Basic Shell Script Examples

Bash (Bourne Again Shell) is a popular Unix and Linux **command-line interpreter** and scripting language. **Shell scripting** in Bash refers to the process of writing scripts using the Bash shell, which enables users to create scripts that run a series of commands, control flow, and carry out complex tasks. It also provides a flexible and powerful environment for automating tasks and performing complex operations efficiently on Unix and Linux systems. In this article, you can find some **basic shell script examples** that will give you a good insight into basic shell scripting as a beginner.

Table of Contents

Getting Started with Shell Script

Shell scripting is the process of writing a series of commands, control structures, and variables in a script file that the shell can execute line by line. To get started with shell scripting, you first have to learn how to **write a bash script** and **make it executable**. In this article, you will learn how to write bash scripts, how to make those scripts executable, and how to run them. So let's get started.

SheBang (#!) in Shell Scripting

While writing a bash script, you **must** start with a line that is called **SheBang(#!)**. It specifies the interpreter or shell that should be used to run the script. The shebang line starts with a hash symbol (**#**) and ends with an exclamation mark (**!**). The path to the interpreter executable is provided immediately after the exclamation mark. So, in shell scripting, the syntax would be **#!/bin/bash**. However, the path specified after the exclamation mark may differ depending on the interpreter's location on the system.

The shebang line is required because it **allows scripts to be executed directly** from the command line by invoking the script file rather than explicitly specifying the interpreter each time. It ensures that the script is run with the appropriate interpreter, allowing the script's commands and logic to be executed.

The syntax for SheBang (#!) in Shell Scripting is given below:

```
#!/bin/bash
```

NOTE: You must write the **SheBang(#!)** on the very first line of the Script.

How to Write and Execute a Bash Script in Linux

Here, I will demonstrate how to write a shell script and then how to execute and run it. I will create a script that will print **Hello World** to the users after running it. I will be using the user's private “**bin**” folder since the “**bin**” directory can automatically be added to the **$PATH** variable. For editing the script, I will use the “**nano**” text editor. Now, follow the steps below to write the script in Linux.

Step 1: Create a Shell Script in Linux

To create and write the shell script follow the steps below.

Steps to follow >

➊First, Launch an Ubuntu Terminal using the shortcut keys **CTRL+ALT+T**.

➋ Then, create a bin folder in your home directory by typing the following command.

```
mkdir bin
```

Explanation

- **mkdir:** Creates a Directory.
- **bin:** User’s private **bin directory**.

💡 **NOTE:** You can skip this step if the directory is already created.

➌ After that, create a bash script file inside the **bin directory** with the command below.

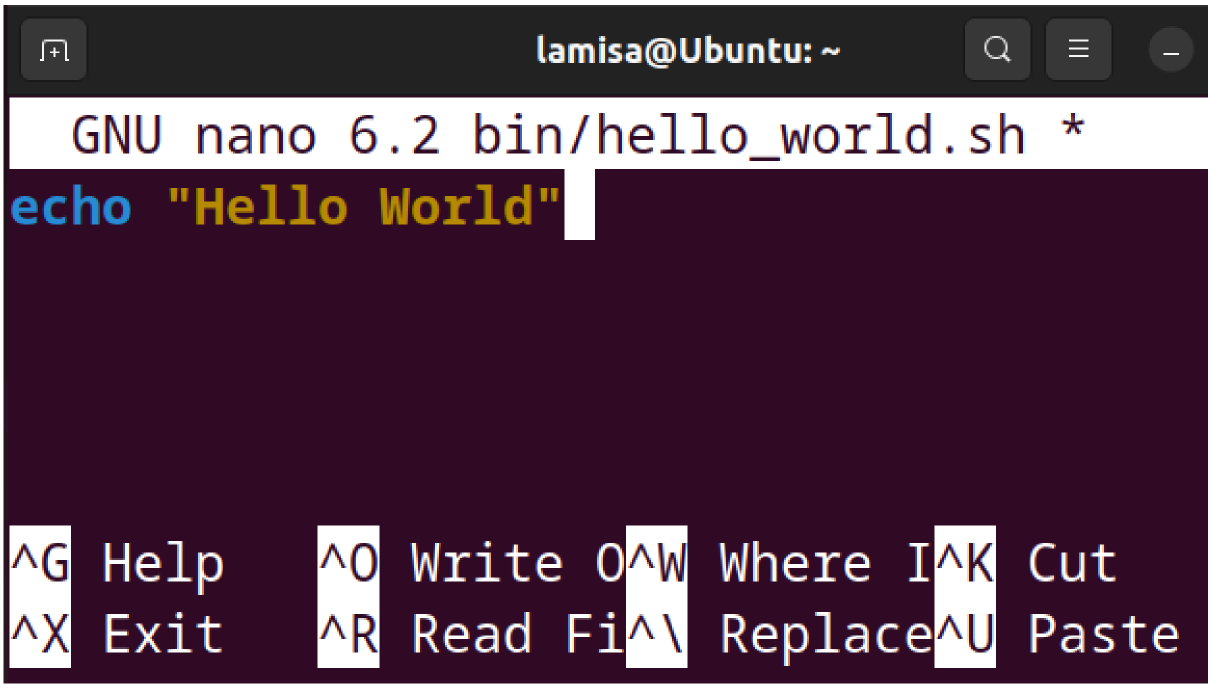
```
nano bin/hello_world.sh
```

Explanation:

- **nano:** Creates/edits text files with Nano text editor.
- **hello_world.sh:** File for writing the bash script.

➍ Now, write the following script in **hello_world.sh** file.

```
#!/bin/bash
echo "Hello World"
```



```
  GNU nano 6.2 bin/hello_world.sh *
echo "Hello World"

^G Help     ^O Write O^W Where I^K Cut
^X Exit     ^R Read Fi^\ Replace^U Paste
```

➎ To save and exit from the script, press **CTRL+S** and **CTRL+X** respectively.

➏ Now, Type the following to make the script executable for the current user in your system.

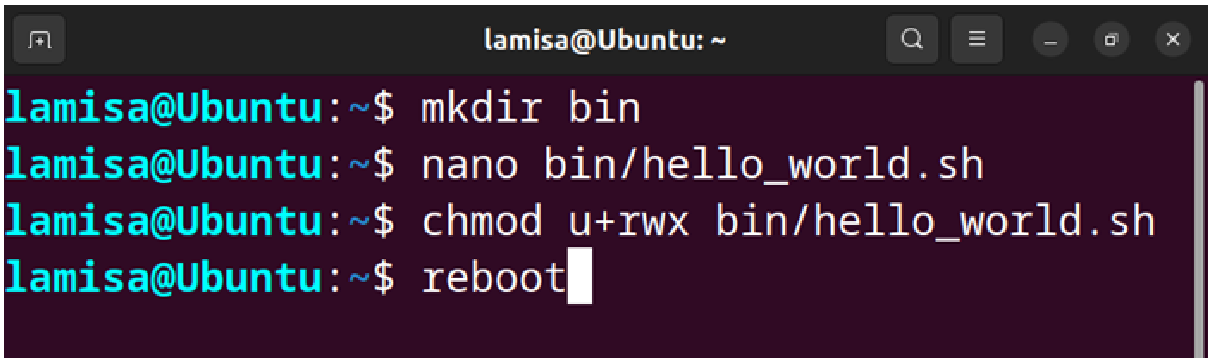
```
chmod u+rwx bin/hello_world.sh
```

Explanation

- **chmod:** Changes folder permissions.
- **u+rwx:** Adds **read**, **write,** and **execute** permissions for the current user.
- **hello_world.sh:** the bash script file.

❼ Finally, restart your system to add the newly created **bin directory** to the **$PATH** variable by typing the command below.

```
reboot
```



```
lamisa@Ubuntu: ~
lamisa@Ubuntu:~$ mkdir bin
lamisa@Ubuntu:~$ nano bin/hello_world.sh
lamisa@Ubuntu:~$ chmod u+rwx bin/hello_world.sh
lamisa@Ubuntu:~$ reboot
```

Restarting the system by default runs the **.profile** script which adds the user's private **bin directory** to **$PATH** and makes the files inside the **bin directory** accessible for every shell session.

Step 2: Running the "Hello World" Script

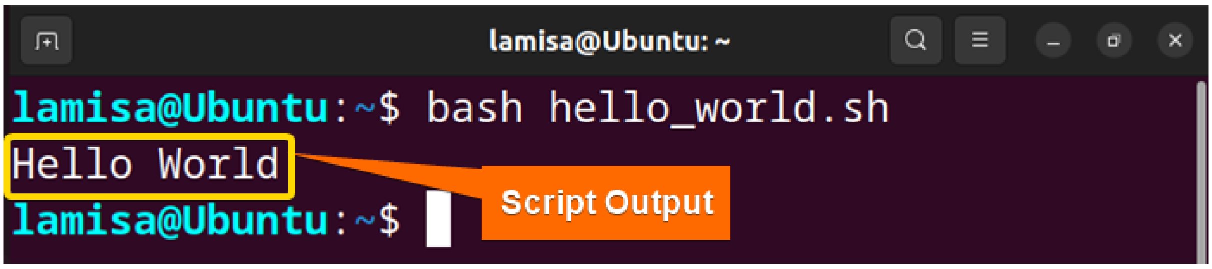
After restarting the system you will be able to run the "**hello_world.sh**" script from any path under the current **user account**. To learn how you can execute the script follow the steps below.

Steps to follow:

➊ At first, press **CTRL+ALT+T** to open the **Ubuntu Terminal**.

➋ Run the previously written script by simply typing the file name and hitting **ENTER**.

```
bash hello_world.sh
```



```
lamisa@Ubuntu: ~
lamisa@Ubuntu:~$ bash hello_world.sh
Hello World
lamisa@Ubuntu:~$
```

In the above image, you can see that, I successfully ran the created "**hello_world.sh**" script. The "Hello World" message is displayed on the terminal from that script.

Basic Bash Scripts

Bash, like any other programming language, adheres to a set of rules and syntaxes. It is essential to begin a bash script with the **shebang line (#!)**. This line instructs the system which interpreter to use when running the script. Following the shebang, you specify the path to the bash executable program, which is typically found at **/bin/bash**.
In addition to becoming acquainted with Linux commands, it is important to learn other fundamental aspects of shell scripting. These are divided into three categories, including **variables, operators, and conditionals**.

Variables in Shell Scripting

Variables are containers that hold important information in shell scripting. They serve as system memory locations capable of storing characters, numeric values, or alphanumeric values. By referencing the variable names, these values can be accessed and manipulated. In shell scripting, the variable name is combined with a dollar sign (**$**), such as **$VARIABLE_NAME.**

The syntax for Variables in Shell Scripting is given below:

```
VARIABLE_NAME=VALUE
```

The rules for Variables in Shell Scripting are as follows:

- Use the equal sign **(=)** to assign values to variable names which indicates that the value on the right-hand side is assigned to the variable on the left-hand side.
- Variable names are case-sensitive, so pay attention to capitalization when referring to variables.
- To refer to the value stored in a variable, use the dollar sign **($)** followed by the variable name.
- When updating or changing the value of a variable, only use the variable name followed by the assignment operator **(=)** and **the new value**.
- You don't need to explicitly define the variable type when declaring variables as the shell interprets the value assigned to a variable accordingly.
- To enclose multiple words or string values within a variable, use single quotes **(' ')** as it ensures that all the characters within the quotes are considered as part of the input for the variable.

Example 1: Defining Variables in a Bash Script

In Bash Script, declare a variable by assigning(=) value to its reference. Furthermore, print the assigned values using **echo $(VARIABLE_NAME)**.

Code:

```
#!/bin/bash
# Declaration of variables
name=Tom
age=12
# Displaying variables
echo $name $age
```

Output:

```
Tom 12
```

Example 2: Read, Store and Display User Input using Bash Script

You can take user input with the **read** command and store it in a variable. Next, use **echo $(VARIABLE_NAME)** to print the user input.

Code:

```
#!/bin/bash
echo "Enter a number:"
read num
echo "The number is: $num"
```

Output:

```
Enter a number:
12
The number is: 12
```

Example 3: Read User Input with Prompt Message using Bash Script

The **read** command used with option **-p** allows you to prompt a message along with taking user input. You can use **echo $(VARIABLE_NAME)** to display the user input on the screen.

Code:

```
#!/bin/bash
read -p "Enter a number:" num
echo "The number is: $num"
```

Output:

```
#!/bin/bash
read -p "Enter a number:" num
echo "The number is: $num"
```

Example 4: Concatenating Multiple Variables

You can concatenate multiple variables and store it into a single variable by enclosing them with a **double quotation (“ ”)**.

Code:

```
#!/bin/bash
# Declaration of variables
name='My name is Tom.'
age='My age is 12.'
# Concatenation
info="${name} ${age}"
echo "Result: $info"
```

Output:

```
Result: My name is Tom. My age is 12.
```

Example 5: Passing Values to Variables as Command Line Arguments

For passing values as command line arguments, you have to run the script along the values in a sequence. Later access these values using the **$** and input sequence number.

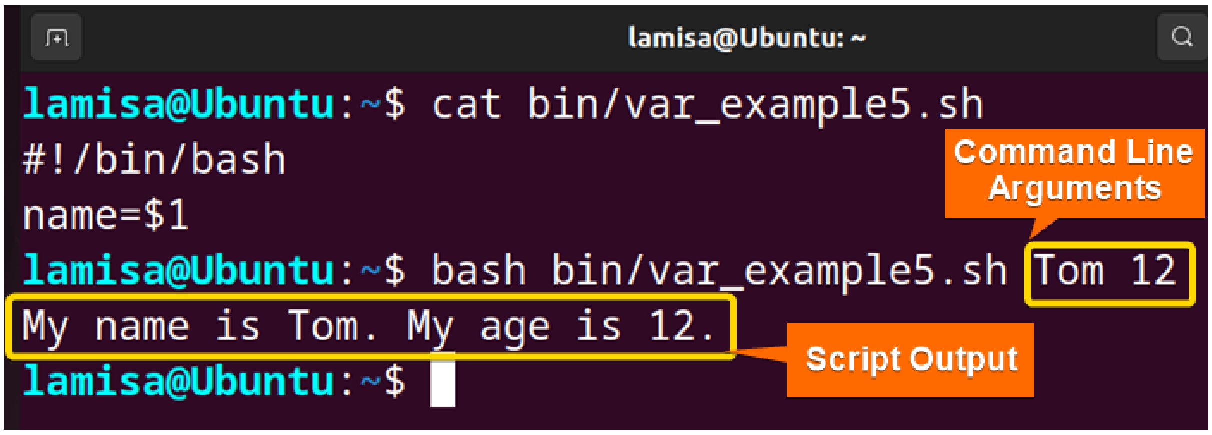
Code:

```
#!/bin/bash
name=$1
age=$2
echo "My name is $name. My age is $age."
```

✅**Syntax to run the Script:** `bash bin/var_example5.sh Tom 12`

Output:

```
My name is Tom. My age is 12.
```



```
lamisa@Ubuntu:~$ cat bin/var_example5.sh
#!/bin/bash
name=$1
lamisa@Ubuntu:~$ bash bin/var_example5.sh Tom 12
My name is Tom. My age is 12.
lamisa@Ubuntu:~$
```

Example 6: Print Environment Variable using Bash Script

You can store an Environment Variable in a regular manner and print it using **${!..}** syntax.

Code:

```
#!/bin/bash
read -p "Enter an Environment Variable name:" var
echo "Environment:${!var}"
```

Output:

```
Enter an Environment Variable name:
HOME
Environment:/home/anonnya
```

Operators in Shell Scripting

Shell scripting provides a wide range of operators to help with a variety of tasks. These operators can be chosen based on your script's output requirements and variables. To make things easier, I've divided the operators in **Bash Scripting** into **five** categories. This classification will help you better understand and apply the operators.

Arithmetic Operators	Numeric Operators	Logical Operators	Bitwise Operators
+ (Addition)	**-lt** (Less than)	**&&** Or, **-a** (AND)	**&** (AND)
- (Subtraction)	**-gt** (Greater than)	**||** Or, **-o** (OR)	**|** (OR)
***** (Multiplication)	**-eq** (Equal)	**!** (NOT)	**!** (NOT)
/ (Division)	**-ne** (Not equal)		**^** (XOR)
% (Modulous)	**-le** (Less or equal)		**<<** (Left shift)
++ (Increment)	**-ge** (Greater or equal)		**>>** (Right shift)
- - (Decrement)			

Example 1: Adding Two Numbers using Bash Script

Run an addition operation using the "**+**" operator between defined variables.

Code:

```
#!/bin/bash
num1=10
num2=20
sum=$(($num1+$num2))
echo "The Sum is: $sum"
```

Output:

```
The Sum is: 30
```

Example 2: Subtracting Two Numbers using Bash Script

Subtract two numbers using the “**-**” operator between defined variables.

Code:

```
#!/bin/bash
num1=30
num2=20
dif=$(($num1-$num2))
echo "The difference is: $dif"
```

Output:

```
The difference is: 10
```

Example 3: Division of Two Numbers using Bash Script

Run a division using the “**/**” operator between defined variables.

Code:

```
#!/bin/bash
num1=30
num2=5
div=$(($num1/$num2))
echo "The division is: $div
```

Output:

```
The division is: 6
```

Example 4: Calculating the Remainder of a Division using Bash Script

For generating the remainder of a division use the “**%**” operator between defined variables.

Code:

```
#!/bin/bash
num1=30
num2=20
mod=$(($num1%$num2))
echo "The remainder is: $mod"
```

Output:

```
The remainder is: 10
```

Example 5: Generating a Random Number Between 1 and 50 Using Bash Script

Utilize the **RANDOM** function of bash for generating random numbers in a range.

Code:

```
#!/bin/bash
echo $((1 + RANDOM % 50))
```
Output:
```
27
```

Example 6: Generating a Random Number Between Two Given Numbers

Generate random numbers of specified numbers by calculating range and with the **RANDOM** function.
Code:
```
#!/bin/bash
read -p "Enter minimum range:" min
read -p "Enter maximum range:" max
r_num=$(( $RANDOM % ($max - $min + 1) + $min ))
echo "Random Number: $r_num"
```
Output:
```
Enter minimum range:10
Enter maximum range:35
Random Number: 24
```

Example 7: Performing Multiple Mathematical Operations in a Script

Perform multiple operations using **echo** without storing the results into another variable.
Code:
```
#!/bin/bash
read -p "Enter a number:" num1
read -p "Enter a smaller number:" num2
echo "Addition: $(($num1 + $num2))"
echo "Subtraction: $(($num1 - $num2))"
echo "Multiplication: $(($num1 * $num2))"
echo "Division: $(($num1 / $num2))"
```
Output:
```
Enter a number:35
Enter a smaller number:15
Addition: 50
Subtraction: 20
Multiplication: 525
Division: 2
```

Example 8: Performs a Bitwise Operation Based on User Input

The given script performs either of the bitwise AND, OR, NOT operations on the 2 input numbers. If the user enters any other operand as input then the script displays and error message.

Code:

```
#!/bin/bash
read -p "Enter two numbers: " num1 num2
read -p "Enter operation to perform (AND, OR, NOT): " op
case $op in
  AND) echo "Result: $num1 & $num2 = $((num1&num2))";;
  OR) echo "Result: $num1 | $num2 = $((num1|num2))";;
  NOT) echo "Result: $num1 ^ $num2 = $((num1^num2))";;
  *) echo "Invalid operator.";;
esac
```

Output:

```
Enter two numbers: 4 5
Enter operation to perform (AND, OR, NOT): AND
Result: 4 & 5 = 4
```

Conditionals in Shell Scripting

Conditional statements are essential for automating tasks with shell scripting. These statements allow specific codes to be executed based on the fulfillment of certain conditions. A basic conditional statement in programming languages evaluates a condition and executes the associated code block if the condition is met. There are **four** types of conditional statements in Bash Scripting. To learn more about the syntax of conditional statements follow the table given below.

The syntax for Conditional Statements in Shell Scripting

if	if-else	if-elif-else	case

if [condition]; then
  #code
fi
	if [condition]; then
	  #code
	else
	  #code
	fi
		if [condition1]; then
		  #code
		elif [condition2]; then
		  #code
		else
		  #code
		fi
			case expression in
			  pattern1)
			    #code;;
			  pattern2)
			    #code;;
			  ***)**
			    #code if expression doesn't match any patterns;;

			esac

Example 1: Check if a Number is an Even or Odd

Check odd and even numbers with simple **if-else** conditions.

Code:

```
#!/bin/bash
read -p "Enter a number:" num
if [ $((num%2)) == 0 ]
then
  echo "The number is even"
else
  echo "The number is odd"
fi
```

Output:

```
Enter a number:25
The number is odd
```

Example 2: Perform an Arithmetic Operation Based on User Input

To perform user input based operations implement the **if-elif-else** condition.

Code:

```
!/bin/bash
read -p "Enter a number:" num1
read -p "Enter a smaller number:" num2
read -p "Enter an operand:" op
if [ $op == + ]
then
     echo "$num1 + $num2 = $((num1+num2))"
elif [ $o == - ]
then
     echo "$num1 - $num2 = $((num1-num2))"
elif [ $op == * ]
then
     echo "$num1 * $num2 = $((num1*num2))"
elif [ $op == / ]
then
     echo "$num1 / $num2 = $((num1/num2))"
else
     echo "Operator not listed"
fi
```

Output:

```
Enter a number:34
Enter a smaller number:14
Enter an operand:+
34 + 14 = 48
```

Example 3: Performs a Logical Operation Based on User Input

You can perform user input based operations with the **case** statement as well.

Code:

```
#!/bin/bash
read -p "Enter two values: "  val1 val2
read -p "Enter an operation(and/or/not) to perform:" op
case $op in
  and)
    if [[ $val1 == true && $val2 == true ]]
    then
      echo "Result: true"
    else
      echo "Result: false"
    fi;;
  or)
    if [[ $val1 == true || $val2 == true ]]
    then
      echo "Result: true"
    else
      echo "Result: false"
    fi;;
  not)
    if [[ $val1 == true ]]
    then
      echo "Result: false"
    else
      echo "Result: true"
    fi;;
  *) echo "Invalid operator."
esac
```

Output:

```
Enter two values: true false
Enter an operation(and/or/not) to perform:or
Result: true
```

Example 4: Check if a Given Input is a Valid Email ID

A valid email can be checked by defining the email syntax inside the **if** condition**.**

Code:

```
#!/bin/bash
read -p "Enter an email ID: " id
if [[ $id =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]
then
  echo "This is a valid email ID!"
else
  echo "This is not a valid email ID!"
fi
```

Output:

```
Enter an email ID: tom@gmail.com
```

Example 5: Check if a Given Input is a Valid URL

To check a valid URL use a simple **if-else** condition with the URL pattern inside the condition.

Code:

```
#!/bin/bash
read -p "Enter a URL: " url
if [[ $url =~ ^(http|https)://[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]
then
  echo " This is a valid URL!"
else
  echo "This is not a valid URL!"
fi
```

Output:

```
Enter a URL: abcdefg1234
This is not a valid URL!
```

Example 6: Check if a Given Number is Positive or Negative

Check if a given number is positive or negative with comparison operators inside the **if-elif-else** condition.

Code:

```
#!/bin/bash
read -p "Enter a number:" num
if [ $num -gt 0 ]
then
  echo "The number is Positive!"
elif [ $num -lt 0 ]
then
  echo "The number is Negative!"
```

```
else
  echo "The number is Zero!!"
fi
```

Output:

```
Enter a number:12
The number is Positive!
```

Example 7: Check if a File is Writable

You can verify file permissions inside **if-else** condition. For this, the write permission is checked with the **-w** notation.

Code:

```
#!/bin/bash
read -p "Enter a File Name:" fname
if [ -w $fname ]
then
  echo "The File $fname is writable."
else
  echo "The File $fname is not writable."
fi
```

Output:

```
Enter a File Name:file1.txt
The File file1.txt is writable.
```

Example 8: Check if a File Exists or Not

Check a file's existence in the current directory using the **-f** notation.

Code:

```
#!/bin/bash
read -p "Enter a File Name:" fname
if [ ! -f $fname ]
then
  echo "The File $fname does not exist!"
  exit 1
fi
echo "The File $fname exists."
```

Output:

```
Enter a File Name:myfile.txt
The File myfile.txt does not exist!
```

Example 9: Check if a Directory Exists or Not

Check a directory's existence in the current folder using the **-d** notation.

Code:

```
#!/bin/bash
read -p "Enter a Filename: " dir
if [ ! -d $dir ]
then
  echo "The directory $dir does not exist!"
  exit 1
fi
echo "The directory $dir exists."
```

Output:

```
Enter a Filename: bin
The directory bin exists.
```

Miscellaneous Bash Scripts

Besides learning the categorized shell scripts example, the following basic scripts will give you a hands-on experience in bash scripting.

Example 1: Echo with New Line

Modify the usage of the **echo** command with **-e** and **\n** to print messages in a new line.

Code:

```
#!/bin/bash
echo -e 'Hi\nthere!'
```

Output:

```
Hi
there!
```

Example 2: Changing Internal Field Separator(IFS)/Delimiter

You can modify the default Internal Field Separator of bash by accessing the **IFS** variable. By changing the **IFS** you will be able to access values separated by your desired delimiter. After this task again restore the original **IFS** to avoid any error.

Code:

```
#!/bin/bash
#store default IFS
old_IFS= $IFS
IFS=,
read val1 val2 val3 <<< "5,60,70"
echo 1st value: $val1
echo 2nd value: $val2
echo 3rd value: $val3
```

```
#restore default IFS
IFS= $old_IFS;
```

Output:

```
1st value: 5
2nd value: 60
3rd value: 70
```

Example 3: Take Two Command Line Arguments and Calculates their Sum

You can do direct mathematical operations on command line arguments using the **$((..))**.

Code:

```
#!/bin/bash
sum=$(( $1 + $2 ))
echo "The sum of $1 and $2 is $sum"
```

✅**Syntax to run the Script:** `bash misc_example3.sh 20 30`

Output:

```
The sum of 20 and 30 is 50
```

Example 4: Take Password Input

In bash, you can utilize the **read** command for taking **password** type inputs. Application of **read** with **-sp** option hides the input characters when you type them**.**

Code:

```
#!/bin/bash
read -sp "Enter your password: " pass
echo -e "\nYour password is: $pass"
```

Output:

```
Enter your password:
Your password is: linuxsimply
```

Example 5: Take Timed Input

You can take timed input in bash using the **read** command with **-t** option. The prompt message will disappear if you do not complete entering your values within the specified time.

Code:

```
#!/bin/bash
read -t 5 -p "Enter your name within 5 seconds: " name
```

Output:

```
Enter your name within 5 seconds: Anonnya
```

Conclusion

Finally, in this article, I have tried to provide some **simple shell script examples** to familiarise you with the power and flexibility of **shell** scripting. You can now customize and automate any system based on specific needs after learning the basics of shell scripting. Although this article has only scratched the surface of shell scripting, this solid foundation will make you well-prepared to explore the vast world of shell scripting and give you the ability to master it in less time.