🌸 bash scripting basics 🌸

🍥 introduction

> bash scripts = a list of commands you'd usually run by hand, saved in a file ✨

🌷 **why script?**

- automate tasks so u don't suffer manually 💻 💔
- reuse your magic with *one* command 🪄

💾 command-line bash vs bash script

- **command-line** = typing commands live like a chaotic gremlin
- **bash script** = calm, collected, all commands saved in a file 😌 📜

🌸 sample bash script

```
#!/bin/bash
mkdir demo
cd demo
mkdir code
mkdir doc
cd code
cp ../../hello.c ./
gcc -o hello hello.c
./hello
```

💡 this script:

- creates folders 🗂️
- copies a file 📄
- compiles it with gcc ⚙️
- runs the result 🚀

🐚 running your script

1️⃣ run with bash

```
bash 00-bash.sh
```

2️⃣ run as executable with shebang (✨ hash-bang✨)

```
#!/usr/bin/env bash  # more flexible + portable
```

💬 shebang = `#` (sharp) + `!` (bang) → tells the system what interpreter to use 🧠

- python script? use `#!/usr/bin/env python`

🔒 make it executable

```
chmod +x 00-shebang.sh  # gives it permission to run
./00-shebang.sh         # runs the script
```

🖨️ echo: print like a pro

```
#!/bin/bash
echo "Hello, World!"
echo "Line1\nLine2\tTabbed"
echo -e "Line1\nLine2\tTabbed"  # -e = enable escape sequences
```

🌈 **printing in color** (yes we're ✨ fabulous✨ even in terminal)

```
echo -e "\e[32mSuccess: Task completed!\e[0m"  # green
echo -e "\e[31mError: Something went wrong!\e[0m"  # red
```

📁 **print to file**

```
echo "Starting script..." >> debug.log
```

📝 **echo & spaces**

```
echo word1 word2
# → word1 word2

echo "word1 word2"
# → word1 word2
```

🌸 bash variables 101 🌸

🍓 basic rules

✨ no type casting, no need for declarations! variables are just character strings 💅

```
name="Bhargav"    # ✅ correct
#name = "Alice"    # ❌ incorrect: no spaces allowed
age=12             # ✅ still correct
```

🧠 **variables are case sensitive**

📌 must start with a letter or underscore → `var1`, `_var`, `user_age`

💡 using variables

to *use* a variable → prefix it with `$`

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

👀 numbers are treated like strings unless you specifically do math with them

```
num1=1234
num2=7890
echo $num1$num2    # → 12347890
```

🫧 quotes & escaping

✨ **single quotes** `'...'` → preserve literal value of *everything* inside ✨ **double quotes** `"..."` → allow variable expansion, command substitution, escaping

```
myVariable="Hello, world\!"
echo $myVariable          # → Hello, world!
echo "$myVariable"        # → Hello, world!
echo '$myVariable'        # → $myVariable
```

😢 **escaping special characters**

```
echo "\"Hello\""      # → "Hello"
echo \$HOME           # → $HOME (literally)
```

🌷 handling spaces + curly braces

💭 when variables are next to text, use {} to separate them

```
var="abc xyz"
num="123"

echo 1 $var$num          # → abc xyz123
echo 2 "$varXX$num"      # → abc xyzXX123
echo 3 "${var}XX${num}" # → abc xyzXX123
```

🧃 command substitution = run a command & save the output

```
lsResult=$(ls)               # modern syntax
directory=`pwd`              # old-school style

echo "My files are: $lsResult in $directory"
```

📝 note: `$(...)` is preferred now!

🧁 environment variables

variables that describe your system 🌍 use `env` in terminal to see 'em all:

variable	description
$HOME	home directory
$PWD	current folder
$USER	username of current user
$PATH	directories to find commands
$BASH	bash shell path
$BASH_VERSION	bash version
$OSTYPE	your operating system
$LOGNAME	login username

```
echo $HOME
echo $PWD
echo $BASH
echo $BASH_VERSION
echo $OSTYPE
echo "User $LOGNAME is working in folder $PWD using OS $OSTYPE"
```

🧮 bash arithmetic expressions & operators 🌸

🔢 arithmetic methods

> bash math? not cute. but we got hacks 😮‍💨 💅

- `let` → inline math that *modifies* variables ✏️

  ```
  let "a = 1"
  let "a = a + 1"
  ```

- `$[]` → deprecated 🚫 (don't touch unless u want drama)
- `$(( ))` → modern + preferred 🫶 for assigning results

  ```
  num=$((5 + 3))
  ```

- **no float support natively** 😩 use `bc` instead!

  ```
  echo "scale=2; 5 / 2" | bc -l
  ```

🧸 optional:

```
declare -i num=5  # makes it integer-only 🎯
```

💖 examples

```
#!/bin/bash

# strings?? nope
num=5*2+1
echo 1 $num  # → string, not math

# ✅ let
div=0
let div=5*2+1
echo 2 $div
```

```
let "a = 5"
let "a++"
echo 3 $a

# ✅ preferred modern method
echo 4 $((5*2+1))
a=10
b=2
result=$((a ** b))
echo 5 $result

# modulus (reminder, no float!)
echo 6 $((10 % 3))  # → 1

# ❌ no float
echo 7 $((3/4))  # → 0

# ✅ float with bc ✨
echo 8 $(echo "3/4" | bc -l)  # → 0.75

# setting precision with scale:
echo 9 $(echo "scale=5; sqrt(49)" | bc -l)  # → 7.00000

# using bc vars

echo 10 $(echo "a=10; b=3; a/b" | bc -l)  # → 3.333...

# bc logic support:
echo 11 $(echo "5 > 2" | bc)  # → 1 = true

# declare -i

declare -i num
num=$((5/2+1))
echo 12 $num
num="hello"
echo 13 $num  # → 0
```

🧩 operators

> all the lil guys that do the ✨ math✨ and ✨ checks✨

➕ arithmetic

op	meaning
+	addition
-	subtraction
*	multiplication

op	meaning
/	integer division
%	modulo
**	exponentiation

```
a=10; b=3
echo $((a + b))  # 13
echo $((a ** b))  # 1000
```

🤝 comparison

op	meaning
-eq	equal to
-ne	not equal
-gt	greater than
-lt	less than
-ge	greater than or equal to
-le	less than or equal to

```
[[ $a -eq $b ]] && echo "same" || echo "diff"
```

⚙️ logical

op	meaning
&&	AND
!	NOT

```
[[ $a -gt 5 && $b -lt 30 ]] && echo "both true"
```

📝 string operators

op	meaning

op	meaning
=	equal
!=	not equal
-z	empty string
-n	not empty string

```
[[ -z "$str3" ]] && echo "empty bestie"
```

📁 file test ops

op	checks for...
-e	existence
-f	regular file
-d	directory
-r	readable
-w	writable
-x	executable

```
touch file.txt
[[ -f file.txt ]] && echo "yup it’s a file"
```

🧃 assignment ops

op	meaning
=	assign
+=	add & assign
-=	subtract & assign
*=	multiply & assign
/=	divide & assign
%=	modulo & assign

```
x=5
((x+=2))  # x = 7
```

🔌 bitwise ops

op	meaning
&	AND
|	OR
^	XOR
~	NOT
<<	left shift
>>	right shift

```
a=5; b=3
# a = 0101, b = 0011

echo $((a & b))  # 0001 → 1
```

🤔 Bash Conditionals & Loops (with ADHD traps + sass)

🧠 Conditionals aka: "Should I do the thing?" 🤨

📦 Basic Syntax:

```
if [[ condition ]]; then
  # do the thing if true
fi

if [[ condition ]]; then
  # true
else
  # false
fi

if [[ cond1 ]]; then
  # cond1 true
elif [[ cond2 ]]; then
  # cond2 true
else
  # neither true, panic maybe?
fi
```

⚠️ ADHD TRAPS™:

- `[[ ]]` is Bash bestie! Not `[ ]`, not `(( ))` unless you're doing math.
- You **can** put spaces inside `[[ ... ]]`, but don't put math in there.
- Use `(( ... ))` for **math comparisons**, not strings.
- Don't forget `then` or you'll cry in the terminal 😭

💖 Examples:

```
x=10

# basic if
if (( x > 5 )); then
  echo "x is big and strong 💪 "
fi

# if-else
if (( x % 2 == 0 )); then
  echo "x is even ✨ "
else
  echo "x is odd 👻 "
fi

# if-elif-else
if (( x > 15 )); then
  echo "whoa too big 😳 "
elif (( x > 5 )); then
  echo "nice and medium sized 👍 "
else
  echo "tiny baby x 🐣 "
fi

# string comparison
name="Arun"
if [[ "$name" == "Sunita" ]]; then
  echo "Hello, Sunita! ☀"
else
  echo "Hello, Stranger! 👽 "
fi

# command-based condition
if ls no_directory &>/dev/null; then
  echo "Folder exists! 🎉 "
else
  echo "Nope. Folder's a ghost 👻 "
fi
```

🔁 Loops: Repeat Until You Forget What You're Doing ✨

🌀 for loop (aka the classic loop)

```
for item in one two three; do
  echo "$item!"
done
```

🔢 Range loop

```
for i in {1..5}; do
  echo "Number: $i"
done
```

📂 Directory checker (great lab exam bait)

```
for dir in docs code memes; do
  if [[ -d $dir ]]; then
    echo "$dir exists ✅"
  else
    mkdir "$dir"
    echo "$dir created 🛠"
  fi
done
```

🔄 Loop over command output (use quotes!)

```
for f in $(ls); do
  echo "Found: $f"
done
```

🚨 break + continue

```
for i in {1..5}; do
  if (( i == 3 )); then
    echo "Stopping at 3 😵"
    break
  fi
  echo "i = $i"
done

for i in {1..5}; do
  if (( i == 3 )); then
    echo "Skipping 3 🙈"
    continue
  fi
  echo "i = $i"
done
```

⏳ while loop (do it while it's true!)

```
count=1
while (( count <= 5 )); do
  echo "Count: $count"
  ((count++))
done
```

🧹 string trimming while loop

```
word="HELLO"
while [[ $word != "" ]]; do
  echo "$word"
  word=${word:1}  # chop 1st char like a fruit ninja 🍉
done
```

🚫 until loop (opposite of while – do until it's true)

```
count=1
until (( count > 5 )); do
  echo "Counting: $count"
  ((count++))
done
```

🧠 Pro Tips for Lab Exams:

- Don't forget `do ... done` in loops or you'll feel deep pain 💀
- Use `[[ ]]` for strings, `(( ))` for numbers
- Loop over `$(command)` for fun + profit
- Have folders ready to test `[ -d folder ]` stuff
- `break` = emergency exit 🏃, `continue` = skip this round

🧮 bash arithmetic expressions & operators 🌸

🔢 arithmetic methods

> bash math? not cute. but we got hacks 😮‍💨 💅

- `let` → inline math that *modifies* variables ✏️

  ```
  let "a = 1"
  let "a = a + 1"
  ```

- `$[]` → deprecated 🚫 (don't touch unless u want drama)
- `$(( ))` → modern + preferred 🫶 for assigning results

  ```
  num=$((5 + 3))
  ```

- **no float support natively** 😩 use `bc` instead!

  ```
  echo "scale=2; 5 / 2" | bc -l
  ```

🧸 optional:

```
declare -i num=5  # makes it integer-only 🎯
```

💖 examples

```
#!/bin/bash

# strings?? nope
num=5*2+1
echo 1 $num  # → string, not math

# ✅ let
div=0
let div=5*2+1
echo 2 $div

let "a = 5"
let "a++"
echo 3 $a

# ✅ preferred modern method
echo 4 $((5*2+1))
a=10
b=2
result=$((a ** b))
echo 5 $result
```

```
# modulus (reminder, no float!)
echo 6 $((10 % 3))  # → 1

# ❌ no float
echo 7 $((3/4))  # → 0

# ✅ float with bc ✨
echo 8 $(echo "3/4" | bc -l)  # → 0.75

# setting precision with scale:
echo 9 $(echo "scale=5; sqrt(49)" | bc -l)  # → 7.00000

# using bc vars

echo 10 $(echo "a=10; b=3; a/b" | bc -l)  # → 3.333...

# bc logic support:
echo 11 $(echo "5 > 2" | bc)  # → 1 = true

# declare -i

declare -i num
num=$((5/2+1))
echo 12 $num
num="hello"
echo 13 $num  # → 0
```

🧩 operators

> all the lil guys that do the ✨ math✨ and ✨ checks✨

➕ arithmetic

op	meaning
+	addition
-	subtraction
*	multiplication
/	integer division
%	modulo
**	exponentiation

```
a=10; b=3
echo $((a + b))  # 13
echo $((a ** b))  # 1000
```

🤝 comparison

op	meaning
-eq	equal to
-ne	not equal
-gt	greater than
-lt	less than
-ge	greater than or equal to
-le	less than or equal to

```
[[ $a -eq $b ]] && echo "same" || echo "diff"
```

⚙️ logical

op	meaning
&&	AND
!	NOT

```
[[ $a -gt 5 && $b -lt 30 ]] && echo "both true"
```

📝 string operators

op	meaning
=	equal
!=	not equal
-z	empty string
-n	not empty string

```
[[ -z "$str3" ]] && echo "empty bestie"
```

📁 file test ops

op	checks for...
-e	existence
-f	regular file
-d	directory
-r	readable
-w	writable
-x	executable

```
touch file.txt
[[ -f file.txt ]] && echo "yup it's a file"
```

🧃 assignment ops

op	meaning
=	assign
+=	add & assign
-=	subtract & assign
*=	multiply & assign
/=	divide & assign
%=	modulo & assign

```
x=5
((x+=2))  # x = 7
```

🔌 bitwise ops

op	meaning
&	AND
^	XOR
~	NOT
<<	left shift

op	meaning
>>	right shift

```
a=5; b=3
# a = 0101, b = 0011

echo $((a & b))  # 0001 → 1
```

🌼 see also: echo basics & variables for even more bashy bash things ✨

🧮 bash arithmetic expressions & operators 🌸

🔢 arithmetic methods

> bash math? not cute. but we got hacks 😮 😬 💅

- `let` → inline math that *modifies* variables ✏️

  ```
  let "a = 1"
  let "a = a + 1"
  ```

- `$[]` → deprecated ❌ (don't touch unless u want drama)
- `$(( ))` → modern + preferred 🧦 for assigning results

  ```
  num=$((5 + 3))
  ```

- **no float support natively** 😩 use `bc` instead!

  ```
  echo "scale=2; 5 / 2" | bc -l
  ```

🤓 optional:

```
declare -i num=5  # makes it integer-only 🌟
```

💖 examples

```bash
#!/bin/bash

# strings?? nope
num=5*2+1
echo 1 $num  # → string, not math

# ✅ let
div=0
let div=5*2+1
echo 2 $div

let "a = 5"
let "a++"
echo 3 $a

# ✅ preferred modern method
echo 4 $((5*2+1))
a=10
b=2
result=$((a ** b))
echo 5 $result

# modulus (reminder, no float!)
echo 6 $((10 % 3))  # → 1

# ❌ no float
echo 7 $((3/4))  # → 0

# ✅ float with bc ✨
echo 8 $(echo "3/4" | bc -l)  # → 0.75

# setting precision with scale:
echo 9 $(echo "scale=5; sqrt(49)" | bc -l)  # → 7.00000

# using bc vars

echo 10 $(echo "a=10; b=3; a/b" | bc -l)  # → 3.333...

# bc logic support:
echo 11 $(echo "5 > 2" | bc)  # → 1 = true

# declare -i

declare -i num
num=$((5/2+1))
echo 12 $num
num="hello"
echo 13 $num  # → 0
```

🥉 operators

> all the lil guys that do the ✨ math✨ and ✨ checks✨

➕ arithmetic

op	meaning
+	addition
-	subtraction
*	multiplication
/	integer division
%	modulo
**	exponentiation

```
a=10; b=3
echo $((a + b))  # 13
echo $((a ** b))  # 1000
```

🤝 comparison

op	meaning
-eq	equal to
-ne	not equal
-gt	greater than
-lt	less than
-ge	greater than or equal to
-le	less than or equal to

```
[[ $a -eq $b ]] && echo "same" || echo "diff"
```

⚙️ logical

op	meaning
&&	AND
!	NOT

```
[[ $a -gt 5 && $b -lt 30 ]] && echo "both true"
```

📝 string operators

op	meaning
=	equal
!=	not equal
-z	empty string
-n	not empty string

```
[[ -z "$str3" ]] && echo "empty bestie"
```

📁 file test ops

op	checks for...
-e	existence
-f	regular file
-d	directory
-r	readable
-w	writable
-x	executable

```
touch file.txt
[[ -f file.txt ]] && echo "yup it’s a file"
```

🧑 assignment ops

op	meaning
=	assign
+=	add & assign
-=	subtract & assign
*=	multiply & assign

op	meaning
/=	divide & assign
%=	modulo & assign

```
x=5
((x+=2))  # x = 7
```

🔌 bitwise ops

op	meaning
&	AND
^	XOR
~	NOT
<<	left shift
>>	right shift

```
a=5; b=3
# a = 0101, b = 0011

echo $((a & b))  # 0001 → 1
```

🌼 see also: echo basics & variables for even more bashy bash things ✨

🌈 **up next: conditionals, loops, & special shell variables** → Bash Control Flow

---

📄 **then: local vs global variables & file reading** → Scope & Files in Bash

🎯 special shell variables, functions, & scope in bash

🧠 special shell variables

> built-ins that spill tea on the shell's inner life ☕

var	meaning
`$0`	name of script

var	meaning
`$1`, `$2`, ...	positional args
`$#`	number of args
`$@`	all args (split)
`$*`	all args (single string)
`$?`	exit status of last command
`$$`	PID of current shell
`$PPID`	parent PID
`$_`	last arg of last command

```bash
#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Total args: $#"
echo "All args as separate (�@): $@"
echo "All args as single string (�*): $*"

for arg in "$@"; do
  echo "arg: $arg"
done

ls "hello.c" 2>/dev/null
echo "Exit status: $?"
echo "PID: $$, PPID: $PPID"
echo "Last arg: $_"
echo "Test"  # updates $_
echo "Now last arg: $_"
```

🧩 functions in bash

> lil reusable command gangs 🛠️

```bash
#!/bin/bash
greet() {
  echo "Hello, $1!"
}

current_date() {
  echo "Today's date is $(date +"%Y-%m-%d")"
}

check_file() {
  if [[ -f "$1" ]]; then
    echo "File '$1' exists."
    return 0
  else
    echo "File '$1' does not exist."
    return 1
  fi
}

countdown() {
  local i=$1
  while [[ $i -ge 0 ]]; do
    echo "Countdown: $i"
    ((i--))
  done
}

factorial() {
  if [[ $1 -le 1 ]]; then
    echo 1
  else
    local temp=$(( $1 - 1 ))
    local result=$(factorial $temp)
    echo $(( $1 * result ))
  fi
}

# 🧪 function calls
greet "Alice"
echo $(current_date)

if check_file "/etc/passwd"; then
  echo "File check passed!"
else
  echo "File check failed!"
fi

countdown 3
fact=$(factorial 5)
echo "Factorial of 5 is: $fact"
```

🔐 local vs global variables

> avoid stepping on your own toes 🙃

```
#!/bin/bash
message="global message"

modify_message() {
  local message="local message"
  echo "Inside: $message"
}

echo "Before: $message"
modify_message
echo "After: $message"

# exporting vars
export X="exported var"
Y="non-exported"
echo "X: $X, Y: $Y"
./child-script.sh
```

📖 file reading techniques

```
# method 1: command substitution
content=$(<file.txt)
echo "$content"

# method 2: read line-by-line
if [[ -f file.txt ]]; then
  i=1
  while read -r line; do
    echo "Line $i: $line"
    ((i++))
  done < file.txt
fi

# method 3: filename as arg
if [[ $# -ne 1 ]]; then
  echo "Usage: $0 <filename>"; exit 1
fi
file="$1"
if [[ ! -f "$file" ]]; then
  echo "Not found: $file"; exit 1
fi
while read -r line; do
  echo "Line: $line"
done < "$file"
```

📄 **up next: file writing & here-documents** → File Output in Bash

🍇 bash arrays (indexed + associative)

📦 indexed arrays

> lil containers of values, indexed by numbers starting from 0 😌

```bash
# declare + assign
fruits=("Apple" "Banana" "Cherry")

# add one more
fruits[3]="Orange"

# access
echo ${fruits[0]}     # → Apple

# all values
echo ${fruits[@]}     # → Apple Banana Cherry Orange

# count
echo ${#fruits[@]}     # → 4

# loop
for fruit in "${fruits[@]}"; do
  echo $fruit
done

# delete Banana
unset fruits[1]
echo ${fruits[@]}  # → Apple Cherry Orange
```

🧠 associative arrays (aka bash dictionaries)

> key-value style! need bash 4+ 🧷

```bash
# declare first
declare -A capitals

# assign
capitals["France"]="Paris"
capitals["Japan"]="Tokyo"
capitals["USA"]="Washington D.C."
capitals["India"]="New Delhi"

# access
echo ${capitals["India"]}  # → New Delhi

# all keys
echo ${!capitals[@]}    # → France Japan USA India

# all values
echo ${capitals[@]}     # → Paris Tokyo Washington D.C. New Delhi

# loop
for country in "${!capitals[@]}"; do
  echo "$country - ${capitals[$country]}"
```

```
done

# delete a key
unset capitals["USA"]
echo ${!capitals[@]}  # → France Japan India
```

🌟 indexed arrays are great for ordered data, associative arrays are perfect for mappings!

🧠 Bash Goofs, Tricks & Callouts

🙃 Common Bash Goofs

1. Forgetting Quotes

```
var="hello world"
echo $var   # 🚨 word splitting! → hello world (2 args)
echo "$var" # ✅ "hello world" (1 arg)
```

2. `[` vs `[[` confusion

```
[ $a == $b ]     # risky if vars are empty
[[ $a == $b ]]   # safer and preferred
```

3. `==` vs `=` in `[[` and `[`

```
[[ $a == $b ]]  # ✅ in bash
[ $a = $b ]     # ✅ in POSIX
```

4. `exit` in sourced script kills your shell 😵

```
. ./myscript.sh  # don't exit in here unless u mean it
```

5. Using `=` in arithmetic 🤦

```
a=5; b=2
echo $((a=b+1))  # assignment, not comparison! use == or -eq
```

🛠️ Bash Tips & Tricks

🔁 Loop over lines safely

```
while IFS= read -r line; do
  echo "$line"
done < file.txt
```

🧼 Safer scripts with `set`

```
set -euo pipefail  # stop on error, unset vars, pipe fails
```

🧾 Print arrays with indices

```
arr=(apple banana cherry)
for i in "${!arr[@]}"; do
  echo "$i: ${arr[$i]}"
done
```

🧪 Check if command exists

```
command -v curl >/dev/null && echo "curl found!"
```

⏳ Timing something

```
time curl -s https://example.com >/dev/null
```

💾 Save stdout + stderr to a file

```
./script.sh > out.log 2>&1
```

🔁 One-liner loop

```
for f in *.txt; do echo "File: $f"; done
```

🧮 Inline math eval

```
echo $((2+3))  # → 5
```

🔒 Quoting ALL the things

```
"$var"
"${arr[@]}"
"$(command)"
```

🧙 Callouts (Bash Style)

- 🧊 Always quote your vars unless you *need* word splitting.
- 🐛 Debug with `set -x` (turn off with `set +x`).
- 🔍 Use `[[` instead of `[` for advanced conditionals.
- 📛 Name your functions carefully – no clashes with commands!
- 🛑 Never `exit` from a sourced script unless you're 💯 sure.
- 🤔 Prefer `$(...)` over backticks `` `...` `` for subshells.
- 💥 Use `trap` to clean up temp files or catch signals:

```
trap 'echo bye!' EXIT
```

🌸 Bash is weird. Bash is wild. Test often, quote everything, and may your loops never be infinite 💫