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 wether 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-ar
# 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 indizes.

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: $l"loc
        # store filename in an array
        file[$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/x961.html

30 Bash Script Examples

2 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.

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"
```

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"
```

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
```

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
```

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
```

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
```

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

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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

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
```

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
```

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
```

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
```

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

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
```

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
```

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
```

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
```

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
```

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

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
```

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

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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

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/

Teste se $1 e $2 não foram digitados

```
if [ -z $1 && -z $2 ]; then
echo "digitar os parâmetros"

fi

if [ "$1" == "" && "$2" == ""]; then
    echo NULL
fi

 if [ "x$var1" == "x"  ] && [ "x$var2" == "x"  ];
 then
     echo "both variables are null"
 fi
```