BASH

-Deepansh Sabharwal

What is Bash scripting?

A bash script is a file containing a sequence of commands that are executed by the bash program line by line. It allows you to perform a series of actions, such as navigating to a specific directory, creating a folder, and launching a process using the command line.

Advantages of Bash scripting

* **Automates tasks** – Saves time by running repetitive commands automatically.
* **Works everywhere** – Runs on Linux, macOS, Unix, and even Windows with tools.
* **Easy to edit** – Just use any text editor, no special tools needed.
* **Customizable** – Easily tweak scripts to fit your exact needs.
* **Combines well** – Works with other programs, languages, and tools.
* **Debug-friendly** – Built-in tools help find and fix script errors fast.

The terms "shell" and "bash" *are used interchangeably*. But there is a subtle difference between the two.

* The term "shell" refers to a program that provides a *command-line interface for interacting with an operating system*. Bash (Bourne-Again Shell) **is one of the most used Unix/Linux shells and is the default shell in many Linux distributions**.

In the below output, ubuntu@ubuntu is the shell prompt. When a shell is used interactively, it displays a $ when it is waiting for a command from the user.

A screenshot of a computer

AI-generated content may be incorrect.

You can determine your shell type by typing *ps*

A screenshot of a computer screen

AI-generated content may be incorrect.



**Here, I’m using BASH shell.**

Generally, commands follow this syntax:

*command [OPTIONS] arguments*

Some basic commands with uses

* **date** - Shows the current date and time.
* **mkdir –** Create a new folder in your home directory
* **cd –** Move into new folder
* **touch –** creating files
* **pwd** - Prints the current working directory.
* **ls** - Lists files and folders in the current directory.
* **echo** - Displays text or variable values on the terminal.
* A screenshot of a computer

  AI-generated content may be incorrect.
* If you want to access the manual (man) page for any command
  + Just type *man command\_name*

This is how we create a directory, move it into that folder, how we make the files and then move them into their respective folders, how we write and append into that file and how to read.

* **mkdir**: Create a new directory.
* **touch**: Create an empty file or update file’s timestamp.
* **cd**: Change current directory.
* **echo "text" > file**: Write text to a file (overwrite).
* **echo "text" >> file**: Append text to a file.
* **ls -lah**: List all files (including hidden) with detailed info in human-readable sizes.
* **cat file**: Display the contents of a file.
* **mv source destination**: Move or rename files and directories.
  + A screen shot of a computer program

    AI-generated content may be incorrect.



***I’m using my mac for bash scripting using HOMEBREW. VM or Dual boot can be used.***

How to Create and Execute Bash scripts

* **Script naming conventions**
* By naming convention, bash scripts end with .sh. However, bash scripts can run perfectly fine without the sh extension.

A computer screen with green text

AI-generated content may be incorrect.

**Adding the Shebang**

* Bash scripts start with a shebang. Shebang consists of #! followed by the path to the Bash interpreter. This is the first line of the script. Shebang tells the shell to execute it via bash shell. Shebang is simply an absolute path to the bash interpreter.

Creating our first bash script ☺

First, we will enter the vim (vi)

* Type: vi run\_all.sh (this creates the script)
* To write the commands, press i, after creating the script. This gets us into the *insert* mode
* Then type the following:

**#!/bin/bash** - This tells the system: “Hey, run this script using Bash”

**echo "Today is $(date)”** - Shows today’s date.

Now,

**echo -e "\nenter the path to directory"**

**read the\_path**

Asks the user to type in a folder path and saves it in a variable called the\_path

* Finally,

**echo -e "\n you path has the following files and folders: "**

**ls $the\_path**

Uses ls to list the contents of the path you entered.

* Now before running, **press *esc* to exit the insert mode and write *:wq*** *(write and quit)*

* Now we can run it. We do so by typing

**chmod +x scriptname (run\_all.sh)**

**./run\_all.sh**

* Now after we’ve run the script, it shows the current date and asks for the user path and prints the directory.

(THE COMPLETE SCRIPT IS AVAILABLE IN THE FOLDER)

Basics of Bash Scripting

Use # for comments

Variables and data types in bash (**NO SPACES ALLOWED**)

1. Assign the value directly
   1. country=India
2. Assign the value based on the output obtained from a program or command, using command substitution. Note that $ is required to access an existing variable's value.
   1. same\_country=$country
3. To access the variable value, append $ to the variable name.
   1. country=India
   2. echo $country
   3. This prints India
      1. Now if we write
         1. new\_country=$country
         2. echo $new\_country
         3. This prints India
   4. A black background with green text

      AI-generated content may be incorrect.

Input/Output in BASH

### a) Reading user input with read

* You can ask the user to type something and store it in a variable.

**bash**

**#!/bin/bash**

**echo "What's your name?" # Prompt the user**

**read entered\_name # Read input and store in variable**

**echo -e "\nWelcome to bash tutorial, $entered\_name"**

*read entered\_name* waits for user input and stores it in *entered\_name*.

Access variables by prefixing with $ (like *$entered\_name*).

* **The -e option with echo enables interpretation of backslash escapes like \n (newline).**

### b) Reading from a file line-by-line

You can also read from a file, line by line:

**bash**

**while read line**

**do**

**echo $line**

**done < input.txt**

* This reads each line from input.txt and prints it.
* The while read line; do ... done < filename syntax is common for file input.
* **Note: This will break if lines contain backslashes or leading/trailing spaces, advanced handling requires more complex code, but this works fine for basic use.**

### c) Command-line arguments

Bash scripts can take inputs when you run them.

* $1 is the **first argument**
* $2 is the second, and so on.

Example script:

**bash**

**#!/bin/bash**

**echo "Hello, $1!"**

Run it with an argument:

**bash script.sh Deepansh**

Output:

**Hello, Deepansh!**

* **$0 is the script name itself.**
* **$# holds the number of arguments.**
* **$@ or $\* hold all arguments.**

Displaying output

Printing to the terminal with echo

echo “Hello World”

Writing the output into a file

echo “This is the text” **>** file.txt

Append the output into the file

echo “This is the appended text” **>>** file.txt

Redirect output from commands

ls > file.txt (Tip: use >> not >)

Instead of printing the result of ls to the terminal, Bash **writes it into** file.txt.

If file.txt doesn’t exist — it creates it.  
If file.txt does exist — it **overwrites** it.

Conditionals

Basic Syntax:

```bash

if [ condition ]; then

# commands

elif [ condition ]; then

# commands

else

# commands

fi

*Important commands and concepts for conditionals:*

**[ ] vs [[ ]]:**

* [ ] is POSIX-compliant (portable).
* [[ ]] is Bash-specific **(more powerful, safer with strings).**

 **Integer Comparisons:**

* -eq (equal)
* -ne (not equal)
* -lt (less than)
* -le (less than or equal)
* -gt (greater than)
* -ge (greater than or equal)

 **String Comparisons:**

* = (equal)
* != (not equal)
* -z (is empty)
* -n (is not empty)

 **File Tests:**

* -f (is regular file)
* -d (is directory)
* -e (exists)
* -r (readable)
* -w (writable)
* -x (executable)

[Check check\_positive\_negative.sh for revision and understanding the use of these commands]

A black screen with green text

AI-generated content may be incorrect.

This is how we use these commands

## Loops

* **For Loop:**

for i in 1 2 3; do

echo $i

done

for i in {1..5}; do

echo $i

done

for file in $(ls); do

echo $file

done

* **While Loop:**

while [ condition ]; do

# commands

done

* **Until Loop:**

until [ condition ]; do

# commands

done

* **Break and Continue:**
  + break → exit loop
  + continue → skip to next iteration

## Case Statements

* **Syntax:**

case $var in

pattern1)

# commands

;;

pattern2|pattern3)

# commands

;;

\*)

# default

;;

esac

* **Use Case:** Useful in **menu-driven scripts**, cleaner than multiple if-else chains.

## Good Practices

* Always quote strings: "$var"
* Use consistent indentation
* Prefer [[ ... ]] for string conditions
* Use read safely inside loops or conditional blocks

References for practice and better explanation

<https://www.freecodecamp.org/news/bash-scripting-tutorial-linux-shell-script-and-command-line-for-beginners/#heading-introduction>

<https://www.w3schools.com/bash/bash_exercises.php>