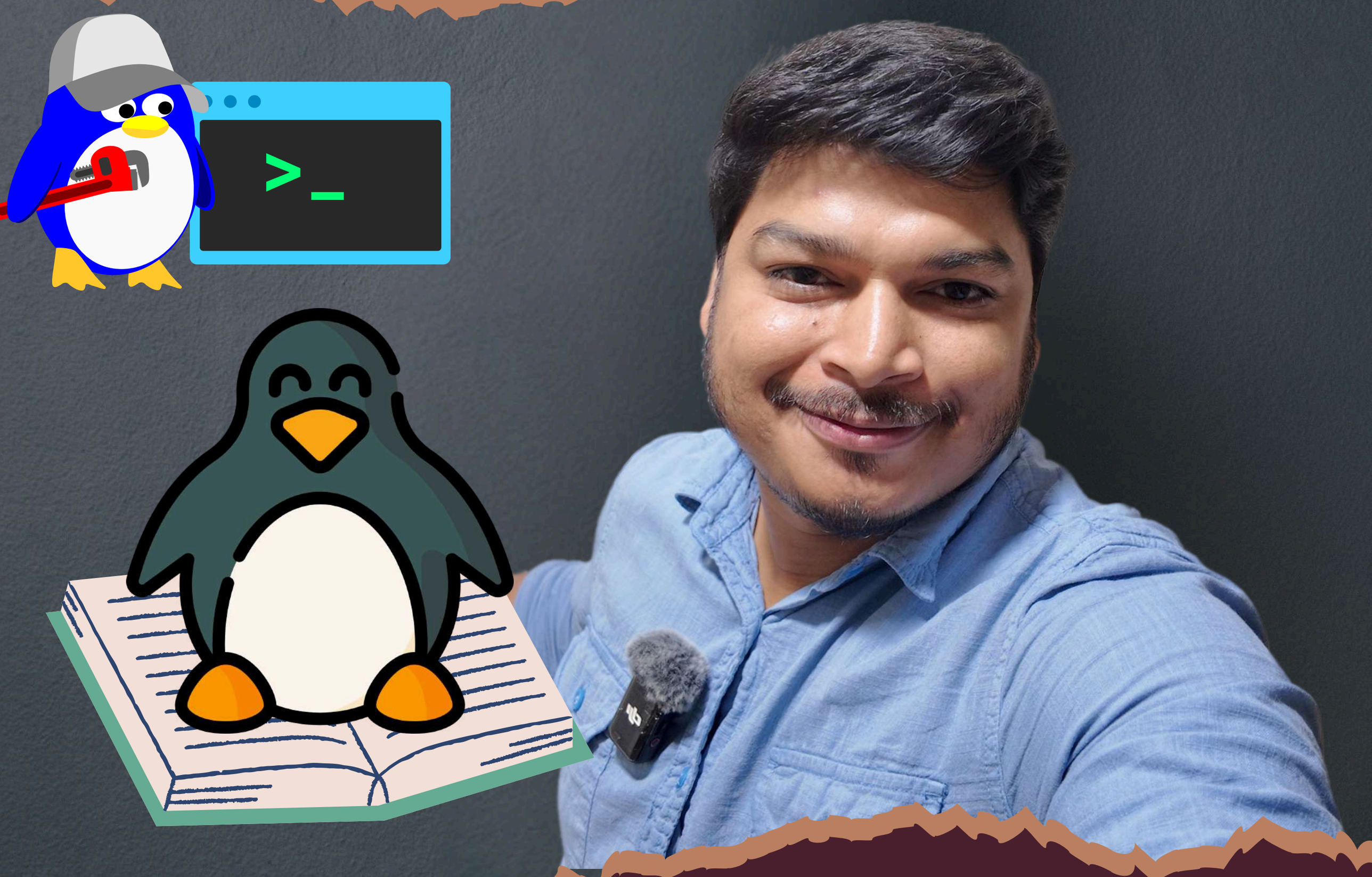


SHELL SCRIPTING HANDBOOK

From Beginner to Expert

For Cloud & DevOps Engineers

By :

SANDIP DAS

LearnXOps.com

Table of Contents

ABOUT ME

Hi! I’m Sandip Das, a Senior Cloud, DevOps & MLOps Engineer with 12+ years of experience, primarily working on production-grade Linux servers. The majority of my work revolves around automating, scaling, and optimizing infrastructure for clients across the US, UK, EU, and UAE.

LearnXOps.com

Introduction

🐧 What is Shell Scripting?

A **shell is a command-line interface** that **allows users to interact with an operating system**.

Shell scripting is the practice of writing sequences of shell commands in a file (a "script") to automate tasks.
Instead of typing commands manually, you write a reusable program to do the work for you.

Key Benefits:

1. **Automation**: Save time by scripting repetitive tasks (e.g., backups, log parsing).
2. **Reproducibility**: Ensure consistency in workflows.
3. **Power**: Combine system tools (e.g., grep, sed, awk) to solve complex problems.

Common Use Cases

1. **File Management**: Batch-renaming files, organizing directories.
2. **System Administration**: User management, disk usage reports.
3. **Data Processing**: Extracting logs, transforming CSV files.
4. **Automation**: Scheduled backups, email alerts, software installations.

What is Bash ?

Bash (GNU Bourne Again Shell) is a popular Unix shell and scripting language, designed as a free software replacement for the original Bourne shell. Bash was written by Brian Fox for the GNU Project and first released in 1989. It has since become the default login shell on most Linux distributions and was the default

Bash serves a dual role:

- **Interactive Shell**: When you open a terminal on Linux or macOS, Bash provides a prompt where you can type commands. It includes features like history, aliases, and tab completion to enhance the command-line experience.
- **Scripting Language**: Bash can run scripts (text files containing shell commands) to automate sequences of operations. Bash scripts can range from simple automation of a few commands to complex programs with functions, flow control, and error handling.

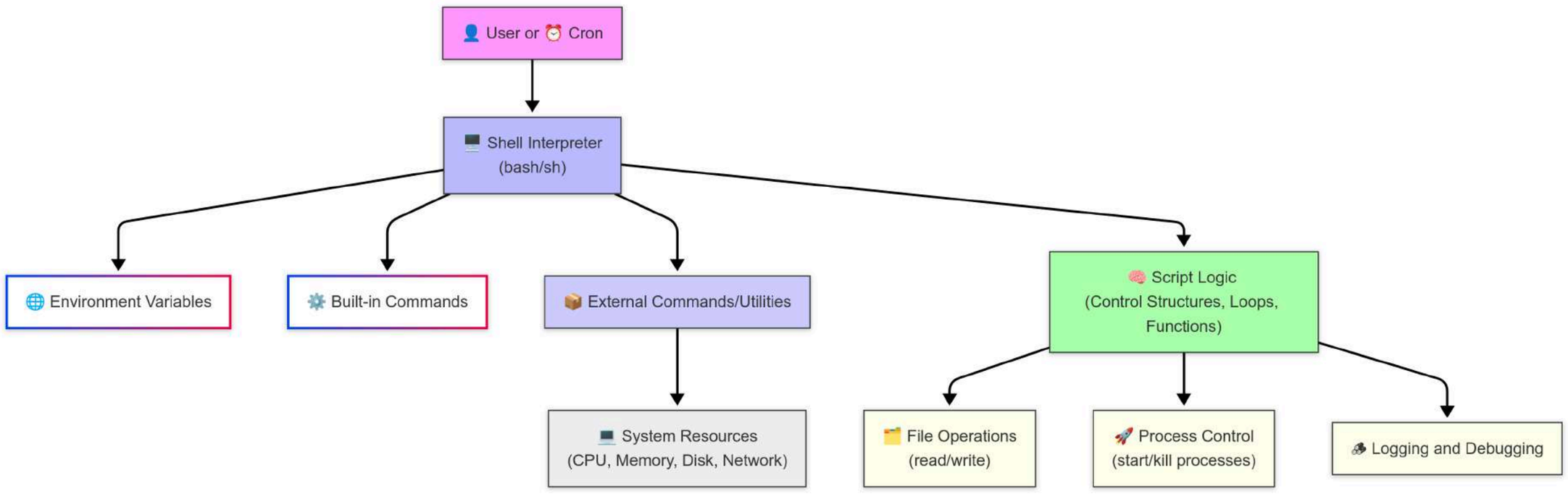
Why Learn Shell Scripting?

- **Ubiquity**: Shells are available on all Unix-like systems (Linux, macOS, WSL).
- **Efficiency**: Scripts handle tasks faster than manual execution.
- **Foundation for DevOps:** Essential for tools like Docker, CI/CD pipelines, and server management.

Bash is not the fastest or most elegant programming language, but its strength is leveraging the power of other programs. A Bash script can call dozens of existing utilities (grep, sed, awk, curl, tar, etc.) to perform work, which is often more efficient than writing that functionality from scratch

@Sandip Das

Shell Scripting Flow



👤 User or ⏰ Cron

Shell scripts can be triggered manually by a User or automatically using Cron jobs (time-based job scheduler). This is the entry point to script execution.

💻 Shell Interpreter (bash/sh)

The shell interpreter (like bash or sh) parses and executes shell commands. It is responsible for interpreting the logic, commands, and syntax written in a shell script.

The interpreter drives:

🔵 **Environment Variables**
- These provide configuration and context for script execution (e.g., $PATH, $HOME, $USER).
- Used for controlling program behavior and customizing environments.

⚙️ **Built-in Commands**
- Commands that are part of the shell itself (e.g., cd, echo, read, set).
- Executed directly by the shell without forking a new process.

📦 **External Commands/Utilities**

- These are standard system utilities (e.g., grep, awk, curl, ls) that the shell invokes via forking.
- Typically found in /bin, /usr/bin, etc.

🔽 These may operate on:
- 🖥️ **System Resources (CPU, Memory, Disk, Network)**: Resource monitoring, data processing, or automation.

🧠 **Script Logic (Control Structures, Loops, Functions)**
This forms the core logic of any script, including: if-else, case, while, for, functions, etc and do work with below cases:
- 🗂️ **File Operations (read/write)**: Handling files via cat, >>, <<, >, read, etc.
- 🚀 **Process Control (start/kill processes)**: Using commands like ps, kill, nohup, &, etc.
- 🔍 **Logging and Debugging**: Using echo, set -x, trap, and redirecting stdout/stderr to logs for troubleshooting.

Setting Up Bash Environment (Linux/macOS/WSL)

Bash on Linux

Bash is pre-installed on most Linux distros.

- **Check version**: echo "$BASH_VERSION"
- **Start Bash manually**: bash
- **Default shell**: check /etc/passwd
- **Editor**: Use Vim, Nano, VS Code, etc.
- **Manual**: man bash

Running a Script

```
#!/usr/bin/env bash  # Shebang
chmod +x myscript.sh  # Make executable
./myscript.sh        # Run script
bash myscript.sh     # Alternative without execute bit
```

What is Shebang ?

Shebang (#!) is the first line in a script that tells the system which interpreter to use to execute the file.

```
#!/bin/bash
echo "Hello, World!"
```

Bash on macOS

Pre-installed Bash is v3.2
(due to GPLv2 licensing)

- Default shell is zsh on macOS Catalina+
- Start Bash manually: bash
- **To switch shell**: chsh -s /bin/bash
- **Upgrade Bash** (optional):

brew install bash

Then update:
/etc/shells + chsh -s /usr/local/bin/bash

Running a Script

```
chmod +x script.sh
./script.sh
```

⚙️ Shell Config Files

- **~/.bash_profile** or **~/.bash_login**: Run on login shells
- **~/.bashrc**: Run on interactive non-login shells
- **~/.bash_profile** usually sources ~/.bashrc

Scripts/cron jobs don't source these files automatically.

To run scripts globally:

```
# Add this in ~/.bashrc or ~/.bash_profile
export PATH="$HOME/bin:$PATH"
```

Bash on Windows (via WSL)

Use WSL (Windows Subsystem for Linux)

- Install: **wsl --install** (Ubuntu by default)
- Run Bash like Linux inside WSL
- Access Windows files under /mnt/c/

Other Types of Shells

sh: The original Bourne Shell (simpler, less feature-rich).

zsh: Enhanced with plugins and themes (popularized by Oh-My-Zsh).

fish: User-friendly syntax but less POSIX-compliant.

@Sandip Das

Terminal Basics

To Prepare Yourself for Effective Scripting

Check Your Default Shell:

```
echo $SHELL
# Prints the path to your default shell (e.g., /bin/bash)
```

Change Your Default Shell:

```
chsh -s /bin/bash
# Replace with your preferred shell path
```

Common Terminal Commands

Command	Description	Example
clear	Clear the terminal screen	clear
history	View command history	`history
man	Open a command's manual	man ls
Ctrl+C	Kill a running process	(Press during execution)
Ctrl+D	Exit the terminal	(Press to exit)

Key Commands

Command	Purpose	Example
pwd	Show current directory	pwd
cd	Change directory	cd /var/www
ls	List files	ls -la
cp	Copy files	cp -r src/ dest/
mv	Move/rename files	mv old.txt new.txt
rm	Delete files	rm -rf tmp/
touch	Create files	touch notes.md

Navigating the Filesystem:

```
pwd       # Print current directory
cd ~/docs # Change to the "docs" folder in your home directory
ls -l     # List files with details (permissions, size)
mkdir project # Create a directory named "project"
```

File Permissions

Use chmod to make scripts executable:

```
chmod +x script.sh
# Allow execution
chmod 755 script.sh
# Set read/write/execute for owner, read/execute for others
```

Permission Codes:

Code	Permission
7	Read + Write + Execute (4+2+1)
6	Read + Write (4+2)
5	Read + Execute (4+1)

Environment Variables

Configure variables for scripts to use:

```
export EDITOR="nano"   # Set default editor
echo $EDITOR           # Print the variable
```

Text Editors

Edit scripts with lightweight editors like vim or nano:

Nano (Beginner-Friendly):

```
nano script.sh  # Open/create a file
# Use Ctrl+O to save, Ctrl+X to exit
```

The PATH Variable:

Add custom script directories to PATH for global access:

```
export PATH="$PATH:$HOME/scripts"  # Add ~/scripts to PATH
# Add this line to ~/.bashrc or ~/.zshrc to make it permanent
```

Vim (Advanced):

```
vim script.sh
# Press `i` to enter insert mode,
#`Esc` to exit, `:wq` to save and quit
```


@Sandip Das

Your First Shell Script

The Shebang Line (#!/bin/bash)

The **shebang line (#!)** tells the system which interpreter to use for executing the script. It must be the first line in your script.

Common Shebang Examples:

```
#!/bin/bash     # Use Bash (most common)
#!/bin/sh      # Use the system's default shell (often Bash, but not always)
#!/usr/bin/env python3 # Use Python 3 (for cross-platform scripts)
```

Writing and Executing a Script

Step 1: Create the Script File

nano hello.sh

Step 2: Add Code

```
#!/bin/bash
echo "Hello from $(hostname)!"
echo "Today is $(date +'%Y-%m-%d')"
```

$(command) executes the command and inserts its output into the string

Step 3: Make It Executable

```
chmod +x hello.sh
```

Step 4: Run the Script

```
./hello.sh      # Execute from the current directory
bash hello.sh   # Alternative (no need for +x)
```

Output:

```
Hello from Sandips-MacBook-Pro.local!
Today is 2025-04-18
```


Syntax Basics

Variables and Data Types

Storing and Manipulating Data in Scripts

Declaring Variables

Variables store data for reuse in your scripts. Shell variables are untyped (treated as strings by default) and case-sensitive.

Syntax:
variable_name="value" # No spaces around the '='

Bash variables are essentially untyped containers for text (strings).
They can also hold numbers, but Bash will treat those numbers as strings unless used in a math context.

Unlike many languages, we do not use keywords like int or string to declare variables, and we shouldn't have spaces around the equals sign. Here's how you assign a variable:

name="Sandip" # Correct
age=35 # Correct (30 is stored as characters '3' '0')
greeting = "Hello" # WRONG (no spaces allowed around '=' in assignment)

Rules:

- **Names**: Can contain letters, numbers, and underscores. Start with a letter or _.
- **Access Values**: Use $variable_name or ${variable_name}.
- **Quotes**: Use double quotes to preserve spaces or special characters:

message="Hello, $user!" # Variable expanded inside double quotes

Key points:

- No $ when assigning. Use $ only when referring to (expanding) the variable's value.
- Variables are by default global in scope (we'll discuss scope in the Functions section). You can create local variables within functions using the local keyword.
- To retrieve the value, prefix with $: e.g., echo $name. It's good practice to wrap the variable in braces if it's adjacent to other characters: echo "${name}!" ensures Bash knows where the variable name ends.

A bit more on the naming convention:
Variable names can contain letters, numbers, and underscores, but cannot start with a number. By convention:

- Environment or configuration variables are often UPPERCASE (e.g., $HOME, $PATH).
- Local variables in scripts can be lowercase or mixed case. (This convention helps avoid accidentally overwriting important env vars.)

Unsetting:
We can unset a variable with **unset VAR**. Until a variable is set, using it will result in an empty string (or an error if set -u is on).

e.g. **unset name**

Syntax Basics

Variables and Data Types

Special variables:

Some variables are set by the shell or have special meaning:

- **$HOME, $USER, $PWD** – environment variables for home directory, username, present working directory.
- **$0, $1, $2, ... $#, $@, $*** – set when a script runs: $0 is the script's name, $1 through $n are the arguments, $# is number of args, $@ is all args as separate words.
- **$? –** exit status of the last command (0 means success).
- **$$ –** PID of the current shell (useful if you need a unique temp filename like /tmp/mytmp.$$).
- **$! –** PID of the last background process started by this shell.
- **$IFS –** Internal Field Separator, default is whitespace. Controls word splitting (discussed in Quoting section).

Quoting

Single Quotes ('...')

- Treat everything literally inside.
- No variable or command expansion.
- Can't nest ' directly — break and escape if needed.

Example:

```
echo '$HOME'   # Prints: $HOME
```

Double Quotes ("...")

- Allows expansion:
  - $VAR, $(command), $(()) work.
  - \ escapes special characters like " or \.
- Prevents word splitting and glob expansion.

Example:

```
echo "Hello, $name!"  # Expands to: Hello, Alice!
```

Rule of Thumb
Always quote variables like $VAR unless you intentionally want expansion and splitting.

Quote it when in doubt.
Safer scripts, fewer surprises.

Backslash Escapes (\)

- Escape characters like space, $, ", or \.
- Works outside quotes and inside double quotes.

Example:

```
file="My Documents/report.txt"
rm $file     # ❌ BAD
rm "$file"  # ✅ GOOD
```

Combining Quotes:
e.g. To echo It's 5 o'clock safely:

```
echo 'It'\''s 5 o'\''clock'  # Messy
# OR
echo "It's 5 o'clock"        # Cleaner
```

📜 **Heredoc with Quotes**

```
cat <<'EOF'
This is literal. $VAR will not expand.
EOF
```

Syntax Basics

Variables and Data Types

Strings

Basic Operations:

Concatenation:

```
greeting="Hello, "$user"!"  # Or greeting="Hello, ${user}!"
```

Length:

```
echo ${#greeting}  # Prints the number of characters
```

Substrings:

```
str="abcdef"
echo ${str:1:3}    # Output: "bcd" (start at index 1, length 3)
```

Numbers

Though variables are strings, you can perform arithmetic with $(( )) or let:

Integer Arithmetic:

```
num=10
result=$((num + 5))     # result=15
((num++))               # Increment: num=11
```

Example Script:

```
#!/bin/bash
read -p "Enter a number: " n
square=$((n * n))
echo "Square of $n is $square"
```

Arrays

Bash supports indexed arrays (lists) and associative arrays (key-value pairs).
Indexed Arrays:

```
fruits=("Apple" "Banana" "Cherry")
fruits[3]="Date"

# Access elements
echo ${fruits[1]}      # Output: "Banana" (index starts at 0)
echo ${fruits[@]}      # Output: All elements
echo ${#fruits[@]}     # Output: Number of elements (4)
```

@Sandip Das

Syntax Basics

Command Syntax and Command Substitution

Bash Command Basic Structure

command_name arg1 arg2 arg3 ...

- The first word is the command (built-in, external, or function).
- Remaining words are arguments.
- Bash splits by whitespace unless quoted.

Example:

cp "My File.txt" /backup/MyFile.txt

🗨️ **Comments**

Start with #. Ignored by shell unless inside quotes.

 # List files in long format
ls -l # Inline comment

Line Continuation
Use \ to continue a command on the next line.

**echo "This is a long line that I'm splitting **
into two lines for readability."

Command Separators

echo "Start"; ./script.sh; echo "End"

- **; = separates commands (like a newline)**
- **&& = run next only if previous succeeded**
- **|| = run next only if previous failed**

Command Substitution

Capture the output of a command into a variable:

Syntax:
result=$(command) # Preferred method
result=`command` # Older backtick syntax (avoid for readability)

Example: List files modified today:

today_files=$(find . -mtime -1)
echo "Recent files: $today_files"

Brace Expansion
echo {1..5}   # 1 2 3 4 5
echo file{.txt,.bak} # file.txt file.bak

Built-in vs External Commands
type cd   # shell builtin
type grep   # external binary (/bin/grep)

Control Structures

Making Decisions and Repeating Actions

Conditional Statements

Control structures allow scripts to execute code conditionally or repeatedly.

1. The if Statement
Basic syntax:

```
if [ condition ]; then
  # code to run if true
elif [ another_condition ]; then
  # code for this case
else
  # default code
fi
```

Example: Check if a file exists

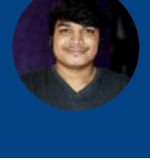
```
#!/bin/bash
file="report.txt"

if [ -f "$file" ]; then
  echo "$file found."
else
  echo "$file not found."
fi
```

Common Test Operators

Type	Test	Description
File	-f file	Is a regular file?
	-d dir	Is a directory?
	-e file	Does it exist?
	-r file	Is it readable?
	-w file	Is it writable?
	-x file	Is it executable?
String	-z "$str"	Is string length zero?
	-n "$str"	Is string length non-zero?
	"$a" = "$b" or == in [[ ]]	Are strings equal?
	"$a" != "$b"	Are strings not equal?
Numeric	-eq, -ne, -lt, -le, -gt, -ge	Integer comparisons
Logic	[[$a -gt 5 && $b -lt 10]]	Combine with && / `

Important syntax points:

- The **then** must appear **after the condition** (on the same line or next line). If on the same line, **you need a semicolon before then (as shown)**.
- Every **if is closed with a corresponding fi (fi = if backwards)**.
- **elif (else-if) is optional;** you can have multiple elif clauses to check additional conditions in order.
- **else is optional;** without it, if none of the conditions is true, the script just skips the if block entirely.



@Sandip Das

Control Structures

Making Decisions and Repeating Actions

Case: Multiple Choices

A case statement is like a multi-branch if, useful for matching one variable or value against several patterns. I

```
case "$variable" in
  pattern1)
    # code for pattern1
    ;;
  pattern2|pattern3)
    # code for pattern2 or pattern3
    ;;
  *)
    # default case
    ;;
esac
```

Example: Handle User Input

```
read -p "Continue? (y/n) " answer
case "$answer" in
    Y|y|yes|Yes)
        echo "Continuing..."
        ;;
    N|n|no|No)
        echo "Quitting..."
        exit 0
        ;;
    *)
        echo "Invalid response." >&2
        exit 1
        ;;
esac
```

Example: Pattern matching

```
os="linux"
case "$os" in
    linux|darwin)
        echo "Unix-like system"
        ;;
    cygwin*|msys*)
        echo "Windows with POSIX layer (Cygwin/MSYS)"
        ;;
    *)
        echo "Unknown OS"
        ;;
esac
```

How it works:

- Bash evaluates the value of $variable and tries to match it against each pattern in order. The patterns are shell patterns (globs), not regular expressions.
- The first pattern that matches, its corresponding block executes. ;; ends that block (and causes case to terminate; no “fall-through” by default in shell case).
- | can combine multiple patterns for the same block.
- *) is a catch-all pattern that matches anything (like default). It’s good practice to include *) to handle unexpected values.
- Patterns can include wildcards: e.g., file*.txt) matches any string starting with "file" and ending in ".txt".

Control Structures

Making Decisions and Repeating Actions

Loop

Loops allow you to execute a block of code repeatedly. Bash has three main loop constructs:

for Loop: Iterate over a list or range:

```
for item in list; do
  # code
done
```

C-Style for Loop:

```
for ((i=0; i<5; i++)); do
  echo "Count: $i"
done
```

Example: Process Files

```
#!/bin/bash
for file in *.txt; do
  echo "Processing $file..."
  mv "$file" "backup_$file"
done
```

Until

Run code until a condition becomes true.
Syntax:

```
until [ condition ]; do
  # code
done
```

Example: Wait for a File to Exist

```
#!/bin/bash
until [ -f "lockfile" ]; do
  echo "Waiting for lockfile..."
  sleep 2
done
echo "Lockfile detected!"
```

Loop Control

- **break**: Exit the loop immediately.
- **continue**: Skip the rest of the current iteration.

Example: Continue: Skip Even Numbers

```
for num in {1..10}; do
  if [ $((num % 2)) -eq 0 ]; then
    continue
  fi
  echo "$num is odd."
done
```

Example:Break : User Age Checker

```
#!/bin/bash
while true; do
  read -p "Enter your age (or 'quit'): " age

  case "$age" in
    [Qq]uit)
      break
      ;;
    *)
      if [ "$age" -lt 18 ]; then
        echo "You're a minor."
      else
        echo "You're an adult."
      fi
      ;;
  esac
done
```

💡 Pro Tips : Avoid this common mistakes

Missing Spaces in Conditions:

```
if [$var -eq 5]   # Wrong
if [ $var -eq 5 ] # Correct
```

Unquoted Variables in Tests:

```
if [ -f $file ]   # Fails if $file has spaces
if [ -f "$file" ] # Correct
```

@Sandip Das

Functions & Scope

Modularizing Code for Reusability and Clarity

Functions allow us to group a set of commands and give it a name, which can then be called (executed) multiple times within our script. Functions help avoid repetition and make scripts more modular and readable.

Additionally, understanding variable scope (global vs local) is important to prevent unintended side effects between different parts of your script.

Defining and Calling Functions

Syntax

```
function_name() {
  # Code to execute
}

# Or (alternative syntax)
function function_name {
  # Code
}
```

Example: Simple Greeting Function

```
greet() {
  echo "Hello, $1!"  # $1 is the first argument
}

greet "Sandip"  # Call the function with "Sandip" as an argument
```

When we call greet "Sandip", inside the function:

- $1 is "Alice".
- $2, $3, etc. would correspond to additional arguments if provided.
- $0 is still the script name, not the function name.
- The special variables like $?, $PIPESTATUS, etc., and environment behave as usual.
- Functions do not create a new process (they run in the current shell process), unlike executing an external script which spawns a subshell. This means functions can directly modify shell variables, change directories, etc., in the main shell environment (which is powerful but also a source of bugs if not careful with naming).

Passing Arguments

Functions use positional parameters ($1, $2, ..., $@, $#) to access arguments, just like scripts.

Example: Calculate Sum

```
add() {
  sum=$(( $1 + $2 ))
  echo "$sum"
}

result=$(add 5 3)
echo "5 + 3 = $result"
```

Important Points:

- Inside a function, **$1, $2**, ... **refer to that function's arguments**, **not the script's arguments**. **The script's own $1...$n are temporarily shadowed**. After the function returns, the script's $1...$n are unchanged.
- **$@** inside a function expands to the function's args (separately quoted), and **$#** is the count of args passed to the function.
- We can still access the script's parameters by other means if needed (like by saving them to global variables or using shift, but that gets messy – better pass what we need to the function).

Functions & Scope

Modularizing Code for Reusability and Clarity

Return Values

- **Exit Status**: Functions return an exit code (0 for success, 1-255 for failure) via return.
- **Output**: Use echo or printf to "return" data to the caller.

Example: Check File Existence

```
file_exists() {
  if [ -f "$1" ]; then
    return 0  # Success
  else
    return 1  # Failure
  fi
}

if file_exists "config.txt"; then
  echo "Config file found."
fi
```

Exit Status Code Table

Exit Code	Meaning
0	Success – The command completed successfully.
1	General error – Catch-all for general errors.
2	Misuse of shell builtins (according to Bash documentation).
126	Command invoked cannot execute (e.g., permission issue).
127	Command not found – Invalid command or missing binary.
128	Invalid argument to exit (e.g., `exit 300` returns 44).
128+n	Fatal error signal "n" – Process was terminated by signal n.
130	Script terminated by Ctrl+C (i.e., signal 2 – SIGINT).
137	Process killed (e.g., OOM or `kill -9`, which is SIGKILL/9).
139	Segmentation fault (e.g., signal 11 – SIGSEGV).
143	Terminated by SIGTERM (signal 15).
> 255	Exit codes wrap around modulo 256, only 0–255 are valid.

Example: Return Straing Data

return in **Bash functions is NOT for returning a data** value like in other languages (**you can't return a string directly**). **It's only for exit status**. To get data out of a function, you typically:

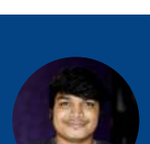
Use **echo or printf** to output the data, and capture it via command substitution when calling the function:

```
get_date() {
    echo "$(date +%Y-%m-%d)"
}
today=$(get_date)
```

Recursion

Functions can call themselves (allowing recursion), but Bash isn't really optimized for deep recursion, and there's a limit (the default maximum function call depth is not very high, and you can easily hit a stack overflow or the FUNCNEST limit). Iterative approaches are usually better in Bash for long loops.

```
#!/bin/bash

factorial() {
  local n=$1
  if (( n <= 1 )); then
    echo 1
  else
    local prev=$(factorial $((n - 1)))
    echo $((n * prev))
  fi
}

# Example usage:
echo "Factorial of 5 is: $(factorial 5)"
```



@Sandip Das

Functions & Scope

Modularizing Code for Reusability and Clarity

Variable Scope

By **default**, **Bash variables are global**. If you set a variable inside a function, it will affect that variable outside the function as well (if it exists, it will overwrite it; if not, it creates a new global variable visible after the function). This can lead to unintended interactions. For example:

```
count=0
increment() {
    count=1   # modifies the global count!
}
increment
echo "$count"  # outputs 1
```

Local:

Using local inside a function makes the variable's scope limited to that function (and it also shadows any global of the same name). When the function returns, the local variables are gone (destroyed).

```
demo_scope() {
  local var1="Local"  # Only accessible inside the function
  var2="Global"       # Modifies the global variable
}

var2="Initial"
demo_scope
echo "var1: $var1"    # Output: "" (not defined)
echo "var2: $var2"    # Output: "Global"
```

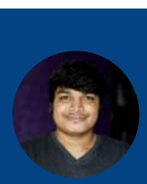
Some Practical Examples

Logger Function

```
log() {
  local message="$1"
  echo "[$(date +'%Y-%m-%d %H:%M:%S')] $message" >> script.log
}

log "Starting backup process..."
```

Calculator with Options

```
calculate() {
  case "$1" in
    add) echo $(($2 + $3)) ;;
    sub) echo $(($2 - $3)) ;;
    mul) echo $(($2 * $3)) ;;
    *) echo "Invalid operation" ;;
  esac
}

calculate add 10 5  # Output: 15
```



@Sandip Das

Input and Output

Mastering Data Flow in Scripts

Reading User Input

The read command captures input from the user or files.

Basic Syntax:

```
read -p "Prompt: " variable_name
```

Example: Interactive Script

```
#!/bin/bash
read -p "Enter your email: " email
read -sp "Enter password: " password  # -s hides input
echo  # Move to a new line after hidden input
echo "Login details saved for $email."
```

Redirection

Shell scripts can redirect input/output streams using operators:

Output Redirection

Operator	Description	Example
>	Overwrite file	echo "Hello" > log.txt
>>	Append to file	date >> log.txt
2>	Redirect stderr	ls missing.txt 2> error.log
&>	Redirect stdout and stderr	command &> output.log

Input Redirection

```
# Read from a file instead of typing input
grep "error" < server.log
```

Pipes (|)

Shell scripts can redirect input/output streams using operators:

```
cat access.log | grep "404" | wc -l  # Count 404 errors
```

@Sandip Das

Input and Output

Mastering Data Flow in Scripts

Here Documents

A here document (<<) feeds multi-line input to a command.

Basic Syntax:

```
command << DELIMITER
  text
  ...
DELIMITER
```

Example: Generate a Configuration File

```
cat > config.yml << EOF
database:
  host: localhost
  port: 5432
EOF
```

Here Strings

A here string (<<<) passes a single string to a command's input:

```
tr 'a-z' 'A-Z' <<< "hello" # Output: "HELLO"
```

File Descriptors

Bash uses file descriptors (FDs) to manage streams:

- **0: stdin (input)**
- **1: stdout (output)**
- **2: stderr (errors)**

Example: Redirect Errors to a File

```
# Redirect stderr (FD 2) to errors.log
command 2> errors.log
```

Custom File Descriptors:

```
exec 3> custom.log  # Open FD 3 for writing
echo "Log message" >&3
exec 3>&-           # Close FD 3
```

Practical Examples

Logging Script with Timestamps

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

# Redirect all output to logfile
exec &> >(tee -a "$logfile")

echo "Starting process at $(date)..."
# ... rest of script ...
```

Password Confirmation

```
read -sp "Enter password: " pass
echo
read -sp "Confirm password: " confirm
echo

if [ "$pass" != "$confirm" ]; then
  echo "Passwords do not match!" >&2  # Print to stderr
  exit 1
fi
```

@Sandip Das

Working with Files and Directories

Advanced Operations for Automation and Data Management

File Permissions

chmod: Change File Permissions

Permissions control read (r), write (w), and execute (x) access for the owner, group, and others.

Numeric Notation:

Code	Permission
7	rwx (4+2+1)
6	rw- (4+2)
4	r--

Symbolic Notation:

```
chmod u+x script.sh    # Add execute for owner
chmod go-w file.txt    # Remove write for group/others
```

Example:

```
chmod 755 script.sh
# Owner: rwx, Group/Others: r-x
```

chown: Change Ownership

```
chown user:group file.txt  # Change owner and group
sudo chown root /etc/config  # Requires root privileges
```

Special Permissions:

- **setuid (4xxx): Execute as owner.**
- **setgid (2xxx): Execute with group privileges.**
- **sticky bit (1xxx): Restrict file deletion to owner (e.g., /tmp).**

Searching Files

find: Locate Files by Criteria

Basic syntax:

```
find [path] [options] [action]
```

Common Options:

Option	Description
-name	Search by filename
-type	Filter by type (f, d)
-mtime	Modified time (days)
-exec	Run command on matched files

Examples:

```
find /var/log -name "*.log"        # Find logs by name
find ~ -type d -mtime -7           # Directories modified in the last 7 days
find . -size +10M -exec ls -lh {} \;  # Files >10MB, list details
```

@Sandip Das

Working with Files and Directories

Advanced Operations for Automation and Data Management

grep: Search Inside Files

Usage: **grep [options] "pattern" [file...]**
It prints lines that match the regex pattern.
Common options: -i (ignore case), -v (invert match, i.e., print lines not matching),
-r (recursive search in directories), -n (prefix with line numbers), -q (quiet, just exit status), -c (count matches).

Advanced Tools:
zgrep: Search compressed files (e.g., .gz).
rg (ripgrep): Faster alternative with regex support.

```
grep "error" /var/log/syslog     # Find lines containing "error"
grep -r "TODO" ~/projects/       # Recursive search in directories
grep -i "warning" file.txt       # Case-insensitive search
```

Processing Text

sed: Stream Editor

Perform text transformations (e.g., replace, delete).
In scripts, sed is often used to do quick text replacements or extract parts of lines via regex groups.

```
sed 's/old/new/g' input.txt > output.txt
# Replace all "old" with "new"
sed -i.bak 's/foo/bar/' file.txt
# Edit in-place with backup
```

Common Commands:

Command	Description
s/old/new/	Substitute text
/pattern/d	Delete lines matching pattern
5,10d	Delete lines 5-10

awk: Text Processing and Reporting

A scripting language for pattern scanning and data extraction.

Basic Syntax:

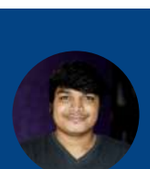
```
awk '/pattern/ { action }' file.txt
```

Examples:

```
# Print the first column of lines containing "GET"
awk '/GET/ {print $1}' access.log

# Sum values in the third column
awk '{sum += $3} END {print sum}' data.csv

# Filter lines where column 2 > 100
awk '$2 > 100 {print $0}' report.txt
```

Practical Example: Log Cleanup Script

```
#!/bin/bash
LOG_DIR="/var/log/app"
ARCHIVE_DIR="/backup/logs"

# Archive logs older than 30 days
find "$LOG_DIR" -name "*.log" -mtime +30 -exec gzip {} \;

# Move compressed logs to backup
find "$LOG_DIR" -name "*.gz" -exec mv {} "$ARCHIVE_DIR" \;

# Update log index
echo "Last cleanup: $(date)" > "$ARCHIVE_DIR/last_cleanup.txt"
```



@Sandip Das

Process Management

Scheduling Tasks, Managing Background Jobs, and Handling Signals

Shell scripting isn't just about running one command after another; you often need to control how processes run — maybe run some tasks in parallel, wait for jobs to finish, handle signals like interrupts, etc

Background Jobs

Running Processes in the Background

Use **&** to run a command in the background:

```
long_running_task &
echo "Task started in background"
```

Here long_running_task (could be a script or command) will run in parallel, and the script will not wait for it (the echo will run immediately). The shell assigns a job ID (in interactive shells) and you can retrieve its process ID.

Wait

After starting a background job, **the special variable $! holds the PID of the last command started in the backgroud**. **You can use this to monitor or wait for that process.**

```
long_running_task &
pid=$!
# do other stuff...
wait $pid
echo "Background task $pid completed"
```

wait $pid will pause the script until the process with that PID finishes. Its exit status will be the exit status of that process (or 127 if no such process existed to wait for). If you have multiple background jobs and call wait with no arguments, it waits for all background jobs to finish.

You can also start multiple background processes:

```
proc1 & pid1=$!
proc2 & pid2=$!
wait $pid1
wait $pid2
echo "Both proc1 and proc2 are done."
```

Managing Jobs

Command	Description
jobs	List background jobs
fg %1	Bring job 1 to the foreground
bg %2	Resume job 2 in the background
kill %3	Terminate job 3

Running tasks in parallel example (use-case):

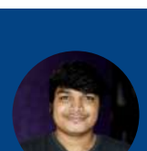
Suppose you have to process 10 files with a CPU-intensive operation. Doing them sequentially might be slow. If you have multiple CPU cores, you can parallelize:

```
for f in file1 file2 file3 file4; do
    process_file "$f" &
done
wait  # wait for all background processes to finish
echo "All files processed."
```

This backgrounds each **process_file** call.

The final **wait** with no arguments waits for all background jobs started by this shell to complete.

This is an easy way to get concurrency. (Be mindful not to oversubscribe - if you spawn 1000 processes at once, that could thrash the system; you might need to limit concurrency using a semaphore or checking number of **jobs** with jobs or monitoring how many & you kick off at once.)



@Sandip Das

Process Management

Scheduling Tasks, Managing Background Jobs, and Handling Signals

Signal Handling via Trap

Signals are a way for the OS to notify processes of events or to ask them to terminate. Common signals:

- **SIGINT (2) – Interrupt from keyboard (Ctrl+C).**
- **SIGTERM (15) – Termination request (the default signal kill sends).**
- **SIGKILL (9) – Force kill (cannot be caught or ignored).**
- **SIGHUP (1) – Hangup (terminal closed, or for daemons to reload config).**
- **SIGUSR1, SIGUSR2 – User-defined signals, for custom purposes.**
- **SIGCHLD – Sent to a parent when a child process exits.**

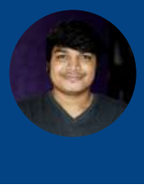
Example: Graceful Shutdown

```bash
#!/bin/bash

# Define a cleanup function that will be triggered on script interruption or termination
cleanup() {
  echo "Stopping services..."
  kill -TERM "$service_pid"  # Send termination signal to background service
  exit 0  # Exit the script gracefully
}

# Register the cleanup function to handle SIGINT (Ctrl+C) and SIGTERM (kill)
trap cleanup SIGINT SIGTERM

# Start a background service (e.g., a server, watcher, etc.)
start_service.sh &  # Run in background
service_pid=$!      # Capture the PID of the background process

# Wait for the background service to complete
wait "$service_pid"
```

Example: Graceful interruption:

```bash
cleanup() {
    echo "Caught Ctrl+C, cleaning up..."
    rm -f "$TMPFILE"
    exit 1  # exit with failure code
}
trap cleanup INT

TMPFILE=$(mktemp) || exit 1
# ... commands that use $TMPFILE ...
```

Here, **if the user hits Ctrl+C (SIGINT)**, the **cleanup function runs**: it echoes a message, deletes the temp file, and exits. Without this trap, hitting Ctrl+C would terminate the script immediately and the temp file might be left behind. With the trap, we handle it gracefully.



Advanced Tools and Tricks

Scheduling with cron

cron is a time-based job scheduler that runs commands at specified intervals.

The crontab File

Edit your user's cron jobs with:

```
crontab -e  # Edit cron jobs
crontab -l  # List cron jobs
```

Cron Syntax:

```
* * * * * command_to_run
| | | | |
| | | | └─ Day of week (0-7, 0=Sunday)
| | | └─── Month (1-12)
| | └───── Day of month (1-31)
| └─────── Hour (0-23)
└───────── Minute (0-59)
```

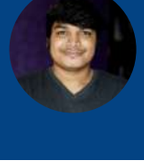
Examples:

Schedule	Description
0 * * * *	Every hour at minute 0
*/15 * * * *	Every 15 minutes
30 3 * * 1	3:30 AM every Monday
@daily	Run once a day (shortcut)

Example: Daily Backup Script

Runs backup.sh daily at 2 AM and logs output.

```
# Add to crontab -e
0 2 * * * /home/user/scripts/backup.sh >> /var/log/backup.log 2>&1
```

Advanced Tools and Tricks

Regular Expressions

Regex is a pattern-matching language for parsing and manipulating text.

Pattern	Matches	Example
.	Any character	a.c → "abc", "a1c"
*	0+ repetitions	a*b → "b", "aaab"
+	1+ repetitions	a+b → "ab", "aaab"
[ ]	Character set	[aeiou] → vowels
^	Start of line	^Hello
$	End of line	world$
\d	Digit (in some tools)	\d{3} → "123"

Usage in Shell Tools:

grep: **grep -E "^error: [A-Z]+" log.txt # Extended regex (-E)**
sed: **sed -E 's/([0-9]{3})-([0-9]{4})/\1****/g' # Mask phone numbers**
awk: **awk '/^[0-9]+$/ {print "Number:", $0}' data.txt # Lines with only digits**

Parameter Expansion

Manipulate variables directly in Bash without external tools

Common Expansions:

Syntax	Description	Example (var="file.txt")
${var%pattern}	Remove shortest suffix match	${var%.*} → "file"
${var%%pattern}	Remove longest suffix match	${var%%.*} → "file" (if var="file.tar.gz")
${var#pattern}	Remove shortest prefix match	${var#*.} → "txt"
${var##pattern}	Remove longest prefix match	${var##*/} → "file.txt" (for paths)
${var/old/new}	Replace first match	${var/file/doc} → "doc.txt"
${var//old/new}	Replace all matches	${var//t/_} → "fi_le.*x*"
${var:-default}	Use default if var is unset/null	${name:-"Guest"} → "Guest"

Example: Batch Rename Files

```
for file in *.jpg; do
  mv "$file" "${file%.jpg}_backup.jpg"
done
```

Advanced Tools and Tricks

Using curl and jq for APIs

Fetch Data with curl

curl -s "https://api.github.com/users/octocat" # -s silences progress

Parse JSON with jq

Extract specific fields:
curl -s "https://api.github.com/users/octocat" | jq '.login, .public_repos'

Filter arrays:
curl -s "https://api.weather.gov/alerts" | jq '.features[] | .properties.headline'

Example: Weather Check Script

```
#!/bin/bash
response=$(curl -s "https://api.weather.gov/gridpoints/TOP/31,80/forecast")
temp=$(echo "$response" | jq '.properties.periods[0].temperature')
echo "Current temperature: $temp°F"
```

Parallel Execution

Speed up tasks by running commands concurrently.

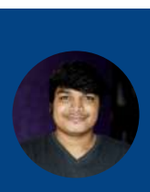
1. xargs for Parallel Jobs
Process files in parallel with -P:

find . -name "*.log" | xargs -P 4 -I {} gzip {} # 4 threads

2. GNU parallel
Advanced parallelization with syntax similar to xargs:
parallel -j 4 convert {} -resize 800x600 {} ::: *.jpg # Resize images in parallel
Example: Process CSV Files Concurrently
ls *.csv | parallel -j 8 '
  process_file.sh {} > {.}.log 2>&1
'

Practical Example: API Data Pipeline

```
#!/bin/bash
# Fetch user data from API, extract emails, process in parallel
API_URL="https://jsonplaceholder.typicode.com/users"

curl -s "$API_URL" | jq -r '.[].email' > emails.txt

# Validate emails with a regex in parallel
cat emails.txt | parallel -j 8 '
  if [[ "{}" =~ ^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$ ]]; then
    echo "{}: Valid"
  else
    echo "{}: Invalid" >&2
  fi
'
```



@Sandip Das

Error Handling

Building Robust and Reliable Scripts

Exit Codes

Every command returns an exit code (0-255) indicating success or failure.

- **0: Success.**
- **Non-zero: Failure (specific codes vary by command).**

Checking Exit Codes with $?

```
ls /nonexistent_dir
echo "Exit code: $?"  # Output: 2 (typically "No such file or directory")
```

Using Exit Codes in Scripts

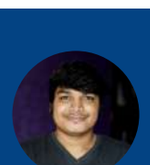
```
if ! mkdir "/invalid/path"; then
  echo "Failed to create directory. Exit code: $?" >&2
  exit 1
fi
```

Trapping Errors with trap

The trap command lets you execute code when a signal (e.g., SIGINT, SIGTERM) or error occurs.

Example: Clean Up on Exit

```
#!/bin/bash
tempfile="tmp.txt"

cleanup() {
  rm -f "$tempfile"
  echo "Cleaned up temporary files."
}

trap cleanup EXIT  # Run cleanup on script exit

touch "$tempfile"
# ... rest of script ...
```

Error Handling

Building Robust and Reliable Scripts

Debugging and Logging

Enable Strict Mode

Add these lines at the start of scripts to enforce stricter error handling:
set -euo pipefail

- **-e: Exit on error.**
- **-u: Treat unset variables as errors.**
- **-o pipefail: Fail pipelines if any command fails.**

Logging with Timestamps

```
log() {
  echo "[$(date +'%Y-%m-%d %H:%M:%S')] ERROR: $1" >> error.log
}

command || log "Command failed: $command"
```

Debug Mode

Run scripts with -x to trace execution:

bash -x script.sh

Or enable debugging within the script:

```
set -x  # Start debugging
# ... code ...
set +x  # Stop debugging
```

@Sandip Das

Real-World Applications

Practical Projects to Automate Workflows and Solve Problems

Project 1: Log Analyzer

Parse server logs to identify errors, count occurrences, and generate reports.

Features:

- Filter logs by date, error level (e.g., ERROR, WARN), or keywords.
- Generate summary statistics (e.g., top errors, frequency).
- Email alerts for critical errors.

```bash
#!/bin/bash
LOG_FILE="/var/log/app/application.log"
REPORT_DIR="/var/reports"
ERROR_THRESHOLD=5

# Extract errors from the last 24 hours
errors=$(grep -i "ERROR" "$LOG_FILE" | grep "$(date -d '24 hours ago' +'%Y-%m-%d')")

# Generate report
error_count=$(echo "$errors" | wc -l)
echo "Total errors: $error_count" > "$REPORT_DIR/error_report.txt"
echo "$errors" | awk '{print $5}' | sort | uniq -c >> "$REPORT_DIR/error_report.txt"

# Send alert if threshold exceeded
if [ "$error_count" -ge "$ERROR_THRESHOLD" ]; then
  mail -s "High Error Rate Detected" admin@example.com < "$REPORT_DIR/error_report.txt"
fi
```

Example Usage:

```
./log_analyzer.sh
```

Real-World Applications

Practical Projects to Automate Workflows and Solve Problems

Project 2: Backup Script

Automate file backups with compression, encryption, and cloud sync.

Features:

- Incremental backups (only new/changed files).
- Retention policy (e.g., keep backups for 30 days).
- Support for local and AWS S3 backups.

```
#!/bin/bash
SOURCE_DIR="/home/user/documents"
BACKUP_DIR="/backup/daily"
DATE=$(date +'%Y-%m-%d')
PASSPHRASE="secret"  # For encryption

# Create daily backup
tar -czvf - "$SOURCE_DIR" | gpg --batch --passphrase "$PASSPHRASE" --symmetric -o
"$BACKUP_DIR/backup_$DATE.tar.gz.gpg"

# Sync to S3 (requires AWS CLI)
aws s3 sync "$BACKUP_DIR" s3://my-bucket/backups/

# Cleanup backups older than 30 days
find "$BACKUP_DIR" -name "*.gpg" -mtime +30 -exec rm {} \;
```

Example Usage:

```
./backup.sh --encrypt --destination s3
```

Real-World Applications

Practical Projects to Automate Workflows and Solve Problems

Project 3: System Health Checker

Monitor server resources and send alerts for anomalies.

Features:

- Check CPU, memory, and disk usage.
- Verify critical services (e.g., Nginx, MySQL).
- Generate HTML reports.

```bash
#!/bin/bash
ALERT_EMAIL="admin@example.com"
THRESHOLD_CPU=90
THRESHOLD_DISK=85

# CPU Check
cpu_usage=$(top -bn1 | grep "Cpu(s)" | awk '{print $2}' | cut -d'%' -f1)
if (( $(echo "$cpu_usage > $THRESHOLD_CPU" | bc -l) )); then
  echo "High CPU usage: ${cpu_usage}%" | mail -s "CPU Alert" "$ALERT_EMAIL"
fi

# Disk Check
disk_usage=$(df -h / | awk 'NR==2 {print $5}' | tr -d '%')
if [ "$disk_usage" -ge "$THRESHOLD_DISK" ]; then
  echo "Disk usage: ${disk_usage}%" | mail -s "Disk Alert" "$ALERT_EMAIL"
fi

# Service Check
if ! systemctl is-active --quiet nginx; then
  echo "Nginx is down!" | mail -s "Service Alert" "$ALERT_EMAIL"
fi
```

Example Usage:

```bash
./health_check.sh --report-html
```

Real-World Applications

Practical Projects to Automate Workflows and Solve Problems

Project 4: User Management Utility

A script to create a batch of user accounts on a system from a CSV input

Features:

- Reading a file (CSV or similar).
- Using useradd command (on Linux) or an abstraction.
- Generating random passwords.
- Using an associative array to track results or any errors.

```bash
#!/usr/bin/env bash
# Usage: add_users.sh users.csv
# CSV format: username,full name,email

set -euo pipefail

INPUT="${1:-users.csv}"
if [[ ! -f "$INPUT" ]]; then
    echo "Input file $INPUT not found." >&2
    exit 1
fi

# Function to generate a random password of length 8
gen_password() {
    tr -dc 'A-Za-z0-9!@#$%^&*' < /dev/urandom | head -c 8
}

declare -A passwds

while IFS=, read -r username fullname email; do
    # Trim whitespace
    username=$(echo "$username" | xargs)
    fullname=$(echo "$fullname" | xargs)
    email=$(echo "$email" | xargs)
    if [[ -z "$username" || -z "$fullname" ]]; then
        continue  # skip if essential fields missing
    fi
    # Check if user already exists
    if id "$username" &>/dev/null; then
        echo "User $username already exists, skipping."
        continue
    fi
    pass=$(gen_password)
    # Create user
    useradd -m -c "$fullname" "$username" || { echo "Failed to create $username"; continue; }
    echo "$username:$pass" | chpasswd   # set password
    passwds["$username"]="$pass"
    # Optionally force password change on first login: chage -d 0 $username
    # Email credentials (assuming mail is set up):
    if [[ -n "$email" ]]; then
        mail -s "Your new account" "$email" <<EOF
Hello $fullname,

Your account $username has been created.
Initial password: $pass
Please change your password on first login.

Regards,
Admin
EOF
    fi
done < "$INPUT"

echo "Created ${#passwds[@]} users:"
for u in "${!passwds[@]}"; do
    echo " $u - password: ${passwds[$u]}"
done
```

@Sandip Das

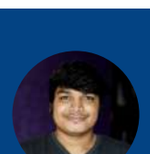
Real-World Applications

Practical Projects to Automate Workflows and Solve Problems

Project 5: Simple To-Do List (Persistent Data with Shell)

Implement a simple to-do list in Bash that allows adding, listing, and removing tasks. Use a file to store tasks. This showcases menu handling, file read/write, and maybe text filtering for removal

```bash
#!/usr/bin/env bash
# Simple Todo List Manager

TODO_FILE="$HOME/.todo_list"

# Ensure the file exists
touch "$TODO_FILE"

print_usage() {
    echo "Usage: $0 [add \"task\" | list | done <task_number>]"
}

case "${1:-}" in
    add)
        task="${2:-}"
        if [[ -z "$task" ]]; then
            echo "Error: no task provided." >&2
            print_usage
            exit 1
        fi
        echo "$task" >> "$TODO_FILE"
        echo "Added task: $task"
        ;;
    list)
        if [[ ! -s "$TODO_FILE" ]]; then
            echo "No tasks yet. Use '$0 add \"task description\"' to add one."
            exit 0
        fi
        echo "To-Do List:"
        nl -w2 -s '. ' "$TODO_FILE"
        ;;
    done)
        num="${2:-}"
        if [[ -z "$num" ]]; then
            echo "Error: no task number provided." >&2
            print_usage
            exit 1
        fi
        if ! [[ "$num" =~ ^[0-9]+$ ]]; then
            echo "Error: task number must be an integer." >&2
            exit 1
        fi
        total=$(wc -l < "$TODO_FILE")
        if (( num < 1 || num > total )); then
            echo "Error: task number $num is out of range (1-$total)." >&2
            exit 1
        fi
        # Delete the nth line in file
        sed -i "${num}d" "$TODO_FILE"
        echo "Marked task #$num as done and removed it."
        ;;
    *)
        print_usage
        ;;
esac
```



@Sandip Das

GET SET GO

Keep Learning, Keep Building! 🐧🚀

Congratulations on completing the **Shell Scripting Handbook** — your ultimate companion to mastering automation, control, and scripting magic with Bash!

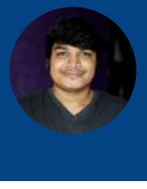
This is just the beginning of your journey into the world of automation — the heartbeat of DevOps, SRE, and Cloud Engineering. Whether you're scheduling backups, parsing logs, or deploying servers, remember: every automation wizard once started with echo "Hello, World!".

💡 What's next?

🔁 Automate repetitive tasks in your daily workflow — cron jobs, backups, and system reports.
📂 Build real-world utilities — log parsers, deployment scripts, environment setup tools.
🧪 Explore advanced concepts — signal handling, debugging with set -x, and writing reusable functions.
📝 Contribute to open-source scripts, create your own GitHub CLI tools, or enhance existing projects.
🤝 Share your scripts, write blogs, teach beginners — the scripting community grows stronger when we collaborate.

If this handbook helped you, I'd love to hear about it! Connect with me on LinkedIn or follow my work at LearnXOps.

Happy scripting — may your scripts run fast, error-free, and always finish with exit 0! 💻🐧
— Sandip Das



@Sandip Das