Bash scripting

Computing Lab
CS 69201
Department of Computer Science and Engineering
IIT Kharagpur

Topic

- ➢ What is bash scripting?
- ➢ Variable and coding logic
- ➢ Popular command in bash script and their uses
- ➢ Compound commands

Pre-requisites

- ➢ A running version of Linux with access to the command line.
- ➢ Make sure your home computer/laptop with this setup

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.

- Bash scripts are essentially just a sequence of the same Linux commands that you would ordinarily use every day. That means that if you regularly use the Linux command line, then Bash scripts are pretty simple. Anything that exists in a Bash script can also be executed in the command line.

Why?

- **Automation:**
  - Shell scripts allow you to automate repetitive tasks and processes, saving time.
- **Portability:**
  - Shell scripts can be run on various platforms and operating systems, including Unix, Linux, macOS.
- **Flexibility:**
  - Shell scripts are highly customizable and can be easily modified to suit specific requirements.
- **Accessibility:**
  - Shell scripts are easy to write and don't require any special tools or software. They can be edited using any text editor, and most operating systems have a built-in shell interpreter.
- **Integration:**
  - Shell scripts can be integrated with other tools and applications, such as databases, web servers, and cloud services, allowing for more complex automation and system management tasks.

Scripting vs. C Programming

- Advantages of bash scripts -
  - Easy to work with other programs
  - Easy to work with files
  - Easy to do small tasks
  - Great for prototyping. No compilation

- Disadvantages of bash scripts -
  - Slower
  - Not well suited for algorithms & data structures

Important tips (before we start)

- Bash file name will be like <scriptname>.sh
- Lot of cmds, what if you forget their working or parameters?
  - Type “ man <cmd_name>” to get all the details of that cmd
- To execute the bash file -
  - chmod +x <scriptname>.sh (gives the permission to run, need to do only once)
  - ./<scriptname>.sh (for running)
- Alternatively, you can execute directly by running the following command
  - bash <scriptname>.sh

Example

Code in bash file -

```
#!/bin/bash
Age=17
if [ "$Age" -ge 18 ]; then
    echo "You can vote"
else
    echo "You cannot vote"
fi
```

Output -

```
You cannot vote
```

Hard ?

Variable - part 1 (hello world)

Code in bash file -

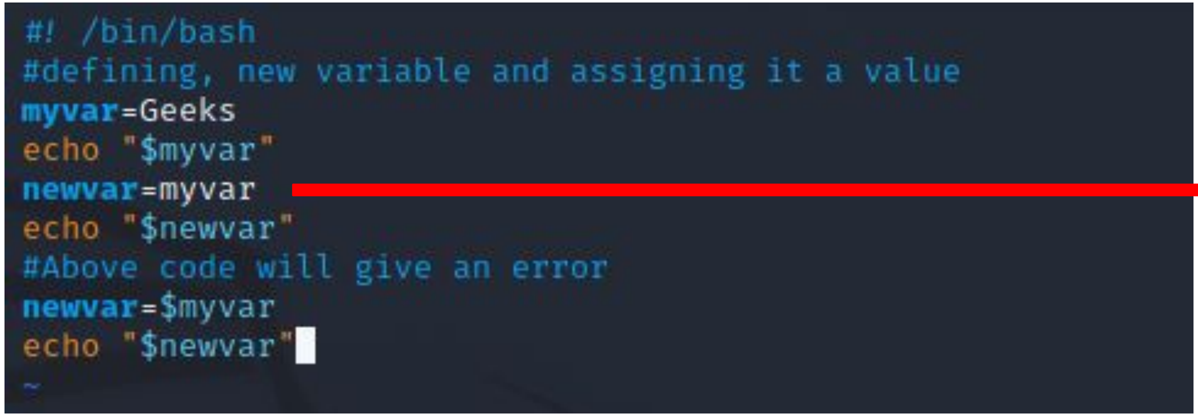
```
Name="Gourav Sarkar"

Age=22



echo "The name is $Name and Age is $Age"
```

Output -

```
The name is Gourav Sarkar and Age is 22
```

- ➢ here is have declared two variables Name and another one is Age.
- ➢ These variables are accessible using $Name and $Age.
- ➢ That means, we can declare a variable in a bash script using VariableName=Value and can access it using $VariableName.

Variable - part 2 (rules)



```
#! /bin/bash
#defining, new variable and assigning it a value
myvar=Geeks
echo "$myvar"
newvar=myvar
echo "$newvar"
#Above code will give an error
newvar=$myvar
echo "$newvar"
~
```

Error

- ➢ Variables can begin with an alphanumeric character or underscore, followed by a letter, number, or underscore.
- ➢ Variables are case sensitive, so “gfg” and “Gfg” are two different variables.
- ➢ Variables cannot start with a number.
- ➢ Do not use special characters while defining a variable name.

Variable - part 3 (local and global)

```bash
#! /bin/bash
myvar=5

function calc(){
# use keyword 'local' to define a local variable
local myvar=5
(( myvar=myvar*5 ))
# print the value of local variable
echo $myvar
}

# call the function calc
calc
# print the value of global variable and
# observe that it is unchanged.
echo $myvar
```

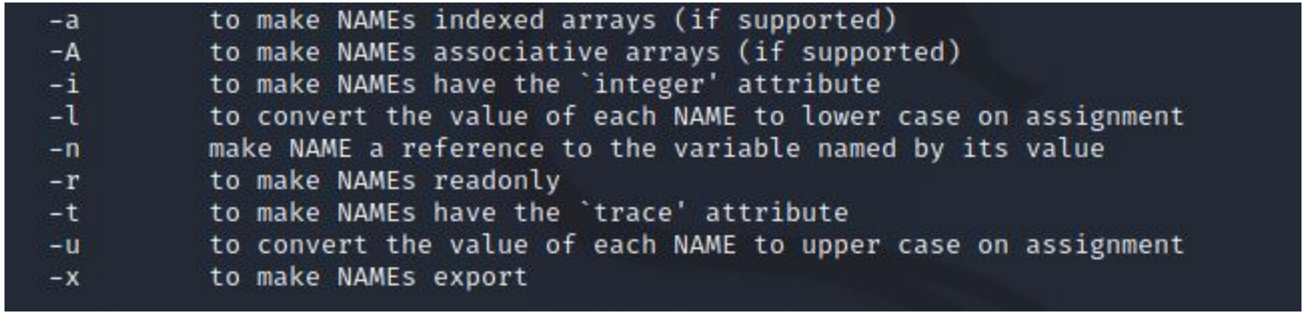
- ➢ we will define a global variable ‘myvar’ and a local variable ‘myvar’.
- ➢ Then we will try changing the value of the variable ‘myvar’ inside our function.

Output -

```
25
5
```

Variable - part 4 (declaring datatype)

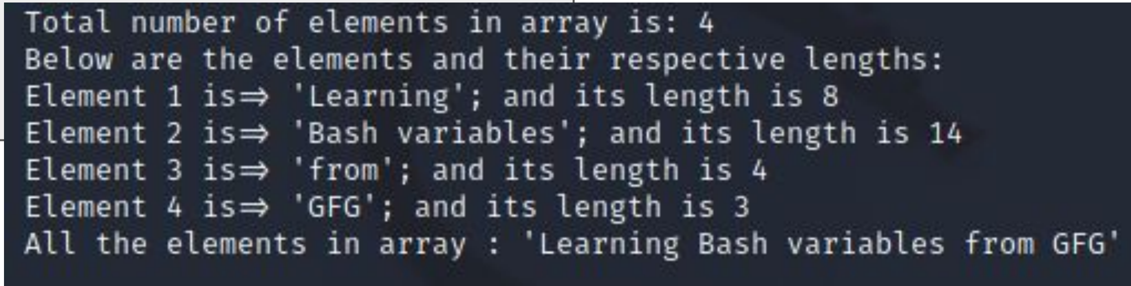
- You can declare an integer using the below command.
  - declare -i myvar
- You can declare an array using the below command.
  - declare -a myarray=([element1 element2 element3])
- You can declare an map/associative array using the below command.
  - declare -A newArray=([key1]=value1 [key2]=value2 [key3]=value3)



```
  -a        to make NAMEs indexed arrays (if supported)
  -A        to make NAMEs associative arrays (if supported)
  -i        to make NAMEs have the `integer' attribute
  -l        to convert the value of each NAME to lower case on assignment
  -n        make NAME a reference to the variable named by its value
  -r        to make NAMEs readonly
  -t        to make NAMEs have the `trace' attribute
  -u        to convert the value of each NAME to upper case on assignment
  -x        to make NAMEs export
```

Variable - part 5 (array example)

```
#! /bin/bash

# declare an array with pre-defined values
declare -a my_data=(Learning "Bash variables" from GFG);

# get length of the array
arrLength=${#my_data[@]}

# print total number of elements in the array
echo "Total number of elements in array is: $arrLength"

# iterate through the array
echo "Below are the elements and their respective lengths:"
for (( i=0; i<arrLength; i++ ));
do
    echo "Element $((i+1)) is=> '${my_data[$i]}'; and its length is ${#my_data[i]}"
done

# print the whole array at once
echo "All the elements in array : '${my_data[@]}'"
```

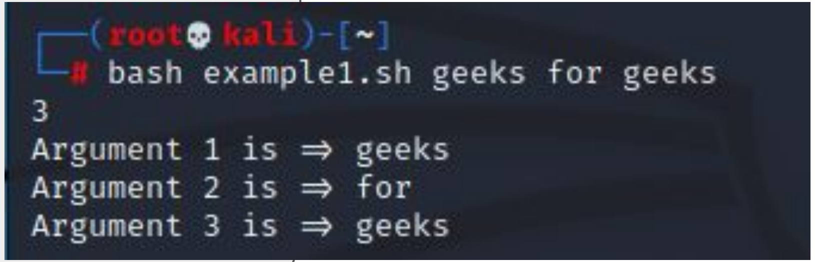


```
Total number of elements in array is: 4
Below are the elements and their respective lengths:
Element 1 is⇒ 'Learning'; and its length is 8
Element 2 is⇒ 'Bash variables'; and its length is 14
Element 3 is⇒ 'from'; and its length is 4
Element 4 is⇒ 'GFG'; and its length is 3
All the elements in array : 'Learning Bash variables from GFG'
```

Command-line arguments

bash script.sh argument1 argument2 argument3 argument4 ....

```
#! /bin/bash
# declare a variable which will store all the values of our arguments
myvar=("$@")

# Lets store the length of the number of arguments
# Here we can make use of the special variable '$#'
le="$#"
echo $le

# let us now print the arguments that user (looping)
for (( i=0; i<le; i++ ))
do
   echo "Argument $((i+1)) is => ${myvar[i]}"
done
```



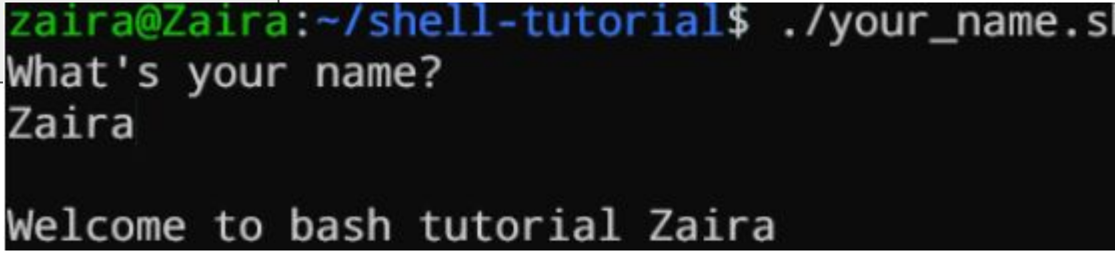
```
┌──(root💀kali)-[~]
└─# bash example1.sh geeks for geeks
3
Argument 1 is ⇒ geeks
Argument 2 is ⇒ for
Argument 3 is ⇒ geeks
```

Note : We can also take cmd argument as $1, $2 , so on like-

first_arg = $1

Input from user (terminal) and file

```
#!/bin/bash

echo "What's your name?"

read entered_name

echo -e "\nWelcome to bash tutorial" $entered_name
```



```
zaira@Zaira:~/shell-tutorial$ ./your_name.s
What's your name?
Zaira

Welcome to bash tutorial Zaira
```

```
while read line
do
  echo $line
done < input.txt
```

Output print in file

- Writing to a file.
  - echo "This is some text." > output.txt
- Appending to a file.
  - echo "More text." >> output.txt
- Redirecting output.
  - ls > files.txt

If-else

```
#!/bin/bash

echo "Please enter a number: "
read num

if [ $num -gt 0 ]; then
  echo "$num is positive"
elif [ $num -lt 0 ]; then
  echo "$num is negative"
else
  echo "$num is zero"
fi
```

Switch case

```
fruit="apple"

case $fruit in
    "apple")
        echo "This is a red fruit."
        ;;
    "banana")
        echo "This is a yellow fruit."
        ;;
    "orange")
        echo "This is an orange fruit."
        ;;
    *)
        echo "Unknown fruit."
        ;;
esac
```

- ➢ The double semicolon ";;" separates each block of code to execute for each pattern, similar to break in programming language.
- ➢ The asterisk "*" represents the default case, which executes if none of the specified patterns match the expression.

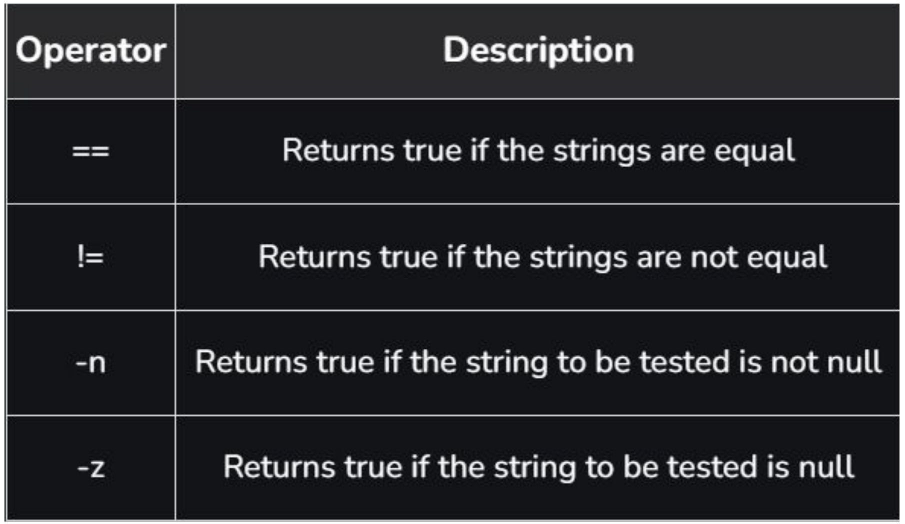
Function call

```
#!/bin/bash
#It is a function
myFunction () {
echo “Hello World”
}

#function call
myFunction
```

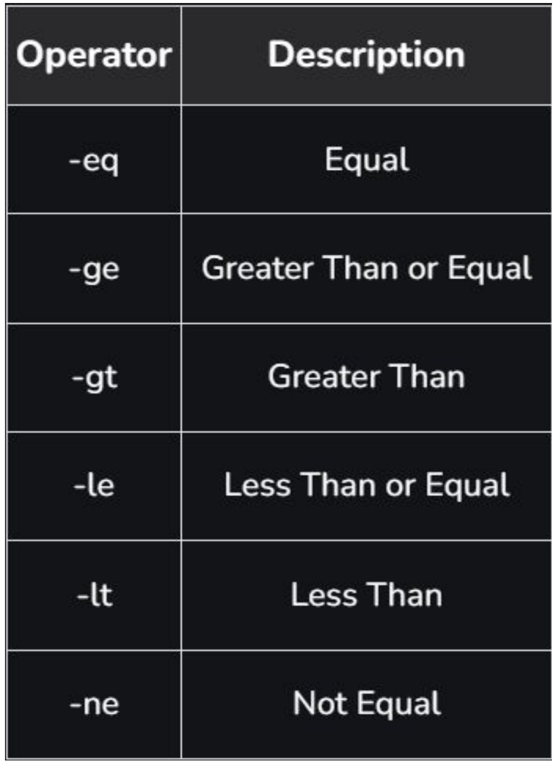
Looping

```
#!/bin/bash
# while loop
i=1
while [[ $i -le 10 ]] ; do
   echo "$i"
  (( i += 1 ))
done

# for loop
for i in {1..5}
do
    echo $i
done
```

Operator



Operator	Description
==	Returns true if the strings are equal
!=	Returns true if the strings are not equal
-n	Returns true if the string to be tested is not null
-z	Returns true if the string to be tested is null



Operator	Description
-eq	Equal
-ge	Greater Than or Equal
-gt	Greater Than
-le	Less Than or Equal
-lt	Less Than
-ne	Not Equal

Date - part 1 (datatype)

- date: Display the date and time specified by STRING.
- +%FORMAT: Display the date and time in the specified format.
- %Y: Year (e.g., 2022)
- %m: Month (e.g., 01)
- %d: Day of the month (e.g., 01)
- %H: Hour (e.g., 12)
- %M: Minute (e.g., 30)
- %S: Second (e.g., 45)

```
#!/bin/bash

current_date=$(date +"%Y-%m-%d")
echo "Current date: $current_date"
```

Date - part 2 (formatting and extracting)

```
#!/bin/bash

Current_datetime = $(date +"%Y-%m-%d %H:%M:%S")
echo "Current date and time: $current_datetime"
```

Print current date time with given format

```
#!/bin/bash

given_date="2022-01-01"
year = $(date -d "$given_date" +"%Y")
echo "Year: $year"
```

Print the year from the given date

Date - part 3 (modify and difference)

```
#!/bin/bash

given_date="2022-01-01"
days_to_add=7

new_date=$(date -d "$given_date + $days_to_add days" +"%Y-%m-%d")
echo "New date: $new_date"
```

Add days to
date

```
#!/bin/bash

start_date="2022-01-01"
end_date="2022-01-07"

date_diff=$(($(date -d "$end_date" +%s) - $(date -d "$start_date" +%s)))
days_diff=$((date_diff / 86400))
echo "Number of days between $start_date and $end_date: $days_diff"
```

Find diff betw
dates and co
into day diff

Command substitution

- myvar=\`ls\` or myvar=$(ls)
- Sometimes, while writing a shell script you will encounter some instances when you want to assign the output of a particular command to a variable. This can be done with the help of command substitution. Command substitution typically works by running a particular shell command and storing its result in a variable for further use or to display.

Important Shell commands

Important Commands part -1

- **cat:** Concatenates and displays the content of files.
- Example:
  - cat file.txt
- Displays the contents of file.txt.

- **sort:** Sorts lines of text in files.
- Example:
  - sort data.txt
- Sorts the lines in data.txt alphabetically.

Important Commands part -2

- ➢ **awk:** A powerful text processing language used for pattern scanning and processing.
- ➢ Example:
  - ○ `awk '{print $1}' file.txt`
- ➢ Prints the first column of each line in file.txt.

- ➢ **Common Awk One-Liner:**
  - ○ `awk '/error/ {print $0}' log.txt`
- ➢ Prints lines containing the word "error" from log.txt.

Important Commands part -3

- **head:** Displays the first few lines of a file.
- Example:
  - head -n 5 file.txt
- Shows the first 5 lines of file.txt.

- **tail:** Displays the last few lines of a file.
- Example:
  - tail -n 10 file.txt
- Shows the last 10 lines of file.txt.

Important Commands part -4

- **cd:** Changes the current directory.
- Example:
  - cd /home/user/Documents
- Changes to the Documents directory.

- **mkdir:** Creates a new directory.
- Example:
  - mkdir new_folder
- Creates a directory named new_folder.

- **ls:** Lists files and directories.
- Example:
  - ls -l
- Lists files and directories in long format, showing details like permissions and file sizes.

Important Commands part -5

- **touch:** Creates an empty file or updates the timestamp of an existing file.
- Example:
  - touch newfile.txt
- Creates an empty newfile.txt if it doesn't exist or updates its last modified time.

- **file:** Determines the type of a file.
- Example:
  - file example.txt
- Displays the type of example.txt, such as "ASCII text" or "image data."

Important Commands part -7

- ➢ **tr:** Translates or deletes characters.
- ➢ Example:
- ➢ bash
- ➢ Copy code
  - ○ echo "hello world" | tr 'a-z' 'A-Z'
- ➢ Converts all lowercase letters in "hello world" to uppercase, resulting in HELLO WORLD.

- ➢ **grep:** Searches for patterns in files.
- ➢ Example:
  - ○ grep "error" logfile.txt
- ➢ Searches for the word "error" in logfile.txt and displays matching lines.

Important Commands part -8

- **alias:** Creates shortcuts for commands.
- Example:
  - alias ll='ls -la'
- Defines ll as a shortcut for ls -la, which lists all files in long format.

- **dd:** Copies and converts files, often used for creating disk images.
- Example:
  - dd if=/dev/sda of=backup.img bs=1M
- Creates a disk image backup.img of the sda device with a block size of 1 megabyte.

Important Commands part -9

- **nohup:** Runs a command immune to hangups, with output redirected to a file.
- Example:
  - nohup long_running_script.sh &
- Runs long_running_script.sh in the background, even after logging out, with output saved in nohup.out.

- **wc:** Counts lines, words, and characters.
- Example:
  - wc -l file.txt
- Counts the number of lines in file.txt.

Important Commands part -10

- **split:** Splits a file into smaller pieces.
- Example:
  - `split -l 1000 largefile.txt smallfile_`
- Splits largefile.txt into files with 1000 lines each, named smallfile_aa, smallfile_ab, etc.

- **cut:** Extracts sections from each line of input.
- Example:
  - `cut -d ',' -f 1 data.csv`
- Extracts the first field from each line of data.csv, assuming fields are separated by commas.

Important Commands part -11

- **sed:** Stream editor for filtering and transforming text.
- Example:
  - `sed 's/old/new/g' file.txt`
- Replaces all occurrences of "old" with "new" in file.txt.

- **curl:** Transfers data to or from a server using various protocols.
- Example:
  - `curl -O https://example.com/file.txt`
- Downloads file.txt from https://example.com

Compound commands

Multiple Commands (Separated by Semicolon or Newline)

- cmd1; cmd2
- **Description:** You can separate commands using a semicolon (;) or by placing them on new lines. Each command is executed sequentially, regardless of the success or failure of the previous command.
- Example:
  - `echo "First command"; echo "Second command"`
- This example executes echo "First command" and then echo "Second command", regardless of whether the first command **succeeds or fails**.

Multiple boolean Commands part -1

- cmd1 && cmd2
- **Description:** cmd2 is executed only if cmd1 succeeds (returns an exit status of 0).
- Example:
    - mkdir new_directory && cd new_directory
- Here, cmd2 is executed only if cmd1 is **successful** (i.e., the directory is created without errors).

Multiple boolean Commands part -2

- cmd1 || cmd2:
- **Description:** cmd2 is executed only if cmd1 fails (returns a non-zero exit status).
- Example:
  - `cd non_existent_directory || echo "Directory does not exist"`
- In this example, if cmd1 **fails** (because the directory doesn't exist), then cmd2 will execute and the message "Directory does not exist" is printed.

Subshell

- (command1; command2) > file
- **Description:** A subshell is created when commands are enclosed in parentheses ((...)). The commands inside the parentheses run in a separate shell process. Output redirection (e.g., > file) can be applied to the entire block.
- Example:
  - `(echo "Line 1"; echo "Line 2") > output.txt`
- This command runs echo "Line 1" and echo "Line 2" in a subshell, and redirects their combined output to output.txt. The subshell ensures that any variables or changes made inside it do not affect the parent shell environment.

Background Execution

- ➢ cmd1 &
- ➢ **Description:** This runs cmd1 in the background, allowing the terminal to remain free for other commands while cmd1 executes.
- ➢ Example:
  - sleep 10 &
- ➢ This command runs sleep for 10 seconds in the background. You can continue to use the terminal while sleep finishes.

Pipelines

- cmd1 | cmd2 :
- **Description:** The output of cmd1 is used as input for cmd2. This is useful for filtering or transforming data.
- Example:
  - `cat file.txt | grep "error"`
- This example displays the content of file.txt and then filters it using grep to show only lines containing the word "error".

Redirection

- **Input Redirection:**
- Description: Reads input for file1.txt from file2.txt. It is typically used with commands that expect input from standard input.
- Example:
  - sort < unsorted.txt
- This command sorts the contents of unsorted.txt and displays the sorted output.

- **Output Redirection:**
- Description: Redirects the output of a command to file2.txt, overwriting its contents if it exists.
- Example:
  - ls > files_list.txt
- This example saves the list of files and directories in the current directory into files_list.txt.

Thank You