ((count--))
 done
}

count_down 5
```

Using Loops to Process Files in a Directory

```
#!/bin/bash

process_files() {
 for file in *.txt
 do
  echo "Processing $file..."
 done
}

process_files
```

Exercise

1. Write a function that **prints numbers from 1 to 10 using a for loop**.
2. Modify the function to **print numbers in reverse order using a while loop**.

Case Study: Using Loops in Functions for Backup Automation

A function iterates over **multiple directories,** compressing and backing up files dynamically. This ensures **regular data backups** without manual intervention.

CHAPTER 5: BEST PRACTICES FOR USING FUNCTIONS IN SHELL SCRIPTING

Best Practices for Writing Functions

1. **Use meaningful function names**
    - Instead of f1(), use backup_logs().
2. **Comment your functions**
    - Explain what each function does.
3. **Use local variables inside functions**
    - Prevent conflicts with global variables.
4. **Handle errors properly**
    - Check for missing arguments and unexpected values.

Debugging Functions in Shell Scripts

Use the set -x option to enable debugging:

```
#!/bin/bash
set -x
debug_function() {
 echo "Debugging mode is ON"
}
debug_function
set +x
```

Example: Handling Errors in Functions

```
#!/bin/bash
divide_numbers() {
 if [ $2 -eq 0 ]; then
  echo "Error: Division by zero is not allowed."
  return 1
 else
  echo "Result: $(($1 / $2))"
 fi
}
divide_numbers 10 0
```

Exercise

1. Modify a function to **validate user input before processing**.

2. Implement error handling for **file operations inside a function**.

Case Study: Implementing Secure Shell Scripting Functions

A **security-focused shell script** uses **functions for user authentication, input validation, and logging**. It prevents unauthorized access and enhances system security.

CONCLUSION

This guide covered:

- **Defining, calling, and returning values from functions.**
- **Passing arguments and using loops inside functions.**
- **Best practices and debugging techniques.**

DEBUGGING AND ERROR HANDLING IN SHELL SCRIPTS

CHAPTER 1: INTRODUCTION TO DEBUGGING AND ERROR HANDLING

Why Debugging and Error Handling are Important in Shell Scripting?

When writing shell scripts, errors are inevitable. Proper **debugging** and **error handling** techniques help identify and fix issues efficiently, ensuring the script works as expected. Without proper debugging, a script might:

- Execute incorrect commands leading to **unexpected results**.
- Fail silently, making **troubleshooting difficult**.
- Cause **data loss or system failures** due to improper handling of errors.

Error handling is equally crucial as it prevents the script from crashing unexpectedly and **gracefully handles failures**, ensuring that the system remains stable.

There are **three main strategies** for handling errors in shell scripting:

1. **Debugging techniques** – Used to **identify and fix issues** in scripts.
2. **Error handling mechanisms** – Prevent unexpected failures and **handle errors gracefully**.
3. **Logging and reporting** – Helps in monitoring script execution for auditing and troubleshooting.

Example

A script that tries to delete a file without checking if it exists may produce an error:

```
#!/bin/bash

rm /nonexistentfile.txt
```

A better approach is to check if the file exists before attempting to delete it:

```
#!/bin/bash

if [ -f "/nonexistentfile.txt" ]; then

 rm /nonexistentfile.txt

else

 echo "File not found!"

fi
```

Exercise

1. Create a script that attempts to read a file. Add error handling to **display an appropriate message if the file is missing**.
2. Modify the script to **log the error into a file instead of printing it on the screen**.

Case Study: Preventing Server Downtime with Error Handling

A system administrator runs a **database backup script** nightly. One night, the script fails due to insufficient disk space, causing **critical data loss**. By implementing **error handling and logging,** future failures are **detected in advance**, preventing server downtime.

CHAPTER 2: DEBUGGING TECHNIQUES IN SHELL SCRIPTING

Using Shell Built-in Debugging Options

Bash provides **built-in debugging options** to trace and debug script execution efficiently.

Using set -x for Debugging

The set -x command enables **execution tracing,** showing each command before it runs.

```
#!/bin/bash

set -x

echo "Debugging enabled"

mkdir test_directory

cd test_directory

set +x

echo "Debugging disabled"
```

Output:

```
+ echo Debugging enabled

Debugging enabled

+ mkdir test_directory

+ cd test_directory

+ set +x

Debugging disabled
```

Using set -e to Exit on Errors

The set -e command forces the script to **exit immediately** if a command fails.

```
#!/bin/bash

set -e

echo "Starting script..."

cd /nonexistent_directory

echo "This line will not be executed."
```

Output:

```
Starting script...

cd: no such file or directory: /nonexistent_directory

Script terminated
```

Using set -u to Detect Unset Variables

The set -u option treats **unset variables as errors,** preventing unintended behavior.

```
#!/bin/bash

set -u

echo "Hello, $name"
```

Output:

```
bash: name: unbound variable
```

Example: Debugging a Simple Script

```
#!/bin/bash

set -x
```

```
echo "Checking system uptime..."

uptime

set +x
```

Exercise

1. Modify a script to enable set -e, set -x, and set -u for debugging.
2. Run a script without setting variables and observe the errors with set -u.

Case Study: Debugging Network Issues Using Shell Scripts

A company experiences **random network failures**. The IT team writes a script that **pings servers** and **logs failures**. Using set -x, they identify a **syntax error** in the script that was causing incorrect execution.

CHAPTER 3: ERROR HANDLING IN SHELL SCRIPTS

Handling Errors Using Exit Status ($?)

Every command in UNIX returns an **exit status**:

- 0 → Command executed successfully.
- Non-zero value → Command failed.

Example: Checking the exit status of a command:

```
#!/bin/bash

mkdir test_dir

if [ $? -eq 0 ]; then
```

```
 echo "Directory created successfully."
else
 echo "Failed to create directory."
fi
```

Using trap to Handle Script Interruptions

The trap command **catches signals** (e.g., script termination) and performs cleanup.

```
#!/bin/bash
trap "echo 'Script interrupted!'; exit 1" SIGINT SIGTERM

echo "Press Ctrl+C to interrupt..."
while true; do sleep 1; done
```

Using || and && for Error Handling

- command || action → Run action **if command fails**.
- command && action → Run action **if command succeeds**.

Example:

```
#!/bin/bash
mkdir new_dir || echo "Failed to create directory"
```

Example: Error Handling in File Operations

```
#!/bin/bash
filename="important_file.txt"
```

```
if [ ! -f "$filename" ]; then
  echo "Error: $filename not found!"
  exit 1
fi

echo "Processing $filename..."
```

Exercise

1. Write a script that **checks if a directory exists before creating it**.
2. Modify the script to **exit with an error message if directory creation fails**.

Case Study: Preventing Script Failures in Automated Tasks

A financial firm uses shell scripts to **automate data processing**. Occasionally, the script fails without logging errors, making troubleshooting difficult. By implementing **error handling using exit codes and traps**, they improve system stability.

CHAPTER 4: LOGGING AND REPORTING ERRORS IN SHELL SCRIPTS

Using Log Files to Record Errors

Instead of displaying errors on the screen, scripts can **log errors to a file**.

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

```
logfile="script.log"


echo "Starting script..." | tee -a $logfile
mkdir /nonexistent_directory 2>> $logfile
echo "Script completed!" | tee -a $logfile
```

Redirecting Error Output to Log Files

- command 2> error.log → Redirect errors to error.log.
- command &> output.log → Redirect **both standard and error output** to output.log.

Example:

```
#!/bin/bash
ls /wrong_directory 2> errors.log
```

Using logger for System Logging

The logger command sends messages to the system log (/var/log/syslog).

```
#!/bin/bash
logger "Script executed successfully"
```

Example: Logging Errors in a Script

```
#!/bin/bash
logfile="script.log"


if [ ! -d "/testdir" ]; then
```

```
    echo "Error: Directory not found!" >> $logfile
    exit 1
fi
```

Exercise

1. Modify a script to **log both successful and failed operations**.
2. Use logger to send **error messages to the system log**.

Case Study: Logging System Events in a Security Monitoring Script

A **cybersecurity team** writes a script that **monitors login attempts**. When unauthorized access is detected, the script **logs details in /var/log/auth.log** and sends alerts to administrators.

CONCLUSION

This guide covered:

- **Debugging techniques (set -x, set -e, trap)**.
- **Error handling using exit codes and conditions**.
- **Logging errors for troubleshooting**.

AUTOMATING TASKS WITH CRON JOBS IN SHELL SCRIPTING

CHAPTER 1: INTRODUCTION TO CRON JOBS

What are Cron Jobs?

Cron jobs are scheduled tasks in UNIX and Linux systems that run automatically at predefined intervals. These jobs allow users and administrators to automate **system maintenance, backups, monitoring, and repetitive tasks** without manual intervention.

The **cron daemon (crond)** runs in the background and executes scheduled commands listed in the **crontab (cron table)** file. The cron service is crucial for:

- **System automation** – Running periodic updates, cleanups, and reports.
- **User-specific tasks** – Automating repetitive tasks like sending emails or clearing logs.
- **Server maintenance** – Scheduling software updates, backups, and performance monitoring.

Each user on a UNIX system can define their own cron jobs, ensuring flexibility and efficiency in task automation.

Example

A system administrator wants to automatically back up system logs every day at midnight. Instead of doing it manually, they use a cron job to run a script at 12:00 AM daily.

Exercise

1. List all currently scheduled cron jobs on your system.

2. Create a simple cron job that prints "Hello, Cron!" every minute.

Case Study: Automating System Backups with Cron Jobs

A large company requires **daily backups of its databases**. Instead of manually performing backups, the IT team schedules a cron job to **run a shell script at midnight**, ensuring consistent and automated backups.

CHAPTER 2: UNDERSTANDING THE CRONTAB FILE

What is the Crontab File?

The **crontab file** is a special configuration file that **lists scheduled tasks** for execution. Each user has a separate crontab file, managed using the crontab command.

To **edit a user's crontab file**, use:

crontab -e

To **list all cron jobs**, use:

crontab -l

To **remove all cron jobs**, use:

crontab -r

Cron Job Syntax

A cron job follows this syntax:

* * * * * command_to_execute

| | | | |

|||| └──── Day of the week (0-7, 0 or 7 = Sunday)

||| └────── Month (1-12)

|| └──────── Day of the month (1-31)

| └────────── Hour (0-23)

└──────────── Minute (0-59)

Asterisks (*) represent "every" possible value for that field.

Example: Running a Script Every Day at Midnight

0 0 * * * /home/user/backup.sh

This job runs /home/user/backup.sh every day at **midnight (00:00 hours)**.

Exercise

1. Write a cron job to execute a script **every Monday at 3 PM**.
2. Modify a cron job to run a cleanup script **on the first day of every month**.

Case Study: Automating Server Health Checks Using Cron

A web hosting company needs to **monitor server health** every hour. By scheduling a **cron job to run a monitoring script,** system administrators receive regular system status reports, allowing **proactive issue resolution**.

CHAPTER 3: SCHEDULING TASKS WITH CRON JOBS

Common Cron Job Scheduling Examples

Below are practical cron job schedules:

Task	Cron Syntax
Run a script every minute	* * * * * /path/to/script.sh
Run a script every 10 minutes	*/10 * * * * /path/to/script.sh
Run a script at 3 AM daily	0 3 * * * /path/to/script.sh
Run a script every Sunday at 2 PM	0 14 * * 0 /path/to/script.sh
Run a script on the 1st of every month	0 0 1 * * /path/to/script.sh
Run a script only on weekdays (Mon-Fri)	0 9 * * 1-5 /path/to/script.sh

Example: Automating Log Cleanup Every Week

0 2 * * 0 rm -rf /var/logs/*.log

This cron job deletes all log files in /var/logs/ **every Sunday at 2 AM**.

Using Special Keywords in Cron

Cron also provides shortcuts for commonly used schedules:

Keyword	Equivalent
@reboot	Runs once after reboot
@daily	Runs every day at midnight
@weekly	Runs every week
@monthly	Runs every month
@yearly	Runs once a year

Example: Running a Script Every Reboot

@reboot /home/user/startup.sh

This cron job **executes the script on system startup**.

Exercise

1. Schedule a script to run **every 5 minutes**.
2. Modify a cron job to execute a script **at 10 PM on the last day of the month**.

Case Study: Automating Log Rotation Using Cron Jobs

A **large-scale server generates logs continuously,** increasing storage usage. A **cron job is scheduled to rotate logs every night at midnight,** compressing old logs and removing files older than 7 days.

CHAPTER 4: MANAGING CRON JOBS WITH LOGGING AND ERROR HANDLING

Redirecting Cron Output to Log Files

By default, cron jobs do not display output. To log cron job output:

0 5 * * * /home/user/script.sh >> /home/user/log.txt 2>&1

- >> /home/user/log.txt → Appends output to log.txt.
- 2>&1 → Redirects both **standard output and errors** to the log file.

Example: Debugging a Failing Cron Job

If a cron job fails, redirect errors to a separate log file:

0 1 * * * /home/user/script.sh 2> /home/user/error.log

This captures errors in error.log for troubleshooting.

Using MAILTO for Notifications

Cron jobs can send **email notifications** on failure:

MAILTO="admin@example.com"

0 3 * * * /home/user/backup.sh

This sends an email to admin@example.com with **script output and errors**.

Example: Checking If a Cron Job Ran Successfully

To verify if a cron job executed, list recent cron logs:

grep CRON /var/log/syslog

Exercise

1. Modify a cron job to **log its execution output to a file**.
2. Set up a cron job that **sends an email notification on execution failure**.

Case Study: Monitoring Cron Job Execution in a Data Processing Pipeline

A **financial services company** processes thousands of transactions daily. A cron job runs a **data processing script every hour**. If the job fails, **email alerts are sent,** ensuring quick resolution.

CHAPTER 5: ADVANCED CRON JOB MANAGEMENT

Using Multiple Cron Jobs in a Single Crontab

A user can define **multiple cron jobs** in one crontab file:

0 6 * * * /home/user/script1.sh

30 6 * * * /home/user/script2.sh

15 7 * * * /home/user/script3.sh

This runs:

- script1.sh at **6:00 AM**
- script2.sh at **6:30 AM**
- script3.sh at **7:15 AM**

Restricting Cron Job Execution to Specific Users

System administrators can **control who can use cron jobs** using:

- **/etc/cron.allow** – Lists users who **can** use cron jobs.
- **/etc/cron.deny** – Lists users who **cannot** use cron jobs.

Example: Allowing Only admin to Use Cron

echo "admin" > /etc/cron.allow

Exercise

1. Create **multiple cron jobs** in a single crontab file.
2. Restrict cron job execution to a **specific user** using cron.allow.

Case Study: Using Cron Jobs for Automated Software Deployment

A **software development team** schedules cron jobs to **deploy new application updates nightly**, ensuring smooth, automated deployment without manual intervention.

CONCLUSION

This guide covered:

- **Scheduling tasks using cron jobs**.
- **Managing cron jobs with error handling and logging**.
- **Using advanced cron job techniques** for automation.

WRITING ADVANCED SCRIPTS FOR SYSTEM AUTOMATION IN UNIX

CHAPTER 1: INTRODUCTION TO ADVANCED SHELL SCRIPTING FOR SYSTEM AUTOMATION

What is System Automation with Shell Scripting?

System automation using shell scripts involves **writing scripts that perform repetitive tasks without manual intervention**. Advanced shell scripting is crucial for **system administrators, developers, and DevOps engineers** to optimize performance, increase efficiency, and reduce human errors.

Shell scripting is used for:

- **Automating system maintenance** – Log cleanup, user management, and software updates.
- **Automating backups** – Regularly saving critical data.
- **Monitoring system performance** – Checking CPU usage, memory consumption, and disk space.
- **Managing server configurations** – Automating firewall rules, network configurations, and security updates.
- **Deploying applications** – Managing software deployment pipelines.

By using **functions, error handling, loops, and cron jobs**, advanced scripts make complex system tasks more efficient and scalable.

Example: Automating User Account Creation

A script that **creates new user accounts** automatically, assigns home directories, and sets default passwords:

```
#!/bin/bash

echo "Enter username:"

read username

sudo useradd -m "$username"

echo "User $username created successfully!"
```

Exercise

1. Modify the script to **check if the user already exists before creating a new account**.
2. Extend the script to **assign a specific group and shell to the new user**.

Case Study: Automating Server Configuration in Large IT Infrastructure

A company deploys **hundreds of servers**. Instead of manually setting configurations, the IT team creates a **script that installs required packages, sets up network configurations, and applies security policies** automatically.

CHAPTER 2: USING FUNCTIONS FOR MODULAR SCRIPTING

Why Use Functions in Advanced Scripts?

Functions **break down large scripts into manageable parts,** making them easier to maintain and debug. They:

- **Improve reusability** – Use functions multiple times without rewriting code.
- **Enhance readability** – Keep scripts structured and organized.

- **Simplify debugging** – Isolate and test specific parts of a script separately.

Defining and Calling Functions in Shell Scripts

Basic Function Syntax

```
my_function() {
  echo "This is a function"
}
my_function  # Calling the function
```

Example: Automating Log File Cleanup Using Functions

```
#!/bin/bash

clean_logs() {
  echo "Cleaning old log files..."
  find /var/log -name "*.log" -mtime +7 -exec rm {} \;
  echo "Log cleanup completed!"
}

clean_logs
```

This function **deletes logs older than 7 days,** preventing unnecessary storage usage.

Exercise

1. Modify the script to **delete only log files larger than 1MB**.

2. Extend the script to **send an email notification after cleanup is completed**.

Case Study: Using Functions to Standardize Server Configurations

A cloud computing provider uses **functions within scripts** to **configure firewalls, set up networking rules, and install essential software packages** on every new server deployment.

CHAPTER 3: USING LOOPS FOR BATCH PROCESSING

Automating Repetitive Tasks with Loops

Loops enable scripts to **process multiple files, users, or directories** automatically, reducing manual work.

Using for Loops for Automation

```
#!/bin/bash

for user in alice bob charlie

do

 sudo useradd -m "$user"

 echo "Created user: $user"

done
```

This script **creates multiple user accounts** in a single execution.

Using while Loops for Monitoring

```
#!/bin/bash

while true
```

```
do
 echo "CPU Usage: $(top -bn1 | grep "Cpu" | awk '{print $2}')%"
 sleep 10
done
```

This script **monitors CPU usage every 10 seconds**.

Exercise

1. Modify the user creation script to **read user names from a text file and create them dynamically**.
2. Create a **while loop** that **monitors disk space and alerts the user when it is below 10%**.

Case Study: Automating File Transfer Using Loops

A company needs to **move thousands of files** between servers. A loop-based script **automates file transfers securely using rsync,** reducing human intervention.

CHAPTER 4: IMPLEMENTING ERROR HANDLING FOR RELIABLE AUTOMATION

Why is Error Handling Important?

Error handling ensures that a script **recovers from failures gracefully** instead of crashing unexpectedly. Advanced scripts must:

- **Check for missing files** before execution.
- **Validate user input** to prevent invalid commands.
- **Handle system interruptions** to avoid incomplete operations.

Using Exit Status Codes ($?)

Every UNIX command returns an **exit status**:

- 0 → Success
- Non-zero → Failure

Example: Checking Command Success

#!/bin/bash

mkdir /backup || echo "Error: Failed to create directory."

If mkdir fails, an error message is displayed.

Using trap for Error Handling

trap captures **script interruptions (Ctrl+C, errors) and performs cleanup**.

#!/bin/bash

trap "echo 'Script interrupted! Cleaning up...'; exit 1" SIGINT SIGTERM

while true; do sleep 1; done

Exercise

1. Modify a script to **log all errors into a file** instead of displaying them on the screen.
2. Implement trap in a script to **delete temporary files if interrupted**.

Case Study: Preventing Data Loss with Robust Error Handling

A **database migration script** failed midway, corrupting data. After adding **error handling and logging mechanisms,** the IT team could detect issues in real-time and roll back changes automatically.

Chapter 5: Automating Task Scheduling with Cron Jobs

Scheduling Tasks with Cron Jobs

For fully automated system administration, shell scripts are **scheduled using cron jobs,** running them at fixed intervals.

Example: Automating Backups with Cron

0 3 * * * /home/user/backup.sh >> /home/user/backup.log 2>&1

This job **runs every night at 3 AM** and logs the output.

Using anacron for Daily Automation

Unlike cron, anacron ensures missed jobs run when the system is back online:

1 5 backup-job /home/user/backup.sh

Runs **backup script once per day,** even if the system was offline at the scheduled time.

Exercise

1. Schedule a **cron job to monitor disk usage every hour** and log results.
2. Use anacron to **schedule weekly system updates**.

Case Study: Automating Cloud Server Maintenance with Cron

A **cloud service provider** runs **automatic software updates, log rotations, and security patches** using scheduled cron jobs, ensuring **24/7 uptime and security**.

CHAPTER 6: ADVANCED LOGGING AND MONITORING IN SHELL SCRIPTS

Logging Script Execution for Troubleshooting

To monitor automation scripts, logging is crucial.

Example: Logging with Timestamps

```
#!/bin/bash

logfile="/var/log/myscript.log"


echo "$(date): Script started" >> $logfile

echo "Performing system checks..." >> $logfile
```

Using logger to Send Logs to System Log

```
#!/bin/bash

logger "Automated script executed successfully!"
```

This sends logs to /var/log/syslog.

Exercise

1. Modify a script to **log all executed commands and errors into a log file**.
2. Use logger to **send log messages to the system log**.

Case Study: Monitoring Server Health in a Production Environment

A **server health check script logs system performance** every 5 minutes, helping administrators detect issues **before system failure occurs**.

CONCLUSION

This guide covered:

- **Using functions and loops** for modular scripting.
- **Implementing error handling** for reliable automation.
- **Automating scripts with cron jobs**.
- **Logging and monitoring system automation scripts**.

Assignment Solution: Automating User Creation and Permission Settings with a Shell Script

Objective

The goal of this assignment is to **write a shell script that automates the process of user creation, assigns them a home directory, sets appropriate permissions, and adds the user to a specific group**. This script ensures that new users are configured consistently and securely.

Step 1: Define the Script Requirements

The script will:

1. **Take a username as input** from the user.
2. **Check if the user already exists** to avoid duplication.
3. **Create the user account** with a home directory.
4. **Assign the user to a specified group** (e.g., developers).
5. **Set the default shell for the user** (e.g., /bin/bash).
6. **Set specific permissions** on the home directory.
7. **Generate a default password** for the new user.
8. **Log all operations** for future reference.

Step 2: Create the Shell Script

Writing the Shell Script

Create a script file using a text editor:

nano create_user.sh

Add the following script:

```
#!/bin/bash


# Define a log file to store operations
LOGFILE="/var/log/user_creation.log"


# Function to check if the script is run as root
check_root() {
   if [ "$(id -u)" -ne 0 ]; then
      echo "Error: This script must be run as root." | tee -a $LOGFILE
      exit 1
   fi
}


# Function to create a user
create_user() {
   echo "Enter the username for the new user:"
   read username
```

```
# Check if user already exists
if id "$username" &>/dev/null; then
    echo "Error: User '$username' already exists!" | tee -a $LOGFILE
    exit 1
fi

# Create the user with home directory
useradd -m -s /bin/bash "$username"
if [ $? -eq 0 ]; then
    echo "User '$username' created successfully." | tee -a $LOGFILE
else
    echo "Error: Failed to create user." | tee -a $LOGFILE
    exit 1
fi

# Assign user to 'developers' group
usermod -aG developers "$username"
echo "User '$username' added to 'developers' group." | tee -a $LOGFILE
```

```
    # Set user permissions on home directory
    chmod 750 /home/"$username"
    echo "Permissions set for /home/$username." | tee -a $LOGFILE

    # Generate a default password
    PASSWORD=$(openssl rand -base64 12)
    echo "$username:$PASSWORD" | chpasswd
    echo "Default password set for $username." | tee -a $LOGFILE

    # Display credentials to the administrator
    echo "User $username created successfully!"
    echo "Username: $username"
    echo "Password: $PASSWORD"
    echo "Remember to change the password on first login."
}

# Run the functions
check_root
create_user
```

STEP 3: MAKE THE SCRIPT EXECUTABLE

Save the file (Ctrl + X, then Y in nano) and **grant execute permissions** to the script:

chmod +x create_user.sh

Step 4: Run the Script

Execute the script using:

sudo ./create_user.sh

- **It prompts for a username** and checks if the user already exists.
- **Creates the user, assigns them to the developers group, and sets permissions.**
- **Generates a secure random password for the user.**

Step 5: Verify User Creation

Check if the User Exists

id username

Verify Home Directory and Permissions

ls -ld /home/username

Expected output:

drwxr-x--- 1 username developers 4096 Feb 24 14:00 /home/username

Check Assigned Group

groups username

Expected output:

username : username developers

Step 6: Logging and Debugging

The script logs all operations to /var/log/user_creation.log. To view logs:

cat /var/log/user_creation.log

Conclusion

This script **automates user creation and permission settings,** making system administration more efficient and reducing human errors. It:

✅ Checks if the script is run as root.
✅ Ensures that a user does not already exist before creation.
✅ Creates a user with a home directory and assigns them to a group.
✅ Sets default permissions on the home directory.
✅ Generates a random password and logs details securely.

Assignment Solution: Creating a Cron Job to Automatically Back Up Files

Objective

The goal of this assignment is to **create a cron job that automatically backs up important files or directories at regular intervals**. This ensures that critical data is always backed up without manual intervention.

By the end of this guide, you will be able to:

✅ **Write a shell script for file backup.**

✅ **Schedule a cron job to execute the backup script automatically.**

✅ **Verify and manage cron jobs.**

Step 1: Define the Backup Requirements

The backup script should:

1. **Select the files/directories to back up** (e.g., /home/user/documents).
2. **Create a compressed archive** with a timestamp.
3. **Store the backup in a specific location** (e.g., /backup).
4. **Remove old backups to save space** (optional).
5. **Log backup activity** for monitoring.

Step 2: Create the Backup Shell Script

1. Create a Backup Script

Open a terminal and create a new script file:

nano backup.sh

2. Add the Following Script

```
#!/bin/bash


# Define backup source and destination

SOURCE="/home/user/documents"

DESTINATION="/backup"

LOGFILE="/var/log/backup.log"


# Ensure backup destination exists

mkdir -p "$DESTINATION"


# Generate a timestamped archive filename

BACKUP_FILE="$DESTINATION/backup_$(date +'%Y-%m-%d_%H-%M-%S').tar.gz"


# Create a compressed archive of the source directory

tar -czf "$BACKUP_FILE" "$SOURCE" 2>> "$LOGFILE"
```

```
# Verify if backup was successful
if [ $? -eq 0 ]; then
  echo "$(date): Backup successful - $BACKUP_FILE" >> "$LOGFILE"
else
  echo "$(date): Backup failed!" >> "$LOGFILE"
fi

# Remove old backups older than 7 days (optional)
find "$DESTINATION" -type f -name "backup_*.tar.gz" -mtime +7 -exec rm {} \; >> "$LOGFILE" 2>&1
```

Step 3: Make the Script Executable

After saving the file (Ctrl + X, then Y in nano), **grant execution permissions**:

```
chmod +x backup.sh
```

Step 4: Manually Test the Backup Script

Run the script to check if it works correctly:

```
./backup.sh
```

After execution:

- Check if the backup file is created in /backup.
- Verify the log file using:

- cat /var/log/backup.log

Step 5: Schedule the Backup Using Cron

1. Open the Crontab File

To schedule the backup job, edit the crontab:

crontab -e

2. Add the Cron Job

Add the following line to schedule the backup **daily at 2 AM**:

0 2 * * * /home/user/backup.sh >> /var/log/backup.log 2>&1

Cron Job Breakdown

0 2 * * * /home/user/backup.sh

| | | | |

| | | | └──── Day of the week (0-7, 0 or 7 = Sunday)

| | | └────── Month (1-12)

| | └──────── Day of the month (1-31)

| └────────── Hour (0-23)

└──────────── Minute (0-59)

This means **the script runs every day at 2 AM**.

Step 6: Verify the Scheduled Cron Job

1. List All Scheduled Cron Jobs

To confirm the cron job is added:

crontab -l

2. Check Cron Job Execution Logs

Since cron logs are stored in /var/log/syslog, check recent executions:

grep CRON /var/log/syslog

Step 7: Managing Cron Jobs

Modify the Cron Job

Edit crontab:

crontab -e

Change the schedule, e.g., **run every Monday at 3 AM**:

0 3 * * 1 /home/user/backup.sh

Remove the Cron Job

To delete the cron job:

crontab -r

Conclusion

This assignment covered:

✅ Writing a **backup shell script** to archive important files.
✅ Scheduling the backup script using **cron jobs**.
✅ **Verifying and managing cron jobs** for automated execution.