

Bash Scripting: From Noob to Go...

By Ahamed - Learn Code with Noob

Bash Scripting: From Noob to Go...

© 2025 Ahamed - Learn Code with Noob

All rights reserved. No part of this book may be copied, reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without prior written permission from the author, except for brief excerpts for review or educational purposes.

The author has made every effort to ensure the accuracy of the information in this book, but assumes no responsibility for errors or omissions. The content is provided "as is" without warranty of any kind. The author is not responsible for any damages or issues arising from the use of this book.

For permissions, inquiries, or feedback, please contact:

 ahamedrashid.me@gmail.com

 <https://facebook.com/ahamedrashid.my>

1 Introduction to Bash

1.1 What is Bash?

- History of Bash
- Difference Between Bash and Other Shells (zsh, fish, sh)
- Role of Bash in Linux Systems

1.2 Why Learn Bash?

- Importance in System Administration
- Automation with Bash
- Bash vs Python for Scripting

1.3 Setting Up Your Bash Environment

- Checking Bash Version (`bash --version`)
- Installing Bash on Different Systems
- Choosing a Good Text Editor (Nano, Vim, VS Code)

1.4 Running Your First Bash Command

- Opening Terminal
- Basic Commands (`ls`, `pwd`, `whoami`)
- Executing Commands in Bash

1.5 Understanding the Shell

- Interactive vs Non-Interactive Shell
 - How the Shell Interprets Commands
 - Understanding `$PATH`
-

2 Basics of Bash Commands

2.1 Navigating the File System

- `ls`, `cd`, `pwd` Commands
- Using `cd ..` and `cd -`

2.2 File Operations

- Creating Files (`touch`)
- Moving & Renaming (`mv`)
- Copying Files (`cp`)
- Removing Files (`rm`)

2.3 Viewing File Contents

- `cat`, `less`, `head`, `tail`
- Combining Commands (`cat file | less`)

2.4 Understanding File Permissions

- Checking Permissions (`ls -l`)
- Changing Permissions (`chmod`)
- Changing Ownership (`chown`)

2.5 Process Management

- Viewing Running Processes (`ps`, `top`, `htop`)
 - Killing Processes (`kill`, `killall`, `pkill`)
-

3 Introduction to Bash Scripting

3.1 What is a Bash Script?

- Purpose of Bash Scripting
- Difference Between Commands and Scripts

3.2 Creating and Running a Script

- Writing Your First Script (`nano script.sh`)
- Executing (`bash script.sh` vs `./script.sh`)

3.3 Writing and Executing Basic Scripts

- Creating a Hello World Script
- Understanding Line Breaks and Spacing

3.4 Using Shebang (`#!/bin/bash`)

- What is a Shebang?
- Why is it Needed?

3.5 Understanding Comments

- Single-line Comments (`# This is a comment`)
 - Multi-line Comments
-

4 Variables and Data Types

4.1 Declaring Variables

- Syntax (`VAR=value`)

- Rules for Naming Variables and little more...

4.2 Using and Accessing Variables

- Printing Variables (`echo $VAR`)
- Using Variables in Strings

4.3 Read-Only Variables

- Declaring Read-Only Variables (`readonly VAR`)

4.4 Local and Global Variables

- Local Variables in Functions
- Exporting Variables (`export VAR=value`)

4.5 Environment Variables

- Common Environment Variables (`$HOME`, `$PATH`)
 - Modifying `.bashrc`
-

5 Input and Output in Bash

5.1 Reading User Input

- Using `read` Command
- Using `-p` for Prompts

5.2 Printing Output

- Using `echo` and `printf`
- Formatting Output with `printf`

5.3 Redirection

- Redirecting Output (`>`, `>>`)
- Redirecting Input (`<`)

5.4 Pipes

- Combining Commands (`ls | grep txt`)
-

6 Conditional Statements

6.1 If-Else Statements

- Basic Syntax
- Using `elif` for Multiple Conditions

6.2 Case Statements

- Using `case` Instead of `if`
- Example: Menu Selection

6.3 Comparison Operators

- Integer Comparisons (`-eq`, `-lt`, `-gt`)
- String Comparisons (`==`, `!=`)

6.4 Logical Operators

- `&&` (AND) and `||` (OR)
-

7 Loops in Bash

7.1 For Loop

- Iterating Over Lists
- Using `{1..10}` Syntax

7.2 While Loop

- Using `while` with Conditions
- Example: Countdown Timer

7.3 Until Loop

- Running Until a Condition is Met

7.4 Loop Control

- Using `break` to Exit Loops
 - Using `continue` to Skip Iteration
-

8 Functions in Bash

8.1 Defining and Calling Functions

- Function Syntax
- Calling Functions Inside Scripts

8.2 Function Arguments

- Passing Arguments (`$1`, `$2`)
- Using `shift` to Handle Multiple Arguments

8.3 Returning Values

- Returning Values with `return` and `echo`

8.4 Local vs Global Variables in Functions

9 Arrays in Bash

9.1 Declaring and Using Arrays

9.2 Accessing Elements

9.3 Adding and Removing Elements

9.4 Iterating Over Arrays

10 Advanced Bash Scripting

10.1 Command-Line Arguments (`$@`, `$#`)

10.2 Working with Date and Time

10.3 String Manipulation

10.4 File Handling

11 Process Management and Job Control

11.1 Background & Foreground Processes (`&`, `fg`, `bg`)

11.2 Process Signals (`kill`, `trap`)

11.3 Monitoring Processes (`ps`, `top`, `htop`)

1 2 Debugging Bash Scripts

12.1 Enabling Debug Mode (`set -x`)

12.2 Error Handling (`| |`, `&&`, `exit`)

12.3 Logging (`tee`, `logger`)

1 3 Working with Files and Directories

13.1 Finding Files (`find`, `locate`)

13.2 Counting Words and Lines (`wc`, `grep`)

13.3 Sorting and Filtering (`sort`, `uniq`)

13.4 Archiving and Compressing (`tar`, `zip`)

1 4 Networking with Bash

14.1 Fetching Data from the Internet (`curl`, `wget`)

14.2 Checking Network Status (`ping`, `netstat`)

14.3 Transferring Files (`scp`, `rsync`)

14.4 Automating SSH Connections (`ssh`)

1 5 Automation and Cron Jobs

15.1 Scheduling Tasks with `cron`

15.2 Using `crontab` for Automation

15.3 Running Scripts Periodically

16 Practical Bash Projects

16.1 Creating a Simple To-Do List Script

16.2 Automating Backups

16.3 Writing a Log Monitoring Script

16.4 Developing a System Health Checker

16.5 Maybe some fun game???

Bash Scripting: From Noob to Pro 😊🚀

By Ahamed - Learn Code with Noob

1 Introduction to Bash 🚀

1.1 What is Bash?

History of Bash

Bash (Bourne Again Shell) is a Unix shell and command language first released in **1989** by **Brian Fox** for the **GNU Project** as a free replacement for the Bourne shell (**sh**). Over time, Bash became the **default shell** for most Linux distributions and macOS (before being replaced by zsh in macOS Catalina).

Key milestones:

- **1977**: Bourne Shell (**sh**) introduced by Stephen Bourne.
- **1989**: Bash released by Brian Fox.
- **1990s - 2000s**: Bash becomes the standard shell in Linux systems.
- **2019**: macOS replaces Bash with zsh as the default shell.

Difference Between Bash and Other Shells

There are several shells available in Unix-like systems. Here's how Bash compares:

Feature	Bash	Zsh	Fish	Sh
Default in Linux	✓	✗	✗	✗
Auto-suggestions	✗	✓	✓	✗
Configuration File	~/.bashrc	~/.zshrc	~/.config/fish/config.fish	~/.profile
Scripting Capable	✓	✓	Limited	✓

Role of Bash in Linux Systems

- Acts as a **command interpreter** between users and the OS.
 - Helps in **automating** tasks using scripts.
 - Provides **programming features** like loops, conditionals, and functions.
-

1.2 Why Learn Bash?

Importance in System Administration

- Used for **managing servers** and **automating administrative tasks**.
- Helps in **handling user accounts, backups, and log files**.

Automation with Bash

- **File management** (automate copying, renaming, and deleting files).
- **Task scheduling** using `cron` jobs.
- **Monitoring system logs** and performance.

Bash vs Python for Scripting

Feature	Bash	Python
Best for System Tasks	✓	✗
Readability	✗	✓
Cross-Platform	✓	✓
Performance for Simple Tasks	✓	✗
Advanced Data Processing	✗	✓

1.3 Setting Up Your Bash Environment

Checking Bash Version

To check the installed Bash version, run:

```
bash --version
```

Installing Bash on Different Systems

Linux

Most Linux distributions come with Bash preinstalled. To install or update Bash:

```
sudo apt update && sudo apt install bash # Debian-based
sudo yum install bash # RedHat-based
```

macOS

macOS uses zsh by default now, but Bash is still available. To install the latest Bash:

```
brew install bash
```

Windows

Bash can be used in Windows via:

- **WSL (Windows Subsystem for Linux)**
- **Git Bash**

Choosing a Good Text Editor

- **Nano:** Easy to use (`nano script.sh`)
 - **Vim:** Powerful, but has a learning curve (`vim script.sh`)
 - **VS Code:** Best for scripting with extensions
-

1.4 Running Your First Bash Command

Opening Terminal

- **Linux/macOS:** Press `Ctrl + Alt + T`.
- **Windows:** Open **Git Bash** or **WSL Terminal**.

Basic Commands

Try these commands in the terminal:

```
ls          # List files and directories
pwd         # Show current directory
whoami      # Display logged-in user
```

Executing Commands in Bash

Run a simple command:

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

1.5 Understanding the Shell

Interactive vs Non-Interactive Shell

- **Interactive Shell:** A shell session where the user interacts directly by typing commands.
- **Non-Interactive Shell:** Runs scripts without user input (e.g., cron jobs).

How the Shell Interprets Commands

- The shell reads input, finds the matching command, and executes it.
- Commands are found using the `$PATH` environment variable.

Understanding `$PATH`

- `$PATH` stores directories where executable files are located.
- To view your `PATH`:

```
echo $PATH
```

- To add a new directory to `PATH`:

```
export PATH=$PATH:/new/directory
```

Basics of Bash Commands

2.1 Navigating the File System

`ls`, `cd`, `pwd` Commands

- `ls` - Lists files in the current directory.
- `cd` - Changes directory.
- `pwd` - Prints the current working directory.

Using `cd ..` and `cd -`

- `cd ..` moves **one level up**.
 - `cd -` switches to the **previous directory**.
-

2.2 File Operations

Creating Files (`touch`)

```
touch filename.txt
```

Moving & Renaming (mv)

```
mv oldname.txt newname.txt
```

Copying Files (cp)

```
cp file1.txt file2.txt
```

Removing Files (rm)

```
rm filename.txt
```

2.3 Viewing File Contents

Commands: cat, less, head, tail

- **cat** - Display entire file.
- **less** - View file page by page.
- **head** - Show first 10 lines.
- **tail** - Show last 10 lines.

Combining Commands (cat file | less)

```
cat largefile.txt | less
```

2.4 Understanding File Permissions

File Permission Basics

In Linux, every file has three types of permissions: **read (r)**, **write (w)**, and **execute (x)**. These permissions apply to three different categories:

1. **Owner (User - u)**: The user who owns the file.
2. **Group (g)**: Users in the same group as the file owner.
3. **Others (o)**: Everyone else.

Checking Permissions (ls -l)

To view file permissions, use:

```
ls -l filename.txt
```

Example output:

```
-rwxr--r-- 1 user group 1234 Feb 14 12:00 filename.txt
```

Breakdown:

Symbol	Meaning
-	Regular file (d = directory)
rwX	Owner has read , write , execute permission
r--	Group has read-only permission
r--	Others have read-only permission

Permission Table

Numeric Code	Symbolic	Meaning
0	---	No permissions
1	--x	Execute only
2	-w-	Write only
3	-wx	Write & execute
4	r--	Read only
5	r-x	Read & execute
6	rw-	Read & write
7	rwX	Read, write & execute

Changing Permissions (chmod)

Use chmod to modify file permissions:

```
chmod 755 filename.txt
```

This sets permissions to:

- **Owner:** Read, write, execute (7)
- **Group:** Read, execute (5)
- **Others:** Read, execute (5)

Changing Ownership (chown)

To change the owner of a file:

```
sudo chown newuser filename.txt
```

To change both owner and group:

```
sudo chown newuser:newgroup filename.txt
```

2.5 Process Management

Viewing Running Processes (ps, top, htop)

```
ps aux
```

```
top
```

Killing Processes (kill, killall, pkill)

```
kill PID
```

```
killall processname  
pkill -9 processname
```

3 Writing and Executing Bash Scripts

3.1 Introduction to Bash Scripting

Bash scripting allows users to automate tasks by writing a sequence of commands in a script file. These scripts can:

- Automate repetitive tasks
- Perform system administration tasks
- Simplify command execution

A Bash script is a text file containing a series of commands that can be executed in sequence.

3.2 Creating Your First Bash Script

Steps to Create a Bash Script

1. Open a terminal and create a new file:
`touch myscript.sh`
2. Open the file with a text editor (Nano, Vim, or VS Code):
`nano myscript.sh`

3. Add the following content:

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

4. Save the file and exit the editor.
5. Give the script execution permission:

```
chmod +x myscript.sh
```

6. Run the script:

```
./myscript.sh
```

Output:

```
Hello, World!
```

3.3 Understanding the Shebang (`#!/bin/bash`)

What is a Shebang?

- The **shebang** (`#!`) is the first line of a script that specifies which interpreter should be used to execute the script.
- The most common shebang for Bash scripts is:

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

Why is the Shebang Needed?

- Without the shebang, the script might not execute correctly.
- It ensures the script runs with the correct shell, even if a different shell (e.g., `zsh`, `sh`) is set as default.
- Example:

```
#!/bin/bash
echo "This script runs with Bash"
```

Running this ensures Bash is used instead of another shell.

3.4 Executing Bash Scripts

Running Scripts with `./`

- You can run a script in the current directory by prefixing it with `./`:

```
./myscript.sh
```

Running Scripts with `bash`

- You can also run a script using the `bash` command:

```
bash myscript.sh
```

- This method does not require execution permission.

Running Scripts from Anywhere

- If you want to execute a script from any location, move it to `/usr/local/bin/`:

```
sudo mv myscript.sh /usr/local/bin/
```

- Now you can run it from anywhere by typing:

```
myscript.sh
```

3.5 Understanding Comments in Bash

Single-Line Comments (#)

- Use # to write comments in Bash scripts.
- Anything after # on a line is ignored by the shell.
- Example:

```
# This is a single-line comment
echo "Hello, World!" # This comment is ignored
```

Multi-Line Comments

Bash does not have a built-in multi-line comment syntax, but you can use:

1. Multiple # lines:

```
# This is a comment
# that spans multiple lines.
```

2. The : <<EOF method:

```
: <<EOF
This is a multi-line comment.
It spans multiple lines and is ignored by Bash.
EOF
```

4 Variables and Data Types

4.1 Declaring Variables

Syntax (VAR=value)

In Bash, variables are declared without a data type, and values are assigned using the = operator.

```
VAR=value
```

Example:

```
name="Ahamed"
echo "$name"
```

Rules for Naming Variables

- Variable names **must** begin with a letter or underscore (_).
- They **cannot** contain spaces; use underscores instead (my_var).
- They **cannot** start with a number (1var is invalid).
- Use uppercase names for environment variables (PATH, HOME).

- **Bash: Variables, Data Types, and Operators**

Bash scripting is an essential skill for automating tasks in Linux. Understanding variables, data types, and operators is fundamental to writing effective Bash scripts.

1. Variables in Bash

In Bash, variables store data, but they **do not have explicit data types** like in other programming languages (e.g., Python or C). Variables in Bash are **treated as strings by default**, even when they hold numbers.

Declaring Variables

```
name="Ahamed"    # No spaces around '='
age=25
```

- **No data type declaration** is needed.
- **No spaces around the equal sign (=)**.
- **Use \$ to access the variable:**

```
echo "My name is $name and I am $age years old."
```

- **Use {} when needed:**

```
echo "Hello, ${name}!"
```

Types of Variables

1. **Local Variables:** Available only in the current script or function.
2. **Environment Variables:** Available system-wide (e.g., \$PATH, \$HOME).

```
export MY_VAR="Hello"
```

3. **Special Variables:** Used in scripting (\$0, \$1, \$?, etc.).
-

2. Bash Variable Data Types

Bash does **not** have built-in types like `int`, `float`, or `bool`. Instead, variables are stored as **strings** by default, but they can be interpreted as numbers when used in arithmetic operations.

However, based on usage, variables behave like:

- **String:** Default behavior
- **Integer:** When used in arithmetic operations
- **Array:** A collection of values
- **Associative Array:** Key-value pairs

Checking Data Type

```
var="123"
if [[ "$var" =~ ^[0-9]+$ ]]; then
    echo "It's a number"
else
    echo "It's a string"
fi
```

3. Operators in Bash

Operators in Bash are used for arithmetic, comparison, string manipulation, and logical operations.

A. Arithmetic Operators

Bash supports integer arithmetic using `((...))` or the `expr` command.

Operator	Description	Example
+	Addition	echo <code>\$((5 + 3))</code> → 8
-	Subtraction	echo <code>\$((5 - 2))</code> → 3
*	Multiplication	echo <code>\$((5 * 3))</code> → 15
/	Division	echo <code>\$((6 / 2))</code> → 3
%	Modulus (remainder)	echo <code>\$((5 % 2))</code> → 1
**	Exponentiation	echo <code>\$((2 ** 3))</code> → 8

Examples

```
a=10
b=5
sum=$((a + b))
echo "Sum is: $sum"
```

or using `expr`:

```
sum=$(expr $a + $b)
echo "Sum is: $sum"
```

B. Comparison Operators

Used in conditional statements (`if`, `while`).

Operator	Description	Syntax
-eq	Equal to	if ["\$a" -eq "\$b"]
-ne	Not equal to	if ["\$a" -ne "\$b"]
-gt	Greater than	if ["\$a" -gt "\$b"]
-lt	Less than	if ["\$a" -lt "\$b"]
-ge	Greater than or equal	if ["\$a" -ge "\$b"]
-le	Less than or equal	if ["\$a" -le "\$b"]

Example

```
a=10
b=5
if [ "$a" -gt "$b" ]; then
    echo "$a is greater than $b"
fi
```

For floating-point numbers, use `bc`:

```
result=$(echo "3.5 + 2.1" | bc)
echo "Result: $result"
```

C. String Operators

Operator	Description	Example
=	Equal	["\$a" = "\$b"]
!=	Not equal	["\$a" != "\$b"]
-z	Empty string	[-z "\$a"]
-n	Non-empty string	[-n "\$a"]

Example:

```
name="Ahamed"
if [ "$name" = "Ahamed" ]; then
    echo "Matched!"
fi
```

D. Logical Operators

Used for combining conditions.

Operator	Description	Example
&&	Logical AND	if ["\$a" -gt 5] && ["\$b" -lt 10]; then `
!	Logical NOT	if [! -z "\$a"]; then

Example:

```
if [ "$age" -gt 18 ] && [ "$age" -lt 30 ]; then
    echo "Eligible"
fi
```

E. File Operators

Used for checking file properties.

Operator	Description	Example
-e	Exists	[-e file.txt]
-f	Regular file	[-f file.txt]
-d	Directory	[-d dir_name]

Operator	Description	Example
-r	Readable	[-r file.txt]
-w	Writable	[-w file.txt]
-x	Executable	[-x script.sh]

Example:

```
if [ -f "myfile.txt" ]; then
    echo "File exists"
fi
```

4.2 Using and Accessing Variables

Printing Variables (echo \$VAR)

Variables can be accessed using the \$ symbol:

```
VAR="Hello World"
echo $VAR
```

Output:

Hello World

Using Variables in Strings

```
name="Ahamed"
echo "My name is $name"
```

Output:

My name is Ahamed

4.3 Read-Only Variables

Declaring Read-Only Variables (readonly VAR)

A variable can be made read-only using readonly:

```
readonly myvar="Cannot Change"
myvar="New Value" # This will cause an error
```

4.4 Local and Global Variables

Local Variables in Functions

A local variable is only accessible inside a function:

```
my_function() {
```

```
    local local_var="I am local"
    echo $local_var
}
my_function
```

Exporting Variables (**export VAR=value**)

To make a variable available to child processes, use **export**:

```
export GLOBAL_VAR="Available everywhere"
```

4.5 Environment Variables

Common Environment Variables

Some commonly used environment variables:

- **\$HOME** - User's home directory
- **\$PATH** - Directories where executables are searched for

Modifying **.bashrc**

To make a variable permanent, add it to **~/ .bashrc**:

```
echo 'export MY_VAR="Persistent Value"' >> ~/.bashrc
source ~/.bashrc
```

Input and Output in Bash

5.1 Reading User Input

Using **read** Command

The **read** command allows users to input data:

```
read name
echo "Hello, $name!"
```

Using **-p** for Prompts

```
read -p "Enter your name: " username
echo "Welcome, $username!"
```

5.2 Printing Output

Using `echo` and `printf`

```
echo "Hello, World!"  
printf "Hello, %s!\n" "User"
```

Formatting Output with `printf`

```
printf "%-10s %-8s\n" "Name" "Age"  
printf "%-10s %-8s\n" "Ahamed" "25"
```

5.3 Redirection

Redirecting Output (`>`, `>>`)

```
echo "Hello" > file.txt # Overwrites file  
echo "World" >> file.txt # Appends to file
```

Redirecting Input (`<`)

```
sort < file.txt
```

5.4 Pipes

Combining Commands (`ls | grep txt`)

```
ls | grep txt
```

6 Conditional Statements

6.1 If-Else Statements

Basic Syntax

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

Using `elif` for Multiple Conditions

```
if [ "$var" == "A" ]; then  
    echo "Variable is A"  
elif [ "$var" == "B" ]; then
```

```
    echo "Variable is B"
else
    echo "Variable is neither A nor B"
fi
```

6.2 Case Statements

Using **case** Instead of **if**

```
case "$var" in
    "A") echo "Variable is A" ;;
    "B") echo "Variable is B" ;;
    *) echo "Variable is something else" ;;
esac
```

Example: Menu Selection

```
echo "Choose an option:"
echo "1) Start"
echo "2) Stop"
read choice
case $choice in
    1) echo "Starting..." ;;
    2) echo "Stopping..." ;;
    *) echo "Invalid choice" ;;
esac
```

6.3 Comparison Operators

Integer Comparisons

- **-eq** (equal)
- **-lt** (less than)
- **-gt** (greater than)

String Comparisons

- **==** (equal)
 - **!=** (not equal)
-

6.4 Logical Operators

- **&&** (AND)
- **||** (OR)

7 Loops in Bash

7.1 For Loop

Iterating Over Lists

```
for item in apple banana cherry; do
    echo "$item"
done
```

Using {1..10} Syntax

```
for i in {1..10}; do
    echo "Number: $i"
done
```

7.2 While Loop

Using while with Conditions

```
count=5
while [ $count -gt 0 ]; do
    echo "Countdown: $count"
    ((count--))
done
```

Example: Countdown Timer

```
seconds=10
while [ $seconds -gt 0 ]; do
    echo "$seconds seconds remaining..."
    sleep 1
    ((seconds--))
done
echo "Time's up!"
```

7.3 Until Loop

Running Until a Condition is Met

```
x=1
until [ $x -ge 5 ]; do
    echo "x is $x"
    ((x++))
done
```

7.4 Loop Control

Using **break** to Exit Loops

```
for i in {1..10}; do
    if [ $i -eq 5 ]; then
        break
    fi
    echo "Iteration $i"
done
```

Using **continue** to Skip Iteration

```
for i in {1..5}; do
    if [ $i -eq 3 ]; then
        continue
    fi
    echo "Iteration $i"
done
```

8 Functions in Bash

8.1 Defining and Calling Functions

Function Syntax

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

Calling Functions Inside Scripts

```
my_function
```

8.2 Function Arguments

Passing Arguments (\$1, \$2)

```
my_function() {
    echo "First argument: $1"
}
```

```
my_function "Hello"
```

Using **shift** to Handle Multiple Arguments

```
while (( "$#" )); do
    echo "Argument: $1"
    shift
done
```

8.3 Returning Values

Returning Values with `return` and `echo`

```
my_function() {  
    return 5  
}  
my_function  
echo $?
```

8.4 Local vs Global Variables in Functions

```
my_function() {  
    local var="Local Variable"  
    echo $var  
}  
my_function  
echo $var # This will be empty
```

9 Arrays in Bash

9.1 Declaring and Using Arrays

Declaring an Array

```
my_array=("element1" "element2" "element3")
```

9.2 Accessing Elements

```
echo ${my_array[0]} # Access first element  
echo ${my_array[@]} # Access all elements  
echo ${#my_array[@]} # Get array length
```

9.3 Adding and Removing Elements

```
my_array+=("new_element") # Add an element  
unset my_array[1]         # Remove second element
```

9.4 Iterating Over Arrays

```
for item in "${my_array[@]}; do  
    echo "$item"  
done
```

Associative Arrays (Key-Value Pairs)

```
declare -A my_dict
my_dict=( [name]="Ahamed" [age]=25 )
echo "Name: ${my_dict[name]}"
echo "Age: ${my_dict[age]}"
```

10 Advanced Bash Scripting

10.1 Command-Line Arguments (\$@, \$#)

Accessing Arguments

Command-line arguments allow users to pass input values to scripts. These arguments can be accessed using special variables:

- \$1, \$2, ... - Access individual arguments
- \$@ - All arguments as separate words
- \$* - All arguments as a single string
- \$# - Number of arguments passed

Example:

```
echo "First argument: $1"
echo "All arguments: $@"
echo "Number of arguments: $#"
```

Shifting Arguments

shift moves arguments to the left:

```
shift # Moves all arguments one place left
```

10.2 Working with Date and Time

Displaying Date and Time

The date command provides date and time information:

```
echo "Current date: $(date)"
echo "Timestamp: $(date +%s)"
```

Formatting Date and Time

You can customize the date format:

```
echo "Formatted Date: $(date +"%Y-%m-%d %H:%M:%S")"
```

Let's break it down:

```
echo "Formatted Date: $(date +"%Y-%m-%d %H:%M:%S")"
```

Breakdown of Format Specifiers:

Format	Meaning	Example Output
%Y	Full year (4 digits)	2025
%m	Month (2 digits, 01-12)	02 (February)
%d	Day of the month (2 digits)	15 (15th day)
%H	Hour (24-hour format, 00-23)	14 (2 PM)
%M	Minutes (00-59)	45
%S	Seconds (00-59)	30

Example Output:

If the current date and time is **February 15, 2025, at 14:45:30**, the command will output:

```
Formatted Date: 2025-02-15 14:45:30
```

Using Date in Filenames

```
backup_file="backup_$(date +%Y%m%d).tar.gz"
```

10.3 String Manipulation

Finding String Length

```
str="Hello, World!"  
echo "Length: ${#str}"
```

Extracting a Substring

```
echo "Substring: ${str:0:5}"
```

Replacing Substrings

```
echo "${str/Hello/Hi}"
```

10.4 File Handling

Creating and Writing to Files

```
echo "Hello" > file.txt
```

Reading File Content

```
cat file.txt
```

Appending to Files

```
echo "New line" >> file.txt
```

Checking If a File Exists

```
if [[ -f file.txt ]]; then
    echo "File exists"
fi
```

Removing Files

```
rm file.txt
```

Copying and Moving Files

```
cp source.txt destination.txt
mv oldname.txt newname.txt
```

Looping Through Files

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

10.5 Process Management

Running Commands in the Background

```
long_running_command &
```

Listing Running Processes

```
ps aux | grep process_name
```

Killing Processes

```
kill -9 PID
```

10.6 Error Handling

Using Exit Codes

```
if command; then
    echo "Success"
else
    echo "Failure"
fi
```

Capturing Errors

```
command || echo "Command failed"
```

10.7 Debugging Bash Scripts

Enabling Debug Mode

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

Using `set` for Debugging

```
set -x # Enable debugging  
set +x # Disable debugging
```

Process Management and Job Control

11.1 Background & Foreground Processes (&, fg, bg)

Running a Process in the Background

```
long_running_command &
```

Listing Background Jobs

```
jobs
```

Bringing a Background Process to the Foreground

```
fg %1 # Brings job number 1 to foreground
```

Sending a Foreground Process to the Background

```
Ctrl + Z # Stops process  
bg %1    # Sends process to background
```

Checking the Status of Jobs

```
jobs -l # Lists jobs with process IDs
```

11.2 Process Signals (kill, trap)

Understanding Process Signals

- SIGTERM (15): Graceful termination
- SIGKILL (9): Force termination
- SIGINT (2): Interrupt (Ctrl + C)

Sending Signals to a Process

```
kill -9 PID # Forcefully terminates process with given PID
kill -TERM PID # Gracefully terminates process
```

Using trap to Handle Signals

```
trap "echo 'Process interrupted!'" SIGINT
```

Ignoring Signals

```
trap "" SIGTERM # Ignores termination signal
```

11.3 Monitoring Processes (ps, top, htop)

Viewing Running Processes

```
ps aux # Lists all running processes
```

Filtering Processes

```
ps aux | grep process_name
```

Interactive Process Monitoring

```
top # Displays real-time process monitoring
htop # Enhanced interactive process viewer
```

Checking CPU and Memory Usage

```
ps -eo pid,ppid,cmd,%mem,%cpu --sort=-%mem | head
```

Killing Unresponsive Processes

```
pkill process_name # Kills process by name
kill -9 PID # Force kill using process ID
```

Monitoring a Specific Process

```
watch -n 1 "ps -p PID -o %cpu,%mem,cmd"
```

1 2 Debugging Bash Scripts

12.1 Enabling Debug Mode (set -x)

Running Scripts in Debug Mode

```
bash -x script.sh # Runs script with debugging enabled
```

Using set -x and set +x

```
set -x # Enable debugging for script execution
set +x # Disable debugging after a specific point
```

12.2 Error Handling (||, &&, exit)

Using exit for Error Handling

```
if [[ ! -f file.txt ]]; then
    echo "File not found! Exiting."
    exit 1
fi
```

Using && and || for Conditional Execution

```
echo "Task successful" && echo "This runs only if the first command succeeds"
echo "Task failed" || echo "This runs only if the first command fails"
```

12.3 Logging (tee, logger)

Redirecting Output to a Log File

```
command | tee logfile.txt # Saves output while displaying it
```

Using logger for System Logs

```
echo "Script started" | logger
```

1 3 Working with Files and Directories

13.1 Finding Files (find, locate)

- Enabling Efficient File Search with find

- Searching Files by Name or Type:

```
find /path/to/search -type f -name "*.txt"
```


- Searching by File Size:

```
find /path/to/search -size +1M # Files larger than 1MB
```

- Combining Multiple Search Criteria:

```
find /path/to/search -type f -name "*.txt" -size +1M -exec ls -l  
{ } \;
```

- Using **locate** for Fast Search

- **Locating Files:**

```
locate myfile.txt
```

- **Updating the Database** (if needed):

```
updatedb
```

- Using **find** vs. **locate**

- **find**: Real-time search, can be customized with multiple criteria.
 - **locate**: Fast, but relies on the database, which might not be up-to-date.
-

13.2 Counting Words and Lines (wc, grep)

- Counting Lines and Words with **wc**

- **Basic Usage:**

```
wc -l file.txt # Count lines in file  
wc -w file.txt # Count words in file
```

- **Counting Multiple Files:**

```
wc -l *.txt # Count lines across all .txt files
```

- Searching and Counting with **grep**

- **Basic Search:**

```
grep "pattern" file.txt # Find pattern in a file
```

- **Case-insensitive Search:**

```
grep -i "pattern" file.txt
```

- **Count Occurrences of a Pattern:**

```
grep -o "pattern" file.txt | wc -l # Count occurrences of pattern
```

- Using **grep** with **wc** for Line Matching

```
grep -i "error" file.txt | wc -l # Count lines containing "error"
```

13.3 Sorting and Filtering (sort, uniq)

- **Sorting Files with sort**

- **Basic Sorting:**

- ```
sort file.txt # Sort lines in ascending order
```

- **Numeric Sorting:**

- ```
sort -n file.txt # Sort numerically
```

- **Reverse Sorting:**

- ```
sort -r file.txt # Sort in reverse order
```

- **Filtering Duplicates with uniq**

- **Removing Duplicates:**

- ```
sort file.txt | uniq # Removes adjacent duplicates
```

- **Counting Occurrences:**

- ```
sort file.txt | uniq -c # Count duplicates
```

- **Combining sort and uniq for Unique Sorted Output**

- ```
sort file.txt | uniq -u # Unique values in sorted file
```

13.4 Archiving and Compressing (tar, zip)

- **Creating and Extracting Archives with tar**

- **Create a .tar Archive:**

- ```
tar -cvf archive.tar directory/ # Create archive
```

- **Extract a .tar Archive:**

- ```
tar -xvf archive.tar # Extract archive
```

- **Create a Compressed .tar.gz Archive:**

- ```
tar -czvf archive.tar.gz directory/ # Create compressed archive
```

- **Extract a .tar.gz Archive:**

- ```
tar -xzvf archive.tar.gz # Extract compressed archive
```

- **Using zip for Compression**

- **Create a .zip Archive:**

- ```
zip archive.zip file1.txt file2.txt # Create zip archive
```

- **Extract a .zip Archive:**

```
unzip archive.zip # Extract zip archive
```

- **Comparing Tar and Zip**
    - **tar**: Ideal for bundling multiple files, commonly used in Linux.
    - **zip**: Popular in Windows environments, simple compression for individual files.
- 

## 14 Networking with Bash

### 14.1 Fetching Data from the Internet (curl, wget)

- **Fetching Data with curl**
  - **Basic Usage**: Retrieve the content of a webpage or resource.  

```
curl https://example.com # Fetch the content of the URL
```
  - **Download Files**:  

```
curl -O https://example.com/file.zip # Download file and save with the same name
```
  - **Handling Authentication**:  

```
curl -u username:password https://example.com/data
```
- **Fetching Data with wget**
  - **Basic Usage**: Similar to curl, used for downloading files.  

```
wget https://example.com/file.zip # Download file
```
  - **Recursive Downloading** (for downloading websites or directories):  

```
wget -r https://example.com/ # Download entire website
```
- **Comparing curl and wget**
  - curl is typically used for making requests to HTTP(S) servers, can handle multiple protocols (FTP, HTTP, etc.).
  - wget is optimized for downloading files, supports recursive downloads and background downloading.

#### Exercise:

Write a script that uses curl to fetch JSON data from a public API and saves it into a file.

---

## 14.2 Checking Network Status (ping, netstat)

- **Using ping to Test Connectivity**

- **Basic Ping:**

- ```
ping google.com # Test if the network is reachable
```

- **Ping with Specific Number of Packets:**

- ```
ping -c 4 google.com # Send 4 ping requests
```

- **Using netstat for Network Statistics**

- **Check Open Network Connections:**

- ```
netstat -tuln # Show listening ports and active connections
```

- **Display Routing Tables:**

- ```
netstat -r # Show routing table
```

- **Checking Network Interfaces**

- Use netstat to list all network interfaces:

- ```
netstat -i
```

Exercise:

Write a script that checks whether a particular server is reachable using ping and displays the results.

14.3 Transferring Files (scp, rsync)

- **Using scp for Secure File Copy**

- **Copy Files Between Local and Remote Systems:**

- ```
scp file.txt user@remote:/path/to/destination
```

- **Copy Directories Recursively:**

- ```
scp -r directory/ user@remote:/path/to/destination
```

- **Using rsync for Efficient File Synchronization**

- **Basic Synchronization:**

- ```
rsync -av source/ destination/
```

- **Synchronize Over SSH:**

- ```
rsync -av -e ssh source/ user@remote:/path/to/destination
```

- **Why rsync Is Better than scp:**

- `rsync` only transfers changed or new files, making it faster for large directories.
- It also allows file compression during transfer.

Exercise:

Write a script to sync a local backup directory with a remote server using `rsync`.

14.4 Automating SSH Connections (ssh)

- **Using ssh for Secure Shell Connections**

- **Basic SSH Command:**

- ```
ssh user@remote # Connect to remote server
```

- **Run Commands on Remote Server:**

- ```
ssh user@remote 'ls -l /path/to/dir'
```

- **Using SSH Key-Based Authentication**

- **Generate SSH Key Pair:**

- ```
ssh-keygen -t rsa -b 4096 # Generate a new SSH key pair
```

- **Copy Public Key to Remote Server:**

- ```
ssh-copy-id user@remote # Copy the public key to the remote server
```

- **Automating SSH Connections with Scripts**

- **Automating Tasks with SSH:**

- Use SSH in scripts to automate remote server tasks. For example, backup, update, or restart services.

- ```
ssh user@remote 'sudo systemctl restart apache2'
```

**Exercise:**

Create a script that automatically logs into a remote server and backs up a specific directory using `rsync` via SSH.

---

Here's a more detailed version of Chapter 15: Automation and Cron Jobs, with added explanations and exercises for each section:

---

# 15 Automation and Cron Jobs

## 15.1 Scheduling Tasks with cron

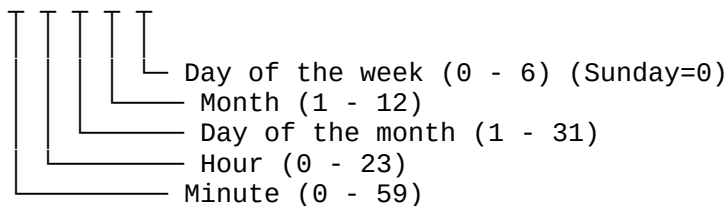
- **What is cron?**

- cron is a time-based job scheduler in Unix-like operating systems. It allows you to run commands or scripts at specific intervals, which is perfect for automating repetitive tasks like backups, log rotations, and system monitoring.

- **Cron Syntax**

The cron schedule consists of five fields representing different time units, followed by the command to be executed:

```
* * * * * command-to-be-executed
```



- **Cron Time Field Details**

- **Minute (0 - 59):** The minute at which the task will run.
- **Hour (0 - 23):** The hour of the day.
- **Day of the Month (1 - 31):** The day of the month on which the task will run.
- **Month (1 - 12):** The month during which the task will run.
- **Day of the Week (0 - 6):** The day of the week when the task will run (0 = Sunday).

- **Examples of Cron Scheduling:**

- **Every day at 3:30 AM:**

```
30 3 * * * command
```

- **Every Monday at 2:00 PM:**

```
0 14 * * 1 command
```

- **Every 15 minutes:**

```
*/15 * * * * command
```

- **Every Sunday at midnight:**

```
0 0 * * 0 command
```

### Exercise:

1. Schedule a cron job that runs every day at 6:00 AM to back up a directory to another location. Create a script for this task (backup.sh).

```
#!/bin/bash
rsync -av /path/to/source/ /path/to/backup/
```

- Schedule it with the cron job:

```
0 6 * * * /path/to/backup.sh
```

---

## 15.2 Using crontab for Automation

- **Editing Cron Jobs with crontab -e**

- To add or modify cron jobs, use the command `crontab -e`. This opens the cron table (crontab) file in the default text editor (typically `vi` or `nano`).
  - Add a new line with the desired schedule and command.
  - Save and exit the editor. Cron will automatically load the new schedule.

- **Listing Current Cron Jobs**

- To list the current user's cron jobs:

```
crontab -l
```

- **Removing Cron Jobs**

- To remove all cron jobs for the current user:

```
crontab -r
```

- **Cron Job Output and Logging**

By default, cron sends output (stdout and stderr) to the user's email. You can redirect the output to a file for logging purposes:

```
0 5 * * * /path/to/script.sh >> /path/to/logfile.log 2>&1
```

- `>>` appends output to a log file.
- `2>&1` ensures both stdout and stderr are logged.

### Exercise:

2. Create a script that checks the disk usage (`check_disk_usage.sh`) and sends a notification to the user if the usage exceeds 80%. `bash #!/bin/bash threshold=80 usage=$(df / | grep / | awk '{ print $5 }' | sed 's/%//g') if [ $usage -gt $threshold ]; then echo "Disk usage is above $threshold%!" | mail -s "Disk Usage Alert" user@example.com fi` - Schedule it to run every hour using cron: `bash 0 * * * * /path/to/check_disk_usage.sh`

---

## 15.3 Running Scripts Periodically

- **Automating Scripts with Cron**

You can automate the execution of scripts at regular intervals using cron. These can range from simple tasks like cleaning up temporary files to complex ones like performing backups.

- **Example: Backup Script**

Here's an example of a backup script `backup.sh` that copies files from a source directory to a backup directory:

```
#!/bin/bash
rsync -av --delete /path/to/source/ /path/to/backup/
```

- **Explanation:**

- `-a`: Archive mode (preserves permissions, timestamps, etc.).
    - `-v`: Verbose output.
    - `--delete`: Deletes files from the destination that no longer exist in the source.

- **Scheduling the Backup:** Schedule the backup script to run every day at 2:00 AM:

```
0 2 * * * /path/to/backup.sh
```

- **Using Cron for Log Rotation**

Log rotation is crucial for preventing log files from growing too large. You can automate log rotation by creating a script that renames and archives log files with a timestamp.

```
#!/bin/bash
mv /path/to/logfile.log /path/to/logfile_$(date +%Y%m%d).log
touch /path/to/logfile.log
```

- **Scheduling Log Rotation:** Schedule this script to run every day at midnight:

```
0 0 * * * /path/to/logrotate.sh
```

- **Automating System Updates**

You can automate system updates by creating a script that runs package updates for your system:

```
#!/bin/bash
apt update && apt upgrade -y
```

- **Scheduling System Updates:** Schedule the update script to run every Sunday at 3:00 AM:

```
0 3 * * 0 /path/to/update_system.sh
```

**Exercise:**

3. Create a script that monitors the memory usage (`check_memory_usage.sh`) and restarts a

service if the memory usage exceeds 90%. `bash` `#!/bin/bash` `threshold=90`  
`mem_usage=$(free | grep Mem | awk '{print $3/$2 * 100.0}')` `if (( $`  
`(echo "$mem_usage > $threshold" | bc -l) )); then systemctl`  
`restart service_name fi` - Schedule it to run every hour: `bash` `0 * * * *`  
`/path/to/check_memory_usage.sh`



---

# 16 Practical Bash Projects

Here's a detailed breakdown of **Chapter 16: Practical Bash Projects**, with each project explained and exercises included:

## 16.1 Creating a Simple To-Do List Script

- **Objective:**

Build a simple to-do list script where you can add tasks, view tasks, and mark them as completed. This will help reinforce basic Bash operations like reading input, appending to files, and conditionals.

- **Project Breakdown:**

1. **Storing Tasks:** Store tasks in a text file (e.g., `todo.txt`). Each line will represent one task.
2. **Adding Tasks:** Allow users to add tasks interactively.
3. **Viewing Tasks:** List the tasks with their corresponding task number.
4. **Marking Tasks as Done:** Allow users to mark a task as done by removing it from the list or striking it through.

- **Script Example:**

```
#!/bin/bash

TODO_FILE="todo.txt"

Function to display the to-do list
function show_todo() {
 echo "To-Do List:"
 nl $TODO_FILE
}

Function to add a new task
function add_task() {
 read -p "Enter task: " task
 echo $task >> $TODO_FILE
 echo "Task added!"
}

Function to mark a task as done
function mark_done() {
 show_todo
 read -p "Enter task number to mark as done: " task_number
 sed -i "${task_number}d" $TODO_FILE
 echo "Task marked as done!"
}

Main menu
while true; do
 echo "1. Show To-Do List"
 echo "2. Add Task"
 echo "3. Mark Task as Done"
```

```

echo "4. Exit"
read -p "Choose an option: " choice

case $choice in
 1) show_todo ;;
 2) add_task ;;
 3) mark_done ;;
 4) exit ;;
 *) echo "Invalid option!" ;;
esac
done

```

- **How It Works:**

1. The script uses a `todo.txt` file to store tasks.
2. `show_todo` displays the tasks with line numbers.
3. `add_task` lets the user input a new task, appending it to `todo.txt`.
4. `mark_done` removes the task from `todo.txt` based on the task number.

**Exercise:**

Create the script, and test adding, viewing, and marking tasks as done. Expand it by adding an option to save tasks to a different file (e.g., `completed.txt`) when they are marked as done.

---

## 16.2 Automating Backups

- **Objective:**

Write a script that automatically backs up important files or directories to a backup location. The script should log the result and be scheduled using cron for periodic backups.

- **Project Breakdown:**

1. **Define Directories:** Specify the source directory (e.g., `/home/user/documents`) and backup destination (e.g., `/home/user/backups`).
2. **Create Backups:** Use `rsync` or `tar` to copy files or compress them for the backup.
3. **Log Backups:** Append the result to a log file, noting the time of the backup and success/failure.

- **Script Example:**

```

#!/bin/bash

SOURCE_DIR="/home/user/documents"
BACKUP_DIR="/home/user/backups"
LOG_FILE="/home/user/backup_log.txt"
DATE=$(date "+%Y-%m-%d_%H-%M-%S")
BACKUP_NAME="backup_$(date +%Y-%m-%d_%H-%M-%S).tar.gz"

Perform backup
echo "Starting backup at $DATE" >> $LOG_FILE
if tar -czf $BACKUP_DIR/$BACKUP_NAME $SOURCE_DIR; then
 echo "Backup successful: $BACKUP_NAME" >> $LOG_FILE
else
 echo "Backup failed: $BACKUP_NAME" >> $LOG_FILE
fi

```

- **How It Works:**

1. The script uses `tar` to compress the source directory and save it in the backup directory.
2. It appends success or failure messages to a log file with the timestamp.

**Exercise:**

Schedule this backup script to run daily at 3:00 AM using cron. Modify the script to rotate backups, keeping only the last 7 backups.

---

## 16.3 Writing a Log Monitoring Script

- **Objective:**

Create a script that monitors a log file (e.g., `/var/log/syslog` or `/var/log/auth.log`) for specific events or errors and sends an alert if certain keywords are found.

- **Project Breakdown:**

1. **Monitor Log File:** Continuously monitor the log file for changes.
2. **Search for Keywords:** Look for specific error keywords (e.g., "ERROR", "FAILED", "CRITICAL").
3. **Send Alerts:** If a keyword is found, send an email or notify the user.

- **Script Example:**

```
#!/bin/bash

LOG_FILE="/var/log/syslog"
KEYWORDS=("ERROR" "FAILED" "CRITICAL")
EMAIL="user@example.com"

Monitor the log file for changes
tail -F $LOG_FILE | while read line; do
 for keyword in "${KEYWORDS[@]}; do
 if echo "$line" | grep -iq "$keyword"; then
 echo "Alert: Found $keyword in log" | mail -s "Log Alert"
 $EMAIL
 break
 fi
 done
done
```

- **How It Works:**

1. `tail -F` monitors the log file for new lines.
2. If any of the keywords are found in the new lines, an email is sent to the specified address.

**Exercise:**

Test the script by forcing some errors in a log file (e.g., manually adding "ERROR" to `/var/log/syslog`). Modify the script to support multiple log files and send alerts through logger instead of email.

---

## 16.4 Developing a System Health Checker

- **Objective:**

Develop a script that checks the system's health by monitoring CPU usage, memory usage, disk space, and active processes. The script should alert the user if any resource usage exceeds a specified threshold.

- **Project Breakdown:**

1. **Check System Resources:** Use commands like `top`, `df`, and `free` to check CPU, memory, and disk usage.
2. **Compare to Thresholds:** Set thresholds (e.g., CPU > 90%, Memory > 80%, Disk space > 90%).
3. **Send Alerts:** If any resource exceeds its threshold, send an alert to the user.

- **Script Example:**

```
#!/bin/bash

CPU_THRESHOLD=90
MEM_THRESHOLD=80
DISK_THRESHOLD=90

Check CPU usage
CPU_USAGE=$(top -bn1 | grep "Cpu(s)" | sed "s/.*, *\([0-9.]*\)%* id.*/\1/" | awk '{print 100 - $1}')
if (($(echo "$CPU_USAGE > $CPU_THRESHOLD" | bc -l))); then
 echo "Alert: CPU usage is above $CPU_THRESHOLD%" | mail -s "System Health Alert" user@example.com
fi

Check Memory usage
MEM_USAGE=$(free | grep Mem | awk '{print $3/$2 * 100.0}')
if (($(echo "$MEM_USAGE > $MEM_THRESHOLD" | bc -l))); then
 echo "Alert: Memory usage is above $MEM_THRESHOLD%" | mail -s "System Health Alert" user@example.com
fi

Check Disk usage
DISK_USAGE=$(df / | grep / | awk '{ print $5 }' | sed 's/%//g')
if [$DISK_USAGE -gt $DISK_THRESHOLD]; then
 echo "Alert: Disk usage is above $DISK_THRESHOLD%" | mail -s "System Health Alert" user@example.com
fi
```

- **How It Works:**

1. The script checks the current CPU, memory, and disk usage.
2. If any of the resource usages exceed the specified threshold, an alert email is sent.

### Exercise:

Run this script manually and observe its behavior when resource usage exceeds the thresholds. Modify the script to check for active processes (e.g., checking if a critical service is running).

## 16.5 Fun Bash Games

Here are some small, fun Bash project ideas with a focus on games and interactive scripts. These will add a bit of entertainment to your learning while reinforcing your Bash scripting skills.

### 16.5.1 Guess the Number Game

- **Objective:**  
Create a simple "Guess the Number" game where the script randomly generates a number between 1 and 100, and the user has to guess it.
- **How It Works:**
  - The script generates a random number.
  - The user has to input guesses.
  - The script tells the user whether their guess is too high, too low, or correct.
- **Script Example:**

```
#!/bin/bash

echo "Welcome to the Guess the Number Game!"
secret_number=$((RANDOM % 100 + 1))
attempts=0

while true; do
 read -p "Guess a number between 1 and 100: " guess
 ((attempts++))

 if [[$guess -lt $secret_number]]; then
 echo "Too low! Try again."
 elif [[$guess -gt $secret_number]]; then
 echo "Too high! Try again."
 else
 echo "Correct! You guessed the number in $attempts attempts."
 break
 fi
done
```

#### Exercise:

- Add a feature that tells the user how close they are to the secret number (e.g., "You are 5 away").
  - Make it so the game keeps track of the best score (fewest attempts).
- 

### 16.5.2 Rock, Paper, Scissors Game

- **Objective:**  
Create a command-line version of the classic rock, paper, scissors game, where the user plays against the computer.
- **How It Works:**
  - The user selects one of the three options.
  - The computer randomly selects an option.

- The script compares the choices and determines the winner.
- **Script Example:**

```
#!/bin/bash

echo "Rock, Paper, Scissors Game"
echo "Choose your move: (rock, paper, or scissors)"
read -p "Your choice: " user_choice

options=("rock" "paper" "scissors")
computer_choice=${options[$RANDOM % 3]}

echo "Computer chose: $computer_choice"

if [[$user_choice == $computer_choice]]; then
 echo "It's a tie!"
elif [[($user_choice == "rock" && $computer_choice == "scissors") ||
 ($user_choice == "scissors" && $computer_choice == "paper") ||
 ($user_choice == "paper" && $computer_choice == "rock")]]; then
 echo "You win!"
else
 echo "You lose!"
fi
```

#### Exercise:

- Add a feature to play multiple rounds.
  - Add a scoring system to keep track of wins and losses.
  - Make the user's input case-insensitive (e.g., allow "Rock", "rock", "RoCk").
- 

### 16.5.3 Hangman Game

- **Objective:**  
Create a simple version of the classic hangman game where the user guesses letters in a hidden word.
- **How It Works:**
  - The script randomly selects a word from a predefined list.
  - The user guesses one letter at a time.
  - The game continues until the user guesses the word or runs out of attempts.
- **Script Example:**

```
#!/bin/bash

words=("bash" "linux" "scripting" "programming" "terminal")
secret_word=${words[$RANDOM % ${#words[@]}]}
word_length=${#secret_word}
guessed_word=$(echo $secret_word | sed 's/./_/g')
attempts=6
guessed_letters=""

echo "Welcome to Hangman!"

while [[$attempts -gt 0 && "$guessed_word" != "$secret_word"]]; do
 echo "Word: $guessed_word"
 echo "Attempts left: $attempts"
 echo "Guessed letters: $guessed_letters"
```

```

read -p "Guess a letter: " guess

if [["$guessed_letters" == *"$guess"*]]; then
 echo "You already guessed that letter!"
 continue
fi

guessed_letters+="$guess"

if [["$secret_word" == *"$guess"*]]; then
 echo "Good guess!"
 guessed_word=$(echo $secret_word | sed
"s/[^$guessed_letters]_/g")
else
 ((attempts--))
 echo "Incorrect guess!"
fi
done

if [["$guessed_word" == "$secret_word"]]; then
 echo "You won! The word was $secret_word."
else
 echo "You lost! The word was $secret_word."
fi

```

#### Exercise:

- Add difficulty levels with longer or shorter words.
  - Implement a graphical representation of the hangman (using text like "O", "|", "/", etc.).
- 

### 16.5.4 Number Sequence Game

- **Objective:**

Create a game where the user has to guess the next number in a sequence based on a given rule (e.g., arithmetic progression).

- **How It Works:**

- The script generates a sequence of numbers with a specific pattern (e.g., increments by 2).
- The user has to guess the next number in the sequence.

- **Script Example:**

```

#!/bin/bash

echo "Welcome to the Number Sequence Game!"
echo "Guess the next number in the sequence."
sequence=(2 4 6 8 10) # This sequence increments by 2
echo "Sequence: ${sequence[@]}"
read -p "What is the next number? " guess

correct_answer=$((${sequence[-1]} + 2))

if [[$guess -eq $correct_answer]]; then
 echo "Correct! The next number was $correct_answer."
else
 echo "Incorrect! The correct number was $correct_answer."
fi

```

### Exercise:

- Create multiple sequence types (arithmetic progression, geometric progression, etc.).
  - Display the sequence with missing numbers, and the user has to guess the missing number.
- 

## 16.5.5 Tic-Tac-Toe Game

- **Objective:**

Create a simple text-based Tic-Tac-Toe game where the user plays against the computer or another player.

- **How It Works:**

- The game uses a 3x3 grid.
- Players take turns marking "X" and "O" in the grid.
- The game checks for a win or a tie.

- **Script Example:**

```
#!/bin/bash

Initialize the board
board=("1" "2" "3" "4" "5" "6" "7" "8" "9")

Function to display the board
function display_board() {
 echo "${board[0]} | ${board[1]} | ${board[2]}"
 echo "-----"
 echo "${board[3]} | ${board[4]} | ${board[5]}"
 echo "-----"
 echo "${board[6]} | ${board[7]} | ${board[8]}"
}

Check for a win
function check_win() {
 for i in 0 3 6; do
 if [[${board[$i]} == ${board[$((i+1))]} && ${board[$i]} == ${board[$((i+2))]}]]; then
 return 1
 fi
 done
 for i in 0 1 2; do
 if [[${board[$i]} == ${board[$((i+3))]} && ${board[$i]} == ${board[$((i+6))]}]]; then
 return 1
 fi
 done
 if [[${board[0]} == ${board[4]} && ${board[0]} == ${board[8]}]]; then
 return 1
 fi
 if [[${board[2]} == ${board[4]} && ${board[2]} == ${board[6]}]]; then
 return 1
 fi
 return 0
}

Play the game
player="X"
```



```

while true; do
 display_board
 read -p "Player $player, choose a position (1-9): " position
 ((position--))
 if [[${board[$position]} != "X" && ${board[$position]} != "O"]];
then
 board[$position]=$player
 if check_win; then
 display_board
 echo "Player $player wins!"
 break
 fi
 if [[! " ${board[@]} " =~ " "]]; then
 display_board
 echo "It's a tie!"
 break
 fi
 player="O"
 else
 echo "Invalid move, try again!"
 fi
done

```

### Exercise:

- Add a two-player mode and a computer opponent that picks random moves.
  - Display the current state of the board after every move.
- 

## Thank You for Reading!

I hope this book has helped you in your journey to mastering Bash scripting! If you found this book useful, please consider sharing it with others who might benefit from it.

Stay connected for more tutorials, updates, and resources:



**YouTube:** [[Linux-View](#)]



**Email:** [[Email](#)]



**Github:** [[Github](#)]

If you have any questions, suggestions, or feedback, feel free to reach out.





**Our Telegram channel** [[Linux-View](#)]



**Quora community** [[Linux-View-Bangla](#)]

Your support helps create more valuable content for the community.

**Happy Coding & Scripting!**  

— Ahamed | Learn Code with Noob