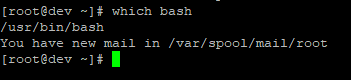
Hello World Bash Shell Script – Bash Scripting Tutorial

First you need to find out where is your Bash interpreter located. Enter *which bash* in command prompt



This command reveals that the Bash shell is stored in /bin/bash. This will come into play momentarily.

The next thing you need to do is open your favorite text editor and create a file called hello\_world.sh. We will use nano command for this step.

Copy and paste the following lines into the new file:

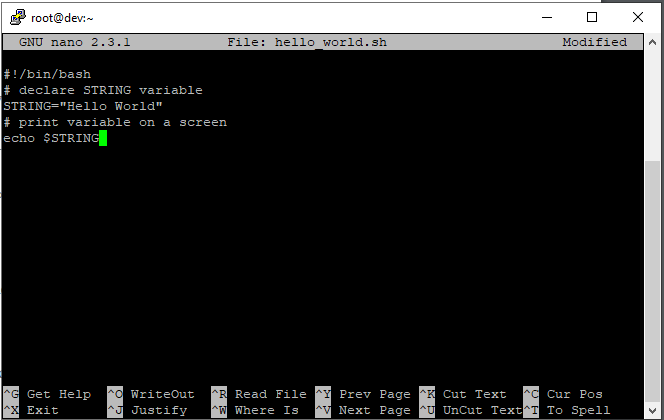
#!/bin/bash

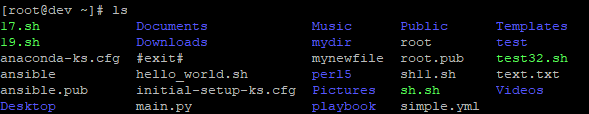
# declare STRING variable

STRING="Hello World"

# print variable on a screen

echo $STRING





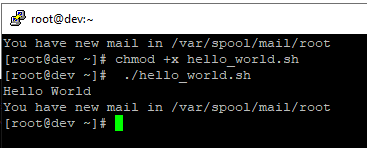
Navigate to the directory where your hello\_world.sh script is located and make the file executable by using following command

*chmod +x hello\_world.sh*

Now you are ready to execute your first bash script. Use below command to run it

*./hello\_world.sh*

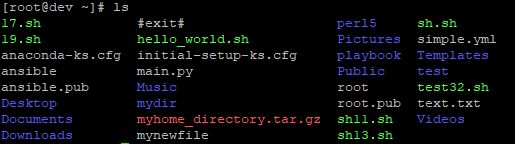
You can see the output as



Simple Backup bash shell script

When writing a Bash script, you are basically putting into it the same commands that you could execute directly on the command line. A perfect example of this is the following script:

*tar -czf myhome\_directory.tar.gz /home/linuxconfig*

**

This will create a compressed tar file of the home directory for user linuxconfig. The tar command we use in the script could easily just be executed directly on the command line. So, what’s the advantage of the script?

Well, it allows us to quickly call this command without having to remember it or type it every time. We could also easily expand the script later on to be more complex.

Variables in Bash scripts

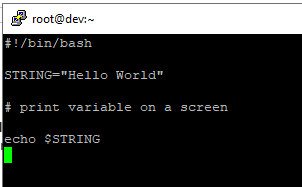
In this example we declare simple bash variable $STRING and print it on the screen (stdout) with echo command.

#!/bin/bash

STRING="Hello World"

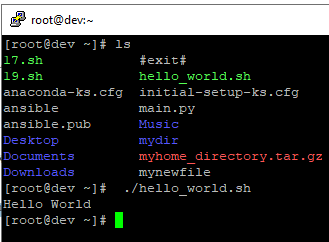
# print variable on a screen

echo $STRING

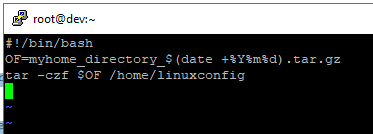


The result when we execute the script:

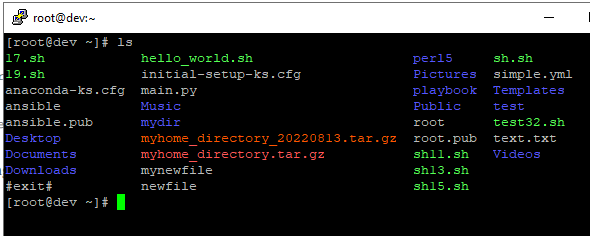
$ ./hello\_world.sh



Circling back to our backup script example, let’s use a variable to name our backup file and put a time stamp in the file name by using the date command.



The result of executing the script:



Now, when we see the file, we can quickly determine that the backup was performed on Aug 13, 2022.

**Global vs. Local variables**

In Bash scripting, a global variable is a variable that can be used anywhere inside the script. A

local variable will only be used within the function that it is declared in. Check out the example

below where we declare both a global variable and local variable. We’ve made some comments

in the script to make it a little easier to digest.

*#!/bin/bash*

*# Define bash global variable*

*# This variable is global and can be used anywhere in this bash script*

*VAR="global variable"*

*function bash {*

*# Define bash local variable*

*# This variable is local to bash function only*

*local VAR="local variable"*

*echo $VAR*

*}*

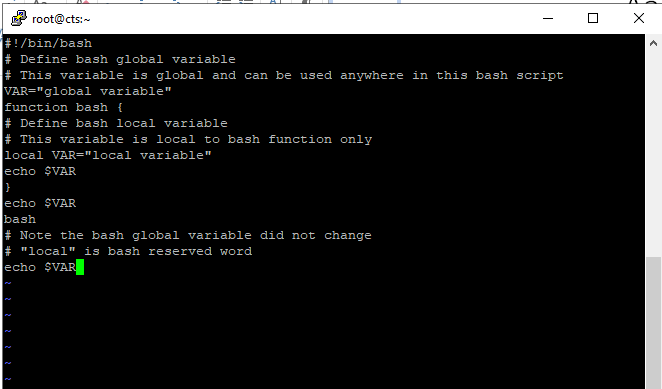
*echo $VAR*

*bash*

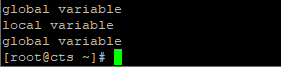
*# Note the bash global variable did not change*

*# "local" is bash reserved word*

*echo $VAR*



The result of executing this script:



**Passing arguments to the bash script**

When executing a Bash script, it is possible to pass arguments to it in your command. As you can see in the example below, there are multiple ways that a Bash script can interact with the

arguments we provide.

*#!/bin/bash*

*# use predefined variables to access passed arguments*

*#echo arguments to the shell*

*echo $1 $2 $3 ' -> echo $1 $2 $3'*

*# We can also store arguments from bash command line in special array*

*args=("$@")*

*#echo arguments to the shell*

*echo ${args[0]} ${args[1]} ${args[2]} ' -> args=("$@"); echo ${args[0]}*

*${args[1]} ${args[2]}'*

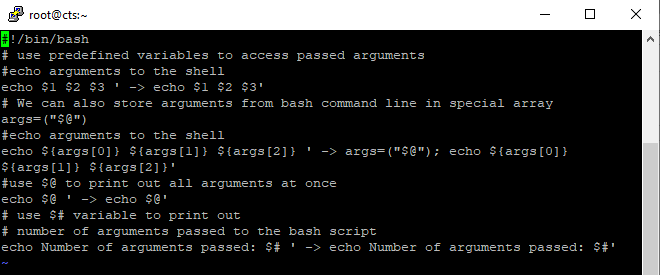
*#use $@ to print out all arguments at once*

*echo $@ ' -> echo $@'*

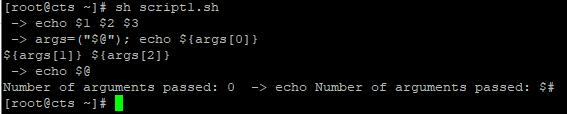
*# use $# variable to print out*

*# number of arguments passed to the bash script*

*echo Number of arguments passed: $# ' -> echo Number of arguments passed: $#'*



The results when we execute this script:

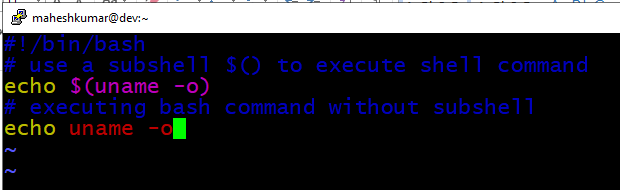


**Executing shell commands with bash**

The best way to execute a separate shell command inside of a Bash script is by creating a new

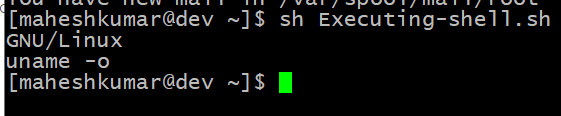
subshell through the $() syntax. Check the example below where we echo the result of running

the uname –o command.



Notice that in the final line of our script, we do not execute the uname command within a

subshell, therefore the text is taken literally and output as such.



**Reading User Input**

We can use the read command to read input from the user. This allows a user to interact with a

Bash script and help dictate the way it proceeds. Here’s an example:

*#!/bin/bash*

*echo -e "Hi, please type the word: \c "*

*read word*

*echo "The word you entered is: $word"*

*echo -e "Can you please enter two words? "*

*read word1 word2*

*echo "Here is your input: \"$word1\" \"$word2\""*

*echo -e "How do you feel about bash scripting? "*

*# read command now stores a reply into the default build-in variable $REPLY*

*read*

*echo "You said $REPLY, I'm glad to hear that! "*

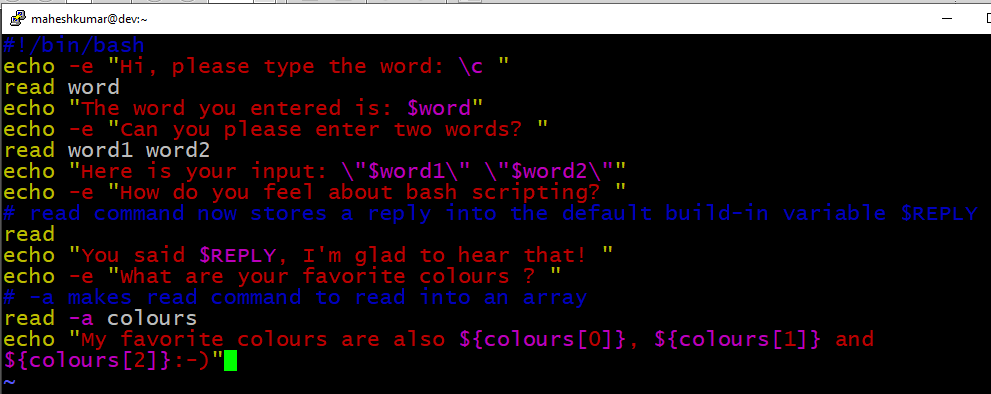
*echo -e "What are your favorite colours ? "*

*# -a makes read command to read into an array*

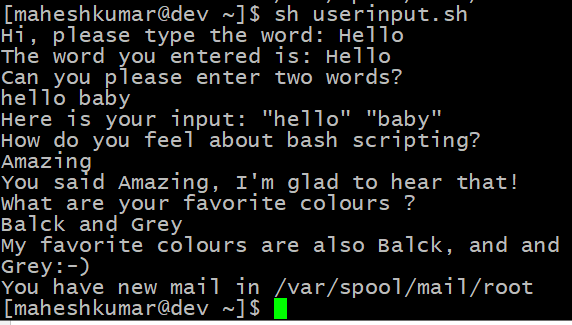
*read -a colours*

*echo "My favorite colours are also ${colours[0]}, ${colours[1]} and*

*${colours[2]}:-)"*



Now if we run this created file, Our Bash script asks multiple questions and then is able to repeat the information back to us through variables and arrays:



**Bash Trap Command**

The trap command can be used in Bash scripts to catch signals sent to the script and then

execute a subroutine when they occur. The script below will detect a Ctrl + C interrupt.

*#!/bin/bash*

*# bash trap command*

*trap bashtrap INT*

*# bash clear screen command*

*clear;*

*# bash trap function is executed when CTRL-C is pressed:*

*# bash prints message => Executing bash trap subrutine !*

*bashtrap()*

*{*

*echo "CTRL+C Detected !...executing bash trap !"*

*}*

*# for loop from 1/10 to 10/10*

*for a in `seq 1 10`; do*

*echo "$a/10 to Exit."*

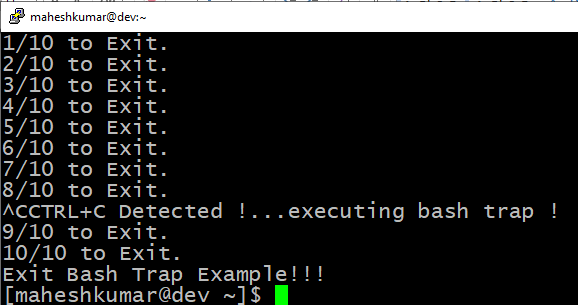
*sleep 1;*

*done*

*echo "Exit Bash Trap Example!!!"*

In the output below you can see that we try to Ctrl + C two times but the script continues to

execute.



**Arrays**

Bash is capable of storing values in arrays. Check the sections below for two different examples.

Declare simple bash array

This example declares an array with four elements.

*#!/bin/bash*

*#Declare array with 4 elements*

*ARRAY=( 'Debian Linux' 'Redhat Linux' Ubuntu Linux )*

*# get number of elements in the array*

*ELEMENTS=${#ARRAY[@]}*

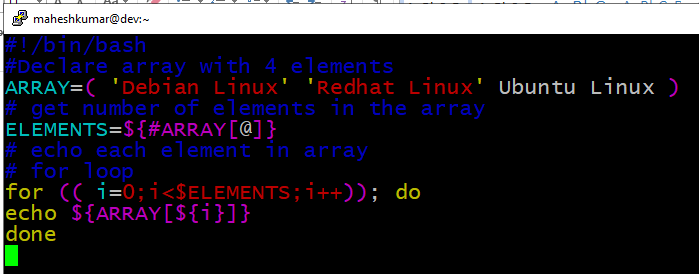
*# echo each element in array*

*# for loop*

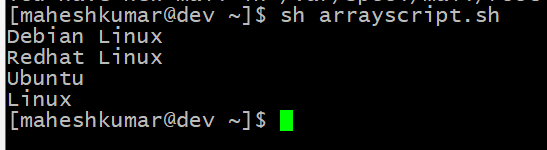
*for (( i=0;i<$ELEMENTS;i++)); do*

*echo ${ARRAY[${i}]}*

*done*



Executing the script will output the elements of our array:



**Bash if / else / fi statements**

Here is a simple if statement that check to see if a directory exists or not. Depending on the

result, it will do one of two things. Please note the spacing inside the [ and ] brackets! Without

the spaces, it won’t work!

*#!/bin/bash*

*directory="./BashScripting"*

*# bash check if directory exists*

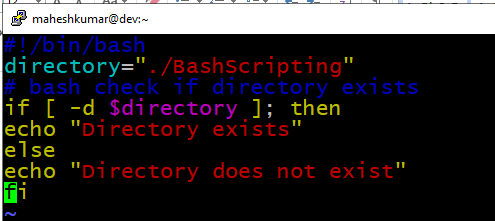
*if [ -d $directory ]; then*

*echo "Directory exists"*

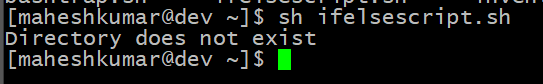
*else*

*echo "Directory does not exist"*

*fi*



The output:



**Nested if/else**

It is possible to place an if statement inside yet another if statement. This is called nesting.

Scripts can get a bit complex depending on how many if statements deep it is

*# Declare variable choice and assign value 4*

*choice=4*

*# Print to stdout*

*echo "1. Bash"*

*echo "2. Scripting"*

*echo "3. Tutorial"*

*echo -n "Please choose a word [1,2 or 3]? "*

*# Loop while the variable choice is equal 4*

*# bash while loop*

*while [ $choice -eq 4 ]; do*

*# read user input*

*read choice*

*# bash nested if/else*

*if [ $choice -eq 1 ] ; then*

*echo "You have chosen word: Bash"*

*else*

*if [ $choice -eq 2 ] ; then*

*echo "You have chosen word: Scripting"*

*else*

*if [ $choice -eq 3 ] ; then*

*echo "You have chosen word: Tutorial"*

*else*

*echo "Please make a choice between 1-3 !"*

*echo "1. Bash"*

*echo "2. Scripting"*

*echo "3. Tutorial"*

*echo -n "Please choose a word [1,2 or 3]? "*

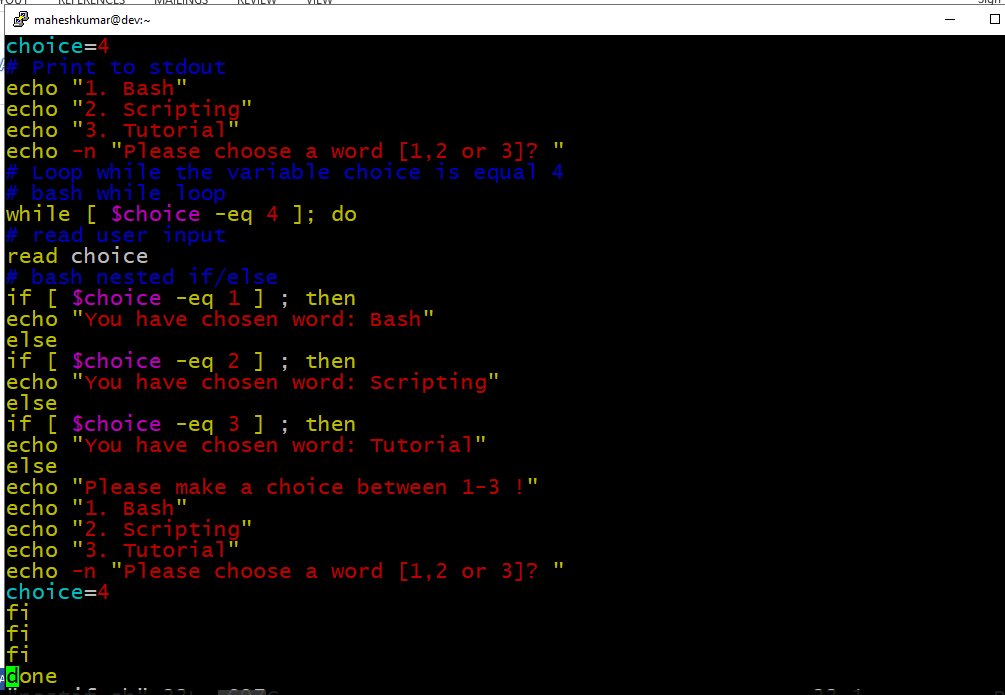
*choice=4*

*fi*

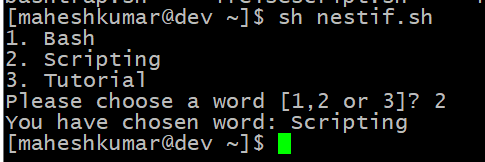
*fi*

*fi*

*done*



The output of the script :



**Bash Comparisons**

Bash can compare two or more values, either integers or strings, to determine if they are equal to

each other, or one is greater than the other, etc.

*#!/bin/bash*

*# declare integers*

*NUM1=2*

*NUM2=2*

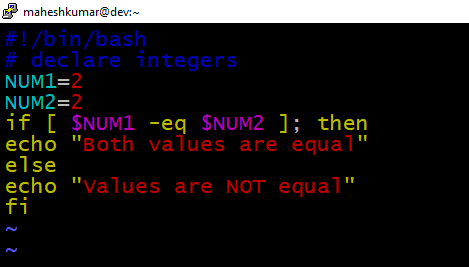
*if [ $NUM1 -eq $NUM2 ]; then*

*echo "Both values are equal"*

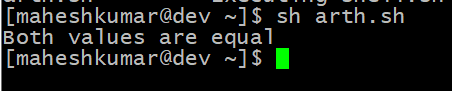
*else*

*echo "Values are NOT equal"*

*fi*

**

The output of the script is :



Let’s try changing one of the numbers:

*#!/bin/bash*

*# declare integers*

*NUM1=2*

*NUM2=1*

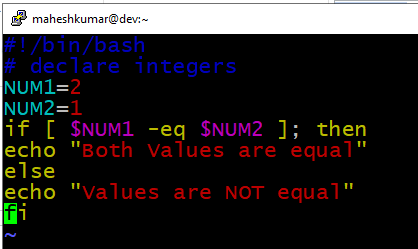
*if [ $NUM1 -eq $NUM2 ]; then*

*echo "Both Values are equal"*

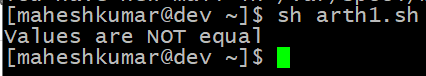
*else*

*echo "Values are NOT equal"*

*fi*



The output of the script:



Let’s add a little more complexity by including an elif statement and determine which number

is larger.

*#!/bin/bash*

*# declare integers*

*NUM1=2*

*NUM2=1*

*if [ $NUM1 -eq $NUM2 ]; then*

*echo "Both values are equal"*

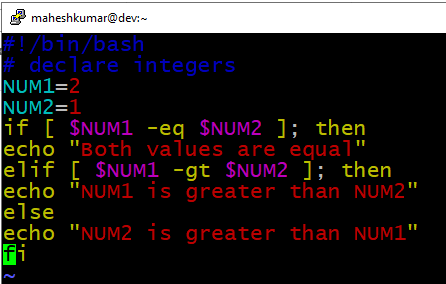
*elif [ $NUM1 -gt $NUM2 ]; then*

*echo "NUM1 is greater than NUM2"*

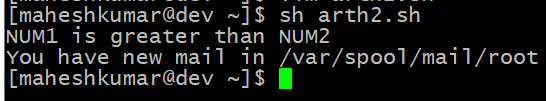
*else*

*echo "NUM2 is greater than NUM1"*

*fi*

**

The output of the script:



**String Comparisons**

Let’s try comparing two strings to see if they are equal.

*#!/bin/bash*

*#Declare string S1*

*S1="Bash"*

*#Declare string S2*

*S2="Scripting"*

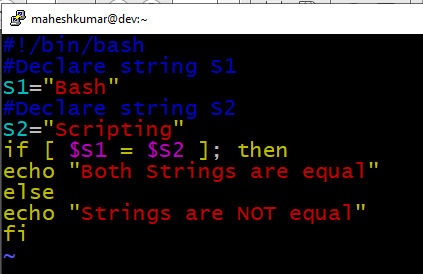
*if [ $S1 = $S2 ]; then*

*echo "Both Strings are equal"*

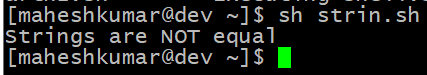
*else*

*echo "Strings are NOT equal"*

*fi*

****

The output of the script:



And again with both string matching:

*#!/bin/bash*

*#Declare string S1*

*S1="Bash"*

*#Declare string S2*

*S2="Bash"*

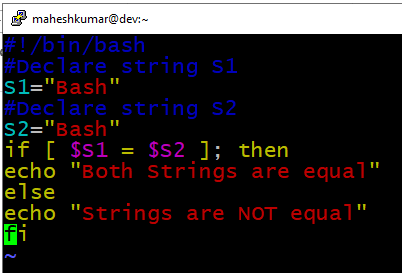
*if [ $S1 = $S2 ]; then*

*echo "Both Strings are equal"*

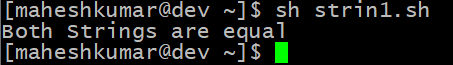
*else*

*echo "Strings are NOT equal"*

*fi*



The output of the script:



**Bash File Testing**

In Bash, we can test to see different characteristics about a file or directory. See the table below

for a full list.

The following script will check to see if a file exists or not.

*#!/bin/bash*

*file="./file"*

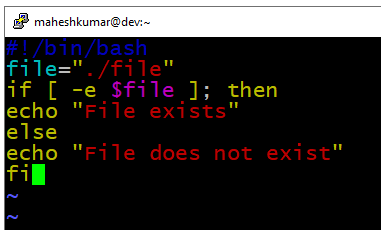
*if [ -e $file ]; then*

*echo "File exists"*

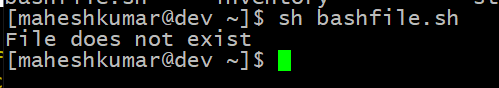
*else*

*echo "File does not exist"*

*fi*



The result:



Similarly for example we can use while loop to check if file does not exist. This script will sleep

until file does exist. Note bash negator ! which negates the -e option.

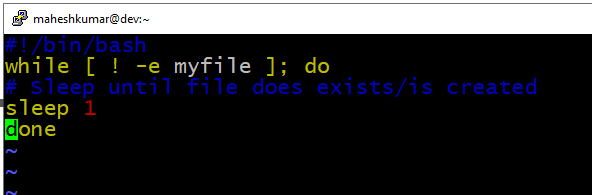
*#!/bin/bash*

*while [ ! -e myfile ]; do*

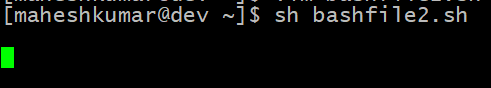
*# Sleep until file does exists/is created*

*sleep 1*

*done*



The result:



**Loops**

There are multiple types of loops that can be used in Bash, including for, while, and until. See

some of the examples below to learn how to use.

***Bash for loop***

This script will list every file or directory it finds inside the /var/ directory.

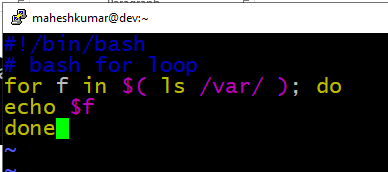
*#!/bin/bash*

*# bash for loop*

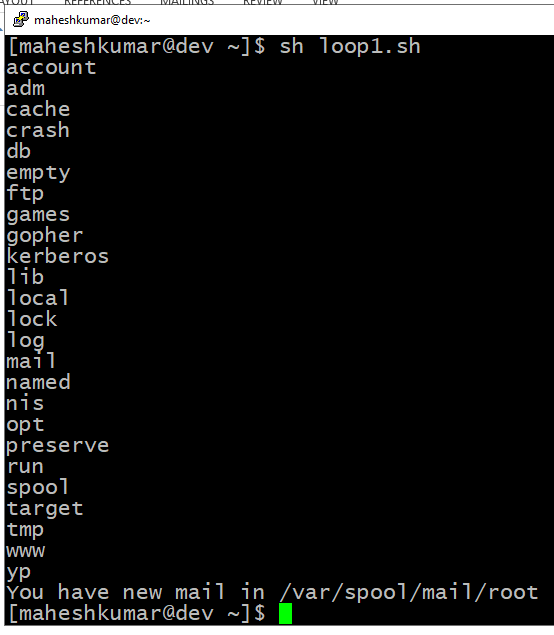
*for f in $( ls /var/ ); do*

*echo $f*

*done*



The result:



***Bash while loop:***

This while loop will continue to loop until our variable reaches a value of 0 or less.

*#!/bin/bash*

*COUNT=6*

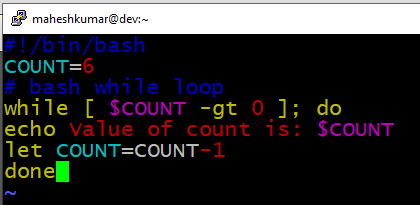
*# bash while loop*

*while [ $COUNT -gt 0 ]; do*

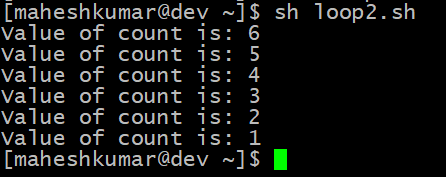
*echo Value of count is: $COUNT*

*let COUNT=COUNT-1*

*done*



The result:



***Bash until loop:***

An until loop works similarly to while.

*#!/bin/bash*

*COUNT=0*

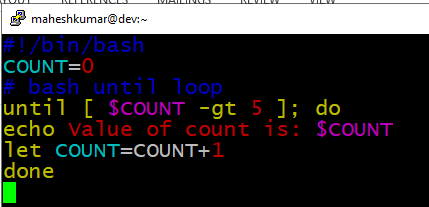
*# bash until loop*

*until [ $COUNT -gt 5 ]; do*

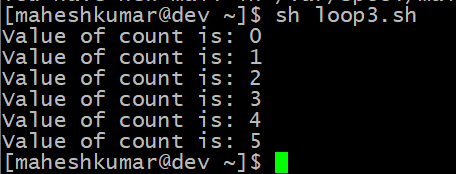
*echo Value of count is: $COUNT*

*let COUNT=COUNT+1*

*done*



The result:



**Bash Functions**

This example shows how to declare a function and call back to it later in the script.

*!/bin/bash*

*# BASH FUNCTIONS CAN BE DECLARED IN ANY ORDER*

*function function\_B {*

*echo Function B.*

*}*

*function function\_A {*

*echo $1*

*}*

*function function\_D {*

*echo Function D.*

*}*

*function function\_C {*

*echo $1*

*}*

*# FUNCTION CALLS*

*# Pass parameter to function A*

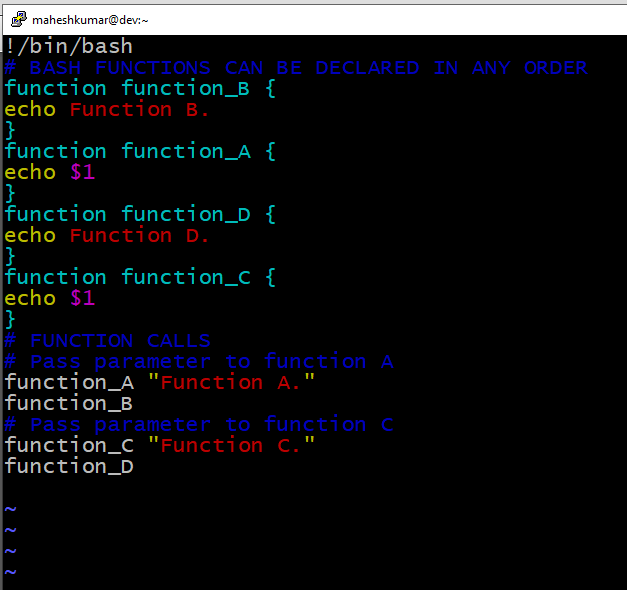
*function\_A "Function A."*

*function\_B*

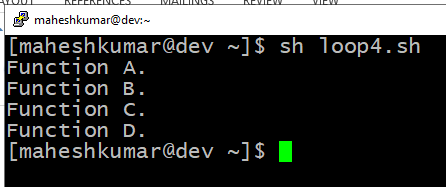
*# Pass parameter to function C*

*function\_C "Function C."*

*function\_D*

**

The result:



**Bash Select**

The select command allows us to prompt the user to make a selection

*#!/bin/bash*

*PS3='Choose one word: '*

*# bash select*

*select word in "linux" "bash" "scripting" "tutorial"*

*do*

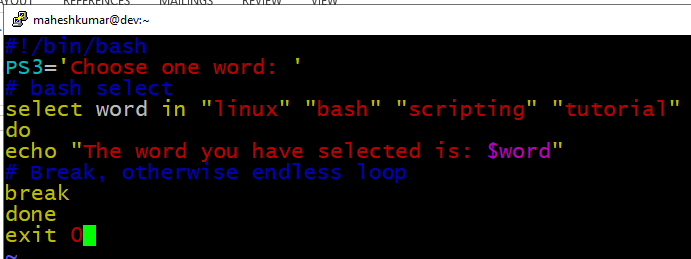
*echo "The word you have selected is: $word"*

*# Break, otherwise endless loop*

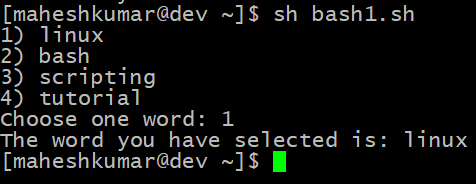
*break*

*done*

*exit 0*



The result:



**Bash quotes and quotations**

Quotations and quotes are important part of bash and bash scripting. Here are some bash quotes

and quotations basics.

***Escaping Meta characters***

Before we start with quotes and quotations we should know something about escaping meta

characters. Escaping will suppress a special meaning of meta characters and therefore meta

characters will be read by bash literally. To do this we need to use backslash \ character.

Example:

*#!/bin/bash*

*#Declare bash string variable*

*BASH\_VAR="Bash Script"*

*# echo variable BASH\_VAR*

*echo $BASH\_VAR*

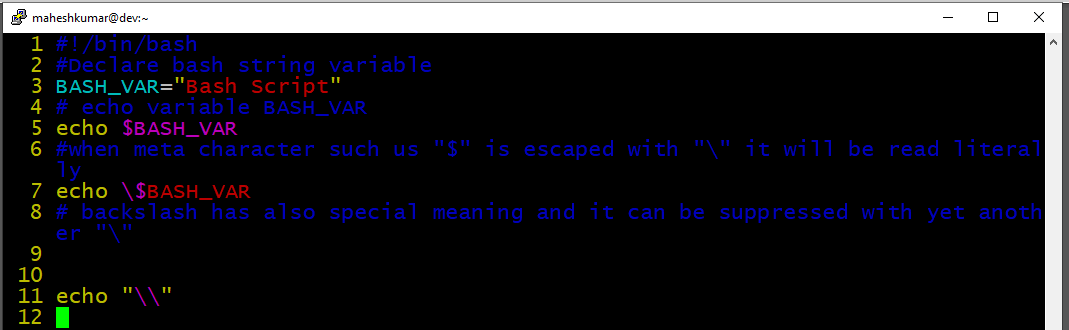
*#when meta character such us "$" is escaped with "\" it will be read literally*

*echo \$BASH\_VAR*

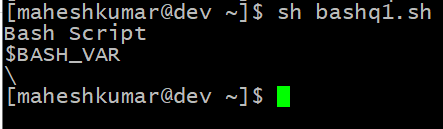
*# backslash has also special meaning and it can be suppressed with yet another*

*"\"*

*echo "\\"*

******

Here’s what it looks like when we execute the script:



**Single quotes**

Single quotes in bash will suppress special meaning of every meta characters. Therefore meta

characters will be read literally. It is not possible to use another single quote within two single

quotes not even if the single quote is escaped by backslash.

*#!/bin/bash*

*# Declare bash string variable*

*BASH\_VAR="Bash Script"*

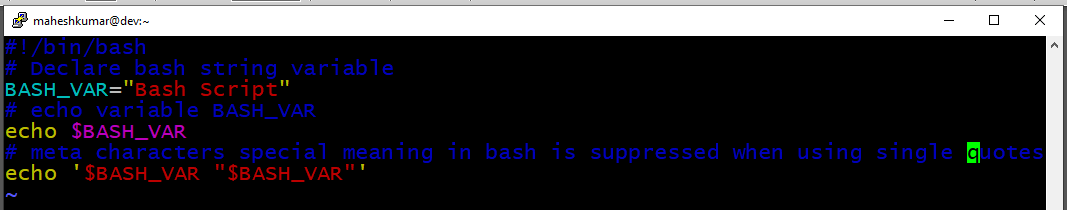
*# echo variable BASH\_VAR*

*echo $BASH\_VAR*

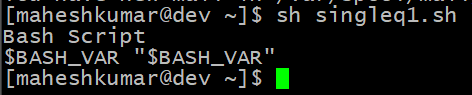
*# meta characters special meaning in bash is suppressed when using single*

*quotes*

*echo '$BASH\_VAR "$BASH\_VAR"'*

**

The result:



**Double quotes**

Double quotes in bash will suppress special meaning of every meta characters except $, \ and `.

Any other meta characters will be read literally. It is also possible to use single quote within

double quotes. If we need to use double quotes within double quotes bash can read them literally when escaping them with \.

Example:

*#!/bin/bash*

*#Declare bash string variable*

*BASH\_VAR="Bash Script"*

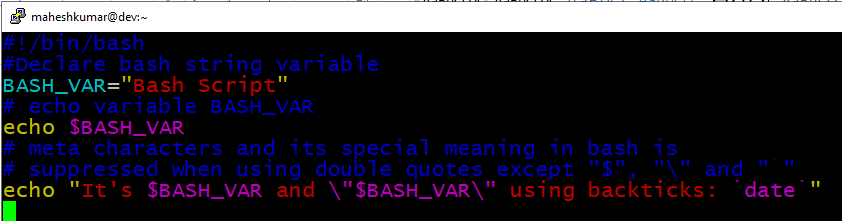
*# echo variable BASH\_VAR*

*echo $BASH\_VAR*

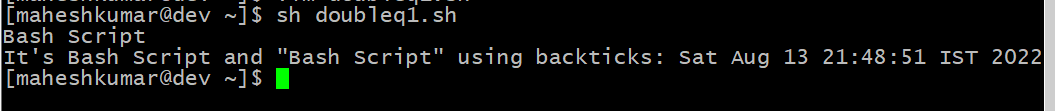
*# meta characters and its special meaning in bash is*

*# suppressed when using double quotes except "$", "\" and "`"*

*echo "It's $BASH\_VAR and \"$BASH\_VAR\" using backticks: `date`"*

**

The result:



**Bash quoting with ANSI-C style**

There is also another type of quoting and that is ANSI-C. In this type of quoting characters

escaped with \ will gain special meaning according to the ANSI-C standard.The syntax for ansi-c bash quoting is: $' ' .

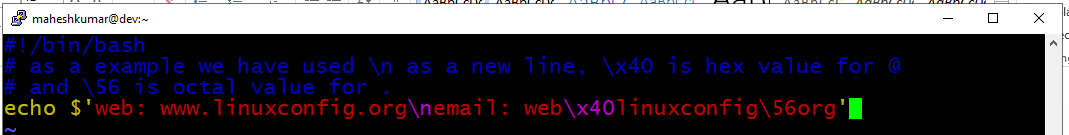
Here is an example:

*#!/bin/bash*

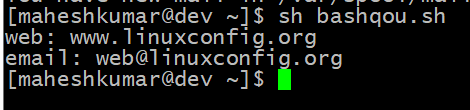
*# as a example we have used \n as a new line, \x40 is hex value for @*

*# and \56 is octal value for .*

*echo $'web: www.linuxconfig.org\nemail: web\x40linuxconfig\56org'*

**

The result:



**Arithmetic Operations**

Bash can be used to perform calculations. Let’s look at a few examples to see how it’s done.

*echo "2 ^ 2 =" $POWEROFTWO*

*echo '### Bash Arithmetic Expansion ###'*

*# There are two formats for arithmetic expansion: $[ expression ]*

*# and $(( expression #)) its your choice which you use*

*echo 4 + 5 = $((4 + 5))*

*echo 7 - 7 = $[ 7 - 7 ]*

*echo 4 x 6 = $((3 \* 2))*

*echo 6 / 3 = $((6 / 3))*

*echo 8 % 7 = $((8 % 7))*

*echo 2 ^ 8 = $[ 2 \*\* 8 ]*

*echo '### Declare ###'*

*echo -e "Please enter two numbers \c"*

*# read user input*

*read num1 num2*

*declare -i result*

*result=$num1+$num2*

*echo "Result is:$result "*

*# bash convert binary number 10001*

*result=2#10001*

*echo $result*

*# bash convert octal number 16*

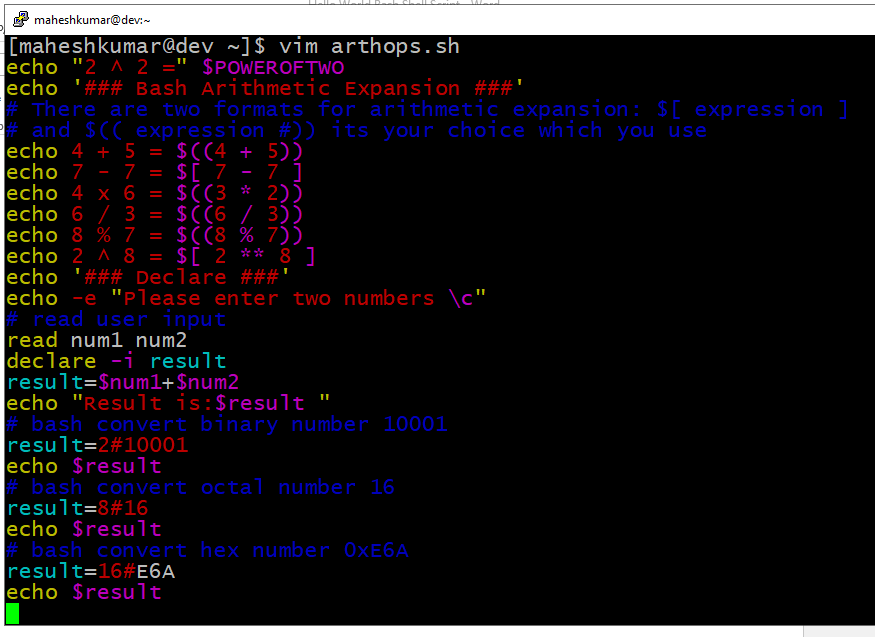
*result=8#16*

*echo $result*

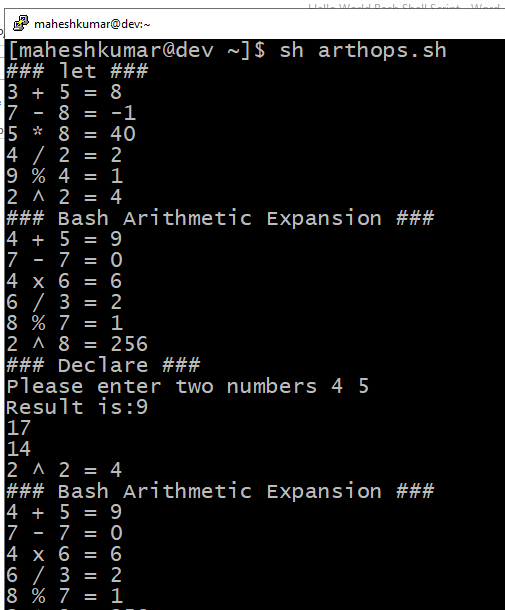
*# bash convert hex number 0xE6A*

*result=16#E6A*

*echo $result*

**

The result:

**

**Round floating point number**

Here is how to use rounding in Bash calculations.

*#!/bin/bash*

*# get floating point number*

*floating\_point\_number=3.3446*

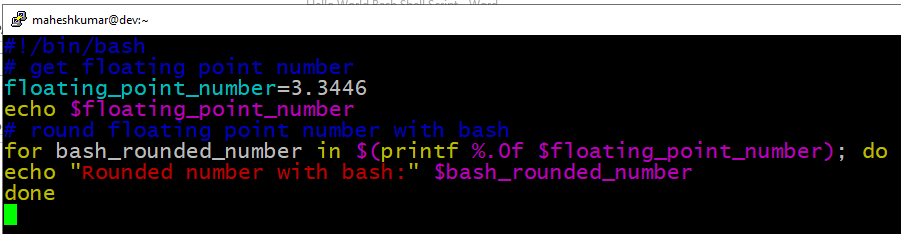
*echo $floating\_point\_number*

*# round floating point number with bash*

*for bash\_rounded\_number in $(printf %.0f $floating\_point\_number); do*

*echo "Rounded number with bash:" $bash\_rounded\_number*

*done*

**

The result:

**

**Bash floating point calculations**

Using the bc bash calculator to perform floating point calculations.

*#!/bin/bash*

*# Simple linux bash calculator*

*echo "Enter input:"*

*read userinput*

*echo "Result with 2 digits after decimal point:"*

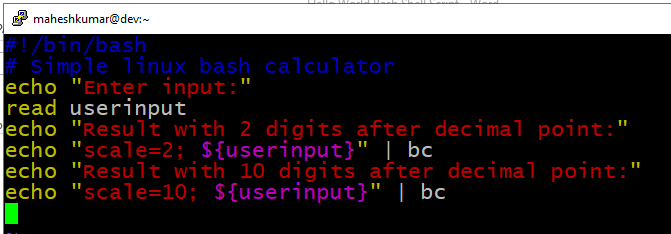
*echo "scale=2; ${userinput}" | bc*

*echo "Result with 10 digits after decimal point:"*

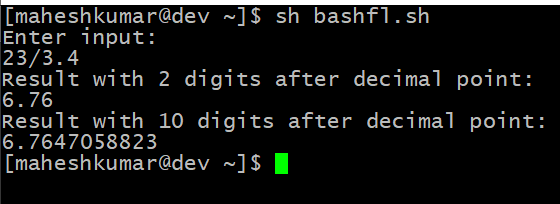
*echo "scale=10; ${userinput}" | bc*

*echo "Result as rounded integer:"*

*echo $userinput | bc*

**

The result:

**

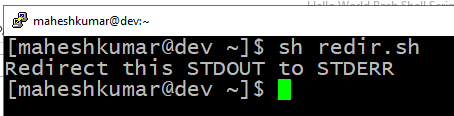
**Redirections**

In the following examples, we will show how to redirect standard error and standard output.

STDOUT from bash script to STDERR

*#!/bin/bash*

*echo "Redirect this STDOUT to STDERR" 1>&2*

**

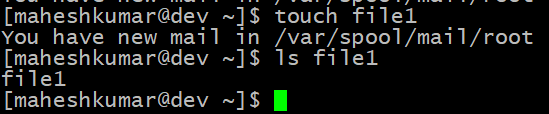
**stdout to screen**

The simple way to redirect a standard output (stdout) is to simply use any command, because by default stdout is automatically redirected to screen. First create a file file1:

*$ touch file1*

*$ ls file1*

*file1*

**

As you can see from the example above execution of ls command produces STDOUT which by

default is redirected to screen.

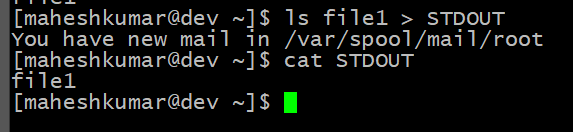
***stdout to file***

To override the default behavior of STDOUT we can use > to redirect this output to file:

*$ ls file1 > STDOUT*

*$ cat STDOUT*

*file1*

**

**stderr to file**

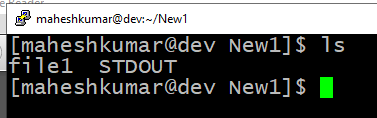
By default STDERR is displayed on the screen:

*$ ls*

*file1 STDOUT*

*$ ls file2*

*ls: cannot access file2: No such file or directory*

**