Session 9: Shell scripting

Recap & Mohsens call you!!

In Last Session we talk about:

1. What is a shell scripting
2. How to write & execute a script
3. Variables
4. Take User Input
5. Arithmetic Expansion

What is Bash scripting?

- A simple way to think of Bash scripts is to think of them like movie scripts. They tell actors what to say at what time, right? Well, Bash scripts tell Bash what to do at what time.
- Bash scripts are a simple text file contained a series of commands we want to **automate running** rather than running them **manually**.

Agenda

- Conditions
- Case Statements
- Shell Loops
- Shell Functions
- Power of scripting
- PATH Variable

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
```

- **For Loop Example *(Another way)***

```
#!/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
```

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***.

alt text

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 20 minutes 🏃

Build a Simple Calculator

- Expected Output

```
Enter first number: 10
Enter second number: 5
Select an operation:
1) Addition
2) Subtraction
3) Multiplication
4) Division
5) Exit
Enter your choice: 1
Result: 15
-----------------------------------
Enter first number:
```

`$PATH` Variable

- The `$PATH` variable is a list of directories where the shell looks for commands when you type them.

How It Works

- When you type a command like `ls`, Bash searches for it in the directories listed in `$PATH`.
- The **current directory (`.`)** is **not** included in `$PATH` by default.
- That's why, to run a script in the current directory, you need to type `./script_name.sh` instead of just `script_name.sh`.

Viewing Your `$PATH`

To see the directories in your `$PATH`, run:

```
echo $PATH
```

Example output:

```
/usr/local/bin:/usr/bin:/bin:/home/user/.local/bin
```

Each directory is separated by a `:`.

Adding a Directory to `$PATH`

If you want to run a script from anywhere without typing the full path, add its directory to `$PATH`:

```
export PATH=$PATH:/path/to/your/directory
```

However, this change is **temporary** and will reset when you restart your terminal.

Making It Permanent

To make the change permanent, add the export command to your shell's configuration file:

- **For Bash**, add this line to `~/.bashrc` or `~/.bash_profile`:

```
echo 'export PATH=$PATH:/path/to/your/directory' >> ~/.bashrc
source ~/.bashrc
```

- **For other shells**:
  - `zsh` → Add it to `~/.zshrc`
  - `ksh` → Add it to `~/.kshrc`

Now, the change will apply every time you open a new terminal! 🚀

Thanks, Wish you all the best.