The 40 Simple Yet Effective Linux Shell Script Examples

By Mehedi Hasan In A-Z Commands Linux

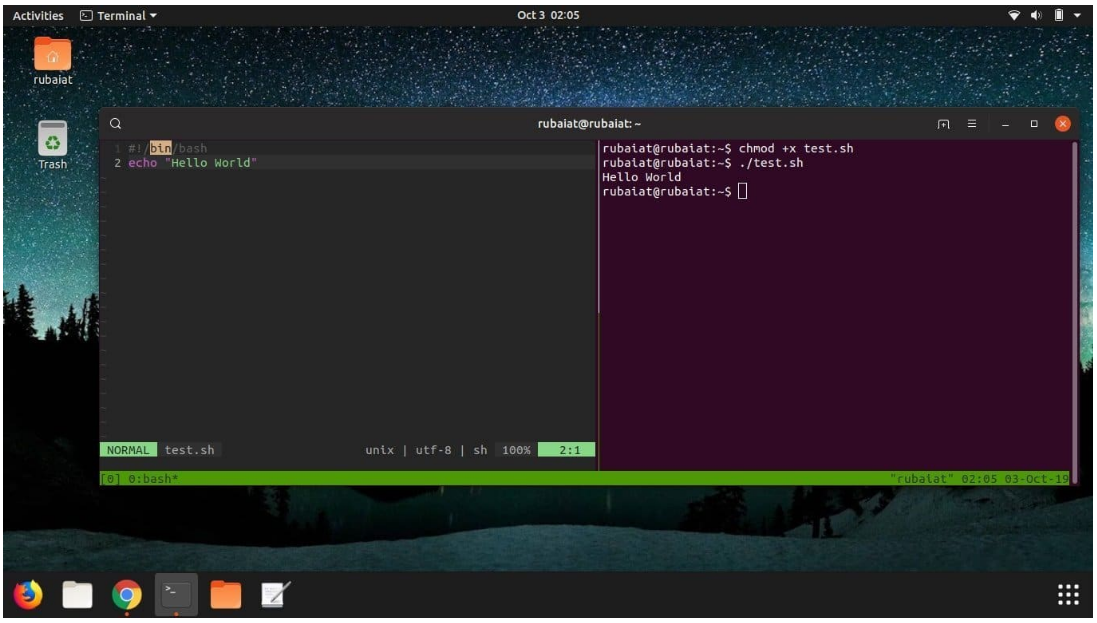
Historically, the shell has been the native command-line interpreter for Unix-like systems. It has proven to be one of Unix's major features throughout the years and grew into a whole new topic itself. Linux offers a variety of powerful shells with robust functionality, including Bash, Zsh, Tcsh, and Ksh. One of the most amazing features of these shells is their programmability. Creating simple yet effective Linux shell scripts for tackling day-to-day jobs is quite easy. Moreover, a modest knowledge of this topic will make you a Linux power user in no time. Stay with us to for a detailed introduction to Unix shell scripting.

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. As a result, you'll find a substantial amount of familiarity between different shell scripts.

Linux Bash Scripts

Bash, aka the Bourne Again Shell, is the default command-line interpreter in most Linux distros 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. So, try these simple examples yourself to gain the first-hand experience.



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

```
NORMAL  test.sh                    unix | utf-8 | sh  100%   2:1
[0] 0:bash*                                  "rubaiat" 02:05 03-Oct-19
```

```
rubaiat@rubaiat:~$ chmod +x test.sh
rubaiat@rubaiat:~$ ./test.sh
Hello World
rubaiat@rubaiat:~$ 
```

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

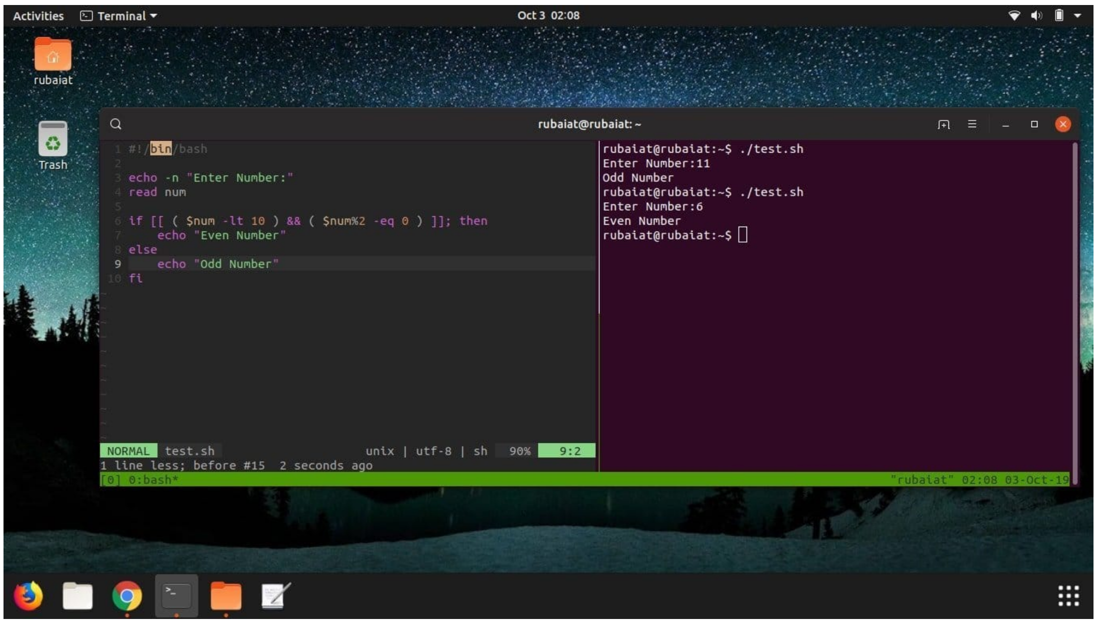
The AND operator allows our program to check if multiple conditions are satisfied at once or not. All parts separated by an AND operator must be true. Otherwise, the statement containing the AND will return false. Check the following bash script example for a better understanding of how AND works.

```
#!/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.



```
#!/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
```

```
NORMAL  test.sh                 unix | utf-8 | sh   90%   9:2
1 line less; before #15  2 seconds ago
[0] 0:bash*                                  "rubaiat" 02:08 03-Oct-19
```

```
rubaiat@rubaiat:~$ ./test.sh
Enter Number:11
Odd Number
rubaiat@rubaiat:~$ ./test.sh
Enter Number:6
Even Number
rubaiat@rubaiat:~$
```

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.

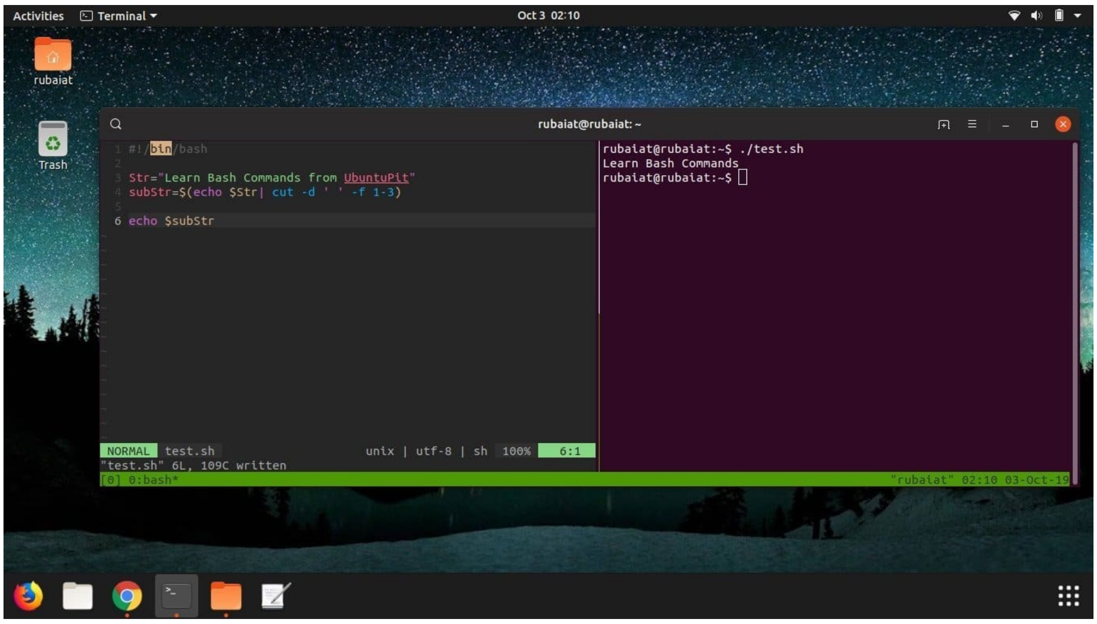
18. Extracting Substrings Using Cut

The Linux cut command 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
```

Check out this guide to understand how Linux Cut command works.



```
#!/bin/bash

Str="Learn Bash Commands from UbuntuPit"
subStr=$(echo $Str| cut -d ' ' -f 1-3)

echo $subStr
```

```
NORMAL test.sh                unix | utf-8 | sh  100%   6:1
"test.sh" 6L, 109C written
[0] 0:bash*                                "rubaiat" 02:10 03-Oct-19
```

```
rubaiat@rubaiat:~$ ./test.sh
Learn Bash Commands
rubaiat@rubaiat:~$ 
```

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
```

This script simply calls your standard shell command mkdir and passes it the directory name if you look closely. This program should create a directory in your filesystem. You can also pass the command to execute inside backticks(") as shown below.

```
`mkdir $newdir`
```

24. Create a Directory after Confirming Existence

The above program will not work if your current working directory already contains a folder with the same name. For example, the below program will check for the existence of any folder named **$dir** and only create one if it finds none.

```
#!/bin/bash
echo -n "Enter directory name ->"
read dir
if [ -d "$dir" ]
then
echo "Directory exists"
else
`mkdir $dir`
echo "Directory created"
fi
```

Write this program using eval to increase your bash scripting skills.

25. Reading Files

Bash scripts allow users to read files very effectively. The below example will showcase how to read a file using shell scripts. First, create a file called editors.txt with the following contents.

```
1. Vim
2. Emacs
3. ed
4. nano
5. Code
```

This script will output each of the above 5 lines.

```
#!/bin/bash
file='editors.txt'
while read line; do
echo $line
done < $file
```

26. Deleting Files

The following program will demonstrate how to delete a file within Linux shell scripts. The program will first ask the user to provide the filename as input and will delete it if it exists. The Linux rm command does the deletion here.

```
#!/bin/bash
echo -n "Enter filename ->"
```

```
read name
rm -i $name
```

Let’s type in editors.txt as the filename and press y when asked for confirmation. It should delete the file.

27. Appending to Files

The below shell script example will show you how to append data to a file on your filesystem using bash scripts. It adds an additional line to the earlier editors.txt file.

```
#!/bin/bash
echo "Before appending the file"
cat editors.txt
echo "6. NotePad++" >> editors.txt
echo "After appending the file"
cat editors.txt
```

You should notice by now that we’re using everyday terminal commands directly from Linux bash scripts.

28. Test File Existence

The next shell script example shows how to check the existence of a file from bash programs.

```
#!/bin/bash
filename=$1
if [ -f "$filename" ]; then
echo "File exists"
else
echo "File does not exist"
fi
```

We are passing the filename as the argument from the command-line directly.

29. Send Mails from Shell Scripts

It is quite straightforward to send emails from bash scripts. The following simple example will demonstrate one way of doing this from bash applications.

```
#!/bin/bash
recipient=”admin@example.com”
subject=”Greetings”
message=”Welcome to UbuntuPit”
`mail -s $subject $recipient <<< $message`
```

It will send an email to the recipient containing the given subject and message.

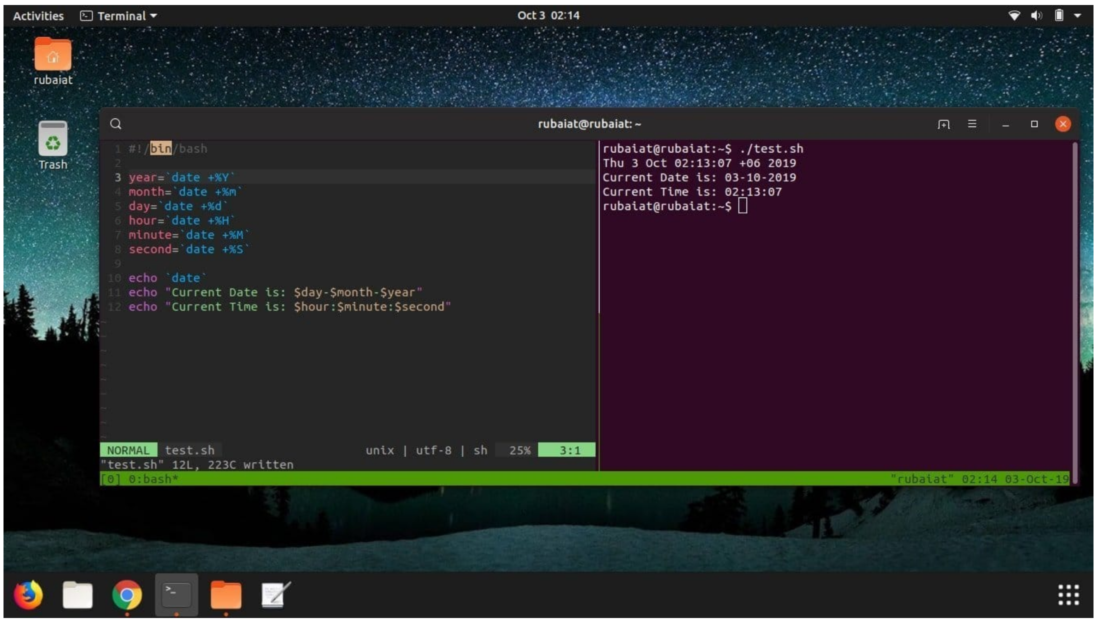
30. Parsing Date and Time

The next bash script example will show you how to handle dates and times using scripts. Again, the Linux date command is used for getting the necessary information, and our program does the parsing.

```
#!/bin/bash
year=`date +%Y`
```

```
month=`date +%m`
day=`date +%d`
hour=`date +%H`
minute=`date +%M`
second=`date +%S`
echo `date`
echo "Current Date is: $day-$month-$year"
echo "Current Time is: $hour:$minute:$second"
```

Run this program to see how it works. Also, try running the date command from your terminal.



```
#!/bin/bash

year=`date +%Y`
month=`date +%m`
day=`date +%d`
hour=`date +%H`
minute=`date +%M`
second=`date +%S`

echo `date`
echo "Current Date is: $day-$month-$year"
echo "Current Time is: $hour:$minute:$second"
NORMAL  test.sh                unix | utf-8 | sh   25%   3:1
"test.sh" 12L, 223C written
[0] 0:bash*                                   "rubaiat" 02:14 03-Oct-19
```

```
rubaiat@rubaiat:~$ ./test.sh
Thu 3 Oct 02:13:07 +06 2019
Current Date is: 03-10-2019
Current Time is: 02:13:07
rubaiat@rubaiat:~$
```

31. The Sleep Command

The sleep command allows your shell script to pause between instructions. It is useful in a number of scenarios, such as performing system-level jobs. The next example shows the sleep command in action from within a shell script.

```
#!/bin/bash
echo "How long to wait?"
read time
sleep $time
echo "Waited for $time seconds!"
```

This program pauses the last instruction's execution until **$time** seconds, which the user in this case provides.

32. The Wait Command

The wait command is used for pausing system processes from Linux bash scripts. Check out the following example for a detailed understanding of how this works in bash.

```
#!/bin/bash
echo "Testing wait command"
sleep 5 &
pid=$!
kill $pid
```

```
wait $pid
echo $pid was terminated.
```

Run this program yourself to check out how it works.

33. Displaying the Last Updated File

Sometimes you might need to find the last updated file for certain operations. The following simple program shows us how to do this in bash using the awk command. It will list either the last updated or created file in your current working directory.

```
#!/bin/bash

ls -lrt | grep ^- | awk 'END{print $NF}'
```

For the sake of simplicity, we'll avoid describing how awk functions in this example. Instead, you can simply copy this code to get the task done.

34. Adding Batch Extensions

The below example will apply a custom extension to all of the files inside a directory. Create a new directory and put some files in there for demonstration purposes. My folder has a total of five files, each named test followed by (0-4). I've programmed this script to add (**.UP)** at the end of the files. You can add any extension you want.

```
#!/bin/bash
dir=$1
for file in `ls $1/*`
do
mv $file $file.UP
done
```

Firstly, do not try this script from any regular directory; instead, run this from a test directory. Plus, you need to provide the directory name of your files as a command-line argument. Use period(.) for the current working directory.

35. Print Number of Files or Directories

The below Linux bash script finds the number of files or folders present inside a given directory. It utilizes the Linux find command to do this. First, you need to pass the directory name to search for files from the command line.

```
#!/bin/bash

if [ -d "$@" ]; then
echo "Files found: $(find "$@" -type f | wc -l)"
echo "Folders found: $(find "$@" -type d | wc -l)"
else
echo "[ERROR] Please retry with another folder."
exit 1
fi
```

The program will ask the user to try again if the specified directory isn't available or have permission issues.

36. Cleaning Log Files

The next simple example demonstrates a handy way we can use shell scripts in real life. This program will simply delete all log files present inside your /var/log directory. You can change the variable that holds this directory for cleaning up other logs.

```
#!/bin/bash
LOG_DIR=/var/log
cd $LOG_DIR

cat /dev/null > messages
cat /dev/null > wtmp
echo "Logs cleaned up."
```

Remember to run this Linux shell script as root.

37. Backup Script Using Bash

Shell scripts provide a robust way to back up your files and directories. The following example will backup each file or directory that have been modified within the last 24 hour. This program utilizes the find command to do this.

```
#!/bin/bash

BACKUPFILE=backup-$(date +%m-%d-%Y)
archive=${1:-$BACKUPFILE}

find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
echo "Directory $PWD backed up in archive file \"$archive.tar.gz\"."
exit 0
```

It will print the names of the files and directories after the backup process is successful.

38. Check Whether You're Root

The below example demonstrates a quick way to determine whether a user is a root or not from Linux bash scripts.

```
#!/bin/bash
ROOT_UID=0

if [ "$UID" -eq "$ROOT_UID" ]
then
echo "You are root."
else
echo "You are not root"
fi
exit 0
```

The output of this script depends on the user running it. It will match the root user based on the **$UID**.

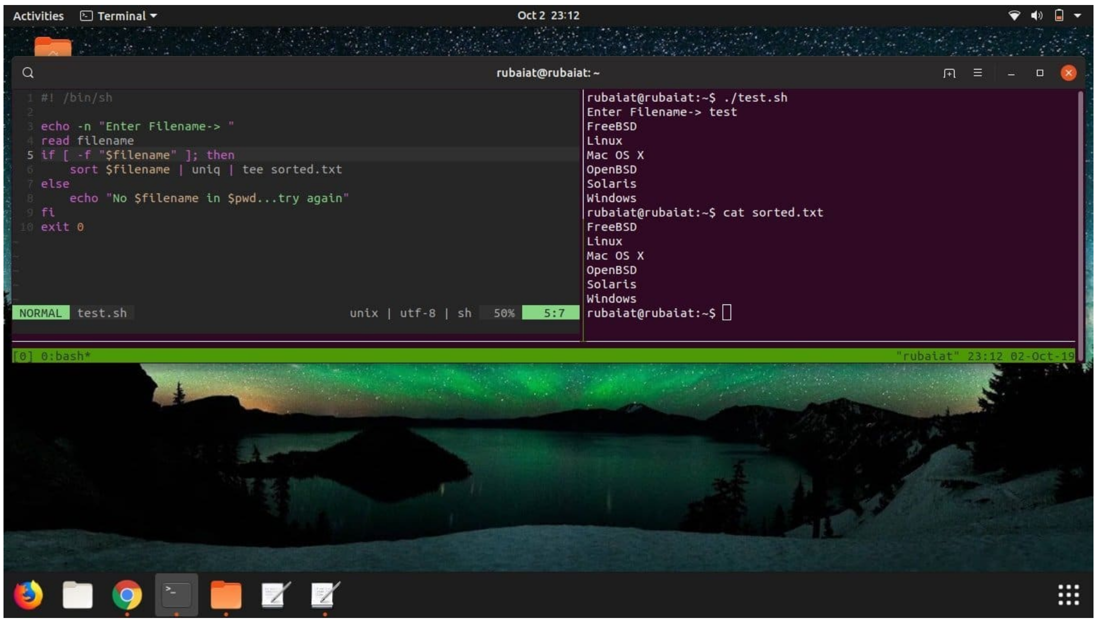
39. Removing Duplicate Lines from Files

File processing takes considerable time and hampers the productivity of admins in many ways. For example, searching for duplicates in your files can become a daunting task. Luckily, you can do this with a short shell script.

```
#! /bin/sh

echo -n "Enter Filename-> "
read filename
if [ -f "$filename" ]; then
sort $filename | uniq | tee sorted.txt
else
echo "No $filename in $pwd...try again"
fi
exit 0
```

The above script goes line by line through your file and removes any duplicative line. It then places the new content into a new file and keeps the original file intact.



```
#! /bin/sh

echo -n "Enter Filename-> "
read filename
if [ -f "$filename" ]; then
    sort $filename | uniq | tee sorted.txt
else
    echo "No $filename in $pwd...try again"
fi
exit 0
```

```
NORMAL  test.sh                unix | utf-8 | sh   50%   5:7
```

```
rubaiat@rubaiat:~$ ./test.sh
Enter Filename-> test
FreeBSD
Linux
Mac OS X
OpenBSD
Solaris
Windows
rubaiat@rubaiat:~$ cat sorted.txt
FreeBSD
Linux
Mac OS X
OpenBSD
Solaris
Windows
rubaiat@rubaiat:~$
```

```
[0] 0:bash*                                  "rubaiat" 23:12 02-Oct-19
```

40. System Maintenance

I often use a little Linux shell script to upgrade my system instead of doing it manually. The below simple shell script will show you how to do this.

```
#!/bin/bash

echo -e "\n$(date "+%d-%m-%Y --- %T") --- Starting work\n"

apt-get update
apt-get -y upgrade

apt-get -y autoremove
```

```
apt-get autoclean


echo -e "\n$(date "+%T") \t Script Terminated"
```

The script also takes care of old packages that are no longer needed. You need to run this script using sudo else it will not work properly.

Ending Thoughts

Linux shell scripts can be as diverse as you can imagine. There's literally no limit when it comes to determining what it can do or can't. If you're a new Linux enthusiast, we highly recommend you master these fundamental bash script examples. You should tweak them to understand how they work more clearly. We've tried our best to provide you with all the essential insights needed for modern Linux bash scripts. We've not touched on some technical matters due to the sake of simplicity. However, this guide should prove to be a great starting point for many of you.