Session 6

Shell Scripting using Bash

Agenda

- What is a shell scripting
- How to write & execute a script
- Variables
- Take User Input
- Arithmetic Expansion
- Conditions
- Case Statements
- Shell Loops
- Shell Functions

Terminal VS Shell VS Kernel

Terminal:

A program which is responsible for providing an interface to a user so that he/she can access the shell.

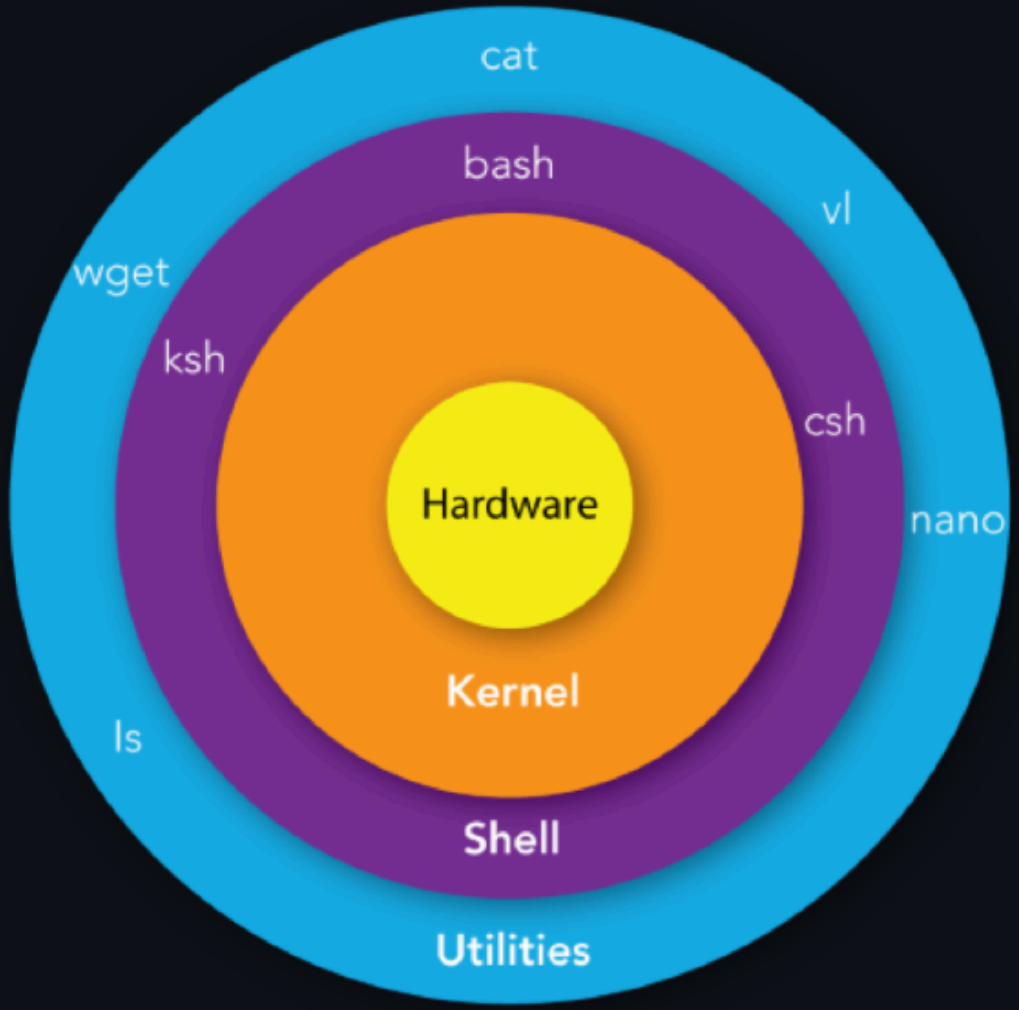
Shell:

A shell is a **special user program** that provides an interface for the user to use operating system services. Shell accepts human-readable commands from users and **converts** them into something which the kernel can understand. It is a **command language interpreter** that executes commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or starts the terminal.

Kernel:

The kernel is a computer program that is the core of a computer’s operating system, with complete control over everything in the system.

Shell VS Kernel



Bash:

Is one of the most popular shells and is the default on many Linux distributions.

Script:

A shell script is a text file containing a series of commands that can be executed by the shell.

Different Types of Shells in Linux : sh, zsh, csh, ksh and bash.

How to write & execute a script ?

Writing a Shell Script

- Create a Script File: Use a text editor (like `nano` or `vim`) to write your script. The file usually has a `.sh` extension, but it's not mandatory.

The Shebang

- The `#!/path/to/shell` at the first line of a script is called a Shebang and is used to tell the operating system what shell it should use to execute the commands.
- By typing this line you do not need to specify the shell at the command line before executing the file.
- This should be the first line to write in a script
- **Ignoring Shebang :**
  If you do not specify an interpreter line, the default is usually the `/bin/sh`. But, it is recommended that you set `#!/bin/bash` line for your bash scripts.

Executing a Shell Script

- **First Way:**

  *1. **Make the Script Executable:*** Before you can run the script, you need to give it execute permissions using the `chmod` command.

  ```
  chmod +x pathToTheFile/my_script.sh
  ```

  *2. **Run the Script:*** You can execute the script by specifying its path.

  ```
  pathToTheFile/my_script.sh
  ```

- **Second Way:**

  You can run it by calling the shell explicitly:

  ```
  bash pathToTheFile/my_script.sh
  ```

Variables in Bash

Variables

- Variables names are **case-sensitive** .
- They can only contain **letters** , **numbers** and **underscores** and They can't start with a number.

> The reason you cannot use other characters such as `!`, `*`, or `-` is that these characters have a special meaning for the shell.

Define a Variable

- You can define a variable by simply assigning a value to a name **without any spaces** around the = sign.

```
varname="I Love OSC"
```

Access Variable Values

- To access the value stored in a variable, **prefix** the variable name with a $.

```
echo $varname
```

  - **varname** → refers to the **variable**.
  - **$varname** → refers to the **value** of the variable.

Variable Types

The shell **does not care about types of variables**; they may store ***strings***, ***integers***, real numbers - anything you like. So there is no syntactic difference between:

- `varname="Hello World"`
- `varname=hi`
- `varname=1`
- `varname=3.142`
- `varname="3.142"`
- `varname=123abc`

Command Substitution: Store the output of a command in a variable using `$(command)`.

```
current_date=$(date)
```

Take User Input

- *Positional Parameters*
- `read` *command*

Positional Parameters

special variables used in shell scripting to refer to arguments passed to a script or a function. These parameters are denoted by `$1`, `$2`, `$3`, and so on, where `$1` refers to **the first argument**, `$2` to **the second**, and so forth. `$0` refers to **the name of the script itself**. There are also special parameters like `$#`, `$@` and `$*` that provide additional functionality.

Special Parameters

Special bash parameter	Meaning
`$0`	It's used to reference the name of the shell script.
`$1`, `$2`, `$3`, ..	The first, second, third, ... arguments passed to the script.
`$#`	The number of arguments passed to the script.
`$*`	All the arguments passed to the script, treated as a single string.

- Example:

```
#!/bin/bash
echo "The total no of args are: $#"
echo "The script name is : $0"
echo "The first argument is : $1"
echo "The second argument is: $2"
echo "The total argument list is: $*"
```

- Output:

```
osc@osc:~$ ./script.sh 1 2 3 4
The total no of args are: 4
The script name is : script.sh
The first argument is : 1
The second argument is: 2
The total argument list is: 1 2 3 4
```

`read` Command

1. Interactive Input with read:

The `read` command is used to take input from the user during script execution. This input is then stored in one or more variables.

2. Handling Spaces:

When you input a string with spaces using the read command, it automatically quotes the input so that the entire string (including spaces) is treated as a single entity, rather than being split into multiple words or arguments.

- **Syntax:**

```
read <options> <arguments>
```

- **`read` Command Options:**
  - `-p <prompt>` Outputs the prompt string before reading user input.
  - `-s`: Does not echo the user's input.

- Example 1

```
  GNU nano 6.2                    myscript.sh *
#!/bin/bash

echo "Enter Your First Name: "
read fname

echo "Enter Your Last  Name: "
read lname

echo "Hello, $fname $lname"
```

- Output:

```
habibayossre@pop-os:~$ bash myscript.sh
Enter Your First Name:
Habiba
Enter Your Last  Name:
Yousry
Hello, Habiba Yousry
```

- Example 2

```
#!/bin/bash

#prompt a user for his username
read -p "Enter Your Username:" user_name

#prompt a user for a password, but do not display it on screen

read -sp "Enter Your Password:" password

echo
echo "Your Username is : $user_name"
echo "Your Password is : $password"
```

- Output:

```
habibayossre@pop-os:~$ bash script.sh
Enter Your Username:Habiba Yousry
Enter Your Password:
Your Username is : Habiba Yousry
Your Password is : 123
```

The Difference between `' '` and `" "`

Single Quotes (' ')

- **Literal Strings:** Everything within single quotes is taken literally. No special characters, variables, or commands are interpreted. ***What you see is what you get***.
- **No Variable Expansion:** Variables inside single quotes are ***not expanded***.
- **No Command Substitution:** Command substitution using backticks (``) or $(...) inside single quotes is ***not performed***.

Double Quotes (" ")

- **Partial Literal Strings:** Most characters inside double quotes are taken literally, but certain special characters still have their meaning.
- **Variable Expansion:** Variables inside double quotes are expanded to their values.
- **Command Substitution:** Command substitution within double quotes is performed.
- **Escape Characters:** Certain escape sequences like `\n` (newline), `\t` (tab) are recognized.

Examples:

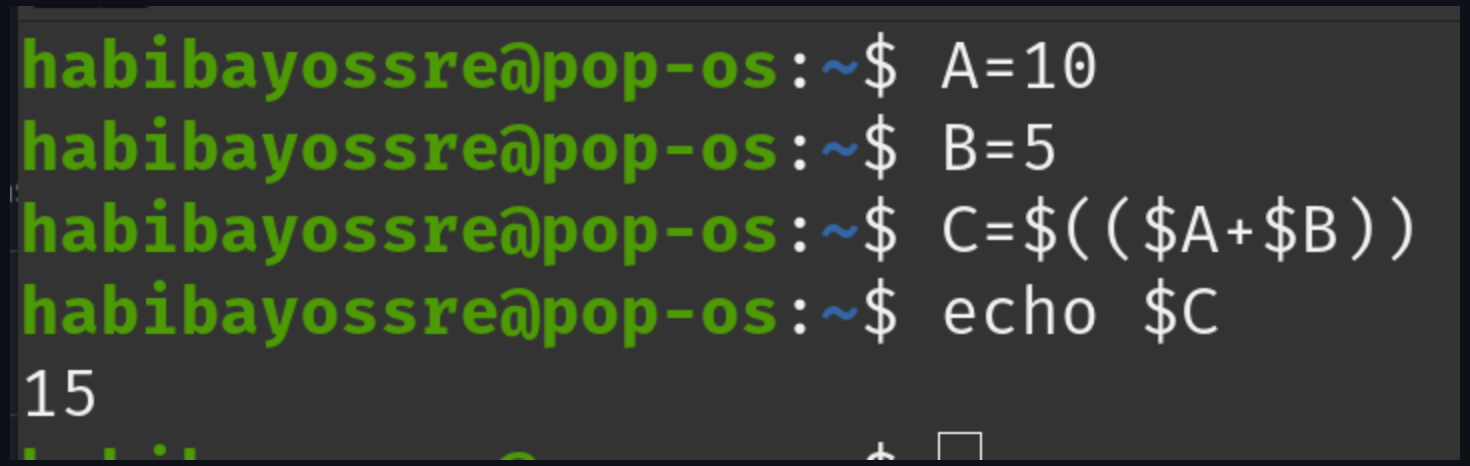
```
osc@pop-os:~$ name="Habiba Yousry"
osc@pop-os:~$ echo "$name"
Habiba Yousry
osc@pop-os:~$ echo $name
Habiba Yousry
osc@pop-os:~$ echo '$name'
$name
osc@pop-os:~$ echo "'$name'"
'Habiba Yousry'
osc@pop-os:~$ echo '"$name"'
"$name"
```

Arithmetic Expansion

- You can do 6 basic arithmetic operators in Bash:
  - `a + b` addition (a plus b)
  - `a - b` subtracting (a minus b)
  - `a * b` multiplication (a times b)
  - `a / b` integer division (a divided by b)
  - `a % b` modulo (the integer remainder of a divided by b)
  - `a ** b` exponentiation (a to the power of b)

- We can do Arithmetic Expansion by just enclose any mathematical expression inside double parentheses `$((  ))`.
- **Example 1:** `a=$((5 - 3 + $b))`

  Which means: variable a is equal to the value of the expression 5 - 3 + $b
- **Example 2:**



`expr` Command

- ***Evaluates a given expression*** and displays its corresponding output.
- It treats numbers containing a decimal point as strings
- **It is used for:**
    - Basic operations like addition, subtraction, multiplication, division, and modulus on integers.
    - Evaluating regular expressions, string operations like substring, length of strings etc.

Note that:

You need to escape the multiplication `*` operator with a backslash `\` to avoid shell expansion. Otherwise, the shell will try to match the operator (*) with the filenames in the current directory and pass them to the expr command, which **will cause an error**.

`expr` Command with Operators

```
habibayossre@pop-os:~$ expr 30+10
30+10
habibayossre@pop-os:~$ expr 30 + 10
40
habibayossre@pop-os:~$ expr 30 - 10
20
habibayossre@pop-os:~$ expr 30 / 10
3
habibayossre@pop-os:~$ expr 30 * 10
expr: syntax error: unexpected argument ‘1-hello’
habibayossre@pop-os:~$ expr 30 \* 10
300
```

`expr` command with variables and string

```
habibayossre@pop-os:~$ x=5
habibayossre@pop-os:~$ y=10
habibayossre@pop-os:~$ expr $x + $y
15
habibayossre@pop-os:~$ name=Habiba
habibayossre@pop-os:~$ expr length $name
6
```

`let` Command

- `let` is a built-in shell command used for arithmetic evaluation in Bash and other Bourne-like shells. It performs arithmetic operations and assigns the result to a variable.
- **Syntax:**

```
let "expression"
```

Examples:

1. Arithmetic Operations:

```
let "a = 5 + 3"   # Adds 5 and 3 and assigns the result (8) to variable 'a'
echo $a           # Outputs 8
let "b = a * 2"   # Multiplies 'a' by 2 and assigns the result (16) to variable 'b'
echo $b           # Outputs 16
```

2. Increment and Decrement:

```
let "i++"         # Increments 'i' by 1
let "j--"         # Decrements 'j' by 1
```

3. Using Variables:

```
a=5
b=3
let "sum = a + b"   # Adds the values of 'a' and 'b', assigns result to 'sum'
echo $sum           # Outputs 8
```

Note That:

To deal with floating point numbers we can use a tool called `bc` but we will not discuss it in this session.

Let's Practise 😊

You have only 7 minutes 🏃

Write a script that calculates the **area** and **perimeter** of a **rectangle** based on user input.

```
habibayossre@pop-os:~/Session6$ bash Hands_On.sh
Enter the length of the rectangle:
5
Enter the width of the rectangle:
4
For a rectangle with length 5 and width 4:
Area: 20
Perimeter: 18
Now let's increase the length and width by 2 and calculate again.
With the new dimensions (length: 7, width: 6):
New Area: 42
New Perimeter: 26
```

Break 😄

Conditions in Shell Scripts

Conditions

- It is very important to understand that all the conditional expressions should be placed inside square braces `[[ Cond ]]` with **spaces around them**.
- What we will discuss in **Conditions** ?

  1. **Comparing String Variables**
  2. **Comparing Numerical Variables**
  3. **File Conditions**

Comparing String Variables

Expression in C	Expression in BASH	Description
`a == b`	`$a = $b` or `$a == $b`	Checks if `a` is equal to `b`
`a != b`	`$a != $b`	Checks if `a` is **not equal** to `b`
`a < b`	`$a < $b`	Checks if `a` is **less than** `b`
`a > b`	`$a > $b`	Checks if `a` is **greater than** `b`

Comparing Numerical Variables

Expression in C	Expression in BASH	Description
`a == b`	`$a -eq $b`	Checks if `a` is **equal** to `b`
`a != b`	`$a -ne $b`	Checks if `a` is **not equal** to `b`
`a < b`	`$a -lt $b`	Checks if `a` is **less than** `b`
`a > b`	`$a -gt $b`	Checks if `a` is **greater than** `b`
`a >= b`	`$a -ge $b`	Checks if `a` is **greater than or equal** to `b`
`a <= b`	`$a -le $b`	Checks if `a` is **less than or equal** to `b`

File Conditions

Expression in BASH	Description
`-d $file`	Checks if file is a **directory**
`-f $file`	Checks if file is an ordinary file as opposed to a directory or special file
`-e #file`	Checks if file/directory **exists**
`-r $file`	Checks if file is **readable**
`-w $file`	Checks if file is **writable**
`-x $file`	Checks if file is **executable**

Comparing String Variables Example

This script checks the value of the variable `x` and prints different outputs based on its value.

```
#!/bin/bash
echo "Enter a String:"
read x

if [[ $x == "String" ]]
  then
      echo 1
elif [[ $x == "String 2" ]]
  then
      echo 2
else
      echo 3
fi
```

Comparing Numerical Variables Example

This script asks the user to input a number and checks whether the number is ***greater*** than, ***equal*** to, or ***less*** than 10, then outputs the appropriate message.

```
#!/bin/bash
echo "Enter a number:"
read number

if [[ $number -gt 10 ]]
  then
    echo "The number is greater than 10"
elif [[ $number -eq 10 ]]
  then
    echo "The number is equal to 10"
else
  echo "The number is less than 10"
fi
```

File Conditions Example

```
#!/bin/bash

# Prompt the user to enter a file path
echo "Enter the file or directory path:"
read file

# Check if the file/directory exists
if [[ -e $file ]]
  then
    echo "$file exists."

    # Check if it is a directory
    if [[ -d $file ]]
      then
        echo "$file is a directory."

    elif [[ -f $file ]]
      then
        echo "$file is an ordinary file."
    fi
fi
```

Case Statements

- ***Case statements*** provide a more elegant way to handle multiple conditions compared to using multiple if statements. They are especially useful when you have a variable that could have ***multiple specific values***.
- **Syntax**

```
case expression in
  pattern1)
    # Commands to execute for pattern1
    ;;
  pattern2)
    # Commands to execute for pattern2
    ;;
  *)
    # Default commands if no pattern matches
    ;;
esac
```

Example on Case Statements

This script prompts the user to enter a letter and then uses a case statement to determine whether the input is a ***lowercase letter***, ***uppercase letter***, ***digit***, or ***special character***, and prints a corresponding message.

```
#!/bin/bash
echo "Enter a letter:"
read letter

case $letter in
  [a-z] )
    echo "You entered a lowercase letter";;
  [A-Z] )
    echo "You entered an uppercase letter";;
  [0-9] )
    echo "You entered a digit";;
  * )
    echo "You entered a special character";;
esac
```

Shell Loops

Loops allow you to repeatedly execute a block of code as long as a certain condition is met. There are three main types of loops in shell scripting: `for`, `while`, and `until`.

For Loop

A `for loop` is used when you want to iterate over a list of items (e.g., numbers, strings, files).

```
for variable in list
do
    # Commands to execute
done
```

- For Loop Example

```
#!/bin/bash

# Example of a for loop
for i in 1..5
do
    echo "Number: $i"
done
```

```
#!/bin/bash

n=5
for (( i=1 ; i<=$n ; i++ ));
do
    "Number: $i"
done
```

- Output:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

While Loop

The **while loop** enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

```
while command
do
   Statement(s) to be executed if command is true
done
```

While Loop Example

This script is uses a `while loop` to count from 1 to 5.

```
#!/bin/bash
count=1
while [ $count -le 5 ]
do
  echo "Count: $count"
  ((count++))
done
```

Break & Continue Statements

`break` Statement

- **Purpose**: The break statement is used to exit a loop prematurely. It immediately terminates the loop in which it is placed, regardless of whether the loop's condition is still true or not.
- **Usage**: Typically used when you want to stop iterating through a loop based on a specific condition.

Example

the loop will stop executing when `counter` reaches 5. The `break` statement causes the script to exit the loop when the condition `[[ $counter -eq 5 ]]` is true.

```
# Example of break in a while loop
counter=1
while [[ $counter -le 10 ]]
do
  if [[ $counter -eq 5 ]]
    then
      echo "Breaking the loop at counter $counter"
      break
  fi

  echo "Counter is $counter"
  ((counter++))
done
```

`continue` Statement

- **Purpose:** The continue statement is used to skip the remaining commands in the current iteration of the loop and proceed to the next iteration. It allows you to skip over parts of the loop based on a condition without terminating the entire loop.
- **Usage:** Commonly used when you want to skip certain iterations of the loop based on a condition but continue looping through the rest.

Example

When the user enters 0, the code skips the lines of code below it and continues to the next iteration.

```
while [[ x -lt 10 ]]
do
    read i

    if [[ i -eq 0 ]]
    then
        echo "Skipping the rest of the code!"
        continue
    fi
    echo $i
    ((x++))
done
```

Shell Functions

- ***Functions*** in shell scripts allow you to encapsulate a block of code that can be reused throughout your script. Functions can take arguments and return values, making them powerful tools for structuring scripts.
- **Syntax**

```
function NAME #Function Definition
{
     #DoThings
}

NAME #Function call
```

```
NAME() #Function Definition
{
     #DoThings
}

NAME #Function call
```

Example

```
#!/bin/bash

function hello
{
  for i in seq 1 3
  do
    echo "Hello !"
  done
}

hello
```

Passing Arguments to a Function

- To use the arguments as variables, you can access their values by using `$n` where `n` is the order of the argument passed to the function.
- **Example**

  This is a function that adds 2 numbers.

```
#!/bin/bash
function add
{
  echo "Num1 + Num2 = $(($1 + $2))"
}

add 2 3
```

Let's Practice 😊

You have only 10 minutes 🏃

Write a script that takes a single number as input and then ***loops*** from ***1 to the input number***, checking if each number is ***even*** or ***odd***.

```
habibayossre@pop-os:~/Session6$ bash Hands_On2.sh
Enter a number:
5
1 is odd
2 is even
3 is odd
4 is even
5 is odd
```

Thanks