Shell Scripting

📜 What is a Shell?

A shell is a program that interprets user commands and sends them to the operating system to execute. Common shell types:

- `bash` (Bourne Again SHell)
- `sh` (Bourne Shell)
- `zsh`, `ksh`, `csh`, etc.

🔍 What is Shell Scripting?

Shell scripting involves writing a series of commands in a script file to automate tasks in Unix/Linux environments (using shells like `bash`, `sh`, etc.).

🚀 Why We Learn Shell Scripting:

1. **🛠️ Automation of Repetitive Tasks**
    - Tasks like backups, software installation, log rotation, file management, and more can be automated.
    - Saves time and reduces human error.

2. 📈 **Increases Productivity**

    - Write once, run forever. A well-written script handles tasks without needing constant user input.

3. 💡 **Enhances System Admin Skills**

    - Most server environments run on Linux. Shell scripting is crucial for managing these systems efficiently.

4. 🔧 **Essential for DevOps**

    - Shell scripts are widely used in CI/CD pipelines, deployments, configuration, and server monitoring.

5. 📊 **Data Processing**

    - You can process large text files, logs, or command outputs with tools like `awk`, `sed`, `grep`, etc., inside a script.

6. 🧠 **Understanding the Operating System**

    - Writing shell scripts teaches you how the OS works—file systems, processes, permissions, etc.

7. 💼 **Highly Valued in Jobs**

    - Many roles in IT list shell scripting as a required or desired skill. It's considered a fundamental tech skill.

8. 🔐 **Security and Pentesting**

    - In cybersecurity, shell scripts are used to scan, audit, or automate exploits and log analysis.

9. 🧪 **Rapid Prototyping**
    - You can quickly test an idea or setup using a shell script without building full software.
10. 📡 **Networking Tasks**

- Automating IP checks, port scans, service availability checks, etc., is common in networking jobs.

💬 **Example Use Cases:**

- A daily backup script for your important files.
- A cron job that monitors disk usage and sends an email alert.
- A setup script that installs your favorite tools on a new Linux system.

✅ **Summary:**

> Shell scripting makes you more efficient, effective, and valuable in IT. It turns you from a **computer user** into a **power user or admin**.

What is Shebang?

-> It is indicates that **#!/bin/bash .**

It tells the system that we are using shell scripting languages here.

If we don't use that script will be run but using this will be standardization.

Print Hello world?

> create a folder and go in this folder and create files where you have to print, shell script , like-

#!/bin/bash

echo “Hello world”

** must give the **rwx** permission to this file . to check permission using this command : ls -ltr

*** If we do not have access to this file or permission, then we will

Give this command : bash (file name)

*** after that or basically we use to execute that : **./(filename)**

**** or give the full name path : `$ /home/paul/myscripts/01_basic.sh`

Comments

Single line comment : #
Multiline comment : <<comment
Comment

Variable in Shell Scripting

Variable use for store data and a container that holds a value, such as a string, number, or file path.

- **Types of Variables**

1. **User-defined Variables** – You create these (like `name` and `age` above).
2. **Environment Variables or System Variable** – Provided by the system (like `$HOME`, `$PATH`, `$USER`).
3. **Special Variables** – Used in scripts for things like arguments (`$1`, `$2`, `$@`, `$?`, etc.).
4.

Example :

a=hello

Do not put any gape in here

echo “$a world”

variable
name = Rahim
Age = 30

gender = male

echo “My name is $name and my age is #age and also my gender is $gender”.

*** **If we want to declare constant variable :**

Just type **readonly, it will not allow this variable once again in the entire code : readonly college**

Use of Bracket

*** We will use double parentheses `(( ))` in the loop whenever the expression is arithmetic.

*** **Always use `[[ ... ]]` for string comparison in Bash** — it’s safer, more powerful, and easier to write.

*** `#` inside `${...}` — Used for **string operations**, like getting the **length** of a string.

*** `$` — Accesses the **value** of a variable

🧩 Bash Scripting Symbol Cheat Sheet

Symbol	Name	Usage Example	Meaning / Use Case
`[ ]`	**Test (POSIX-style)**	`[ "$a" = "b" ]`	String or number test (portable) – always quote variables
`[[ ]]`	**Test (Bash-style)**	`[[ $a == b ]]`	Safer Bash-only string test, supports patterns/regex
`(( ))`	**Arithmetic evaluation**	`(( a > b ))`	Used for math comparisons or operations
`{ }`	**Command group / block**	`{ echo A; echo B; }`	Groups multiple commands – output redirected as a unit

		`> file.txt`	
`{ 1 . . 5 }`	**Brace expansion**	`for i in {1..5}; do echo $i; done`	Sequence generator (like a loop counter)
`$ { v a r }`	**Parameter expansion**	`echo "Hello, ${name}!"`	Safely access or manipulate variable content
`$ ( )`	**Command substitution**	`files=$(ls)`	Runs a command, stores its output
`\`	**Escape character**	`echo \"quoted\"`	Escapes special characters

`>` , `>` `>`	**Redirection**	`echo "Hi" > file` or `>>` for append	Redirects output to a file
`` ` ``	`` ` ``	**Pipe**	`ls
`&` , `&` `&`	**Background / AND**	`command1 & command1 && command2`	Run in background / run second only if first succeeds
`` ` ``		`` ` ``	**OR operator**

Array

An array is a data structure used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare:

myArray=(1 2 3 4.5 “Hello World”)

echo ${myArray[4]}

Length of array

The length of array is the basically, find out the array , like how many arrays value are using here:

myArray : (1 2 3 4 5)

echo ${#myArray[*]}

- This mark indicates the full length of the array

*****

In Bash, `${mydata[name1]}` retrieves the value stored in the associative array `mydata` under the key `name1`, while `${#mydata[name1]}` returns the length of that value (number of characters). The `declare -A` command is required to define an associative array, and values must be assigned individually (not as a single initialization block). Understanding these differences

helps avoid syntax errors and ensures correct data manipulation in Bash scripting.

****

Now if we want to find out a particular value's length from the array. Do not need use the# mark here.

echo ${myArray[*]:1:2}

How to update Array values in here :

myArray+=(20 30 40)

Just type it with plus marks that will indicate your array is updated.

Associative Arrays (Key-Value Pairs)

Associative arrays, also known as key-value pairs, are data structures that store data using unique keys instead of numeric indices. They allow for efficient data retrieval and manipulation by referencing values through their corresponding keys.

Key Features:

1. Key-Value Storage: Each element consists of a unique key mapped to a specific value.

2. Fast Lookups: Values can be retrieved efficiently using their keys.

3. Dynamic Size: They can grow or shrink dynamically as elements are added or removed.

4. Unordered Nature: In some languages, associative arrays do not maintain the order of insertion.

Example :

```
declare -A myArray

myArray=(

[name]=Rahim

[age]=27

[Country]=Malaysia

)

echo my name is ${myArray[name]}

echo my age is ${myArray[age]}

echo i lived in ${myArray[Country]}
```

**** In the simplest terms, the `-A` flag in Bash tells the script that the variable `myArray` is an associative array.

Without the `-A` flag, Bash assumes the variable is a regular indexed array (using numbers as keys). The `-A` flag allows you to use words or strings as keys instead of just numbers. You are telling Bash: "Hey, `myArray` will store data as key-value pairs (like `name: John`, `age: 30`, etc.) instead of just using numbers to access items.

String

It seems like you might have meant "string," which refers to a sequence of characters used in programming, including in shell scripts.

A string is a data type used to represent text rather than numbers. It is a sequence of characters, which can include letters, numbers, symbols, and spaces. Strings are used to store and manipulate text data.

In shell scripting, a string can be any sequence of characters enclosed in either single (`'`) or double (`"`) quotes.

String Length

```
my_string="Hello, World!"

length=${#my_string}

echo "Length of string: $length"
```

`Substring Extraction`

In shell scripting, **substring extraction** refers to extracting a portion of a string, starting from a specified position and optionally with a specified length.

>> ${string:start:length}

- **`string`**: The original string.
- **`start`**: The position where you want to start extracting (zero-based index).
- **`length`**: The number of characters to extract (optional).

You can extract substrings using the substring notation

${string:start:length}:

my_string="Hello, World!"

substring=${my_string:7:5}

echo $substring # Output will be "World"

If we want to execute particular number from the variable of string :

myVarValue= “Hello this the new world”

result=${myVarValue:1:7}

>> here only one character will be executed if we want multiple just add 2 or 3 , whatever number we want just type there , it will show to you the multiple numbers.

uppercase and lowercase

UpLower="Hello good people"

echo ${UpLower,,}

echo ${UpLower^^}

how to replace any variable's data

replace=${UpLower/Hello/HI}

echo ${replace}

User Interaction

This process command will ask for data from users

 # when we add read it will automatic ask for your input
#method -1

read name
echo your name is $name

#method-2
echo what is your name? -

read age
echo your age is ${age}

#method -3

read -p "tell me about yourself" details
echo thanks for your information and check it again ${details}

Projects :

this the simple projects that takes data from users

```
read -p "please type HI for enter the panel" panel
echo your input accpted ${panel}

read -s -p "Enter your password" password
echo ${password}

read -t 3 -p "wait a minute your process will start..." time
echo ${time}

read -p "please enter your name" name
echo your name is ${name}

read -p "Please enter you class id" ID
echo your class ID is ${ID}


read -p "Please enter you section" section
echo your section is ${section}

read -p "check it again. if it ok then type ok or No" answer
echo your result is ${answer}
```

Conditional

Conditional is actually created in the bash script. It executed the language with conditions.

Example:

```
if [ condition ]; then
   # Commands to execute if condition is true
fi
```

```
num=10

if [ $num -gt 5 ]; then
    echo "Number is greater than 5"
fi
```

`if-else` Statement

```
if [ condition ]; then
    # Commands if condition is true
else
    # Commands if condition is false
fi
```

```
num=3

if [ $num -gt 5 ]; then
    echo "Number is greater than 5"
else
    echo "Number is 5 or less"
fi
```

`if-elif-else` Statement

```
if [ condition1 ]; then
```

```
    # Commands if condition1 is true
elif [ condition2 ]; then
    # Commands if condition is true
else
    # Commands if none of the conditions are true
fi
```

—

```
num=5

if [ $num -gt 10 ]; then
    echo "Number is greater than 10"
elif [ $num -eq 5 ]; then
    echo "Number is exactly 5"
else
    echo "Number is less than 5"
fi
```

Comparison Operators in Bash:

Operator	Meaning	Example
`-eq`	Equal to	`[ 5 -eq 5 ]` (true)
`-ne`	Not equal to	`[ 5 -ne 3 ]` (true)
`-gt`	Greater than	`[ 10 -gt 5 ]` (true)
`-lt`	Less than	`[ 3 -lt 5 ]` (true)
`-ge`	Greater than or equal	`[ 10 -ge 10 ]` (true)
`-le`	Less than or equal	`[ 5 -le 10 ]` (true)

*** in logical operator use in Latter and -ge, -le will use in numeric.

like , [[10 -ge 12]] , [[rahim == karim]]

- **AND (`&&` or `-a`)**
- **OR (`||` or `-o`)**
- **NOT (`!`)**

Other String Comparison Operators in Bash

Operator	Description	Example
`-z`	True if the string is empty	`[[ -z "$var" ]]`
`-n`	True if the string is **not** empty	`[[ -n "$var" ]]`
`=`	True if two strings are **equal**	`[[ "$str1" = "$str2" ]]`
`!=`	True if two strings are **not equal**	`[[ "$str1" != "$str2" ]]`
`<`	True if `str1` is **lexicographically less than** `str2`	`[[ "apple" < "banana" ]]`
`>`	True if `str1` is **lexicographically greater than** `str2`	`[[ "banana" > "apple" ]]`

`exit 1` in Bash

The `exit` command in Bash **terminates the script** immediately. The number after `exit` is the **exit status code**, which tells the system (or the user) whether the script ran successfully or encountered an error.

Case Command

In shell scripting, a **case** statement is used for pattern matching and conditional execution, similar to a switch-case in other programming languages. It helps to execute different commands based on user input or variable values.

Explanation:

1. The script prompts the user to enter a choice.
2. The `case` statement checks the input against predefined options.
3. If a match is found, the corresponding commands are executed.
4. If no match is found (`*` wildcard), it prints an error message.

Loop in Bash

A **loop** in programming is a control structure that allows you to repeat a block of code multiple times based on a condition. In Bash, common loops include the `for` loop, `while` loop, and `until` loop. A **`for` loop** is often used when you know how many times you want to iterate. In this case, you can use it to iterate through elements of an array or a sequence of numbers. For example, in the expression `m % 2 == 0`, a `for` loop can be used to check each number in a range to determine whether it is even or odd. The modulo operator `%` helps identify if a number is divisible by another, and the condition `m % 2 == 0` checks if a number is even. If the condition holds true, the code inside the loop executes, such as printing the even number. Similarly, the condition `m % 2 != 0` is used to filter out odd

numbers. Loops provide an efficient way to process data multiple times, making them essential in many programming tasks.

To check if a number is even in Bash, we use the modulo operator `%`, which calculates the remainder of a division. The expression `m % 2 == 0` checks whether the remainder when dividing `m` by 2 is 0, indicating that the number is even. If the condition is true, it means `m` is divisible by 2 without any remainder, and we can execute actions like printing the value. This approach is useful for filtering even numbers in loops or arrays. Similarly, for odd numbers, we use `m % 2 != 0` to ensure the remainder is not 0, indicating the number is not divisible by 2.

Example ;

```
#Loops in Bash scripts allow you to automate repetitive tasks efficiently.

#method 1
for i in 1 2 3 4 5 6
do
echo "The numbers are $i"
done

#method-2

for j in {1..10} # (..) two dots you have to give here becuase more than it i wll not excute to 10
do
echo "the number $j "
done

for k in $(seq 1 5); do #Seq 1 5 is a command that generates a sequence of numbers from 1 to 5.
   echo "Count: $k"
done
```

```
#method-3

fileName="/fileName/file.docx"  #this method executed the file's mention
data
for name in $(cat $fileName)
do
 echo "we can read $name"
done


#mthod - 4 loop array
myArrayValue=({1..10})
lenghtOF=${#myArrayValue[*]}
for (( m=1;m<lenghtOF;m++))
do
echo "the value should be $m"
done


#method -5(print odd numbers with loops)

arrayOdd=({1..10})
lenghtOFoDD=${#arrayOdd[*]}
for(( m=1;m<lenghtOFoDD;m++))
do

if (( m%2 !=0 ));then

   echo "the odd number is $m"
fi

done


#method -6 (print even number)
```

```
arrayEven=({1..10})
lenghtOFeVen=${#arrayEven[*]}
for(( n=0;n<lenghtOFeVen;n++ ))
do

if (( n%2 == 0 )); then
echo "the even number is $n"
fi
done
```

While loop

The `while` loop in shell scripting is a fundamental control structure used to execute a set of commands repeatedly, as long as a given condition remains true. The syntax begins with the `while` keyword followed by a condition enclosed in square brackets `[ ]`. The body of the loop is placed between the `do` and `done` keywords. Before each iteration, the condition is evaluated—if it returns true, the loop continues; otherwise, it stops. One of the most common uses of a `while` loop is reading a file line by line using the `while read line` command. This loop is particularly useful in scenarios where the number of iterations is not known in advance. To ensure efficiency and avoid infinite loops, it's important to update variables or include a clear exit condition inside the loop. Overall, the `while` loop provides great flexibility and control in automating repetitive tasks within a shell script.

Example:

```
#!/bin/bash

count=1

while [ $count -le 5 ]
```

```
do
  echo "Count is: $count"
  ((count++))
done
```

Explanation:
In this example, the loop starts with `count` equal to 1. It checks if the count is less than or equal to 5. If true, it prints the value of `count` and increments it by 1. The loop runs until the condition becomes false (i.e., `count` becomes 6).

***** when we need to execute only particular command or conditions**

While true

do

read -p “Enter the number : ” number

if [[“$number” =~ ^[admin]$+]]; then

break

echo “your input is valid”

else

echo “this is not valid”

fi

done

***** when we need to execute the data from files**

while read fileName
do
  echo “the file data” $fileName
Done < (path of file)

To read content / data from CSV file

You can read the contents of a CSV file line by line using the `while read` loop. This method is helpful when you want to process each row individually. To properly handle commas (`,`) that separate the values, you can use the `IFS` (Internal Field Separator) to split each line into fields.

```
# this process reead from CSV file
while IFS=',' read -p ID Name Age
do
   echo "The ID is: $ID"
   echo "The Name is: $Name"
   echo "The Age is: $Age"
   echo "----------------------"
done < "test - Sheet1.csv"
```

Using `tail -n +2` to skip the header:

```
tail -n +2 "file.csv" | while IFS=',' read -r ID
Name Age
do
  ...
done
```

- **`tail -n +2` starts reading from the 2nd line, skipping the header.**
- **This is useful when you don't want to process column titles like "ID,Name,Age".**
- **You are piping the output of `tail` to the `while read` loop.**
- **This structure is more readable when processing data, especially in CSV files where the first line is usually the header.**

```
#this process using for read csv file and Using
tail -n +2 to skip the header:

tail -n +2 "test - Sheet1.csv" | while IFS=',' read
-r ID Name Age
do
  echo "The ID is: $ID"
```

```
  echo "The Name is: $Name"
  echo "The Age is: $Age"
  echo "--------------------"
done
```

🔁 `while` vs `for` Loop — When to Use What

✅ `while` Loop

Use `while` **when you don't know how many times** a loop will run. It's great for:

- Waiting for a condition to become true (e.g., waiting for a server to start)
- Prompting user input (like your username/password case)
- Monitoring logs or processes continuously

Example:

```
while true; do
  echo "Waiting for service to come online..."
  sleep 5
done
```

✅ `for` Loop

Use `for` **when you know how many items or times** you want to loop through. Great for:

- Looping over files, numbers, or output from commands
- Running commands over a list of servers or configs
- Iterating over arrays

Example:

```
for server in server1 server2 server3; do
  ssh "$server" uptime
done
```

📘 Using Regex to Find Even or Odd Numbers in a Loop

Here's how you can use regex to check if a number is even or odd.

```
for (( ; ; ))
do
  read -p "Enter a number: " num
  if [[ "$num" =~ ^[0-9]+$ ]]; then
    if [[ $((num % 2)) -eq 0 ]]; then
      echo "$num is Even"
    else
      echo "$num is Odd"
    fi
    break
  else
    echo "Invalid number, try again!"
  fi
```

```
done
```

🔍 We use `[[ "$num" =~ ^[0-9]+$ ]]` to ensure it's a valid number (regex for digits only).

💼 Which Loop is More Common in DevOps?

If you're aiming for **DevOps**, here's what you'll *really* use more often:

Use Case	Loop Type	Notes
Parsing logs continuously	`while`	Keeps running, e.g. `tail -f`
Running commands on multiple servers/files	`for`	SSH, copying configs, etc.
User inputs / conditions	`while`	Wait for correct input or status
One-time iterations	`for`	Loop through a known set of things

🔧 **DevOps engineers use both**, but they lean on `for` loops more when automating tasks with known scopes (like deploying to N servers).

🧭 Complete Beginner-Friendly Guideline for Bash Loops in DevOps

Start with `for` loops – simpler to understand and super useful for scripts.

bash

```
for user in user1 user2; do
  useradd "$user"
done
```

1. **Use `while` for waiting or validating**

   bash
   CopyEdit

   ```
   while ! ping -c 1 myserver &> /dev/null; do
     echo "Waiting for myserver..."
     sleep 5
   done
   ```

2. **Practice regex** in `if [[ ... =~ regex ]]` to validate inputs like numbers, IPs, filenames, etc.

3. **Write small scripts** to:

   - Loop through a list of servers and ping them
   - Monitor a service and restart if it's down
   - Validate user input for even/odd or specific patterns

4. **Automate real-world tasks**
    - Set up cron jobs with scripts using loops
    - Backup files from multiple directories
    - Check disk usage across servers

🧠 Final Tip

You're doing great asking this question. Everyone starts somewhere, and in DevOps **your scripting game is your superpower**. Master Bash loops, conditions, and regex — and you'll feel **much more confident** handling infrastructure tasks.

Until Loops

The `until` loop in shell scripting is kind of like the opposite of a `while` loop. It keeps running **until** the condition becomes true (whereas loops run **while** the condition is true).

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

Functions

🔹 **What is a Function in Bash (or in general programming)?**

A **function** is a **named block of code** that performs a specific task. Instead of writing the same code again and again, you write it once inside a function and **call it whenever needed**.

🔹 **Why Functions are Used?**

Functions are used to:

✅ **Avoid code repetition**
✅ **Make the script more organized and readable**
✅ **Break down large tasks into smaller pieces**
✅ **Reuse code** with different inputs (parameters)
✅ **Easily manage or update logic** in one place

🔹 **Real-Life Analogy:**

Imagine you have a washing machine. Every time you need to wash clothes, you **press a button** and it does all the steps: wash, rinse, spin.

For example :

```
data_OF_Name(){

  echo —----------------------------

 echo Name $1

echo Age $2

echo Gender $3

echo location $4

echo Occupation $5

  echo —----------------------------

}

data_OF_Name "Raju" 20 "Male" "Dhaka" "Student"
data_OF_Name "Maju" 30 "Male" "Dhaka" "Student"
data_OF_Name "Saju" 40 "Male" "Dhaka" "Student"
data_OF_Name "jadu" 50 "Male" "Dhaka" "Student"
```

*** if we know that there are file have but we do not know that what is the path is and we need the path, at that time , we need to type, **realpath**, it is easily configure the path in here.

> *** $RANDOM is basically generated the code 0 - 3267, is generated randomly.

***

using UID , is basically, using the ID is root or not
#if the user is 0 which mean it is root users

*** if i need to copy the lots of file the just go to thi **dir,** and type **>, like , go to dir and type ls > myfile.txt**

How to conduct the Automatic command

Just first of all write , find the date, by using **date** command, and 2ndly, use the **at** command by using this , and now give the command what you need to execute.

We can remove it by using : atrm