🧠 Bash Scripting – From Zero to Confident

📘 Your Personalized Bash & Shell Scripting Guide (Beginner to Confident User)

- **Style**: Step-by-step beginner-friendly guide
- **Goal**: Learn Bash scripting for real-world IT/DevOps tasks
- **Delivery**: One section at a time in this chat
- **Extras**: Mini-projects, real-world use cases, clear examples, and later a Cheat Sheet

---

🧭 Roadmap Overview

We'll follow this structure:

1. **Introduction to Bash and Shell Scripting**
2. **How the Shell Works (Bash vs Others)**
3. **Basic Bash Commands Refresher**
4. **Script Structure: What Is a Bash Script?**
5. **Variables and Data Types**
6. **User Input and Arguments**
7. **Conditionals: if, else, elif**
8. **Loops: for, while, until**
9. **Functions in Bash**

10. **Working with Files and Directories**
11. **Error Handling and Exit Codes**
12. **Practical Bash Tools: grep, awk, sed, cut, etc.**
13. **Debugging and Best Practices**
14. **Mini-Projects / Real-World Tasks**
15. **Final Bash Cheat Sheet for Fast Lookup**

🔹 Section 1: Introduction to Bash and Shell Scripting

📌 What Is Bash?

Bash stands for **Bourne Again Shell**. It's the default command-line interpreter (CLI) used in most Linux distributions and macOS. It's a program that lets you **interact with the operating system by typing commands**.

🧠 Why Bash?

- Automate repetitive tasks
- Manage files and users
- Monitor and log system activity
- Schedule tasks (with cron)
- Write scripts for DevOps tools like Docker, Git, Kubernetes, etc.

🧾 What Is a Shell?

A **shell** is a command-line environment that interprets your commands and passes them to the operating system.

There are many shells:

Shell	Description
sh	Original Bourne shell
bash	Bourne Again Shell (improved version of sh) — most popular

zsh	Z Shell — more features than bash, often used by power users
fish	Friendly Interactive Shell — user-friendly with autocompletion

We'll focus on **bash** because it's widely used in Linux, scripting, and DevOps.

🧩 What Is a Bash Script?

A **Bash script** is a **plain text file** containing a series of Bash commands. Think of it like a recipe or list of instructions you want the computer to run one by one.

You can:

- Save commands to a file
- Run them all at once with a single line
- Automate tasks like backups, log cleanups, user creation, etc.

⚙️ Real-World Example: Why Scripting Matters in IT/DevOps

Suppose you need to:

- Backup logs every day at 2 AM
- Restart a service if it crashes
- Scan a server for disk space issues
- Send alerts when CPU usage is high

You could do it manually — **or write a script to do it for you automatically**.

Scripting makes you **faster, more consistent, and more valuable**.

✅ What You'll Be Able to Do

By the end of this guide, you'll know how to:

- Write, run, and debug Bash scripts
- Use logic, loops, and functions
- Automate real-world admin/DevOps tasks
- Build and understand simple to intermediate Bash scripts confidently
- Put "Bash scripting" on your CV with real confidence

🧪 Mini-Activity (Optional for Practice)

Try this in your terminal:

```
echo "Hello, Bash!"
```

This is a basic Bash command. You'll get familiar with using `echo` for printing to the screen.

🔹 Section 2: How the Shell Works (Bash vs Others)

Understanding **how the shell works** helps you think like a script writer, not just a command typer.

🧠 What Is a Shell, Technically?

A **shell** is a command-line interpreter — a program that:

1. **Takes what you type**
2. **Interprets it** (turns it into something the OS understands)
3. **Runs it**, and shows you the result/output

Think of it as a **translator** between *you* and *Linux*.

🛠️ How the Shell Executes Commands

When you type something like:

ls -la /home

Here's what happens:

- The shell (Bash) sees `ls -la /home`
- It checks if `ls` is a **built-in command** or a program in your `$PATH`
- It finds the `ls` binary and runs it with `-la /home` as options
- Then it prints the result in your terminal

🆚 Bash vs Other Shells

Let’s compare the most common shells:

Shell	Full Name	Default In	Notes
sh	Bourne Shell	Very old systems	Basic, minimal features
bash	Bourne Again Shell	Most Linux distros	Default for scripting, widely used
zsh	Z Shell	macOS (recent)	Rich features, plugins (e.g., Oh My Zsh)
ksh	Korn Shell	Some Unix systems	Like sh, with improvements
fish	Friendly Interactive Shell	Custom installs	Very user-friendly, not great for scripting

We stick with **bash** because:

✅ Almost every Linux/Unix server has it
✅ Standard for system scripts
✅ Compatible with most learning materials
✅ Robust for both beginners and pros

🧾 What Is the Shebang?

Every script usually starts with:

#!/bin/bash

This is called a **shebang**. It tells the OS:

> "Use **this interpreter** to run this file."

Without it, Linux may not know which language to use to run your script.

You can change it to:

- `#!/bin/sh` (for more basic POSIX-compliant scripts)
- `#!/usr/bin/env bash` (portable — finds `bash` even if installed in a different path)

✅ **Best Practice: Always use a shebang in your scripts**

🎯 Key Shell Environment Concepts

Concept	Description
`$PATH`	A list of folders where the shell looks for programs to run
`~/.bashrc`	Your personal Bash config file — runs when a shell starts
`~/.profile`	Runs for login shells — can set environment variables
`env`	Shows your environment variables
`export`	Used to create/set environment variables

📂 How Bash Processes a Script (Behind the Scenes)

If you have a file called `myscript.sh`:

#!/bin/bash

echo "Hello from my script!"

Running it with:

bash myscript.sh

…will do this:

- Bash opens the script
- Reads the first line → sees the shebang `#!/bin/bash`
- Reads and executes each line top to bottom
- Exits when it reaches the end (or an `exit` command)

🧪 Try It Yourself

1. Open your terminal.
2. Create a new script:

nano testscript.sh

3. Add this:

#!/bin/bash

echo "This is a test script"

4. Save and exit (`Ctrl + O`, then `Ctrl + X`)
5. Make it executable:

chmod +x testscript.sh

6. Run it:

./testscript.sh

You should see: `This is a test script`

🧩 Summary: Key Points

- The **shell** is your bridge to the OS
- **Bash** is the most popular shell for scripting
- A **shebang** defines the interpreter for your script
- Scripts run **top to bottom**, line by line
- Use `chmod +x` to make a script file executable

➡️ **Next Up: Section 3 – Bash Commands Refresher (Essentials for Script Writers)**
This section will cover the most important and most used Bash commands that you'll use inside scripts.

Want to move on to Section 3 now?

Perfect — let's continue!

🔹 Section 3: Bash Commands Refresher (Essentials for Script Writers)

Before diving deeper into writing full scripts, you need a strong grip on the **core Bash commands**. These are the building blocks that make your scripts functional, practical, and powerful.

🔧 What Are Bash Commands?

Bash commands are instructions you type into a terminal or include in a Bash script to interact with the operating system.

Let's break down the **most essential ones**, grouped by category:

📁 1. File & Directory Commands

Command	What It Does	Example
`ls`	List files in a directory	`ls -l` (long format)
`cd`	Change directory	`cd /etc`
`pwd`	Print working directory	`pwd`
`mkdir`	Make directory	`mkdir logs`

`touch`	Create an empty file	`touch report.txt`
`rm`	Remove files or folders	`rm file.txt`, `rm -r folder/`
`cp`	Copy files or folders	`cp file1.txt file2.txt`
`mv`	Move or rename files	`mv old.txt new.txt`
`find`	Search for files	`find /var -name "*.log"`

📃 2. File Content & Text Manipulation

Command	What It Does	Example
`cat`	View file contents	`cat file.txt`
`less` / `more`	View large files page-by-page	`less file.txt`
`head` / `tail`	View top/bottom lines of a file	`tail -n 20 file.txt`

`echo`	Print to the screen	`echo "Hello"`
`>>` / `>`	Append or overwrite output to file	`echo "Done" >> log.txt`
`grep`	Search for patterns	`grep "ERROR" log.txt`
`cut`	Extract columns of data	`cut -d':' -f1 /etc/passwd`
`awk`	Pattern scanning and processing	`awk '{print $1}' file.txt`
`sed`	Stream editor (find/replace)	`sed 's/old/new/g' file.txt`

💡 3. System Info & Monitoring

Command	What It Does	Example
`top` / `htop`	View running processes	`top`
`df -h`	Disk usage (human-readable)	`df -h`

`du -sh`	Size of directory	`du -sh /var/log`
`free -m`	RAM usage	`free -m`
`uptime`	Show system uptime	`uptime`
`whoami`	Show your user	`whoami`
`uname -a`	System info	`uname -a`

🔁 4. Scripting-Specific Essentials

Command	What It Does	Example
`read`	Get user input	`read name`
`echo`	Output text/variables	`echo $name`
`sleep`	Pause execution	`sleep 5` (pause 5 seconds)
`date`	Show current date/time	`date "+%Y-%m-%d"`

`exit`	End a script	`exit 0`
`clear`	Clear terminal screen	`clear`

🧪 Mini-Practice: Simple Useful Commands

Try these in your terminal:

```
pwd                 # Where am I?
mkdir test_folder   # Create a folder
cd test_folder      # Go into it
touch hello.txt     # Create a file
echo "Hi there" > hello.txt   # Add text
cat hello.txt       # Show contents
```

🧠 Pro Tip: Combine Commands

You can chain commands together like this:

```
mkdir backup && cd backup && touch log.txt
```

This means:

- Make a folder
- If successful, go into it
- Then create a file

You'll learn even more about this when we do conditionals and error checking.

✅ Summary: Commands to Know by Heart

Make sure you're very comfortable with these:

- `ls`, `cd`, `mkdir`, `rm`, `cp`, `mv`, `touch`, `echo`, `cat`
- `grep`, `awk`, `sed`, `cut`
- `df`, `du`, `free`, `uptime`, `whoami`, `date`
- `read`, `sleep`, `exit`, `clear`

These are **used constantly in scripts** for automation and task management.

➡️ **Next Up: Section 4 – What Is a Bash Script (And How to Write Your First One)**
This section will break down a script line-by-line and help you write your first reusable Bash script with comments and explanations.

Ready to move on?

Great — let's dive into it!

🔹 Section 4: What Is a Bash Script (And How to Write Your First One)

Now that you're familiar with key commands, let's put them together into a **Bash script** — your first real step toward automation.

📜 What Is a Bash Script?

A **Bash script** is a plain text file containing a sequence of **commands** that the shell runs **line by line** — just as if you typed them in manually.

✅ Bash scripts are used to:

- Automate tasks
- Chain multiple commands
- Add logic (if-else, loops, etc.)
- Save time & reduce human error

🧱 Bash Script Anatomy

Here's the basic structure of a script:

```
#!/bin/bash            # Shebang - tells the OS to use Bash

# This is a comment    # Lines starting with # are ignored


echo "Hello, world!"   # Command 1

date                   # Command 2
```

Each line:

- Can run a command
- Can include variables
- Can include logic (if, loops, etc.)
- Can include comments for clarity

📂 Step-by-Step: Writing Your First Script

1. **Create a new file**:

nano myscript.sh

2. **Add this content**:

#!/bin/bash

My first script

echo "Hello, $USER!"

echo "Today is: $(date)"

echo "You are in directory: $(pwd)"

`$(command)` runs the command and substitutes its output.

3. **Save and exit**:

- `Ctrl + O` (save)
- `Enter` (confirm)
- `Ctrl + X` (exit)

4. **Make the script executable**:

chmod +x myscript.sh

5. **Run the script**:

./myscript.sh

🧠 What Just Happened?

Line	Explanation
`#!/bin/bash`	Shebang: uses Bash to run this script
`echo "Hello, $USER!"`	Prints your username using a built-in variable
`$(date)`	Runs the `date` command and includes the result
`$(pwd)`	Shows your current folder

🔡 Variables in Bash

You can declare variables like this:

name="Alice"

echo "Hello, $name"

🚫 **No spaces** around `=`
✅ Use `$name` to access the variable

🧪 Mini Task – Try This Script

Create another script called `userinfo.sh`:

```
#!/bin/bash

echo "Welcome, $USER!"
echo "Your home directory is: $HOME"
echo "The system uptime is: $(uptime -p)"
```

Run it with:

```
chmod +x userinfo.sh
./userinfo.sh
```

✅ Summary: What You Just Learned

- A **Bash script** is a file full of Bash commands
- It starts with `#!/bin/bash` (shebang)
- You run it using `./scriptname.sh`
- You can include:
    - Commands (`echo`, `date`, `pwd`)
    - Variables (`$USER`, `$HOME`)
    - Substitutions (`$(command)`)

This is the **foundation of scripting** — everything from here builds on this format.

➡️ **Next Up: Section 5 – Variables, Input, and Output in Bash Scripts**
We'll explore how to:

- Declare and use variables
- Accept **user input** with `read`
- Output and redirect text to files

Ready to continue?

Awesome — let's move on!

🔹 Section 5: Variables, Input, and Output in Bash Scripts

This is where Bash scripts become **interactive** and **dynamic**. You'll learn how to:

- Create and use **variables**
- Accept **user input**
- Use **input/output redirection** (`>`, `>>`, `<`)
- Write scripts that react based on what the user enters

🧠 Part 1: Variables in Bash

🔹 Defining Variables

name="Alice"

age=25

✅ **No spaces** around `=`
❗ Wrong: `name = "Alice"` ← this won't work.

🔹 **Using Variables**

```
echo "Name: $name"

echo "Age: $age"
```

💡 You can also do:

```
echo "User $name is $age years old."
```

🔄 Part 2: Command Substitution

You can store the result of a command in a variable:

```
today=$(date)

echo "Today is: $today"
```

This runs the `date` command and saves its output in `today`.

🎯 Part 3: User Input (`read`)

✅ Prompting the User

```
echo "What is your name?"

read name

echo "Welcome, $name!"
```

💡 You can add the prompt in the same line:

```
read -p "Enter your favorite color: " color

echo "Your color is $color"
```

📂 Part 4: Redirecting Output

🔹 Output to File

echo "This is a test" > output.txt

- `>`: Overwrites the file
- `>>`: Appends to the file

echo "Another line" >> output.txt

🔹 Input from File

cat < input.txt

🧪 Example Script: `greet.sh`

#!/bin/bash

read -p "Enter your name: " name

read -p "Enter your age: " age

echo "Hello, $name. You are $age years old."

echo "User $name ($age) logged in at $(date)" >> log.txt

Run it:

chmod +x greet.sh

./greet.sh

💡 Now check the contents of `log.txt` — every run adds a new line.

🧰 Bonus: Predefined Environment Variables

Variable	Description
`$USER`	Current user
`$HOME`	Home directory
`$PWD`	Current directory
`$HOSTNAME`	Name of the machine
`$?`	Exit status of last command

Example:

echo "You are logged in as $USER on $HOSTNAME"

✅ Summary

Feature	Usage	Example
Define variable	`name="Alice"`	
Use variable	`$name`	`echo "Hello, $name"`
User input	`read name`	`read -p "Enter name: " name`
Redirect output	`>` or `>>`	`echo "Log" >> file.txt`
Command in variable	`now=$(date)`	`echo "It's $now"`

🔜 Next Up: Section 6 – Conditional Logic (`if`, `else`, `elif`) in Bash Scripts

This is where you add intelligence to your scripts — making them **react based on conditions**.

Ready to continue?

Perfect — now it's time to make your scripts **think**!

🔹 Section 6: Conditional Logic in Bash Scripts (`if`, `else`, `elif`)

Conditional logic lets your script **make decisions** based on user input, variable values, or the output of commands.

🧠 Why Use Conditionals?

With conditionals, your script can:

- Respond differently based on input
- Check if a file exists
- Verify if a command was successful
- Control the flow of logic like a real program

✅ Basic `if` Statement

```
if [ condition ]; then
  # code if condition is true
fi
```

Example:

```
if [ "$USER" == "admin" ]; then
  echo "Welcome, administrator."
fi
```

🔀 `if` + `else`

```
if [ "$age" -ge 18 ]; then
  echo "You are an adult."
else
  echo "You are a minor."
fi
```

🔁 `if` + `elif` + `else`

```
if [ "$score" -ge 90 ]; then
  echo "Grade: A"
elif [ "$score" -ge 75 ]; then
  echo "Grade: B"
else
  echo "Grade: C or below"
fi
```

🧱 Comparison Operators (Used Inside `[ ]`)

Operator	Meaning
`-eq`	Equal to

`-ne`	Not equal
`-gt`	Greater than
`-lt`	Less than
`-ge`	Greater or equal
`-le`	Less or equal
`==`	String equality
`!=`	String not equal
`-z`	String is empty
`-n`	String is not empty

📂 File Test Operators

Test	What It Checks
`-e file`	File exists

`-f file`	File exists and is a regular file
`-d dir`	Directory exists
`-s file`	File exists and is not empty
`-r file`	File is readable
`-w file`	File is writable
`-x file`	File is executable

🧪 Practice Script: `check_age.sh`

```
#!/bin/bash

read -p "Enter your age: " age

if [ "$age" -ge 18 ]; then
  echo "You are allowed to vote."
```

```
else
  echo "You are not allowed to vote."
fi
```

Try it out:

```
chmod +x check_age.sh
./check_age.sh
```

🧪 Another Example: `file_check.sh`

```
#!/bin/bash

read -p "Enter file name: " file

if [ -e "$file" ]; then
  echo "File exists."
else
  echo "File does not exist."
fi
```

⚠️ Syntax Notes

- Always put spaces inside `[ ]` (e.g., `[ "$age" -ge 18 ]`)

- Use `"$variable"` in quotes to avoid errors if variable is empty
- Use `==` or `!=` for **string comparisons**
- Use `-ge`, `-eq`, etc., for **numeric comparisons**

✅ Summary: Conditional Building Blocks

Task	Code Snippet
String comparison	`[ "$name" == "Alice" ]`
Number check	`[ "$age" -ge 18 ]`
File check	`[ -e file.txt ]`
If/else	`if …; then … else … fi`
Multiple branches	`if … elif … else … fi`

➡️ **Next Up: Section 7 – Loops in Bash: `for`, `while`, and Practical Uses**
You'll learn how to **repeat tasks**, loop through files, and write real-world Bash automations using loops.

Ready to continue?

Awesome — let's level up your scripting skills with **loops**!

🔹 Section 7: Loops in Bash – `for`, `while`, and Real-World Uses

Loops let your script **repeat tasks**, making it easy to:

- Process multiple files
- Repeat commands
- Automate monitoring
- Iterate over lists or input

🔁 1. `for` Loops

🔹 Loop Through a List

```
for name in Alice Bob Charlie
do
  echo "Hello, $name"
done
```

🔹 Loop Through Files in a Directory

```
for file in *.txt
do
  echo "Processing $file"
```

done

🧠 This loops over all `.txt` files in the current folder.

🔁 2. `while` Loops

🔹 **Basic Syntax**

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

🔹 **Example: Countdown Timer**

```
count=5

while [ $count -gt 0 ]
do
  echo "Countdown: $count"
  ((count--))  # Decrement
  sleep 1
done

echo "Liftoff!"
```

🔁 3. Infinite Loops

```
while true
do
  echo "Running..."
  sleep 2
done
```

🔔 You must `Ctrl + C` to stop it.

🔄 4. Loop with `read`

You can loop through lines in a file:

```
while read line
do
  echo "Line: $line"
done < myfile.txt
```

🧪 Example Script: `bulk_greet.sh`

```
#!/bin/bash

for name in Alice Bob Charlie
do
  echo "Good day, $name!"
done
```

🧪 Example: `file_report.sh`

```
#!/bin/bash

for file in *.log
do
  echo "$file has $(wc -l < "$file") lines"
done
```

💡 `wc -l < "$file"` counts the number of lines in each `.log` file.

✅ Summary

Loop	Purpose	Example
`for`	Loop over list/items	`for f in *.txt`
`while`	Loop until condition is false	`while [ $x -gt 0 ]`
`read`	Read each line of a file	`while read line; do ... done < file`
`true`	Infinite loop	`while true; do ... done`

➡️ **Next Up: Section 8 – Functions in Bash Scripts (Modular Scripting)**
You'll learn how to create **functions**, call them from your scripts, and reuse code for cleaner, more organized Bash programs.

Ready to continue?

Great — let's move into **functions**, one of the most powerful features of Bash scripting!

🔹 Section 8: Functions in Bash Scripts (Modular Scripting)

Functions allow you to **group code into reusable blocks**. This makes your scripts easier to read, manage, and maintain — just like in programming languages like Python or JavaScript.

🧠 Why Use Functions?

- Avoid repeating code
- Organize scripts logically
- Make scripts easier to update or debug

✅ Defining a Function

```
function greet() {
  echo "Hello, welcome!"
}
```

or more commonly:

```
greet() {
  echo "Hello, welcome!"
}
```

▶️ Calling a Function

Just write the function name:

```
greet
```

🗒️ Example: Simple Function

```
greet() {
  echo "Hi $1! Good to see you."
}

greet "Alice"
greet "Bob"
```

💡 $1 is the **first argument** passed to the function.

🔢 Function with Multiple Parameters

```
add_numbers() {
```

```
  sum=$(( $1 + $2 ))
  echo "Sum: $sum"
}

add_numbers 5 10
```

🔁 Returning Values (via `echo`)

Bash doesn't return values like other languages. Instead, it **echoes** output which can be captured:

```
get_date() {
  echo $(date)
}

today=$(get_date)
echo "Today is: $today"
```

⛔ Return Code (Exit Status)

You can still return an **exit status** (0 = success, 1+ = error):

```
check_even() {
  if (( $1 % 2 == 0 )); then
    return 0
  else
```

```
    return 1
  fi
}

check_even 4
if [ $? -eq 0 ]; then
  echo "Even"
else
  echo "Odd"
fi
```

🧪 Example Script: `user_check.sh`

```
#!/bin/bash

check_user() {
  if [ "$1" == "$USER" ]; then
    echo "That's you!"
  else
    echo "That's not your username."
  fi
}

read -p "Enter a username to check: " input
```

```
check_user "$input"
```

✅ Summary

Concept	Syntax	Example
Define function	`name() { commands; }`	`greet() { echo "Hello"; }`
Call function	`name`	`greet`
Use parameter	`$1`, `$2`, etc.	`greet "Alice"`
Return output	`echo result`	`today=$(get_date)`
Return code	`return 0`	`if [ $? -eq 0 ]; then …`

➡️ **Next Up: Section 9 – Arrays in Bash: Handling Lists and Collections**
You'll learn how to use arrays in Bash, loop through them, and work with lists of values (files, options, etc.).

Ready to continue?

Excellent — now let's explore **arrays** in Bash scripting!

🔹 Section 9: Arrays in Bash – Handling Lists and Collections

Arrays let you store **multiple values** (like a list of names, files, or numbers) in a **single variable** — perfect for automation, loops, and processing tasks.

✅ 1. Declaring an Array

```
fruits=("apple" "banana" "cherry")
```

You can also declare it element by element:

```
fruits[0]="apple"
fruits[1]="banana"
```

📥 2. Accessing Array Elements

```
echo ${fruits[0]}  # apple
echo ${fruits[1]}  # banana
```

🔢 3. Looping Through an Array

```
for fruit in "${fruits[@]}"
do
  echo "I like $fruit"
```

done

💡 Use `[@]` to get **all elements**.

🔢 4. Array Length

echo "Number of fruits: ${#fruits[@]}"

🗂️ 5. Indexes and Slices

echo ${!fruits[@]} # prints all indexes (0 1 2)

echo ${fruits[@]:1:2} # slice: 2 elements starting from index 1 (banana cherry)

🧪 Practical Example: `process_files.sh`

#!/bin/bash

log_files=("sys.log" "app.log" "error.log")

for file in "${log_files[@]}"

do

  echo "Checking $file..."

  if [-e "$file"]; then

    echo "$file exists."

```
  else
    echo "$file is missing!"
  fi
done
```

➕ Adding and Updating Items

```
fruits+=("date")  # add to array
fruits[2]="grape" # update element
```

❌ Deleting an Item

```
unset fruits[1]  # removes "banana"
```

✅ Summary

Task	Code
Declare	`arr=("a" "b" "c")`
Access item	`${arr[0]}`

All items	`${arr[@]}`
All indexes	`${!arr[@]}`
Length	`${#arr[@]}`
Loop	`for i in "${arr[@]}"`
Add item	`arr+=("new")`
Delete	`unset arr[2]`

➡️ **Next Up: Section 10 – Script Input & Arguments: Using `$1`, `$2`, `$@`, and `getopts`**
You'll learn how to make your scripts **accept arguments** (like command-line tools), handle **flags/options**, and build more flexible utilities.

Ready to move on?

Great — now let's dive into how to **pass input to your scripts** and handle arguments just like professional command-line tools do.

🔹 Section 10: Script Input & Arguments – $1, $2, $@, and `getopts`

Scripts become powerful when they can accept **dynamic input** — users can pass filenames, usernames, modes, flags, etc.

✅ 1. Positional Parameters: $1, $2, $3, etc.

These are the values passed to your script **from the command line**.

🔸 **Example Script: `greet.sh`**

#!/bin/bash

echo "Hello, $1!"

echo "You are $2 years old."

Run it like:

bash greet.sh Alice 30

📌 Output:

Hello, Alice!

You are 30 years old.

🔁 2. Loop Over All Arguments: $@ or $*

#!/bin/bash

```
echo "You passed the following arguments:"

for arg in "$@"

do

  echo "$arg"

done
```

`"$@"` treats arguments as separate strings.
`"$*"` treats them as one long string.

🔢 3. Count Arguments: `$#`

```
echo "Total args: $#"
```

🛑 4. Check If Arguments Were Provided

```
if [ -z "$1" ]; then

  echo "Usage: $0 <name>"

  exit 1

fi
```

🧪 Real-World Example: `backup.sh`

```
#!/bin/bash


src="$1"
```

```
dest="$2"

if [ -z "$src" ] || [ -z "$dest" ]; then
  echo "Usage: $0 <source> <destination>"
  exit 1
fi

cp -r "$src" "$dest"
echo "Backup completed."
```

Run like:

```
bash backup.sh /home/user/docs /mnt/backup
```

⚙️ 5. Handling Options with `getopts` (like `-h`, `-v`)

This allows **flags/options** (like `-u user`, `-p password`), similar to Linux commands.

🔸 **Example Script with `getopts`**

```
#!/bin/bash

while getopts ":u:p:" opt; do
  case $opt in
    u) user="$OPTARG" ;;
    p) pass="$OPTARG" ;;
    \?) echo "Invalid option -$OPTARG" ;;
```

```
  esac
done

echo "Username: $user"
echo "Password: $pass"

Run like:
bash login.sh -u alice -p secret123
```

✅ Summary

Variable	Purpose
`$0`	Script name
`$1`, `$2`, ...	1st, 2nd, ... argument
`$@`	All arguments (individually quoted)
`$*`	All arguments as one string
`$#`	Number of arguments

`getopts`	Parse flags/options

✅ When to Use This?

- Scripts that need input (e.g. filenames, usernames)
- Automations with custom parameters
- Making your Bash scripts feel like CLI tools

➡️ **Next Up: Section 11 – Real-World Bash Script Projects You Can Put on Your CV**
You'll get script project ideas with real use cases to **build, learn, and showcase** on your resume.

Would you like to continue?

🔹 Section 11: Real-World Bash Script Projects You Can Put on Your CV

These scripts **demonstrate practical, job-ready skills** and cover automation, system management, backups, and logging — things employers love to see.

🛠️ 1. Backup Automation Script

Purpose: Back up a folder to another location with a timestamp.

```
#!/bin/bash

src="/home/$USER/Documents"
dest="/mnt/backup"
date=$(date +%Y-%m-%d_%H-%M-%S)
backup_dir="$dest/backup_$date"

mkdir -p "$backup_dir"
cp -r "$src" "$backup_dir"

echo "Backup completed to $backup_dir"
```

Skills shown:

- Variables
- Date handling
- Directory creation
- File copy

🧹 2. Log Cleaner Script

Purpose: Delete log files older than 7 days.

```
#!/bin/bash

logdir="/var/log/myapp"
```

find "$logdir" -type f -name "*.log" -mtime +7 -exec rm {} \;

echo "Old logs cleaned from $logdir"

Skills shown:

- File management
- Automation
- Scheduling with `cron` (optional)

👥 3. User Management Tool

Purpose: Add a new user and set password interactively.

#!/bin/bash

read -p "Enter username: " username

read -sp "Enter password: " password

echo

sudo useradd "$username"

echo "$username:$password" | sudo chpasswd

echo "User $username added and password set."

Skills shown:

- User input
- User creation
- Secure scripting

📜 4. System Info Report Script

Purpose: Generate a summary report of system stats.

```
#!/bin/bash

echo "Hostname: $(hostname)"
echo "Uptime: $(uptime -p)"
echo "Memory usage:"
free -h

echo "Disk usage:"
df -h

echo "Top 5 running processes:"
ps aux --sort=-%mem | head -n 6
```

Skills shown:

- Command chaining
- System monitoring

- Script reporting

🔐 5. Simple Password Generator

```
#!/bin/bash

length=12
password=$(tr -dc A-Za-z0-9 </dev/urandom | head -c $length)
echo "Generated password: $password"
```

Skills shown:

- Random generation
- Practical command use
- Security scripts

💬 6. Interactive Menu Script

```
#!/bin/bash

while true; do
  echo "1. Show date"
  echo "2. List files"
  echo "3. Exit"
  read -p "Choose: " option
```

```
  case $option in
    1) date ;;
    2) ls -l ;;
    3) echo "Bye!"; break ;;
    *) echo "Invalid option." ;;
  esac
done
```

Skills shown:

- Control flow
- User interaction
- Menu creation

🧠 What These Projects Show Employers:

✅ You can write structured, maintainable scripts
✅ You know core Bash features
✅ You understand system administration tasks
✅ You can automate real-world IT operations
✅ You're comfortable with Linux environments

🔹 Section 12: Practice Plan + Roadmap to Become Bash-Scripting Confident

---

🧠 Why Practice Matters

You don't need to memorize everything.
You just need to:

- Know **what's possible** with Bash
- Recognize **common use cases**
- Be comfortable **looking up and adapting solutions**

This section gives you a **daily habit**, a **weekly project idea**, and a **timeline** to get good enough to list **Bash Scripting** confidently on your CV.

✅ Weekly Learning Roadmap (4 Weeks)

🗓️ Week 1 – Core Commands + Scripting Basics

Goals:

- Master shell commands (e.g. `ls`, `cat`, `grep`, `awk`, `cut`, `sort`, `find`)
- Write scripts with variables, if-else, and loops
- Understand file permissions and run scripts with `chmod`

Daily Practice:

- 1 small script a day: calculator, greeting, file checker

🗓️ Week 2 – Arguments, Functions, Menus

Goals:

- Use `$1`, `$@`, `getopts` to handle input
- Write reusable functions inside your scripts
- Build an interactive menu-based CLI tool

Daily Practice:

- Make tools: disk checker, todo list, uptime reporter

🗓️ Week 3 – Arrays, Logging, Input Validation

Goals:

- Process lists (arrays)
- Write log-aware scripts
- Add error-checking to user input

Projects:

- User manager script
- Random password generator with log entries

🗓️ Week 4 – Real Projects + Polish

Goals:

- Build and refine 2–3 scripts for your portfolio
- Add usage instructions (`--help`)
- Focus on clean, reusable, commented code

Projects:

- System backup tool
- Log cleaner with dry-run mode
- Health-checker for servers or apps

🎯 What to List on Your CV

Skill: `Bash scripting & Linux automation`
Optional line (under Projects):

> Developed and maintained Bash scripts for system automation, backup, log rotation, and user management.

🔁 How to Practice Daily (15–30 minutes)

Time	Task
5 min	Read a script or command breakdown
10 min	Modify or run a script yourself

10–15 min	Create your own version or combine with other tools (like `cron`, `awk`, `grep`)

📁 Where to Store Your Scripts

- Use **GitHub** to store and version your scripts
- Organize by folder: `backup-tools/`, `user-tools/`, `fun-scripts/`
- Add a README for each folder explaining what each script does

---

🧰 Suggested Tools You Can Use Together With Bash

Tool	Purpose
`cron`	Run scripts on a schedule
`awk`, `sed`	Text processing inside scripts
`systemctl`, `service`	Manage system services
`ssh`	Automate remote tasks
`scp`, `rsync`	Automate file transfer & backups

`dialog` / `whiptail`	Add GUI-style menus in terminal

✅ That completes the core sections of your **Bash scripting foundation**.

You're now equipped with:

- Bash essentials ✅
- Scripting fundamentals ✅
- Real-world script projects ✅
- Daily learning plan ✅
- CV-ready script experience ✅

🧠 Bash Scripting Cheat Sheet – From Zero to Confident

🔹 1. Bash Basics

Concept	Syntax/Example
Shebang (script header)	`#!/bin/bash`
Run script	`bash script.sh` or `./script.sh` (after `chmod +x`)
Print to screen	`echo "Hello"`
Comments	`# This is a comment`
Variables	`name="Alice"`
Use variable	`echo "$name"`
Read input	`read name`
Arithmetic	`((sum=5+3))` or `sum=$((5+3))`

🔹 2. Conditions

Type	Example
If-else	`if [ $x -gt 10 ]; then ... fi`
File exists	`if [ -f "file.txt" ]; then ...`
String match	`if [ "$x" == "yes" ]; then ...`

Test operators	`-eq -ne -gt -lt -ge -le`

🔹 3. Loops

Loop Type	Example
`for` loop	`for i in {1..5}; do echo $i; done`
`while` loop	`while [ $x -lt 10 ]; do ...; done`
`until` loop	`until [ $x -ge 5 ]; do ...; done`

🔹 4. Functions

```
greet() {
  echo "Hello, $1"
}
greet Alice
```

- `$1`, `$2` are function arguments

🔹 5. Script Input (Arguments)

Symbol	Meaning
`$0`	Script name
`$1`, `$2`	First, second argument
`$@`	All arguments

`$#`	Number of arguments

With getopts:

```
while getopts ":u:p:" opt; do
  case $opt in
    u) user="$OPTARG" ;;
    p) pass="$OPTARG" ;;
  esac
done
```

🔹 6. File Operations

Command	Purpose
`touch file.txt`	Create file
`mkdir dir/`	Create directory
`cp`, `mv`, `rm`	Copy, move, delete
`cat`, `less`, `head`, `tail`	View contents
`echo "text" > file`	Overwrite file
`echo "text" >> file`	Append to file

🔹 7. Text Tools

Tool	Example
`grep`	`grep "error" logfile.txt`

```
cut   cut -d: -f1
      /etc/passwd

awk   awk '{print $1}'
      file.txt

sed   sed 's/old/new/'
      file.txt

tr    tr 'a-z' 'A-Z'
```

🔹 8. Arrays

```
fruits=("apple" "banana" "orange")
echo ${fruits[1]}     # banana
echo ${#fruits[@]}    # number of items
```

🔹 9. Logging & Debugging

Technique	Command
Logging	`echo "$(date) Something happened" >> log.txt`
Debug mode	`bash -x script.sh`
Exit codes	`exit 0`, `$?` for last command's status

🔹 10. Scheduling (Bonus)

Tool	Purpose
`cron`	Schedule scripts
`crontab -e`	Edit cron jobs

`@reboot`, `* * * * *`	Run at boot or every minute/hour/day

🧪 Bonus Projects Recap

Script	What it Does
`backup.sh`	Copies a folder with a timestamp
`log-cleaner.sh`	Deletes old logs
`user-add.sh`	Creates user with password
`sysinfo.sh`	Reports uptime, memory, disk
`gen-pass.sh`	Makes a secure random password
`menu.sh`	CLI app with options

🧭 What You Can Now Confidently Say on Your CV

✅ **Skills Section**:
`Linux Bash Scripting, CLI Automation, System Scripting`

✅ **Projects Section**:

> Developed and maintained Bash scripts for backup automation, user management, log rotation, and system reporting.
