

# 30 Bash Script Examples

3 years ago

by [Fahmida Yesmin](#)

Bash scripts can be used for various purposes, such as executing a shell command, running multiple commands together, customizing administrative tasks, performing task automation etc. So knowledge of bash programming basics is important for every Linux user. This article will help you to get the basic idea on bash programming. Most of the common operations of bash scripting are explained with very simple examples here.

The following topics of bash programming are covered in this article.

1. [Hello World](#)
2. [Echo Command](#)
3. [Comments](#)
4. [Multi-line comment](#)
5. [While Loop](#)
6. [For Loop](#)
7. [Get User Input](#)
8. [If statement](#)
9. [And Condition if statement](#)
10. [Or Condition if statement](#)
11. [Else if and else condition](#)
12. [Case Condition](#)
13. [Get Arguments from Command Line](#)
14. [Get arguments from command line with names](#)
15. [Combine two strings in a variable](#)
16. [Get Substring of Strings](#)
17. [Add 2 numbers into a variable](#)
18. [Create a Function](#)
19. [Use Function Parameters](#)
20. [Pass Return Value from Script](#)
21. [Make directory](#)
22. [Make directory by checking existence](#)
23. [Read a file](#)
24. [Delete a File](#)
25. [Append to file](#)
26. [Test if File Exists](#)
27. [Send Email Example](#)
28. [Get Parse Current Date](#)
29. [Wait Command](#)
30. [Sleep Command](#)

## Create and Execute First BASH Program:

You can run bash script from the terminal or by executing any bash file. Run the following command from the terminal to execute a very simple bash statement. The output of the command will be 'Hello World'.

```
$ echo "Hello World"
```

```
ubuntu@ubuntu-VirtualBox:~$ echo "Hello World"
Hello World
ubuntu@ubuntu-VirtualBox:~$
```

Open any editor to create a bash file. Here, **nano** editor is used to create the file and filename is set as 'First.sh'

```
$ nano First.sh
```

Add the following bash script to the file and save the file.

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

```
GNU nano 2.8.6                               File: First.sh                               Modified

#!/bin/bash
echo "Hello World"

^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text ^T To Linter ^_ Go To Line
```

You can run bash file by two ways. One way is by using bash command and another is by setting execute permission to bash file and run the file. Both ways are shown here.

```
$ bash First.sh
```

**Or,**

```
$ chmod a+x First.sh
```

```
$ ./First.sh
```

```
ubuntu@ubuntu-VirtualBox:~$ bash First.sh
Hello World
ubuntu@ubuntu-VirtualBox:~$ chmod a+x First.sh
ubuntu@ubuntu-VirtualBox:~$ ./First.sh
Hello World
ubuntu@ubuntu-VirtualBox:~$
```

[Go to top](#)

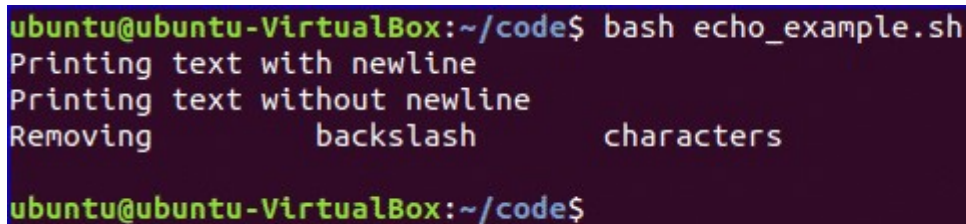
## Use of echo command:

You can use echo command with various options. Some useful options are mentioned in the following example. When you use ‘**echo**’ command without any option then a newline is added by default. ‘**-n**’ option is used to print any text without new line and ‘**-e**’ option is used to remove backslash characters from the output. Create a new bash file with a name, ‘**echo\_example.sh**’ and add the following script.

```
#!/bin/bash
echo "Printing text with newline"
echo -n "Printing text without newline"
echo -e "\nRemoving \t backslash \t characters\n"
```

Run the file with bash command.

```
$ bash echo_example.sh
```

A terminal window with a dark purple background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The command 'bash echo\_example.sh' has been entered. The output is: 'Printing text with newline', 'Printing text without newline', and 'Removing backslash characters' on three separate lines. The prompt returns to 'ubuntu@ubuntu-VirtualBox:~/code\$'.

[Go to top](#)

## Use of comment:

‘**#**’ symbol is used to add single line comment in bash script. Create a new file named ‘**comment\_example.sh**’ and add the following script with single line comment.

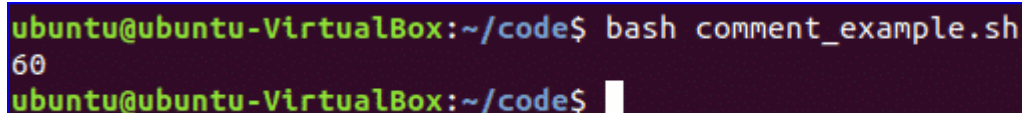
```
#!/bin/bash

# Add two numeric value
((sum=25+35))
```

```
#Print the result
echo $sum
```

Run the file with bash command.

```
$ bash comment_example.sh
```

A terminal window with a dark purple background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The command 'bash comment\_example.sh' has been entered. The output is '60' on a single line. The prompt returns to 'ubuntu@ubuntu-VirtualBox:~/code\$'.

[Go to top](#)

## Use of Multi-line comment:

You can use multi line comment in bash in various ways. A simple way is shown in the following example. Create a new bash named, ‘**multiline-comment.sh**’ and add the following script. Here, ‘**:**’

and “ ’ ” symbols are used to add multiline comment in bash script. This following script will calculate the square of 5.

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

```
: '
```

The following script calculates the square value of the number, 5.

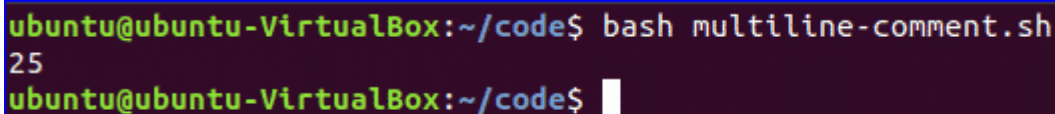
```
,
```

```
((area=5*5))
```

```
echo $area
```

Run the file with bash command.

```
$ bash multiline-comment.sh
```

A terminal window with a dark purple background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The command 'bash multiline-comment.sh' has been entered and executed, resulting in the output '25'. The prompt is now 'ubuntu@ubuntu-VirtualBox:~/code\$' followed by a cursor.

```
ubuntu@ubuntu-VirtualBox:~/code$ bash multiline-comment.sh
25
ubuntu@ubuntu-VirtualBox:~/code$
```

You can check the following link to know more about the use of bash comment.

[https://linuxhint.com/bash\\_comments/](https://linuxhint.com/bash_comments/)

[Go to top](#)

## Using While Loop:

Create a bash file with the name, ‘**while\_example.sh**’, to know the use of **while** loop. In the example, **while** loop will iterate for **5** times. The value of **count** variable will increment by **1** in each step. When the value of **count** variable will 5 then the **while** loop will terminate.

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

```
valid=true
```

```
count=1
```

```
while [ $valid ]
```

```
do
```

```
echo $count
```

```
if [ $count -eq 5 ];
```

```
then
```

```
break
```

```
fi
```

```
((count++))
```

```
done
```

Run the file with bash command.

```
$ bash while_example.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash while_example.sh
1
2
3
4
5
ubuntu@ubuntu-VirtualBox:~/code$
```

You can check the following link to know more about the use of while loop.

<https://linuxhint.com/bash-while-loop-examples/>

[Go to top](#)

## Using For Loop:

The basic **for** loop declaration is shown in the following example. Create a file named '**for\_example.sh**' and add the following script using **for** loop. Here, **for** loop will iterate for **10** times and print all values of the variable, **counter** in single line.

```
#!/bin/bash
for (( counter=10; counter>0; counter-- ))
do
echo -n "$counter "
done
printf "\n"
```

Run the file with bash command.

\$ bash for\_example.sh

```
ubuntu@ubuntu-VirtualBox:~/code$ bash for_example.sh
10 9 8 7 6 5 4 3 2 1
ubuntu@ubuntu-VirtualBox:~/code$
```

You can use for loop for different purposes and ways in your bash script. You can check the following link to know more about the use of for loop.

<https://linuxhint.com/bash-for-loop-examples/>

[Go to top](#)

## Get User Input:

'**read**' command is used to take input from user in bash. Create a file named '**user\_input.sh**' and add the following script for taking input from the user. Here, one string value will be taken from the user and display the value by combining other string value.

```
#!/bin/bash
echo "Enter Your Name"
read name
echo "Welcome $name to LinuxHint"
```

Run the file with bash command.

```
$ bash user_input.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash user_input.sh
Enter Your Name
Fahmida
Welcome Fahmida to LinuxHint
ubuntu@ubuntu-VirtualBox:~/code$
```

You can check the following link to know more about the use of user input.

<https://linuxhint.com/bash-script-user-input/>

[Go to top](#)

## Using if statement:

You can use if condition with single or multiple conditions. Starting and ending block of this statement is define by ‘if’ and ‘fi’. Create a file named ‘**simple\_if.sh**’ with the following script to know the use if statement in bash. Here, **10** is assigned to the variable, **n**. if the value of **\$n** is less than 10 then the output will be “**It is a one digit number**”, otherwise the output will be “**It is a two digit number**”. For comparison, ‘-lt’ is used here. For comparison, you can also use ‘-eq’ for equality, ‘-ne’ for not equality and ‘-gt’ for greater than in bash script.

```
#!/bin/bash
n=10
if [ $n -lt 10 ];
then
echo "It is a one digit number"
else
echo "It is a two digit number"
fi
```

Run the file with bash command.

```
$ bash simple_if.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash simple_if.sh
It is a two digit number
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

## Using if statement with AND logic:

Different types of logical conditions can be used in if statement with two or more conditions. How you can define multiple conditions in if statement using AND logic is shown in the following example. ‘&&’ is used to apply AND logic of if statement. Create a file named ‘**if\_with\_AND.sh**’ to check the following code. Here, the value of **username** and **password** variables will be taken from the user and compared with ‘**admin**’ and ‘**secret**’. If both values match then the output will be “**valid user**”, otherwise the output will be “**invalid user**”.

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

```
echo "Enter username"
read username
echo "Enter password"
read password
```

```
if [[ ( $username == "admin" && $password == "secret" ) ]]; then
echo "valid user"
else
echo "invalid user"
fi
```

Run the file with bash command.

```
$ bash if_with_AND.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash if_with_AND.sh
Enter username
admin
Enter password
1234
invalid user
ubuntu@ubuntu-VirtualBox:~/code$ bash if_with_AND.sh
Enter username
admin
Enter password
secret
valid user
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

## Using if statement with OR logic:

‘||’ is used to define **OR** logic in **if** condition. Create a file named ‘**if\_with\_OR.sh**’ with the following code to check the use of **OR** logic of **if** statement. Here, the value of **n** will be taken from the user. If the value is equal to **15** or **45** then the output will be “**You won the game**”, otherwise the output will be “**You lost the game**”.

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

```
echo "Enter any number"
read n
```

```
if [[ ( $n -eq 15 || $n -eq 45 ) ]]
then
echo "You won the game"
else
```

```
echo "You lost the game"
fi
```

Run the file with bash command.

```
$ bash if_with_OR.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash if_with_OR.sh
Enter any number
40
You lost the game
ubuntu@ubuntu-VirtualBox:~/code$ bash if_with_OR.sh
Enter any number
15
You won the game
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

### Using else if statement:

The use of **else if** condition is little different in bash than other programming language. '**elif**' is used to define **else if** condition in bash. Create a file named, '**elseif\_example.sh**' and add the following script to check how **else if** is defined in bash script.

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

```
echo "Enter your lucky number"
read n
```

```
if [ $n -eq 101 ];
then
echo "You got 1st prize"
elif [ $n -eq 510 ];
then
echo "You got 2nd prize"
elif [ $n -eq 999 ];
then
echo "You got 3rd prize"
```

```
else
echo "Sorry, try for the next time"
fi
```

Run the file with bash command.

```
$ bash elseif_example.sh
```



```
ubuntu@ubuntu-VirtualBox:~/code$ bash elseif_example.sh
Enter your lucky number
101
You got 1st prize
ubuntu@ubuntu-VirtualBox:~/code$ bash elseif_example.sh
Enter your lucky number
999
You got 3rd prize
ubuntu@ubuntu-VirtualBox:~/code$ bash elseif_example.sh
Enter your lucky number
100
Sorry, try for the next time
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

## Using Case Statement:

**Case** statement is used as the alternative of **if-elseif-else** statement. The starting and ending block of this statement is defined by '**case**' and '**esac**'. Create a new file named, '**case\_example.sh**' and add the following script. The output of the following script will be same to the previous **else if** example.

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

```
echo "Enter your lucky number"
read n
case $n in
101)
echo echo "You got 1st prize" ;;
510)
echo "You got 2nd prize" ;;
999)
echo "You got 3rd prize" ;;
*)
echo "Sorry, try for the next time" ;;
esac
```

Run the file with bash command.

```
$ bash case_example.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash case_example.sh
Enter your lucky number
101
echo You got 1st prize
ubuntu@ubuntu-VirtualBox:~/code$ bash case_example.sh
Enter your lucky number
510
You got 2nd prize
ubuntu@ubuntu-VirtualBox:~/code$ bash case_example.sh
Enter your lucky number
999
You got 3rd prize
ubuntu@ubuntu-VirtualBox:~/code$ bash case_example.sh
Enter your lucky number
777
Sorry, try for the next time
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

## Get Arguments from Command Line:

Bash script can read input from command line argument like other programming language. For example, \$1 and \$2 variable are used to read first and second command line arguments. Create a file named “**command\_line.sh**” and add the following script. Two argument values read by the following script and prints the total number of arguments and the argument values as output.

```
#!/bin/bash
echo "Total arguments : $#\"
echo "1st Argument = $1\"
echo "2nd argument = $2\"
```

Run the file with bash command.

\$ bash command\_line.sh Linux Hint

```
ubuntu@ubuntu-VirtualBox:~/code$ bash command_line.sh Linux Hint
Total arguments : 2
1st Argument = Linux
2nd argument = Hint
ubuntu@ubuntu-VirtualBox:~/code$
```

You can check the following link to know more about the use of command line argument.

[https://linuxhint.com/command\\_line\\_arguments\\_bash\\_script/](https://linuxhint.com/command_line_arguments_bash_script/)

[Go to top](#)

## Get arguments from command line with names:

How you can read command line arguments with names is shown in the following script. Create a file named, ‘**command\_line\_names.sh**’ and add the following code. Here, two arguments, **X** and **Y** are read by this script and print the sum of X and Y.

```
#!/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"
```

Run the file with bash command and with two command line arguments.

```
$ bash command_line_names X=45 Y=30
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash command_line_names.sh X=45 Y=30
X+Y=75
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

## Combine String variables:

You can easily combine string variables in bash. Create a file named “**string\_combine.sh**” and add the following script to check how you can combine string variables in bash by placing variables together or using ‘+’ operator.

```
#!/bin/bash

string1="Linux"
string2="Hint"
echo "$string1$string2"
string3=$string1+$string2
string3+=" is a good tutorial blog site"
echo $string3
```

Run the file with bash command.

```
$ bash string_combine.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash string_combine.sh
LinuxHint
Linux+Hint is a good tutorial blog site
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

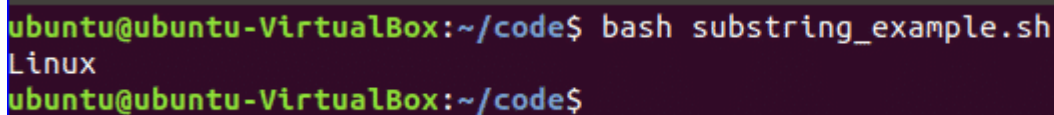
## Get substring of String:

Like other programming language, bash has no built-in function to cut value from any string data. But you can do the task of substring in another way in bash that is shown in the following script. To test the script, create a file named '**substring\_example.sh**' with the following code. Here, the value, **6** indicates the starting point from where the substring will start and **5** indicates the length of the substring.

```
#!/bin/bash
Str="Learn Linux from LinuxHint"
subStr=${Str:6:5}
echo $subStr
```

Run the file with bash command.

```
$ bash substring_example.sh
```

A terminal window with a dark purple background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The user enters 'bash substring\_example.sh'. The output is 'Linux'. The prompt returns to 'ubuntu@ubuntu-VirtualBox:~/code\$'.

[Go to top](#)

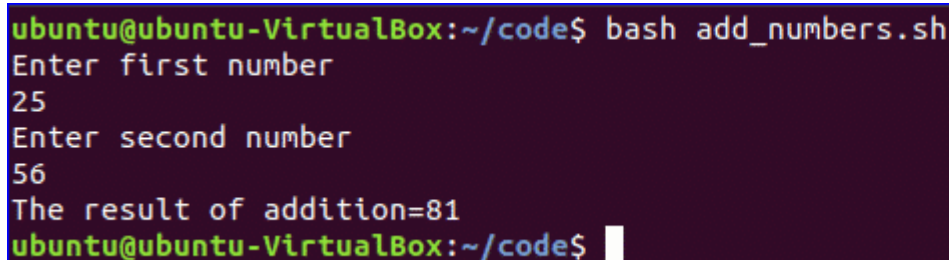
## Add Two Numbers:

You can do the arithmetical operations in bash in different ways. How you can add two integer numbers in bash using double brackets is shown in the following script. Create a file named '**add\_numbers.sh**' with the following code. Two integer values will be taken from the user and printed the result of addition.

```
#!/bin/bash
echo "Enter first number"
read x
echo "Enter second number"
read y
(( sum=x+y ))
echo "The result of addition=$sum"
```

Run the file with bash command.

```
$ bash add_numbers.sh
```

A terminal window with a dark purple background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The user enters 'bash add\_numbers.sh'. The script prompts 'Enter first number' and the user enters '25'. It then prompts 'Enter second number' and the user enters '56'. The output is 'The result of addition=81'. The prompt returns to 'ubuntu@ubuntu-VirtualBox:~/code\$'.

You can check the following link to know more about bash arithmetic.

[https://linuxhint.com/bash\\_arithmetic\\_operations/](https://linuxhint.com/bash_arithmetic_operations/)

[Go to top](#)

## Create Function:

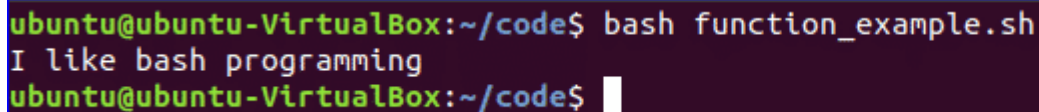
How you can create a simple function and call the function is shown in the following script. Create a file named '**function\_example.sh**' and add the following code. You can call any function by name only without using any bracket in bash script.

```
#!/bin/bash
function F1()
{
echo 'I like bash programming'
}
```

F1

Run the file with bash command.

\$ bash function\_example.sh

A terminal window with a dark purple background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The command 'bash function\_example.sh' has been entered and executed. The output is 'I like bash programming'. The prompt is now 'ubuntu@ubuntu-VirtualBox:~/code\$' followed by a cursor.

```
ubuntu@ubuntu-VirtualBox:~/code$ bash function_example.sh
I like bash programming
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

## Create function with Parameters:

Bash can't declare function parameter or arguments at the time of function declaration. But you can use parameters in function by using other variable. If two values are passed at the time of function calling then \$1 and \$2 variable are used for reading the values. Create a file named '**function\_parameter.sh**' and add the following code. Here, the function, '**Rectangle\_Area**' will calculate the area of a rectangle based on the parameter values.

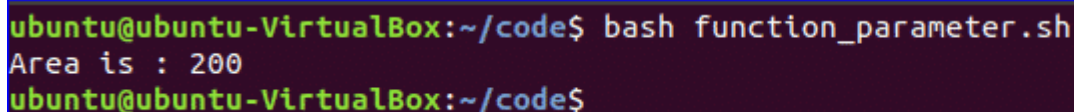
```
#!/bin/bash

Rectangle_Area() {
area=$(( $1 * $2 ))
echo "Area is : $area"
}
```

Rectangle\_Area 10 20

Run the file with bash command.

\$ bash function\_parameter.sh

A terminal window with a dark purple background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The command 'bash function\_parameter.sh' has been entered and executed. The output is 'Area is : 200'. The prompt is now 'ubuntu@ubuntu-VirtualBox:~/code\$' followed by a cursor.

```
ubuntu@ubuntu-VirtualBox:~/code$ bash function_parameter.sh
Area is : 200
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

## Pass Return Value from Function:

Bash function can pass both numeric and string values. How you can pass a string value from the function is shown in the following example. Create a file named, '**function\_return.sh**' and add the following code. The function, **greeting()** returns a string value into the variable, **val** which prints later by combining with other string.

```
#!/bin/bash
function greeting() {

str="Hello, $name"
echo $str

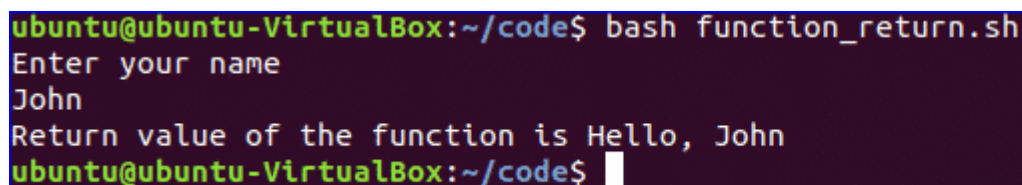
}

echo "Enter your name"
read name

val=$(greeting)
echo "Return value of the function is $val"
```

Run the file with bash command.

```
$ bash function_return.sh
```



```
ubuntu@ubuntu-VirtualBox:~/code$ bash function_return.sh
Enter your name
John
Return value of the function is Hello, John
ubuntu@ubuntu-VirtualBox:~/code$
```

You can check the following link to know more about the use of bash function.

<https://linuxhint.com/return-string-bash-functions/>

[Go to top](#)

## Make Directory:

Bash uses '**mkdir**' command to create a new directory. Create a file named '**make\_directory.sh**' and add the following code to take a new directory name from the user. If the directory name is not exist in the current location then it will create the directory, otherwise the program will display error.

```
#!/bin/bash
echo "Enter directory name"
read newdir
`mkdir $newdir`
```

Run the file with bash command.

```
$ bash make_directory.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code/temp$ bash make_directory.sh
Enter directory name
test_dir
ubuntu@ubuntu-VirtualBox:~/code/temp$ ls
make_directory.sh  test_dir
ubuntu@ubuntu-VirtualBox:~/code/temp$
```

[Go to top](#)

## Make directory by checking existence:

If you want to check the existence of directory in the current location before executing the **'mkdir'** command then you can use the following code. **'-d'** option is used to test a particular directory is exist or not. Create a file named, **'directory\_exist.sh'** and add the following code to create a directory by checking existence.

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

Run the file with bash command.

```
$ bash directory_exist.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code/temp$ bash directory_exist.sh
Enter directory name
newdir
Directory created
ubuntu@ubuntu-VirtualBox:~/code/temp$ ls
directory_exist.sh  make_directory.sh  newdir  test_dir
ubuntu@ubuntu-VirtualBox:~/code/temp$
```

You can check the following link to know more about directory creation.

[https://linuxhint.com/bash\\_mkdir\\_not\\_existent\\_path/](https://linuxhint.com/bash_mkdir_not_existent_path/)

[Go to top](#)

## Read a File:

You can read any file line by line in bash by using loop. Create a file named, **'read\_file.sh'** and add the following code to read an existing file named, **'book.txt'**.

```
#!/bin/bash
file='book.txt'
while read line; do
```



```
echo $line  
done < $file
```

Run the file with bash command.

```
$ bash read_file.sh
```

Run the following command to check the original content of '**book.txt**' file.

```
$ cat book.txt
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash read_file.sh  
1. Pro AngularJS  
2. Learning JQuery  
3. PHP Programming  
4. CodeIgniter 3  
ubuntu@ubuntu-VirtualBox:~/code$ cat book.txt  
1. Pro AngularJS  
2. Learning JQuery  
3. PHP Programming  
4. CodeIgniter 3  
ubuntu@ubuntu-VirtualBox:~/code$
```

You can check the following link to know the different ways to read file.

[https://linuxhint.com/read\\_file\\_line\\_by\\_line\\_bash/](https://linuxhint.com/read_file_line_by_line_bash/)

[Go to top](#)

## Delete a File:

'**rm**' command is used in bash to remove any file. Create a file named '**delete\_file.sh**' with the following code to take the filename from the user and remove. Here, '**-i**' option is used to get permission from the user before removing the file.

```
#!/bin/bash  
echo "Enter filename to remove"  
read fn  
rm -i $fn
```

Run the file with bash command.

```
$ ls  
$ bash delete_file.sh  
$ ls
```



```

ubuntu@ubuntu-VirtualBox:~/code/temp$ ls
delete_file.sh      make_directory.sh  test_dir
directory_exist.sh  newdir             test.txt
ubuntu@ubuntu-VirtualBox:~/code/temp$ bash delete_file.sh
Enter filename to remove
test.txt
rm: remove regular file 'test.txt'? y
ubuntu@ubuntu-VirtualBox:~/code/temp$ ls
delete_file.sh  directory_exist.sh  make_directory.sh  newdir  test_dir
ubuntu@ubuntu-VirtualBox:~/code/temp$

```

[Go to top](#)

## Append to File:

New data can be added into any existing file by using ‘>>’ operator in bash. Create a file named ‘**append\_file.sh**’ and add the following code to add new content at the end of the file. Here, ‘**Learning Laravel 5**’ will be added at the of ‘**book.txt**’ file after executing the script.

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

```
echo "Before appending the file"
cat book.txt
```

```
echo "Learning Laravel 5">> book.txt
echo "After appending the file"
cat book.txt
```

Run the file with bash command.

```
$ bash append_file.sh
```

```

ubuntu@ubuntu-VirtualBox:~/code/temp$ bash append_file.sh
Before appending the file
1. Pro AngularJS
2. Learning JQuery
3. PHP Programming
4. CodeIgniter 3
After appending the file
1. Pro AngularJS
2. Learning JQuery
3. PHP Programming
4. CodeIgniter 3
Learning Laravel 5
ubuntu@ubuntu-VirtualBox:~/code/temp$

```

[Go to top](#)

## Test if File Exist:

You can check the existence of file in bash by using ‘-e’ or ‘-f’ option. ‘-f’ option is used in the following script to test the file existence. Create a file named, ‘**file\_exist.sh**’ and add the following code. Here, the filename will pass from the command line.

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

Run the following commands to check the existence of the file. Here, **book.txt** file exists and **book2.txt** is not exist in the current location.

```
$ ls
$ bash file_exist.sh book.txt
$ bash file_exist.sh book2.txt
```

```
ubuntu@ubuntu-VirtualBox:~/code/temp$ ls
append_file.sh  delete_file.sh  file_exist.sh  newdir
book.txt        directory_exist.sh  make_directory.sh  test_dir
ubuntu@ubuntu-VirtualBox:~/code/temp$ bash file_exist.sh book.txt
File exists
ubuntu@ubuntu-VirtualBox:~/code/temp$ bash file_exist.sh book2.txt
File does not exist
ubuntu@ubuntu-VirtualBox:~/code/temp$
```

[Go to top](#)

## Send Email:

You can send email by using ‘**mail**’ or ‘**sendmail**’ command. Before using these commands, you have to install all necessary packages. Create a file named, ‘**mail\_example.sh**’ and add the following code to send the email.

```
#!/bin/bash
Recipient="admin@example.com"
Subject="Greeting"
Message="Welcome to our site"
`mail -s $Subject $Recipient <<< $Message`
```

Run the file with bash command.

```
$ bash mail_example.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash mail_example.sh
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

## Get Parse Current Date:

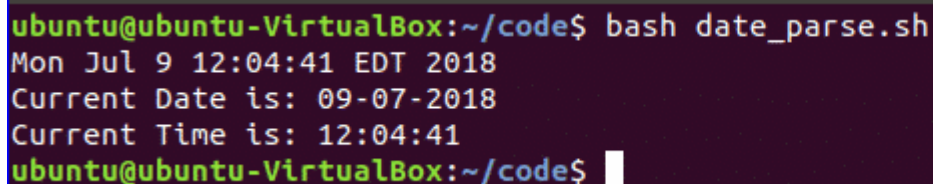
You can get the current system date and time value using ‘**date**’ command. Every part of date and time value can be parsed using ‘**Y**’, ‘**m**’, ‘**d**’, ‘**H**’, ‘**M**’ and ‘**S**’. Create a new file named

'**date\_parse.sh**' and add the following code to separate day, month, year, hour, minute and second values.

```
#!/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 the file with bash command.

\$ bash date\_parse.sh

A terminal window with a dark purple background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The user has entered 'bash date\_parse.sh'. The output shows the current date and time in various formats: 'Mon Jul 9 12:04:41 EDT 2018', 'Current Date is: 09-07-2018', and 'Current Time is: 12:04:41'. The prompt is now 'ubuntu@ubuntu-VirtualBox:~/code\$' with a cursor.

```
ubuntu@ubuntu-VirtualBox:~/code$ bash date_parse.sh
Mon Jul 9 12:04:41 EDT 2018
Current Date is: 09-07-2018
Current Time is: 12:04:41
ubuntu@ubuntu-VirtualBox:~/code$
```

[Go to top](#)

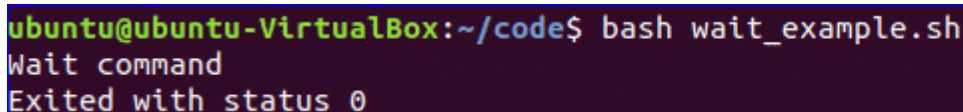
## Wait Command:

**wait** is a built-in command of Linux that waits for completing any running process. **wait** command is used with a particular process id or job id. If no process id or job id is given with wait command then it will wait for all current child processes to complete and returns exit status. Create a file named '**wait\_example.sh**' and add the following script.

```
#!/bin/bash
echo "Wait command" &
process_id=$!
wait $process_id
echo "Exited with status $?"
```

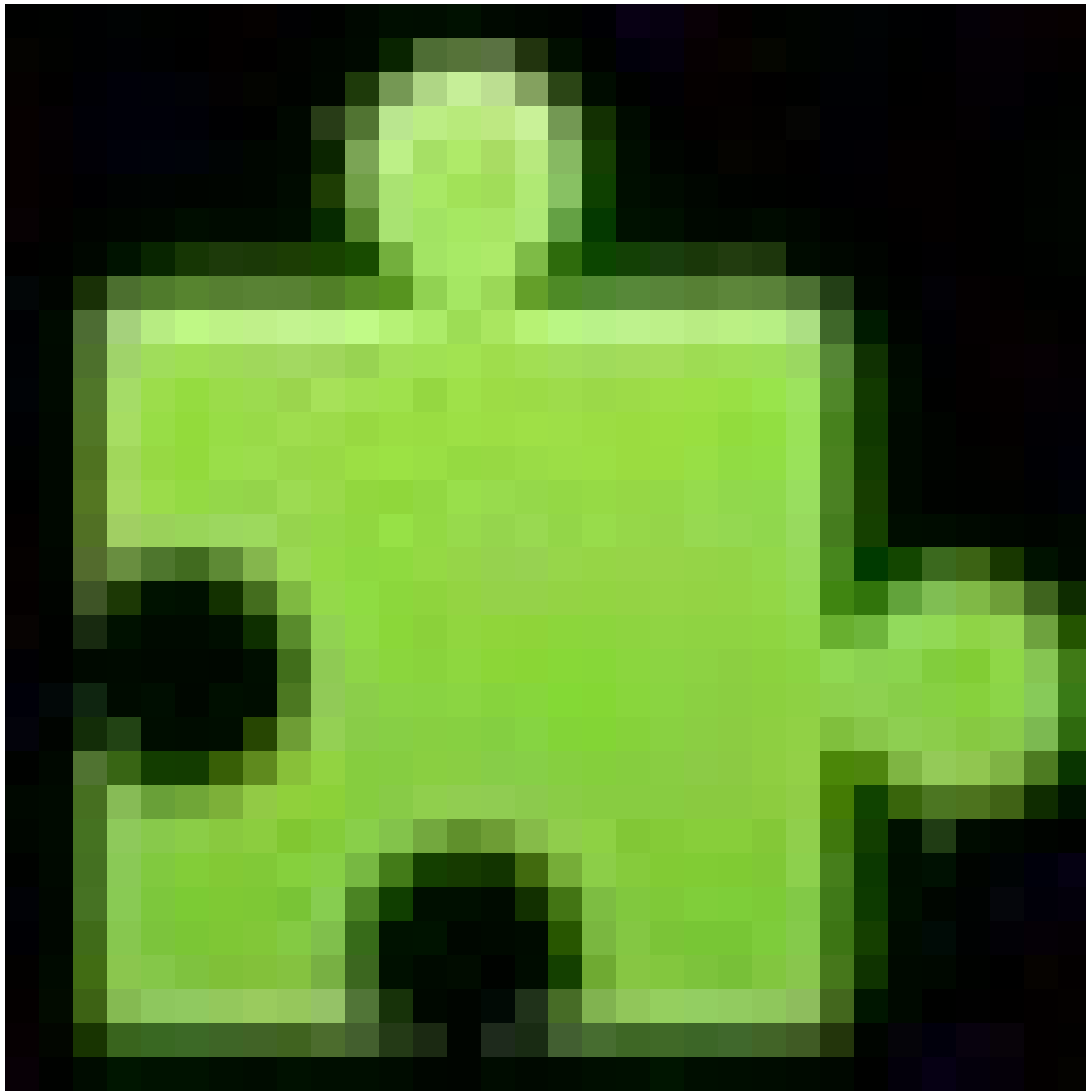
Run the file with bash command.

\$ bash wait\_example.sh

A terminal window with a dark purple background. The prompt is 'ubuntu@ubuntu-VirtualBox:~/code\$'. The user has entered 'bash wait\_example.sh'. The output shows 'Wait command' and 'Exited with status 0'. The prompt is now 'ubuntu@ubuntu-VirtualBox:~/code\$' with a cursor.

```
ubuntu@ubuntu-VirtualBox:~/code$ bash wait_example.sh
Wait command
Exited with status 0
ubuntu@ubuntu-VirtualBox:~/code$
```

You can check the following link to know more about wait command.



[Go to top](#)

### **Sleep Command:**

When you want to pause the execution of any command for specific period of time then you can use **sleep** command. You can set the delay amount by **seconds (s)**, **minutes (m)**, **hours (h)** and **days (d)**. Create a file named '**sleep\_example.sh**' and add the following script. This script will wait for 5 seconds after running.

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

```
echo "Wait for 5 seconds"
```

```
sleep 5
```

```
echo "Completed"
```

Run the file with bash command.

```
$ bash sleep_example.sh
```

```
ubuntu@ubuntu-VirtualBox:~/code$ bash sleep_example.sh
Wait for 5 seconds
Completed
ubuntu@ubuntu-VirtualBox:~/code$
```

You can check the following link to know more about sleep command.

[https://linuxhint.com/sleep\\_command\\_linux/](https://linuxhint.com/sleep_command_linux/)

[Go to top](#)

Hope, after reading this article you have got a basic concept on bash scripting language and you will be able to apply them based on your requirements.

[https://linuxhint.com/30\\_bash\\_script\\_examples/](https://linuxhint.com/30_bash_script_examples/)

# Shell Script Examples

This section presents several shell script examples.

## Hello World

### Example 9. Hello World

```
#!/bin/sh
echo "Hello world"
```

## Using Arguments

### Example 10. Shell Script Arguments

```
#!/bin/bash

# example of using arguments to a script
echo "My first name is $1"
echo "My surname is $2"
echo "Total number of arguments is $#"
```

Save this file as `name.sh`, set execute permission on that file by typing **`chmod a+x name.sh`** and then execute the file like this: **`./name.sh`**.

```
$ chmod a+x name.sh
$ ./name.sh Hans-Wolfgang Loidl
My first name is Hans-Wolfgang
My surname is Loidl
Total number of arguments is 2
```

## Version 1: Line count example

The first example simply counts the number of lines in an input file. It does so by iterating over all lines of a file using a **while** loop, performing a **read** operation in the loop header. While there is a line to process, the loop body will be executed in this case simply increasing a counter by **((counter++))**. Additionally the current line is written into a file, whose name is specified by the variable `file`, by echoing the value of the variable `line` and redirecting the standard output of the variable to **\$file**. the current line to file. The latter is not needed for the line count, of course, but demonstrates how to check for success of an operation: the special variable **\$?** will contain the return code from the previous command (the redirected **echo**). By Unix convention, success is indicated by a return code of 0, all other values are error code with application specific meaning.

Another important issue to consider is that the integer variable, over which iteration is performed should always *count down* so that the analysis can find a bound. This might require some restructuring of the code, as in the following example, where an explicit counter `Z` is introduced for this purpose. After the loop, the line count and the contents of the last line are printed, using **echo**. Of course, there is a Linux command that already implements line-count functionality: **wc** (for word-count) prints, when called with option **-l**, the number of lines in the file. We use this to check whether our line count is correct, demonstrating numeric operations on the way.

```
#!/bin/bash
# Simple line count example, using bash
#
# Bash tutorial: http://linuxconfig.org/Bash\_scripting\_Tutorial#8-2-read-file-into-bash-a
# My scripting link: http://www.macs.hw.ac.uk/~hwloidl/docs/index.html#scripting
#
# Usage: ./line_count.sh file
# -----

# Link filedescriptor 10 with stdin
exec 10<&0
# stdin replaced with a file supplied as a first argument
exec < $1
# remember the name of the input file
in=$1

# init
file="current_line.txt"
let count=0

# this while loop iterates over all lines of the file
while read LINE
do
    # increase line counter
    ((count++))
    # write current line to a tmp file with name $file (not needed for counting)
    echo $LINE > $file
    # this checks the return code of echo (not needed for writing; just for demo)
    if [ $? -ne 0 ]
    then echo "Error in writing to file ${file}; check its permissions!"
    fi
done

echo "Number of lines: $count"
echo "The last line of the file is: `cat ${file}`"

# Note: You can achieve the same by just using the tool wc like this
echo "Expected number of lines: `wc -l $in`"

# restore stdin from filedescriptor 10
# and close filedescriptor 10
exec 0<&10 10<&-
```

As documented at the start of the script, it is called like this (you must have a file `text_file.txt` in your current directory):

```
$ ./line_count.sh text_file.txt
```



### Sample text file

You can get a sizable sample text file by typing:

```
$ cp /home/msc/public/LinuxIntro/WaD.txt text_file.txt
```

## Several versions of line counting across a set of files

This section develops several shell scripts, each counting the total number of lines across a set of files. These examples elaborate specific shell features. For counting the number of lines in one file

we use **wc -l**. As a simple exercise you can replace this command with a call to the line counting script above.

## Version 1: Explicit For loop

We use a for-loop to iterate over all files provided as arguments to the script. We can access all arguments through the variable `$*`. The sed command matches the line count, and replaces the entire line with just the line count, using the back reference to the first substring (`\1`). In the for-loop, the shell variable `n` is a counter for the number of files, and `s` is the total line count so far.

```
#!/bin/bash
# Counting the number of lines in a list of files
# for loop over arguments

if [ $# -lt 1 ]
then
    echo "Usage: $0 file ..."
    exit 1
fi

echo "$0 counts the lines of code"
l=0
n=0
s=0
for f in $*
do
    l=`wc -l $f | sed 's/^\[0-9]*\).*$/\1/'`
    echo "$f: $l"
    n=$((n + 1))
    s=$((s + l))
done

echo "$n files in total, with $s lines in total"
```

## Version 2: Using a Shell Function

In this version we define a function **count\_lines** that counts the number of lines in the file provided as argument. Inside the function the value of the argument is retrieved by accessing the variable `$1`.

```
#!/bin/bash
# Counting the number of lines in a list of files
# function version

count_lines () {
    local f=$1
    # this is the return value, i.e. non local
    l=`wc -l $f | sed 's/^\[0-9]*\).*$/\1/'`
}

if [ $# -lt 1 ]
then
    echo "Usage: $0 file ..."
    exit 1
fi

echo "$0 counts the lines of code"
l=0
n=0
s=0
while [ "$*" != "" ]
```



```
do
    count_lines $1
    echo "$1: $l"
    n=[ $n + 1 ]
    s=[ $s + $l ]
    shift
done

echo "$n files in total, with $s lines in total"
```

### Version 3: Using a return code in a function

This version tries to use the return value of the function to return the line count. However, this fails on files with more than 255 lines. The return value is intended to just provide a return code, e.g. 0 for success, 1 for failure, but not for returning proper values.

```
#!/bin/bash
# Counting the number of lines in a list of files
# function version using return code
# WRONG version: the return code is limited to 0-255
# so this script will run, but print wrong values for
# files with more than 255 lines

count_lines () {
    local f=$1
    local m
    m=`wc -l $f | sed 's/^\[0-9]*\).*$/\1/'`
    return $m
}

if [ $# -lt 1 ]
then
    echo "Usage: $0 file ..."
    exit 1
fi

echo "$0 counts the lines of code"
l=0
n=0
s=0
while [ "$*" != "" ]
do
    count_lines $1
    l=$?
    echo "$1: $l"
    n=[ $n + 1 ]
    s=[ $s + $l ]
    shift
done

echo "$n files in total, with $s lines in total"
```

### Version 4: Generating the file list in a shell function

```
#!/bin/bash
# Counting the number of lines in a list of files
# function version

# function storing list of all files in variable files
```

```

get_files () {
    files="\ls *. [ch]`"
}

# function counting the number of lines in a file
count_lines () {
    local f=$1 # 1st argument is filename
    l=`wc -l $f | sed 's/^([0-9]*\).*$/\1/'` # number of lines
}

# the script should be called without arguments
if [ $# -ge 1 ]
then
    echo "Usage: $0 "
    exit 1
fi

# split by newline
IFS=$'\012'

echo "$0 counts the lines of code"
# don't forget to initialise!
l=0
n=0
s=0
# call a function to get a list of files
get_files
# iterate over this list
for f in $files
do
    # call a function to count the lines
    count_lines $f
    echo "$f: $l"
    # increase counter
    n=$(( n + 1 ))
    # increase sum of all lines
    s=$(( s + $l ))
done

echo "$n files in total, with $s lines in total"

```

## Version 5: Using an array to store all line counts

The example below uses shell arrays to store all filenames (`file`) and its number of lines (`line`). The elements in an array are referred to using the usual `[ ]` notation, e.g. `file[1]` refers to the first element in the array `file`. Note, that bash only supports 1-dimensional arrays with integers as indices.

See [the section on arrays in the Advanced Bash-Scripting Guide](#).

```

#!/bin/bash
# Counting the number of lines in a list of files
# function version

# function storing list of all files in variable files
get_files () {
    files="\ls *. [ch]`"
}

# function counting the number of lines in a file
count_lines () {

```

```

f=$1 # 1st argument is filename
l=`wc -l $f | sed 's/^\[0-9\]*\).*$/\1/'` # number of lines
}

# the script should be called without arguments
if [ $# -ge 1 ]
then
    echo "Usage: $0 "
    exit 1
fi

# split by newline
IFS=$'\012'

echo "$0 counts the lines of code"
# don't forget to initialise!
l=0
n=0
s=0
# call a function to get a list of files
get_files
# iterate over this list
for f in $files
do
    # call a function to count the lines
    count_lines $f
    echo "$f: $l"
    # store filename in an array
    file[$n]=$f
    # store number of lines in an array
    lines[$n]=$l
    # increase counter
    n=$((n + 1))
    # increase sum of all lines
    s=$((s + $l))
done

echo "$n files in total, with $s lines in total"
i=5
echo "The $i-th file was ${file[$i]} with ${lines[$i]} lines"

```

## Version 6: Count only files we own

```

#!/bin/bash
# Counting the number of lines in a list of files
# for loop over arguments
# count only those files I am owner of

if [ $# -lt 1 ]
then
    echo "Usage: $0 file ..."
    exit 1
fi

echo "$0 counts the lines of code"
l=0
n=0
s=0
for f in $*
do
    if [ -o $f ] # checks whether file owner is running the script

```

```

then
    l=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'`
    echo "$f: $l"
    n=$((n + 1))
    s=$((s + l))
else
    continue
fi
done

echo "$n files in total, with $s lines in total"

```

## Version 7: Line count over several files

The final example supports options that can be passed from the command-line, e.g. by **./loc7.sh -d 1** **loc7.sh**. The `getopts` shell function is used to iterate over all options (given in the following string) and assigning the current option to variable `name`. Typically it is used in a while loop, to set shell variables that will be used later. We use a pipe of **cat** and **awk** to print the header of this file, up to the first empty line, if the help option is chosen. The main part of the script is a for loop over all non-option command-line arguments. In each iteration, `$f` contains the name of the file to process. If the date options are used to narrow the scope of files to process, we use the **date** and an if-statement, to compare whether the modification time of the file is within the specified interval. Only in this case do we count the number of lines as before. After the loop, we print the total number of lines and the number of files that have been processed.

### Example 11. Version 7: Line count over several files

```

#!/bin/bash
#####
#
# Usage: loc7.sh [options] file ...
#
# Count the number of lines in a given list of files.
# Uses a for loop over all arguments.
#
# Options:
# -h      ... help message
# -d n ... consider only files modified within the last n days
# -w n ... consider only files modified within the last n weeks
#
# Limitations:
# . only one option should be given; a second one overrides
#
#####

help=0
verb=0
weeks=0
# defaults
days=0
m=1
str="days"
getopts "hvd:w:" name
while [ "$name" != "?" ] ; do
    case $name in
        h) help=1;;
        v) verb=1;;

```

```

    d) days=$OPTARG
        m=$OPTARG
        str="days";;
    w) weeks=$OPTARG
        m=$OPTARG
        str="weeks";;
esac
getopts "hvd:w:" name
done

if [ $help -eq 1 ]
then no_of_lines=`cat $0 | awk 'BEGIN { n = 0; } \
                                /^$/ { print n; \
                                    exit; } \
                                { n++; }'`

    echo "`head - $no_of_lines $0`"
    exit
fi

shift $[ $OPTIND - 1 ]

if [ $# -lt 1 ]
then
    echo "Usage: $0 file ..."
    exit 1
fi

if [ $verb -eq 1 ]
then echo "$0 counts the lines of code"
fi

l=0
n=0
s=0
for f in $*
do
    x=`stat -c "%y" $f`
    # modification date
    d=`date --date="$x" +%y%m%d`
    # date of $m days/weeks ago
    e=`date --date="$m $str ago" +%y%m%d`
    # now
    z=`date +%y%m%d`
    #echo "Stat: $x; Now: $z; File: $d; $m $str ago: $e"
    # checks whether file is more recent then req
    if [ $d -ge $e -a $d -le $z ] # ToDo: fix year wrap-arounds
    then
        # be verbose if we found a recent file
        if [ $verb -eq 1 ]
        then echo "$f: modified (mmdd) $d"
        fi
        # do the line count
        l=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'`
        echo "$f: $l"
        # increase the counters
        n=$((n + 1))
        s=$((s + $l))
    else
        # not strictly necessary, because it's the end of the loop
        continue
    fi
done

```

```
echo "$n files in total, with $s lines in total"
```



### Exercise

Extend Version 7 of the line count example above, to also compute the total number of bytes and the total number of words in the input file.

```
$ ./loc8.sh text_file.txt loc8.sh  
2 files in total, with 1494438 bytes, 746 words, 27846 lines in total
```

Hint: check the man page for `wc`.

<https://www.macs.hw.ac.uk/~hwloidl/Courses/LinuxIntro/x984.html>

# 40 Simple Yet Effective Linux Shell Script Examples

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, Tesh, 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 over 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. 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. Our editors have also tried to outline some shell script examples that deal with shells other than bash. 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 for gaining first-hand experience.

#### 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. 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. 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. 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. 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 are 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 read 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. 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. 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. 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 quick 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

Contrary to many programming languages, bash doesn't provide any in-built function for cutting portions of a string. 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 from 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](#).

## 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 inside 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 "Addition 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 features of functions is they allow 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
```

If you look closely, this script simply calls your standard shell command `mkdir` and passes it the directory name. 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. 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. 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. 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.



### **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 is provided by the user in this case.

### **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. You can simply copy this code for getting 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. You need to pass the directory name where you want 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 find out whether a user is 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. 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.

### **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 to 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.

<https://www.ubuntupit.com/simple-yet-effective-linux-shell-script-examples/>

# Positional Parameters

When we last left our script, it looked something like this:

```
#!/bin/bash # sysinfo_page - A script to produce a system information HTML file ##### Constants
TITLE="System Information for $HOSTNAME" RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER" ##### Functions system_info() { echo
"<h2>System release info</h2>" echo "<p>Function not yet implemented</p>" } # end of
system_info show_uptime() { echo "<h2>System uptime</h2>" echo "<pre>" uptime echo
"</pre>" } # end of show_uptime drive_space() { echo "<h2>Filesystem space</h2>" echo
"<pre>" df echo "</pre>" } # end of drive_space home_space() { # Only the superuser can get this
information if [ "$(id -u)" = "0" ]; then echo "<h2>Home directory space by user</h2>"
echo "<pre>" echo "Bytes Directory" du -s /home/* | sort -nr echo "</pre>" fi } # end of
home_space ##### Main cat <<- _EOF_ <html> <head> <title>$TITLE</title> </head> <body>
<h1>$TITLE</h1> <p>$TIME_STAMP</p> $(system_info) $(show_uptime) $(drive_space) $
(home_space) </body> </html> _EOF_
```

We have most things working, but there are several more features we can add:

1. We should be able to specify the name of the output file on the command line, as well as set a default output file name if no name is specified.
2. Let's offer an interactive mode that will prompt for a file name and warn the user if the file exists and prompt the user to overwrite it.
3. Naturally, we want to have a help option that will display a usage message.

All of these features involve using command line options and arguments. To handle options on the command line, we use a facility in the shell called *positional parameters*. Positional parameters are a series of special variables (\$0 through \$9) that contain the contents of the command line.

Let's imagine the following command line:

```
[me@linuxbox me]$ some_program word1 word2 word3
```

If `some_program` were a bash shell script, we could read each item on the command line because the positional parameters contain the following:

- \$0 would contain "some\_program"
- \$1 would contain "word1"
- \$2 would contain "word2"
- \$3 would contain "word3"

Here is a script we can use to try this out:

```
#!/bin/bash echo "Positional Parameters" echo '$0 = ' $0 echo '$1 = ' $1 echo '$2 = ' $2 echo
'$3 = ' $3
```

## Detecting Command Line Arguments

Often, we will want to check to see if we have command line arguments on which to act. There are a couple of ways to do this. First, we could simply check to see if `$1` contains anything like so:

```
#!/bin/bash if [ "$1" != "" ]; then echo "Positional parameter 1 contains something" else
echo "Positional parameter 1 is empty" fi
```

Second, the shell maintains a variable called  `$#`  that contains the number of items on the command line in addition to the name of the command (`$0`).

```
#!/bin/bash if [ $# -gt 0 ]; then echo "Your command line contains $# arguments" else
echo "Your command line contains no arguments" fi
```

## Command Line Options

As we discussed before, many programs, particularly ones from [the GNU Project](#), support both short and long command line options. For example, to display a help message for many of these programs, we may use either the `-h` option or the longer `--help` option. Long option names are typically preceded by a double dash. We will adopt this convention for our scripts.

Here is the code we will use to process our command line:

```
interactive= filename=~ /sysinfo_page.html while [ "$1" != "" ]; do case $1 in -f | --file )
shift filename="$1" ;; -i | --interactive ) interactive=1 ;; -h | --help ) usage exit ;; * ) usage
exit 1 esac shift done
```

This code is a little tricky, so we need to explain it.

The first two lines are pretty easy. We set the variable `interactive` to be empty. This will indicate that the interactive mode has not been requested. Then we set the variable `filename` to contain a default file name. If nothing else is specified on the command line, this file name will be used.

After these two variables are set, we have default settings, in case the user does not specify any options.

Next, we construct a `while` loop that will cycle through all the items on the command line and process each one with `case`. The `case` will detect each possible option and process it accordingly.

Now the tricky part. How does that loop work? It relies on the magic of `shift`.

`shift` is a shell builtin that operates on the positional parameters. Each time we invoke `shift`, it "shifts" all the positional parameters down by one. `$2` becomes `$1`, `$3` becomes `$2`, `$4` becomes `$3`, and so on. Try this:

```
#!/bin/bash echo "You start with $# positional parameters" # Loop until all parameters are used up
while [ "$1" != "" ]; do echo "Parameter 1 equals $1" echo "You now have $# positional
parameters" # Shift all the parameters down by one shift done
```

## Getting an Option's Argument

Our `-f` option requires a valid file name as an argument. We use `shift` again to get the next item from the command line and assign it to `filename`. Later we will have to check the content of `filename` to make sure it is valid.

## Integrating the Command Line Processor into the Script

We will have to move a few things around and add a usage function to get this new routine integrated into our script. We'll also add some test code to verify that the command line processor is working correctly. Our script now looks like this:

```
#!/bin/bash # sysinfo_page - A script to produce a system information HTML file ##### Constants
TITLE="System Information for $HOSTNAME" RIGHT_NOW="$(date +"%x %r %Z")"
TIME_STAMP="Updated on $RIGHT_NOW by $USER" ##### Functions system_info() { echo
"<h2>System release info</h2>" echo "<p>Function not yet implemented</p>" } # end of
system_info show_uptime() { echo "<h2>System uptime</h2>" echo "<pre>" uptime echo
"</pre>" } # end of show_uptime drive_space() { echo "<h2>Filesystem space</h2>" echo
"<pre>" df echo "</pre>" } # end of drive_space home_space() { # Only the superuser can get this
information if [ "$(id -u)" = "0" ]; then echo "<h2>Home directory space by user</h2>"
echo "<pre>" echo "Bytes Directory" du -s /home/* | sort -nr echo "</pre>" fi } # end of
home_space write_page() { cat <<- _EOF_ <html> <head> <title>$TITLE</title> </head> <body>
<h1>$TITLE</h1> <p>$TIME_STAMP</p> $(system_info) $(show_uptime) $(drive_space) $
(home_space) </body> </html> _EOF_ } usage() { echo "usage: sysinfo_page [[-f file] [-i]] [-h]]" } ##### Main interactive= filename=~/.sysinfo_page.html while [ "$1" != "" ]; do case
$1 in -f|--file) shift filename=$1 ;; -i|--interactive) interactive=1 ;; -h|--help) usage exit ;;
*) usage exit 1 esac shift done # Test code to verify command line processing if
[ "$interactive" = "1" ]; then echo "interactive is on" else echo "interactive is off" fi
echo "output file = $filename" # Write page (comment out until testing is complete) # write_page
> $filename
```

## Adding Interactive Mode

The interactive mode is implemented with the following code:

```
if [ "$interactive" = "1" ]; then response= read -p "Enter name of output file [$filename] > "
response if [ -n "$response" ]; then filename="$response" fi if [ -f $filename ];
then echo -n "Output file exists. Overwrite? (y/n) > " read response if [ "$response" != "y"
]; then echo "Exiting program." exit 1 fi fi fi
```

First, we check if the interactive mode is on, otherwise we don't have anything to do. Next, we ask the user for the file name. Notice the way the prompt is worded:

```
"Enter name of output file [$filename] > "
```

We display the current value of `filename` since, the way this routine is coded, if the user just presses the enter key, the default value of `filename` will be used. This is accomplished in the next two lines where the value of `response` is checked. If `response` is not empty, then `filename`

is assigned the value of `response`. Otherwise, `filename` is left unchanged, preserving its default value.

After we have the name of the output file, we check if it already exists. If it does, we prompt the user. If the user response is not "y," we give up and exit, otherwise we can proceed.

[https://linuxcommand.org/lc3\\_wss0120.php](https://linuxcommand.org/lc3_wss0120.php)