100 Shell Script Examples

The **GNU Bourne-Again Shell** also known as **bash** is the default shell for most of the **Linux** distributions. Although **bash** is commonly run in its interactive form which is the **Command Line Interface (CLI)**, its non-interactive mode also becomes significant when it comes to executing **Shell Scripts**. **Scripts** are lists of commands stored in a file to run sequentially for task automation. In this article, you will find 100 shell script examples along with the very basics of **Shell Scripting**.

Table of Contents

Getting Started with Shell Script

Shell scripts are executed line-by-line by the **bash** program. Therefore, the first step to learning **Shell scripting** is to write a proper bash program and make it executable. There are several ways in which you can create a script. However, the most convenient process for creating and executing a bash script is described below.

SheBang (#!) in Shell Scripting

The **(#)** and **(!)** signs together are called **SheBang(#!)** in **Shell Scripting**. When the script is run with the **SheBang(#!)** in its first line, it instructs the interpreter to execute the script line-by-line. You will need to write the **SheBang(#!)** along with the **bash path directive: /bin/bash** that denotes the execution of a specified type(**bash**) of the scripts.

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

In this example, I will create the first shell script the "**hello_world.sh**". The task is to display the "Hello World" message on the terminal. As the location of the script file, 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

First, create the shell script file using the instructions below.
Steps to follow:
❶ At first, open a terminal window by pressing **CTRL**+**ALT**+**T**.
❷ 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.
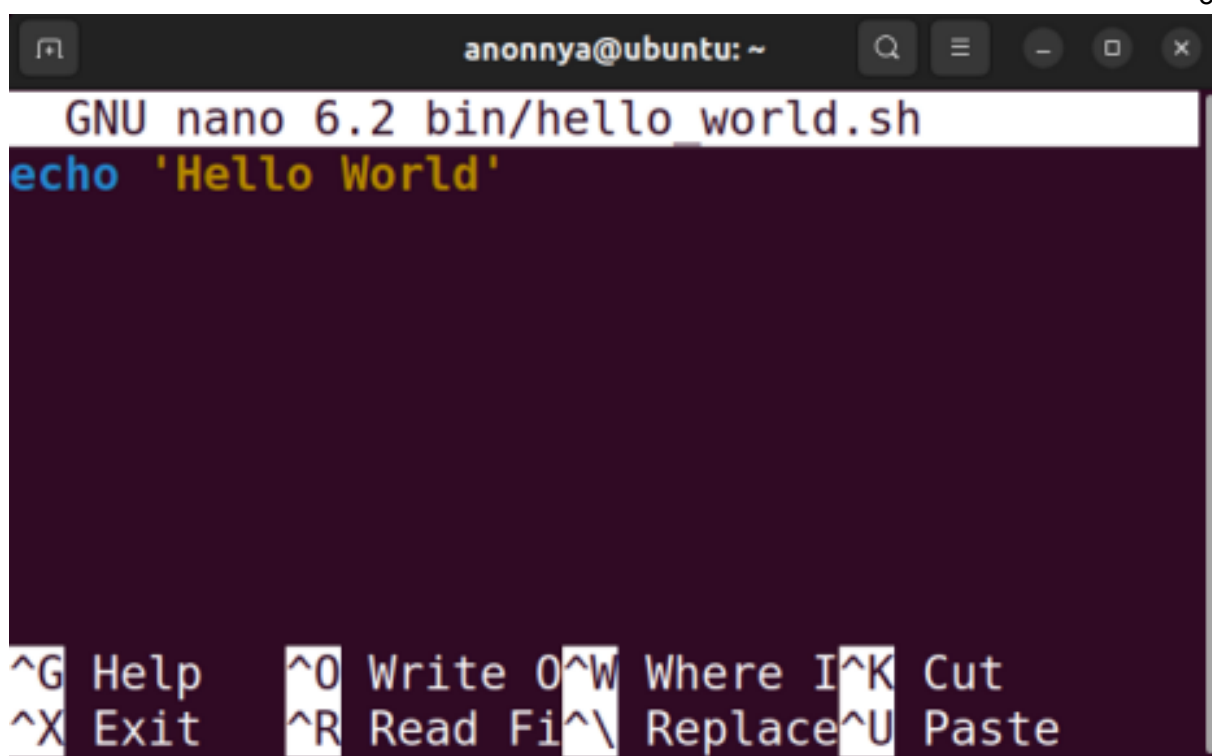
❸ After that, create a bash script file inside the **bin directory** with the command below.

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

Explanation:
- **nano:** Creates/edits text files with Nano text editor.
- **hello_world.sh:** File for writing the bash script.
❹ Now, write the following script in **hello_world.sh** file.

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

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

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" Bash Script in Linux

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:
❶ At first, press **CTRL+ALT+T** to open the **Ubuntu Terminal**.
❷ Run the previously written script by simply typing the file name and hitting **ENTER**.

```
bash hello_world.sh
```


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

Similar to every other programming language **Bash** also has a set of basic terms, rules, and syntaxes. The very first line of a bash script needs to start with the **SheBang (#!)**. Followed by the **SheBang (#!)** then you will need to specify the path to the bash executable program (**/bin/bash**). Furthermore, besides the **Linux Commands** the other basic terms of shell scripting can be divided into categories such as **Variables**, **Operators**, **Conditionals,** etc.

Variables in Shell Scripting

Variables in **shell scripting** are containers for storing necessary information. They specify memory locations in the system via characters or numeric or alphanumeric values. Values

stored in these locations are later accessed and manipulated by referring to their Variable names. In shell scripting, reference to a variable is done by combining a variable name with the **dollar sign ($)** i.e. **$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.
- Variable names are case sensitive i.e. 'A' and 'a' are different.
- To refer to a variable use the **dollar sign ($)** i.e. **$VARIABLE_NAME**. ● While updating/changing the variable values use only the variable name with the assignment operator(=) i.e. **VARIABLE_NAME= NEW_VALUE**.

- No need to define variable type while declaring variables.
- Enclose multiple words or string values within **Single Quote (' ')** to consider all characters as input.

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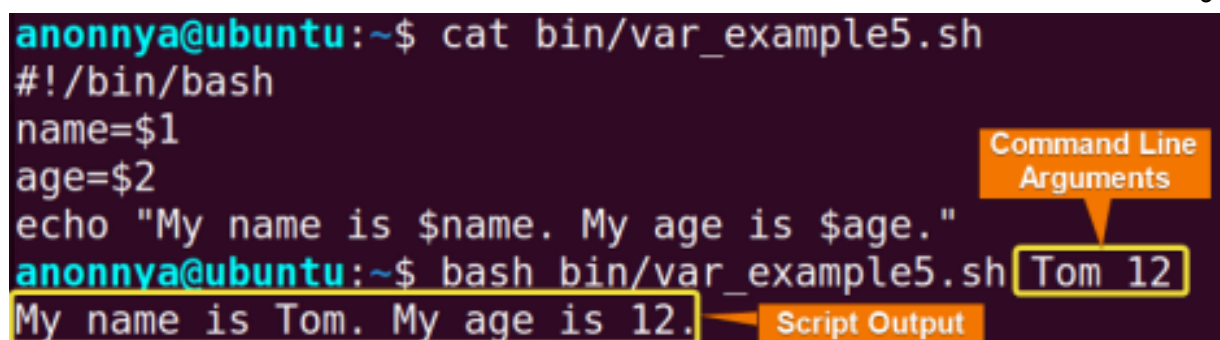
```bash
#!/bin/bash
name=$1
age=$2
echo "My name is $name. My age is $age."
```

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

Output:

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

Example 6: Print Environment Variable using Bash Script

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

```bash
#!/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

For performing different kinds of operations shell scripting offers a variety of operators. Depending on your output criteria and variables you can select these operators for your usage.

For your better understanding, I have divided the operators in **Bash Scripting** into **five** different categories. These are as follows:

Arithmetic Operators Numeric Operators Logical Operators Bitwise Operators

		&& Or, **-a** (AND)	**&** (AND)
+ (Addition)	**-lt** (Less than)		

- (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 **>>** (Right 10

	equal)	shift)
- - (Decrement)		

Example 1: Adding Two Numbers using Bash Script

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

```bash
#!/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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/bin/bash
read -p "Enter a number:" num1
read -p "Enter a smaller number:" num2
echo "Addition: $(($num1 + $num2))"
```

```bash
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:

```bash
#!/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 task automation in shell scripting. A basic conditional statement in programming language works in such a way that it will execute a piece of code depending on the fulfillment of some condition. There are **four** types of conditional statements in **Bash Scripting**. Follow the table below to learn about the syntaxes of these conditional

statements.

The syntax for Conditional Statements in Shell Scripting

if if-else if-elif-else case

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

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

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

```bash
#!/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:

```bash
!/bin/bash
read -p "Enter a number:" num1
read -p "Enter a smaller number:" num2
```

```bash
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:

```bash
#!/bin/bash
read -p "Enter two values: " val1 val2
```

```bash
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)
```

```bash
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:

```bash
#!/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!"
```

```bash
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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/bin/bash
read -p "Enter a Filename: " dir
if [ ! -d $dir ]
```

```bash
then
```

```bash
    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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/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:

```bash
#!/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:**

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

Output:

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

Advanced Bash Scripts

In addition to running basic tasks and commands from the script, you may want to create bash programs with advanced functionalities. **Bash Scripting** offers the concepts of string, array, and loops for achieving such programming goals. In this section, you will learn about these advanced topics through practical examples.

Strings in Shell Scripting

Similar to all the programming languages **Bash** also has the **String** data type which indicates a set of characters. To denote inputs as **String** you must enclose it within the **double quotation("")**. Values passed as strings are considered as text rather than a number or variable. Therefore, **Bash** provides an additional set of operators for the **String** data type.

The syntax for Strings in Shell Scripting is given below:

```
STRING_NAME="STRING_VALUE"
```

The String operators in Shell Scripting are as follows:

String Operators

`<` (Less than) `==` (Equal) `+=` (Concatenation) `>` (Greater than)

`!=` (Not equal)

Example 1: Find the Length of a String

You can simply use the **${#STRING}** to find the length of a string.
Code:

```bash
#!/bin/bash
str="My name is Tom!"
len=${#str}
echo "The length of the string is: $len"
```

Output:

```
The length of the string is: 15
```

Example 2: Check if Two Strings are Equal

Check whether two strings are same or not using the **== (Equal)** operator inside **if** condition.
Code:

```bash
#!/bin/bash

string1="hello"
string2="world"

if [ "$string1" == "$string2" ]; then
    echo "The strings are equal."
else
    echo "The strings are not equal."
fi
```

Output:

```
The strings are not equal.
```

Example 3: Convert All Uppercase Letters in a String to Lowercase

Here is a bash script for converting all upper case letters in a string to lower case letters that use the **tr** command with the **[:upper:]** and **[:lower:]** classes for conversion.
Code:

```bash
#!/bin/bash
read -p "Enter a string: " str
echo "Converted String:" $str | tr '[:upper:]' '[:lower:]'
```

Output:

```
Enter a string: ABCDefgh
converted string: abcdefgh
```

Example 4: Remove All Whitespace from a String

For removing white spaces from a string simply use the **${STRING// /}**.
Code:

```bash
#!/bin/bash
str=" Hello from Linuxsimply ! ! "
str=${str// /}
echo "The resultant string: $str"
```

Output:

```
The resultant string: HellofromLinuxsimply!!
```

Example 5: Reverse a String

To reverse a string use the **rev** command with **echo** and **Pipe(|)**.
Code:

```bash
#!/bin/bash
str="Linuxsimply"
str=$(echo "$str" | rev)
echo "The reversed string: $str"
```

Output:
```
The reversed string: ylpmisxuniL
```

Example 6: Reverse a Sentence

You can reverse a sentence by reversing the order of words with the **awk** command.
Code:

```bash
#!/bin/bash
sentence="Hello from LinuxsimplY!!"
r_sentence=$(echo "$sentence" | awk '{ for(i=NF;i>0;i--) printf("%s ",$i);
print "" }')
echo "The reversed sentence is: $r_sentence"
```

Output:
```
The reversed sentence is: LinuxsimplY!! from Hello
```

Example 7: Capitalize the First Letter of a Word

For capitalizing only the first letter of a word, cut out the first letter to convert it and then concatenate it with the rest of the string.

Code:

```bash
#!/bin/bash
str="linuxsimply!!"
cap_str=$(echo "${str:0:1}" | tr '[:lower:]' '[:upper:]')${str:1}
echo "The capitalized word is: $cap_str"
```

Output:

```
The capitalized word is: Linuxsimply!!
```

Example 8: Replace a Word in a Sentence

You can replace the first occurrence of a word in a string with a given word using the **$(../../..)**.

Code:

```bash
#!/bin/bash
read -p "Enter a sentence: " str1
read -p "Enter the word to be replaced: " str2
read -p "Enter the new word: " str3
echo "Modified sentence: ${str1/$str2/$str3}"
```

Output:

```
Enter a sentence: I love Linux
Enter the word to be replaced: Linux
Enter the new word: Linuxsimply
Modified sentence: I love Linuxsimply
```

Loops in Shell Scripting

Loops are introduced in programming languages to run tasks in a repetitive manner. It iterates a set of statements within a limit depending on conditions. **Bash Scripting** provides **three** types of loops for statement iterations. These are the **for loop**, the **while loop,** and the **until loop**. Syntaxes for each of the loops are listed below.

Syntaxes for Loops in Bash Scripting:

```
                                                            for while until

                              i++ )) do                until [ condition ]
for item in item1            #code to execute          do
item2 ... itemN              done                         #code to
OR,                          while [ condition ] do    execute
for (( i=initial_val;        #code to execute done      done
i<=terminating_val;
```

Example 1: Print Numbers from 5 to 1

You can use the **"until"** loop in bash to print a number sequence. In this case, specify the condition to stop the loop inside "**until [ ]**".

Code:

```bash
#!/bin/bash
n=5
until [ $n == 0 ]
```

```bash
do
    echo $n
    n=$((n-1))
done
```

Output:

```
5
4
3
2
1
```

Example 2: Print Even Numbers from 1 to 10

To print the even number in a range, check the even number condition inside the for loop before printing the number.

Code:

```bash
#!/bin/bash
for (( i=1; i<=10; i++ ))
do
    if [ $((i%2)) == 0 ]
    then
        echo $i
    fi
done
```

Output:

```
2
4
6
8
10
```

Example 3: Print the Multiplication Table of a Number

Use the simple **echo** command inside a "**for**" loop to display the Multiplication Table of a number.

Code:

```bash
#!/bin/bash
read -p "Enter a number: " num
for (( i=1; i<=10; i++ ))
do
```

```bash
    echo "$num x $i = $((num*i))"
done
```

Output:

```
Enter a number: 12
12 x 1 = 12
12 x 2 = 24
12 x 3 = 36
12 x 4 = 48
12 x 5 = 60
12 x 6 = 72
12 x 7 = 84
12 x 8 = 96
12 x 9 = 108
12 x 10 = 120
```

Example 4: Calculate the Sum of Digits of a Given Number

For calculating the sum of digits of a given number, extract each digit using "**%**" operator and store the summation in a fixed variable using the loop.

Code:

```bash
#!/bin/bash
read -p "Enter a number: " num
sum=0
```

```bash
while [ $num -gt 0 ]
do
    dig=$((num%10))
    sum=$((sum+dig))
    num=$((num/10))
done
echo "The sum of digits of the given number: $sum"
```

Output:

```
Enter a number: 1567
The sum of digits of the given number: 19
```

Example 5: Calculate the Factorial of a Number

Calculate the factorial of a number by running multiplications inside a "**for**" loop.
Code:

```bash
#!/bin/bash
read -p "Enter a number: " num
```

```bash
temp=1
for (( i=1; i<=$num; i++ ))
do
    temp=$((temp*i))
done
echo "The factorial of $num is: $temp"
```

Output:

```
Enter a number: 6
The factorial of 6 is: 720
```

Example 6: Calculate the Sum of the First "n" Numbers

To calculate the sum of the first n numbers run a for loop and addition operation till n.
Code:

```bash
#!/bin/bash
read -p "Enter a number: " num
sum=0
for (( i=1; i<=$num; i++ ))
do
    sum=$((sum + i))
```

```bash
done
echo "Sum of first $num numbers: $sum"
```

Output:

```
Enter a number: 100
Sum of first 100 numbers: 5050
```

Arrays in Shell Scripting

Arrays, in general, are a set or collection of data of similar types. Bash arrays differ from arrays in other programming languages since bash does not necessarily differentiate between the numbers or string data types. Therefore, an array in **bash** can store both numbers and strings at the same time. Follow the examples below to learn more about array operations in bash scripting.

Example 1: Loop Through Array Elements

For accessing each array element you can use the **for loop** in the following manner. Indicate the desired array using "**${ARRAY_NAME[@]}**" and access each item stored in the array. **Code:**

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

```bash
arr=("mango" "grape" "apple" "cherry" "orange")
for item in "${arr[@]}"; do
    echo $item
done
```

Output:

```
mango
grape
apple
cherry
orange
```

Example 2: Find the Smallest and Largest Elements in an Array

For finding the smallest and largest element in a given array, first initialize a small and a large number. Then compare the array elements with these numbers inside any loop. **Code:**

```bash
#!/bin/bash
arr=(24 27 84 11 99)
echo "Given array: ${arr[*]}"
```

```
s=100000
l=0
for num in "${arr[@]}"
do
    if [ $num -lt $s ]
    then
        s=$num
    fi
    if [ $num -gt $l ]
    then
        l=$num
    fi
done
echo "The smallest element: $s"
echo "The largest: $l"
```

Output:

```
Given array: 24 27 84 11 99
The smallest element: 11
The largest: 99
```

Example 3: Sort an Array of Integers in Ascending Order

You can sort an array of integers by converting it into a list of integers using "**tr '\n'**". The list of integers is sorted with the "**sort -n**" command and then converted back into an array. **Code:**

```
#!/bin/bash
arr=(24 27 84 11 99)
echo "Given array: ${arr[*]}"
arr=($(echo "${arr[*]}" | tr ' ' '\n' | sort -n | tr '\n' ' '))
echo "Sorted array: ${arr[*]}"
```

Output:

```
Given array: 24 27 84 11 99
Sorted array: 11 24 27 84 99
```

Example 4: Remove an Element from an Array

In bash, you can simply remove an element from an array using the pattern substitution concept. The syntax **${arr[@]/$val}** contains all the elements of the original array "**arr**" except for any occurrences of the value **$val**.

Code:

```bash
#!/bin/bash
arr=(24 27 84 11 99)
echo "Given array: ${arr[*]}"
read -p "Enter an element to remove: " val
arr=("${arr[@]/$val}")
echo "Resultant array: ${arr[*]}"
```

Output:

```
Given array: 24 27 84 11 99
Enter an element to remove: 11
Resultant array: 24 27 84 99
```

Example 5: Inserting an Element Into an Array

For inserting an element into an array, split the array in the given index and insert the element.
Code:

```bash
#!/bin/bash
arr=(24 27 84 11 99)
echo "Given array: ${arr[*]}"
read -p "Enter an element to insert: " new_val
read -p "Enter the index to insert the element: " index
```

```bash
arr=("${arr[@]:0:$index}" "$new_val" "${arr[@]:$index}")
echo "The updated array: ${arr[@]}"
```

Output:

```
Given array: 24 27 84 11 99
Enter an element to insert: 100
Enter the index to insert the element: 3
The updated array: 24 27 84 100 11 99
```

Example 6: Slicing an Array using Bash Script

Slice an array in Bash by placing the indices to slice inside the **${arr[@]:$index1:$index2}**
pattern.
Code:

```bash
#!/bin/bash
arr=(24 27 84 11 99)
echo "Given array: ${arr[*]}"
read -p "Enter 1st index of slice: " index1
```

```bash
read -p "Enter 2nd index of slice: " index2
sliced_arr=("${arr[@]:$index1:$index2}")
echo "The sliced array: ${sliced_arr[@]}"
```

Output:

```
Given array: 24 27 84 11 99
Enter 1st index of slice: 1
Enter 2nd index of slice: 3
The sliced array: 27 84 11
```

Example 7: Calculate the Average of an Array of Numbers

Find the sum of array elements using a "**for**" loop and divide it by the number of elements i.e. **${#arr[@]}**.

Code:

```bash
#!/bin/bash
echo "Enter an array of numbers (separated by space):"
read -a arr
sum=0
for i in "${arr[@]}"
do
    sum=$((sum+i))
done
avg=$((sum/${#arr[@]}))
```

```bash
echo "Average of the array elements: $avg"
```

Output:

```
Enter an array of numbers (separated by space):
23 45 11 99 100
Average of the array elements: 55
```

Example 8: Find the Length of an Array

To find the length of an array simply use the syntax: **${#arr[@]}**.

Code:

```bash
#!/bin/bash
arr=(24 27 84 11 99)
echo "Given array: ${arr[*]}"
len=${#arr[@]}
```

```bash
echo "The length of the array: $len"
```

Output:

```
Given array: 24 27 84 11 99
The length of the array: 5
```

Functions in Shell Scripting

Functions are one of the popular concepts of programming languages. It is a piece of code that can be called and executed as many times as you want. Thus, functions offer efficiency, code optimization, and minimization. Functions in **Bash** work in a similar way as functions in other programming languages. However, there are some rules and syntaxes that you must follow while using them in your script.

The syntax for Function in Shell Scripting:

```
FUNCTION_NAME () {
#codes to execute
}
```

Or,

```
FUNCTION_NAME () { #code to execute; }
```

The rules for Function in Shell Scripting are as follows:

- Functions must be defined before using/calling them.
- You may pass arguments to functions while calling them.

- To access arguments inside the function, use **$1, $2, $3 …** and so on according to the number and sequence of arguments passed.
- The scope of the variables declared inside a function remains within the function.

Example 1: Check if a String is a Palindrome

Write the code to check a palindrome inside the function "**Palindrome()**" and call it by passing the desired string.

Code:

```bash
#!/bin/bash
Palindrome () {
    s=$1
    if [ "$(echo $s | rev)" == "$str" ]
```

```bash
then
    echo "The string is a Palindrome"
else
    echo "The string is not a palindrome"
fi
}
read -p "Enter a string: " str
Palindrome "$str"
```

Output:

```
Enter a string: wow
The string is a Palindrome
```

Example 2: Check if a Number is Prime

Create the "**Prime()**" function that returns whether the parameter passed is prime or not.

Code:

```bash
#!/bin/bash
Prime () {
    num=$1
    if [ $num -lt 2 ]
    then
        echo "The number $num is Not Prime"
        return
    fi
    for (( i=2; i<=$num/2; i++ ))
    do
        if [ $((num%i)) -eq 0 ]
        then
```

```bash
            echo "The number $num is Not Prime"
            return
        fi
    done
    echo "The number $num is Prime"
}
read -p "Enter a number: " num
Prime "$num"
```

Output:

```
Enter a number: 2
The number 2 is Prime
```

Example 3: Convert Fahrenheit to Celsius

Here, the function "**Celsius()**" runs the necessary formula on the passed temperature value in Farenheit to convert it into Celsius.

Code:

```bash
#!/bin/bash
Celsius () {
   f=$1
   c=$((($f-32)*5/9))
   echo "Temperature in Celsius = $c°C"
}
read -p "Enter temperature in Fahrenheit:" f
Celsius $f
```

Output:

```
Enter temperature in Fahrenheit:100
Temperature in Celsius = 37°C
```

Example 4: Calculate the Area of a Rectangle

Write the formula to calculate the area of a rectangle inside the function "**Area()**" and call it by passing the height and width.

Code:

```bash
#!/bin/bash
Area() {
   width=$1
   height=$2
   area=$(($width * $height))
   echo "Area of the rectangle is: $area"
```

```bash
}
read -p "Enter height and width of the ractangle:" h w
Area $h $w
```

Output:

```
Enter height and width of the ractangle:10 4
"Area of the rectangle is: 40"
```

Example 5: Calculate the Area of a Circle

Write the formula to calculate the area of a circle inside the function "**Area()**" and call it by passing the given radius.

Code:

```bash
#!/bin/bash
Area () {
    radius=$1
    area=$(echo "scale=2; 3.14 * $radius * $radius" | bc)
    echo "Area of a circle with radius $radius is $area."
}
read -p "Enter radius of the circle:" r
Area $r
```

Output:

```
Enter radius of the circle:4
Area of a circle with radius 4 is 50.24.
```

Example 6: Grading System

The function "**Grade()**" runs the necessary conditions to divide the number ranges into grades and returns the resultant grade.

Code:

```bash
Grade() {
    score=$1
    if (( $score >= 80 )); then
        grade="A+"
    elif (( $score >= 70 )); then
        grade="A"
    elif (( $score >= 60 )); then
        grade="B"
    elif (( $score >= 50 )); then
        grade="C"
    elif (( $score >= 40 )); then
```

```bash
        grade="D"
    else
```

```bash
    grade="F"
  fi
    echo "The grade for mark $s is $grade"
}
read -p "Enter a score between 1-100:" s
Grade $s
```

Output:

```
Enter a score between 1-100:76
"The grade for mark 76 is A"
```

Task-Specific Bash Scripts (44 Examples)

In addition to the conceptual bash scripts, in this section, you will find some task-specific script examples. These scripts are mostly related to the regular process that you run on your system. Hence, explore the examples below to get more hands-on experience with **Shell Scripting**.

Regular Expression Based Scripts

1. Search for a Pattern inside a File

The script given below will take a filename and a pattern as user input and search it within the file. If the pattern is found then the lines having the pattern will be displayed on the screen along with line numbers. Otherwise, it will print a message saying the pattern did not match. **Code:**

```bash
#!/bin/bash
read -p "Enter filename: " filename
read -p "Enter a pattern to search for: " pattern
grep -w -n $pattern $filename
if [ $? -eq 1 ]; then
echo "Pattern did not match."
fi
```

Output:

```
Enter filename: poem.txt
Enter a pattern to search for: daffodils
4:A host, of golden daffodils;
27:And dances with the daffodils.
```

2. Replace a Pattern in a Fille

The following script will take a file name and a pattern from the user to replace it with a new pattern. Finally, it will display the updated lines on the terminal. If the pattern to replace does not exist, then it will show an error message.

Code:

```bash
#!/bin/bash
read -p "Enter filename: " filename
read -p "Enter a pattern to replace: " pattern
read -p "Enter new pattern: " new_pattern
grep -q $pattern $filename
if [ $? -eq 0 ]; then
sed -i "s/$pattern/$new_pattern/g" $filename
echo "Updated Lines: "
grep -w -n $new_pattern $filename
else
echo "Error! Pattern did not match."
fi
```

Output:

```
Enter filename: poem.txt
Enter a pattern to replace: daffodils
Enter new pattern: dandelions
Updated Lines:
4:A host, of golden dandelions;
27:And dances with the dandelions.
```

File Operations with Shell Scripts

3. Take Multiple Filenames and Prints their Contents

The below script is for reading the contents of multiple files. It will take the file names as user input and display their contents on the screen. If any filename does not exist, it will show a separate error message for that file.

Code:

```bash
#!/bin/bash
read -p "Enter the file names: " files
IFS=' ' read -ra array <<< "$files"
for file in "${array[@]}"
do
if [ -e "$file" ]; then
        echo "Contents of $file:"
```

```
        cat "$file"
    else
        echo "Error: $file does not exist"
fi
done
```

Output:

```
Enter the file names: message.txt passage.txt
Contents of message.txt:
"Merry Christmas! May your happiness be large and your bills be
small." Contents of passage.txt:
The students told the headmaster that they wanted to celebrate the
victory of the National Debate Competition.
```

4. Copy a File to a New Location

You can copy a file to another location using the bash script below. It will read the filename and destination path from the terminal and copy the file if it exists in the current directory. If the file is not there, the script will return an error message.

Code:

```bash
#!/bin/bash
read -p "Enter the file name: " file
read -p "Enter destination path:" dest
if [ -e "$file" ]; then
        cp $file $dest
        file_location=$(readlink -f $dest)
        echo "A copy of $file is now located att: $file_location"
    else
        echo "Error: $file does not exist"
fi
```

Output:

```
Enter the file name: poem.txt
Enter destination path:/home/anonnya/Documents
A copy of poem.txt is now located at: /home/anonnya/Documents
```

5. Create a New File and Write Text Inside

The script given below is for creating a new file and writing text inside the file. You will be able to write into the file from the command line. Upon completion, it will show a message saying the file

has been created.

Code:

```bash
#!/bin/bash
read -p "Enter the file name: " file
echo "Enter text to write:"
read text
echo "$text" > "$file"
echo "---------------------------------"
echo "The File $file is created!"
```

Output:

```
Enter the file name: text_file1.txt
Enter text to write:
In English, there are three articles: a, an, and the. Articles are used
before nouns or noun equivalents and are a type of adjective. The
definite article (the) is used before a noun to indicate that the
identity of the noun is known to the reader.
---------------------------------
The File text_file1.txt is created!
```

6. Compare the Contents of Two Given Files

The following bash script takes two file names as user input and compares there contents. If one or either of the files does not exist in the current directory it shows an error to the user. Otherwise prints the result if the files are identical or not.

Code:

```bash
#!/bin/bash
read -p "Enter the 1st file name: " file1
read -p "Enter the 2nd file name: " file2
if [ ! -f $file1 ] || [ ! -f $file2 ]
then
    echo "Error! One of the files does not exists."
    exit 1
fi
if cmp -s "$file1" "$file2"
then
    echo "The Files $file1 and $file2 are identical."
else
    echo "The Files $file1 and $file2 are different."
fi
```

Output:

```
Enter the 1st file name: article1.txt
Enter the 2nd file name: text_file1.txt
```

```
The Files article1.txt and text_file1.txt are identical.
```

7. Delete a Given File If It Exists

This is a script for checking a file's existence before running deleting the file. The script will take the file's name from the user and delete it if it is found in the current directory. Otherwise, it will display an error.

Code:

```bash
#!/bin/bash
read -p "Enter the file name for deletion: " file
if [ -f $file ]
then
    rm $file
    echo "The file $file deleted successfully!"
else
    echo "Error! The file $file does not exist."
fi
```

Output:

```
Enter the file name for deletion: article1.txt
The file article1.txt deleted successfully!
```

8. Renames a File from Script

You can rename an existing file using the script below. All you have to do is enter the old filename and the new filename. The script will rename the file if it is available in the directory. If the file is not in the path, then it will display an error message.

Code:

```bash
#!/bin/bash
read -p "Enter the file name: " file
read -p "Enter new file name: " new_file
if [ -f $file ]
then
    mv "$file" "$new_file"
```

```
    echo "The file $file has been renamed as $new_file!"
else
    echo "Error! The file $file does not exist."
fi
```

Output:

```
Enter the file name: poem.txt
Enter new file name: daffodils.txt
```

```
The file poem.txt has been renamed as daffodils.txt!
```

File Permission Based Shell Scripts

9. Check the Permissions of a file

The script below checks permissions for the given filename and lists the active permissions of the current user. If there does not exist any file of the input file name, then it displays an error message.

Code:

```
#!/bin/bash
read -p "Enter the file name: " file
if [ -f $file ]; then
if [ -r "$file" ]; then
        echo "Readable"
fi
if [ -w "$file" ]; then
        echo "Writable"
fi
if [ -x "$file" ]; then
            echo "Executable"
fi
else
    echo "Error! The file $file does not exist."
fi
```

Output:

```
Enter the file name: daffodils.txt
Readable
Writable
```

10. Sets the Permissions of a Directory for the Owner

The following script gives the current user read, write, and execute permissions of a directory. The directory name is taken as user input and if the directory does not exist, it displays an error message.

Code:

```bash
#!/bin/bash
read -p "Enter the directory name: " dir
if [ -d $dir ]; then
    chmod u+rwx $dir
```

```bash
    echo "Directory permissions have been updated!"
else
    echo "Error! The directory $dir does not exist."
fi
```

Output:

```
Enter the directory name: Documents
Directory permissions have been updated!
```

11. Change the File Owner

The script here changes the owner of a file if the file exists in the directory. Since changing ownership requires administrator permissions, you will need to give the **sudo** password while running the script. Upon completion of the task, the script will show a success message. **Code:**

```bash
#!/bin/bash
read -p "Enter the file name: " file
read -p "Enter file owner name: " owner
if [ -f $file ]; then
    sudo chown $owner $file
    echo "Changed file owner to $owner!"
else
    echo "Error! The file $file does not exist."
fi
```

Output:

```
Enter the file name: daffodils.txt
Enter file owner name: tom
[sudo] password for anonnya:
Changed file owner to tom!
```

12.Change the Overall Permissions of a File

You can change the permissions of an existing file using the script below. All you have to do is enter the filename, the permissions in **absolute mode,** and the **sudo** password to activate administrative privileges. The script will update the file permissions if it is available in the directory. If the file is not in the path, then it will display an error message.

Code:

```bash
#!/bin/bash
read -p "Enter the file name: " file
read -p "Enter new permissions in Absolute Mode: " permissions
if [ -f $file ]; then
```

```bash
    sudo chmod $permissions $file
    echo "Permissions for the file $file has been changed!"
else
    echo "Error! The file $file does not exist."
fi
```

Output:

```
Enter the file name: daffodils.txt
Enter new permissions in Absolute Mode: 777
[sudo] password for anonnya:
Permissions for the file daffodils.txt has been changed!
```

Network Connection Based Shell Scripts

13.Check a Remote Host for its Availability

The following script checks the network status of a remote host. You will need to enter the host IP address as input and it will return a message saying if the host is up or down. **Code:**

```bash
#!/bin/bash
read -p "Enter remote host IP address:" ip
ping -c 1 $ip
if [ $? -eq 0 ]
then
    echo "----------------------"
    echo "Host is up!"
    echo "----------------------"
else
```

```
echo "---------------------"
echo "Host is down!"
echo "---------------------"
fi
```

Output:

```
Enter remote host IP address:192.168.0.6
PING 192.168.0.6 (192.168.0.6) 56(84) bytes of data.
64 bytes from 192.168.0.6: icmp_seq=1 ttl=64 time=4.10 ms

--- 192.168.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time
0ms rtt min/avg/max/mdev = 4.095/4.095/4.095/0.000 ms
---------------------
```

```
Host is up!
---------------------
```

14. Test if a Remote Port is Open

The script below checks the network connection in a system port. It takes a host address and port number as the input. If the connection to the host through the port number is successful then it verifies that the port is open. Otherwise, it returns a message saying the port is closed.

Code:

```bash
#!/bin/bash

read -p "Enter host address:" HOST
read -p "Enter port number:" PORT

nc -z -v -w5 "$HOST" "$PORT"

if [ $? -eq 0 ]; then
    echo "--------------------------------------------"
    echo "Port $PORT on $HOST is open"
    echo "--------------------------------------------"
else
    echo "--------------------------------------------"
    echo "Port $PORT on $HOST is closed"
    echo "--------------------------------------------"
fi
```

Output:

```
Enter host address:192.168.0.107
Enter port number:80
Connection to 192.168.0.107 80 port [tcp/http-alt]
succeeded! --------------------------------------------
Port 80 on 192.168.0.107 is open
--------------------------------------------
```

15. Checking Network Connectivity

The below script checks network connectivity to a remote host via the internet. If there is a successful connection then it returns the status "internet connection is up". Otherwise, returns "Internet connection is down".

Code:

```bash
#!/bin/bash
read -p "Enter a host address:" HOST
```

```bash
if ping -q -c 1 -W 1 $HOST >/dev/null; then
    echo "--------------------------------------------"
    echo "Internet connection is up"
    echo "--------------------------------------------"
else
    echo "--------------------------------------------"
    echo "Internet connection is down"
    echo "--------------------------------------------"
fi
```

Output:

```
Enter a host address:192.168.0.107
--------------------------------------------
Internet connection is up
--------------------------------------------
```

16. Automating Network Configuration

The following bash script configures a network IP address and subnet mask. Upon configuration, it sets up the gateway and DNS server at the given addresses. All four IP addresses are passed as user input. It will return an error message if it is unsuccessful at running any of the commands.

Code:

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

```bash
echo "Enter network configuration variables:"
read -p "Enter an IP address: " ip_addr
read -p "Enter a subnet mask: " subnet_mask
read -p "Enter a Gateway address: " gateway
read -p "Enter a DNS address: " dns

# Configure network interface
sudo ifconfig eth0 $ip_addr netmask $subnet_mask up
if [ $? -eq 0 ]; then
    # Set default gateway
    sudo route add default gw $gateway
    if [ $? -eq 0 ]; then
        # Set DNS servers
        sudo echo "nameserver $dns" > /etc/resolv.conf
        if [ $? -eq 0 ]; then
            echo "---------------------------------------------"
            echo "Network configuration completed"
```
```bash
            echo "---------------------------------------------"
        else
            echo "---------------------------------------------"
            echo "Error while setting the DNS server."
        fi
    else
        echo "---------------------------------------------"
        echo "Error while setting the default gateway."
    fi
else
    echo "---------------------------------------------"
    echo "Network Configuration Failed."
fi
```

> ✅**Syntax to run the Script: `sudo bash bin/adv_example16.sh`**
> ⚠️**Requirement:** ifconfig must be installed.

Output:

```
Enter network configuration variables:
Enter an IP address: 192.168.0.108
Enter a subnet mask: 255.255.255.0
Enter a Gateway address: 192.168.0.1
Enter a DNS address: 8.8.8.8
------------------------------------------------
Network configuration completed
------------------------------------------------
```

17. Process Management: Check if a Process is Running

The given script can check if a process is currently running on your system or not. You will need to enter your desired process name and the script will display the process's current status.

Code:

```bash
#!/bin/bash
read -p "Enter process name: " process
if pgrep $process > /dev/null
then
    echo "Process is running."
else
    echo "Process is not running."
fi
```

Output:

```
Enter process name: bash
Process is running.
```

Process Management Based Shell Scripts

18. Start a Process if It's Not Already Running

You can use the script given below to start a process. The process name is passed as user input to the script. If the process is already running then it will return a message saying "The Process is already running". Otherwise, It will start the desired process.

Code:

```bash
#!/bin/bash
read -p "Enter process name: " process
if ! pgrep $process > /dev/null
```

```bash
then
    /path/to/process_name &
    echo "The Process $process has now started."
else
    echo "The Process is already running."
fi
```

Output:

```
Enter process name: bash
The Process is already running.
```

19.Stop a Running Process

The script below can stop a process if it runs in the system. The user has to enter a process name as the script input. If the process is currently running then the script will terminate that process. Otherwise, it says, "The process is not running".

Code:

```bash
#!/bin/bash
read -p "Enter process name: " process
if pgrep $process > /dev/null
then
    pkill $process
    echo "The Process $process has stopped."
else
    echo "The Process $process is not running."
fi
```

Output:

```
Enter process name: nslookup
The Process nslookup has stopped.
```

20.Restart a Runnimg process

The following script aims to take a process name as input and then restart it. If the process is already running then the script kills the process and starts over. After the first kill command, it waits for 5 seconds. If by then the process does not terminate then it will force kill that process before restarting.

Code:

```bash
#!/bin/bash
read -p "Enter process name: " process
```

```bash
pid=$(pgrep -f $process)

if [ -n "$pid" ]; then
    kill $pid
    sleep 5
    if pgrep -f $process> /dev/null; then
        echo "Process did not exit properly, force killing..."
        kill -9 $pid
    fi
fi
process_path=$(which $process)
$process_path & echo "Process restarted."
```

Output:

```
Enter process name: firefox
Process restarted.
```

21. Monitor a Process and Restart It If Crashes

The script here, takes a process name as input from the user and checks for its status every 5 seconds. If the process is running without any issues then it shows a message saying "The process is running". Otherwise, it restarts the process and continues to check its status again.

Code

```bash
#!/bin/bash
read -p "Enter process name: " process
process_path=$(which $process)
while true
do
    if pgrep $process > /dev/null
    then
```

```bash
        echo "The Process $process is running."
        sleep 5
    else
        $process_path &
        echo "The Process $process restarted."
        continue
    fi
done
```

Output:

```
Enter process name: firefox
The Process firefox is running.
The Process firefox is running.
```

22. Display the Top 10 CPU-Consuming Processes

The script below lists the top 10 CPU-consuming processes. It prints the Process ID, the percentage of CPU usage along with the command that runs each process. **Code:**

```bash
#!/bin/bash
echo "The current top 10 CPU-consuming processes: "
ps -eo pid,%cpu,args | sort -k 2 -r | head -n 11
```

Output:

```
The current top 10 CPU-consuming processes:
    PID %CPU COMMAND
   2161 0.6 /usr/bin/gnome-shell
   1126 0.5 /usr/sbin/mysqld
   7593 0.5 /usr/libexec/gnome-terminal-server
    832 0.2 /usr/bin/java -Djava.awt.headless=true -jar
/usr/share/java/jenkins.war
   --webroot=/var/cache/jenkins/war --httpPort=8080
    668 0.1 /usr/bin/vmtoolsd
   5498 0.1 gjs
/usr/share/gnome-shell/extensions/ding@rastersoft.com/ding.js -E
-P /usr/share/gnome-shell/extensions/ding@rastersoft.com -M 0 -D
0:0:1918:878:1:34:0:0:0:0
    104 0.0 [zswap-shrink]
     86 0.0 [xenbus_probe]
     26 0.0 [writeback]
     39 0.0 [watchdogd]
```

23. Display the Top 10 Memory-Consuming Processes

The given script lists the top 10 memory-consuming processes. It prints the Process ID, percentage of memory usage as well as the commands for running each process. **Code:**
```
#!/bin/bash
echo "The current top 10 Memory-consuming processes: "
ps -eo pid,%mem,args | sort -k 2 -r | head -n 11
```

Output:

```
The current top 10 Memory-consuming processes:
    PID %MEM COMMAND
```

```
 1126 9.7 /usr/sbin/mysqld
  832 6.8 /usr/bin/java -Djava.awt.headless=true -jar
/usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war
--httpPort=8080
 2161 6.7 /usr/bin/gnome-shell
 2516 2.1 /usr/bin/Xwayland :0 -rootless -noreset -accessx -core -auth
/run/user/1000/.mutter-Xwaylandauth.G8UR41 -listen 4 -listen 5 -displayfd 6
-initfd 7
 2585 1.9 /usr/libexec/gsd-xsettings
 1209 1.5 /usr/bin/dockerd -H fd://
--containerd=/run/containerd/containerd.sock
 5498 1.5 gjs
/usr/share/gnome-shell/extensions/ding@rastersoft.com/ding.js -E -P
/usr/share/gnome-shell/extensions/ding@rastersoft.com -M 0 -D
0:0:1918:878:1:34:0:0:0:0
 2966 1.4 /usr/bin/gedit --gapplication-service
 7593 1.3 /usr/libexec/gnome-terminal-server
 2381 1.3 /usr/libexec/evolution-data-server/evolution-alarm-notify
```

24. Kill Processes of a Specific User

The following script is created to kill all the processes of a specific user. The Specified username is taken as user input. After receiving the username, all the running processes of that user are terminated.

Code:

```bash
#!/bin/bash
read -p "Enter username: " user
sudo pkill -u $user
echo "All processes of user $user have been terminated."
```

Output:

```
Enter username: tom
[sudo] password for anonnya:
All processes of user tom have been terminated.
```

25. Kill All Processes That are Consuming More Than a Certain Amount of CPU

This script takes a CPU usage percentage as user input and terminates all the running

processes that are consuming more than the entered CPU threshold. If there is no process above that threshold, then it returns a message saying there are no such processes. **Code:**

```bash
#!/bin/bash
read -p "Enter CPU usage threshold: " threshold
if [ "$(ps -eo pid,%cpu | awk -v t=$threshold '$2 > t {print $1}' | wc -c)" -gt 0 ]; then
for pid in $(ps -eo pid,%cpu | awk -v t=$threshold '$2 > t {print $1}')
do
    kill $pid
done
echo "All processes consuming more than $threshold% CPU killed."
else
echo "There are no process consuming more than $threshold% CPU."
fi
```

Output:

```
Enter CPU usage threshold: 10
There are no process consuming more than 10% CPU.
```

26. Kill All Processes That are Consuming More Than a Certain Amount of Memory

This script takes a memory space percentage as user input and terminates all the running processes that are consuming more than the entered space threshold. If there is no process above that threshold, then it returns a message saying there are no such processes. **Code:**

```bash
#!/bin/bash
read -p "Enter memory usage threshold (in KB): " threshold
if [ "$(ps -eo pid,%mem | awk -v t=$threshold '$2 > t {print $1}' | wc -c)" -gt 0 ]; then
for pid in $(ps -eo pid,%mem | awk -v t=$threshold '$2 > t {print $1}')
do
```

```bash
    kill $pid
done
echo "All processes consuming more than $threshold KB memory killed."
else
    echo "There are no process consuming more than $threshold KB memory."
fi
```

Output:

```
Enter memory usage threshold (in KB): 10
There are no process consuming more than 10 KB memory.
```

System Information Based Shell Scripts

27.Check the Number of Logged-in Users

You view the find the number of logged-in users in your system with the script below. It counts the users that are logged in only at the current time.

Code:

```
#!/bin/bash
users=$(who | wc -l)
echo "Number of currently logged-in users: $users"
```

Output:

```
Number of currently logged-in users: 2
```

28.Check the Operating System Information

The following script displays information regarding the machine's operating system. It retrieves and lists the os name, release, version as well as system architecture.

Code:

```
#!/bin/bash

os_name=$(uname -s)
os_release=$(uname -r)
os_version=$(cat /etc/*-release | grep VERSION_ID | cut -d '"' -f 2)
os_arch=$(uname -m)

echo "OS Name: $os_name"
echo "OS Release: $os_release"
echo "OS Version: $os_version"
echo "OS Architecture: $os_arch"
```

Output:

```
OS Name: Linux
OS Release: 5.19.0-38-generic
OS Version: 22.04
OS Architecture: x86_64
```

29. Check the System's Memory Usage

The script given below calculates the percentage of memory being used. The "**$3*100/$2**" expression converts the usage into percentages and displays the output with two decimal places.

Code:

```bash
#!/bin/bash
mem=$(free -m | awk 'NR==2{printf "%.2f%%", $3*100/$2}')
echo "Current Memory Usage: $mem"
```

Output:

```
Current Memory Usage: 72.48%
```

30. Check the System's Disk Usage

The following script displays the percentage of disk space used on the root (/) file system. It gets the file system's disk space usage in a human-readable format and prints only the used percentage.

Code:

```bash
#!/bin/bash
disk=$(df -h | awk '$NF=="/"{printf "%s", $5}')
echo "Current Disk Usage: $disk"
```

Output:

```
Current Disk Usage: 80%
```

31. Check the System's Network Information

Use the script below to get the network information of your system. It lists the system's IP address, Gateway address, and DNS server address.

Code:

```bash
#!/bin/bash
echo " System's network information:-"
ip=$(hostname -I)
echo "IP Address: $ip"
```

```
gw=$(ip route | awk '/default/ { print $3 }')
echo "Gateway: $gw"
dns=$(grep "nameserver" /etc/resolv.conf | awk '{print $2}')
echo "DNS Server: $dns"
```

Output:

```
System's network information:-
IP Address: 192.168.0.109
Gateway: 192.168.0.1
DNS Server: 127.0.0.53
```

32. Check the Uptime

The given script can be used to find out the uptime of the system. It will return two values. The first one is the current time, and the second one is the uptime i.e. for how long the system has been running. In this example, "**up 16:19**" indicates that the system has been up for 16 hours and 19 minutes.

Code:

```
#!/bin/bash
uptime | awk '{print $1,$2,$3}' | sed 's/,//'
echo "Uptime: $uptime"
```

Output:

```
Uptime: 00:16:38 up 16:19
```

33. Check the System Load Average

The following script returns the system's Load Average. It will extract the load averages for the past 1, 5, and 15 minutes from the system's uptime and display their average on the screen.

Code:

```
#!/bin/bash
loadavg=$(uptime | awk '{print $10,$11,$12}')
echo "Load Average: $loadavg"
```

Output:

```
Load Average: 0.36
```

34. Check the System Architecture

To determine your current machine's architecture you can run the following script. It returns the system's architecture. In this example, **x86_64** indicates that the machine is using the 64-bit version of the x86 architecture.

Code:

```bash
#!/bin/bash
arch=$(uname -m)
echo "System Architecture: $arch"
```

Output:

```
System Architecture: x86_64
```

35. Count the Number of Files in The System

You can use the script below to find the available number of files on your machine. It runs the find command to check every file on the system and returns the total file count. **Code:**

```bash
#!/bin/bash
count=$(find / -type f | wc -l)
echo "Number of files in the system: $count."
```

Output:

```
Number of files in the system: 500090.
```

Advanced Tasks with Shell Scripts

36. Automated Backup

The following script creates a backup file of a given directory. The source directory path and the destination directory path are user inputs. The backup file is named along with the current date for keeping track. Upon completion of the task, it returns the path where the backup archive resides.

Code:

```bash
#!/bin/bash
read -p "Enter path of the directory to backup: " source_dir
read -p "Enter destination path for backup: " backup_dir
date=$(date +%Y-%m-%d)
backup_file="backup-$date.tar.gz"
# Create backup directory if it doesn't exist
if [ ! -d "$backup_dir" ]; then
```

```bash
    mkdir -p "$backup_dir"
fi
# Create backup archive
tar -czf "$backup_dir/$backup_file" "$source_dir"
echo "Completed Creating backup at: $backup_dir."
```

Output:

```
Enter path of the directory to backup:
/home/anonnya/Documents  Enter destination path for backup:
/home/anonnya/Desktop
tar: Removing leading `/' from member names
Completed Creating backup at: /home/anonnya/Desktop.
```

37. Generate Alert if Disk Space Usage Goes Over a Threshold

The script below generates an alert if the disk space usage goes over a threshold. It takes the threshold and a filename from the user. The alert is then generated in that file along with the disk space usage. If the space consumed is less than the threshold than the file remains empty.

Code:

```bash
#!/bin/bash
read -p "Enter filename to write alert: " file
touch $file
read -p "Enter disk space threshold: " threshold
df -H | grep -vE "^Filesystem|tmpfs|cdrom" | awk '{ print $5 " " $1 }' |
while read output;
do
    usage=$(echo $output | awk '{ print $1}' | cut -d'%' -f1)
    if [ $usage -ge $threshold ]; then
        partition=$(echo $output | awk '{ print $2 }')
        echo "Alert for \"$partition: Almost out of disk space $usage% as on
$(date) " >> $file
        break
    fi
done
cat $file
```

Output:

```
Enter filename to write alert: alert.log
Enter disk space threshold: 70
Alert for "/dev/sda3: Almost out of disk space 80% as on Thu May
11 01:54:50 AM EDT 2023
```

38. Create a New User and Add to Sudo Group

You can use the following script to create a new sudo user in your Linux system. The script will taek the username and password as input to create the user. It will also create a home directory for the user besides adding the account to the sudo group.

Code:

```bash
#!/bin/bash

read -p "Enter username: " username
read -p "Enter password: " password

useradd -m -s /bin/bash -p $(openssl passwd -1 $password) $username
if [ $? -eq 0 ]; then
usermod -a -G sudo $username

mkdir /home/$username/mydir
chown -R $username:$username /home/$username/mydir
usermod -d /home/$username/mydir $username

echo "$username ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers

echo "User $username created successfully!"
echo "User $username added to sudo group!"
else
echo "Error while creating user!"
fi
```

✅**Syntax to run the Script: `sudo bash bin/adv_example38.sh`**

Output:

```
Enter username: Jim
Enter password: linuxsimply
User Jim created successfully!
User Jim added to sudo group!
```

39. Monitor Network traffic

The following script monitors the receiving (RX) and transmitting(TX) packets over a network. User needs to enter the interface name which they want to monitor. Then in every 10 seconds it will display the total packet received and transmitted and their size in KB. **Code:**

```bash
#!/bin/bash
read -p "Enter network interface to monitor traffic (ex. eth0): " net
while true
do
    rx=$(ifconfig $net | grep "RX packets" | awk '{print $3 $6 $7}')
    tx=$(ifconfig $net | grep "TX packets" | awk '{print $3 $6 $7}')
```

```bash
    echo "$(date) RX: $rx, TX: $tx"
    sleep 10
done
```

Output:

```
Enter network interface to monitor traffic (ex. eth0):
ens33 Wed May 10 16:55:40 +06 2023 RX: 342(40.4KB), TX:
171(18.4KB) Wed May 10 16:55:51 +06 2023 RX: 355(41.6KB),
TX: 178(19.0KB) Wed May 10 16:56:01 +06 2023 RX:
361(42.0KB), TX: 178(19.0KB) Wed May 10 16:56:11 +06 2023
RX: 361(42.0KB), TX: 178(19.0KB)
```

40. Monitor CPU and Memory Usage

The script below can be used to monitor the CPU and Memory usage of a system. It extracts the CPU and Memory usage information every 10 seconds and converts them into a percentage for displaying on the screen.

Code:

```bash
#!/bin/bash
while true
do
    cpu=$(top -bn1 | grep "Cpu(s)" | sed "s/.*, *\([0-9.]*\)%* id.*/\1/" |
```

```bash
awk '{print 100 - $1"%"}')
    mem=$(free -m | awk 'NR==2{printf "%.2f%%", $3*100/$2 }')
    echo "$(date) CPU Usage: $cpu, Memory Usage: $mem"
    sleep 10
done
```

Output:

```
Sun May 7 02:19:49 AM EDT 2023 CPU Usage: 29.4%, Memory Usage:
68.78% Sun May 7 02:19:59 AM EDT 2023 CPU Usage: 7.1%, Memory Usage:
68.78% Sun May 7 02:20:10 AM EDT 2023 CPU Usage: 25%, Memory Usage:
68.72% Sun May 7 02:20:20 AM EDT 2023 CPU Usage: 17.6%, Memory
Usage: 68.72% Sun May 7 02:20:30 AM EDT 2023 CPU Usage: 6.2%, Memory
Usage: 68.70%
```

41. Creating a Script and Adding It to PATH

You can use the script below to customize another script and make it runnable. The script here will take another script name and the commands to write within this new script as user inputs. After receiving the input values, it will update the permission modes of the desired script and add it to the **$PATH** variable to make the new script runnable. After creation, you can run this new script with the **bash** keyword.

Code:

```bash
#!/bin/bash
read -p "Enter a name for the command: " my_comm
echo "Enter commands to write on script:"
read comm

read -p "Enter path to the directory containing the command: " comm_path

# Create script for custom command
echo "#!/bin/bash" > $my_comm.sh
echo "$comm" >> $my_comm.sh

# Make script executable
chmod +x $my_comm.sh

# Add script to PATH
export PATH="$PATH$comm_path/$my_comm.sh"

echo "A script called $my_comm has been created."
```

Output:

```
Enter a name for the command: echo_hello
Enter commands to write on script:
echo "Hello from custom command!!"
Enter path to the directory containing the command:
/home/anonnya/bin A script called echo_hello has been created.
```


42.Running a Command at Regular Intervals

The script given below runs a command at a regular time interval. To achieve this task the user has to enter the desired command and the interval for running that command. The interval passed as input must be in the following format: m h dom mon dow.

Code:

```bash
#!/bin/bash
read -p "Enter command to run: " com
command_to_run=$(which $com)
read -p "Enter interval for running the command (m h dom mon dow): "
interval
```

```bash
# Add command to crontab
(crontab -l ; echo "$interval $command_to_run") | sort - | uniq - | crontab -
echo "Command added to crontab and will run at $interval"
```

Output:

```
Enter command to run: echo "1 Minute passed!" >> time.log
Enter interval for running the command (m h dom mon dow): * * * *
* Command added to crontab and will run at * * * * *
```

```
anonnya@ubuntu:~$ cat time.log
1 Minute passed! Date & Time: Sat May  6 04:01:01 PM EDT 2023
1 Minute passed! Date & Time: Sat May  6 04:02:01 PM EDT 2023
1 Minute passed! Date & Time: Sat May  6 04:03:01 PM EDT 2023
1 Minute passed! Date & Time: Sat May  6 04:04:01 PM EDT 2023
1 Minute passed! Date & Time: Sat May  6 04:05:01 PM EDT 2023
```

43. Downloading Files From a List of URLs

The following script takes a filename as input where a list of URLs should be stored. The script will iterate through the list of URLs and download the available contents on the link. It displays each download information on the terminal along with the "Completed Download" message. Upon downloading files from all the URLs, it shows another message saying "All files downloaded successfully!".

Code:

```bash
#!/bin/bash
read -p "Enter the filename containing URLs: " url_file
while read -r url; do
    filename=$(basename "$url")
    curl -o "$filename" "$url"
    echo "Completed Download $filename"
done < "$url_file"
echo
"-----------------------------------------------------------------------------------------"
echo "All files downloaded successfully!"
```

Output:

```
Enter the filename containing URLs: urls.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
```

```
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0curl: (6) Could not
resolve host: linuxsimply.com
Completed Download Emacs-Keybindings-or-Shortcuts-in-Linux.pdf
curl: (3) URL using bad/illegal format or missing URL
Downloaded
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0curl: (6) Could not
resolve host: linuxsimply.com
```

```
Completed Download Bash-Terminal-Keyboard-Shortcuts-for-Information.pdf
-------------------------------------------------------------------------
----------------
All files downloaded successfully!
```

44.Organizes Files in a Directory Based on Their File Types

The script given below organizes files in a directory depending on their type. The user needs to give a destination directory path to organize the files along with the source directory path.

This script will create five directories: 1) Documents, 2) Images, 3) Music, 4) Videos, and 5) Others only if they do not already exist on the destination path. Then, it will check all the files and their extension and move them to the corresponding directory. If there is any unknown file extension, then the script will move the file to the Others Directory.

Code:

```bash
#!/bin/bash

# Specify the source and destination directories
read -p "Enter path to the source directory: " source_dir
read -p "Enter path to the destination directory: " dest_dir

# Create the destination directories if they don't exist
mkdir -p "${dest_dir}/Documents"
mkdir -p "${dest_dir}/Images"
mkdir -p "${dest_dir}/Music"
mkdir -p "${dest_dir}/Videos"
mkdir -p "${dest_dir}/Others"

# Move files to the appropriate directories based on their extensions
for file in "${source_dir}"/*; do
    if [ -f "${file}" ]; then
```

```bash
        extension="${file##*.}"
        case "${extension}" in
            txt|pdf|doc|docx|odt|rtf)
                mv "${file}" "${dest_dir}/Documents"
                ;;
            jpg|jpeg|png|gif|bmp)
                mv "${file}" "${dest_dir}/Images"
                ;;
            mp3|wav|ogg|flac)
                mv "${file}" "${dest_dir}/Music"
```

```
            ;;
    mp4|avi|wmv|mkv|mov)
            mv "${file}" "${dest_dir}/Videos"
            ;;
        *)
            mv "${file}" "${dest_dir}/Others"
            ;;
        esac
    fi
done
echo "Files organized successfully!"
```

Output:

```
Enter path to the source directory: /home/anonnya/Downloads  Enter path
to the destination directory: /home/anonnya/Downloads_Organized  Files
organized successfully!
```

Conclusion

This article covers **100 shell script examples** that a user can frequently use. These examples range from basic to advanced topics along with the **preliminary concepts of script writing and configurations**. Furthermore, the examples are divided into sections and subsections depending on their topic and level of understanding. Therefore, it is a proper guide for users of every category.