**Linux Shell Script Examples**

The majority of shell scripting done on Linux involve the bash shell. However, power users who have specified choices often use other shells such as Zsh and Ksh. We’ll mostly stick with Linux bash scripts in our examples due to their widespread popularity and immense usability. However, our editors have also tried to outline some shell script examples that deal with shells other than bash.

**Linux Bash Scripts**

Bash, aka [the Bourne Again Shell](https://en.wikipedia.org/wiki/Bash_(Unix_shell)), is the default command-line interpreter in most [Linux distros](https://www.ubuntupit.com/best-linux-distro-top-recommendation-to-boost-up-your-linux-experience/) nowadays. It is an upgrade of the earlier Bourne shell that was first introduced in Version 7 Unix. Learning bash shell scripting will allow you to understand other shell scripts much faster.

**1. Hello World**

Programmers often learn new languages via learning the hello world program. It’s a simple program that prints the string ***“Hello World”*** to the standard output. Then, use an editor like vim or nano to create the file hello-world.sh and copy the below lines into it.

#!/bin/bash

echo "Hello World"

Save and quit the file. Next, you need to make this file executable using the below command.

$ chmod a+x hello-world.sh

You can run this using any of the below two commands.

$ bash hello-world.sh

$ ./hello-world.sh

It will print out the string passed to echo inside the script.

**2. Using echo to Print**

The echo command is used for printing out information in bash. It is similar to the C function ‘printf’ and provides many common options, including escape sequences and re-direction.

Copy the below lines into a file called echo.sh and make it executable as done above.

#!/bin/bash

echo "Printing text"

echo -n "Printing text without newline"

echo -e "\nRemoving \t special \t characters\n"

Run the script to see what it does. The **-e** option is used for telling echo that the string passed to it contains special characters and requires extended functionality.

**3. Using Comments**

Comments are useful for documentation and are a requirement for high-quality codebases. It’s a common practice to put comments inside codes that deal with critical logic. To comment out a line, just use the **#**(hash) character before it. For example, check the below bash script example.

#!/bin/bash

# Adding two values

((sum=25+35))

#Print the result

echo $sum

This script will output the number 60. First, check how comments are used using **#** before some lines. The first line is an exception, though. It’s called the shebang and lets the system know which interpreter to use when running this script.

**4. Multi-line comments**

Many people use multi-line comments for documenting their shell scripts. Check how this is done in the next script called comment.sh.

#!/bin/bash

: '

This script calculates

the square of 5.

'

((area=5\*5))

echo $area

Notice how multi-line comments are placed inside **:’** and **‘** characters.

**5. The While Loop**

The while loop construct is used for running some instruction multiple times. Check out the following script called while.sh for a better understanding of this concept.

#!/bin/bash

i=0

while [ $i -le 2 ]

do

echo Number: $i

((i++))

done

So, the while loop takes the below form.

while [ condition ]

do

commands 1

commands n

done

The space surrounding the square brackets is mandatory.

**6. The For Loop**

The for loop is another widely used bash shell construct that allows users to iterate over codes efficiently. A simple example is demonstrated below.

#!/bin/bash

for (( counter=1; counter<=10; counter++ ))

do

echo -n "$counter "

done

printf "\n"

Save this code in a file named for.sh and run it using ./for.sh. Don’t forget to make it executable. This program should print out the numbers 1 to 10.

**7. Receive Input from User**

Getting user input is crucial to implement user interaction in your scripts. The below shell script example will demonstrate how to receive user input within a shell program.

#!/bin/bash

echo -n "Enter Something:"

read something

echo "You Entered: $something"

So, the reading construct, followed by a variable name, is used for getting user input. The input is stored inside this variable and can be accessed using the $ sign.

**8. The If Statement**

If statements are the most common conditional construct available in Unix shell scripting, they take the form shown below.

if CONDITION

then

STATEMENTS

fi

The statements are only executed given the CONDITION is true. The fi keyword is used for marking the end of the if statement. A quick example is shown below.

#!/bin/bash

echo -n "Enter a number: "

read num

if [[ $num -gt 10 ]]

then

echo "Number is greater than 10."

fi

The above program will only show the output if the number provided via input is greater than ten. The **-gt** stands for greater than; similarly **-lt** for less than; **-le** for less than equal; and **-ge** for greater than equal. In addition, the [[ ]] are required.

**9. More Control Using If Else**

Combining the else construct with if allows much better control over your script’s logic. A simple example is shown below.

#!/bin/bash

read n

if [ $n -lt 10 ];

then

echo "It is a one digit number"

else

echo "It is a two digit number"

fi

The else part needs to be placed after the action part of if and before fi.

**10. Using the AND Operator**

The AND operator allows our program to check if multiple conditions are satisfied at once or not. All parts separated by an AND operator must be true. Otherwise, the statement containing the AND will return false. Check the following bash script example for a better understanding of how AND works.

#!/bin/bash

echo -n "Enter Number:"

read num

if [[ ( $num -lt 10 ) && ( $num%2 -eq 0 ) ]]; then

echo "Even Number"

else

echo "Odd Number"

fi

The AND operator is denoted by the **&&** sign.

**11. Using the OR Operator**

The OR operator is another crucial construct that allows us to implement complex, robust programming logic in our scripts. Contrary to AND, a statement consisting of the OR operator returns true when either one of its operands is true. It returns false only when each operand separated by the OR is false.

#!/bin/bash

echo -n "Enter any number:"

read n

if [[ ( $n -eq 15 || $n -eq 45 ) ]]

then

echo "You won"

else

echo "You lost!"

fi

This simple example demonstrates how the OR operator works in Linux shell scripts. It declares the user as the winner only when he enters the number 15 or 45. The || sign represents the OR operator.

**12. Using Elif**

The elif statement stands for else if and offers a convenient means for implementing chain logic. Find out how elif works by assessing the following example.

#!/bin/bash

echo -n "Enter a number: "

read num

if [[ $num -gt 10 ]]

then

echo "Number is greater than 10."

elif [[ $num -eq 10 ]]

then

echo "Number is equal to 10."

else

echo "Number is less than 10."

fi

The above program is self-explanatory, so we won’t dissect it line by line. Instead, change portions of the script like variable names and values to check how they function together.

**13. The Switch Construct**

The switch construct is another powerful feature offered by Linux bash scripts. It can be used where nested conditions are required, but you don’t want to use complex **if-else-elif** chains. Take a look at the next example.

#!/bin/bash

echo -n "Enter a number: "

read num

case $num in

100)

echo "Hundred!!" ;;

200)

echo "Double Hundred!!" ;;

\*)

echo "Neither 100 nor 200" ;;

esac

The conditions are written between the case and esac keywords. The \*) is used for matching all inputs other than 100 and 200.

**14. Command Line Arguments**

Getting arguments directly from the command shell can be beneficial in a number of cases. The below example demonstrates how to do this in bash.

#!/bin/bash

echo "Total arguments : $#"

echo "First Argument = $1"

echo "Second Argument = $2"

Run this script with two additional parameters after its name. I’ve named it test.sh and the calling procedure is outlined below.

$ ./test.sh Hey Howdy

So, $1 is used for accessing the first argument, $2 for the second, and so on. Then, finally, the $# is used for getting the total number of arguments.

**15. Getting Arguments with Names**

The below example shows how to get command-line arguments with their names.

#!/bin/bash

for arg in "$@"

do

index=$(echo $arg | cut -f1 -d=)

val=$(echo $arg | cut -f2 -d=)

case $index in

X) x=$val;;

Y) y=$val;;

\*)

esac

done

((result=x+y))

echo "X+Y=$result"

Name this script test.sh and call it as shown below.

$ ./test.sh X=44 Y=100

It should return X+Y=144. The arguments here are stored inside ‘**$@**‘ and the script fetches them using the Linux cut command.

**16. Concatenating Strings**

String processing is of extreme importance to a wide range of modern bash scripts. Thankfully, it is much more comfortable in bash and allows for a far more precise, concise way to implement this. See the below example for a glance into bash string concatenation.

#!/bin/bash

string1="Ubuntu"

string2="Pit"

string=$string1$string2

echo "$string is a great resource for Linux beginners."

The following program outputs the string “UbuntuPit is a great resource for Linux beginners.” to the screen.

**17. Slicing Strings**

Unlike many programming languages, bash doesn’t provide any in-built function for cutting portions of a string. However, the below example demonstrates how this can be done using parameter expansion.

#!/bin/bash

Str="Learn Bash Commands from UbuntuPit"

subStr=${Str:0:20}

echo $subStr

This script should print out “*Learn Bash Commands*” as its output. The parameter expansion takes the form **${VAR\_NAME:S:L**}. Here, S denotes starting position, and L indicates the length.

**18. Extracting Substrings Using Cut**

The [Linux cut command](https://www.ubuntupit.com/simple-and-useful-linux-cut-command-in-unix/) can be used inside your scripts to ‘cut’ a portion of a string, aka the substring. The next example shows how this can be done.

#!/bin/bash

Str="Learn Bash Commands from UbuntuPit"

#subStr=${Str:0:20}

subStr=$(echo $Str| cut -d ' ' -f 1-3)

echo $subStr

**19. Adding Two Values**

It’s quite easy to perform arithmetic operations inside Linux shell scripts. The below example demonstrates how to receive two numbers as input from the user and add them.

#!/bin/bash

echo -n "Enter first number:"

read x

echo -n "Enter second number:"

read y

(( sum=x+y ))

echo "The result of addition=$sum"

As you can see, adding numbers in bash is reasonably straightforward.

**20. Adding Multiple Values**

You can use loops to get multiple user input and add them to your script. The following examples show this in action.

#!/bin/bash

sum=0

for (( counter=1; counter<5; counter++ ))

do

echo -n "Enter Your Number:"

read n

(( sum+=n ))

#echo -n "$counter "

done

printf "\n"

echo "Result is: $sum"

However, omitting the **(( ))** will result in string concatenation rather than addition. So, check for things like this in your program.

**21. Functions in Bash**

As with any programming dialect, functions play an essential role in Linux shell scripts. They allow admins to create custom code blocks for frequent usage. The below demonstration will outline how functions work in Linux bash scripts.

#!/bin/bash

function Add()

{

echo -n "Enter a Number: "

read x

echo -n "Enter another Number: "

read y

echo "Adiition is: $(( x+y ))"

}

Add

Here we’ve added two numbers just like before. But here, we’ve done the work using a function called Add. So whenever you need to add again, you can just call this function instead of writing that section again.

**22. Functions with Return Values**

One of the most fantastic functions is allowing the passing of data from one function to another. It is useful in a wide variety of scenarios. Check out the next example.

#!/bin/bash

function Greet() {

str="Hello $name, what brings you to UbuntuPit.com?"

echo $str

}

echo "-> what's your name?"

read name

val=$(Greet)

echo -e "-> $val"

Here, the output contains data received from the Greet() function.

**23. Creating Directories from Bash Scripts**

The ability to run system commands using shell scripts allows developers to be much more productive. The following simple example will show you how to create a directory from within a shell script.

#!/bin/bash

echo -n "Enter directory name ->"

read newdir

cmd="mkdir $newdir"

eval $cmd