**Bash Scripting**

Before we move on to the topic of bash scripting use cases, we need to elaborate on what bash and bash scripting are.

Bash is a **command-line interface interpreter** that runs in a text window where users can manage and execute shell commands. Bash – or shell scripting – on the other hand is the process of writing a set of commands to be executed on a Linux system. A file that includes such instructions is called a bash script.

To put it simply, the bash interpreter reads the bash script and executes the commands at the same time. For example, a Linux user can execute hundreds of commands with a single click instead of inputting them one by one. For this reason, bash scripting is the go-to choice for increasing productivity, setting up automation, and eliminating repetitive tasks.

**25 Bash Scripts Examples**

The following section will cover 25 of the most popular bash scripting examples, including variable manipulation and echoing out various values. We will also cover functions, arrays, loops, and much more.

**1. Hello World**

**Hello World** is the most simple bash script to start with. We will create a new variable called **learningbash** and print out the words **Hello World**. First, open a new shell script file with a text editor of your choice:

nano hello.sh

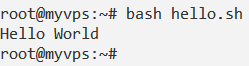
Paste the following lines into it:

#!/bin/bash

#Creates a new variable with a value of "Hello World"

learningbash="Hello World"

echo $learningbash



The first line (**/bin/bash**) is used in every bash script. It instructs the operating system to use a bash interpreter as a command interpreter.

**2. Echo Command**

The **echo** bash command can be used to print out text as well as values of variables. In the following example, we will showcase how quotation marks affect the echo command. We will start by opening a new bash script file:

nano echo.sh

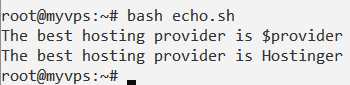
This simple bash script example will create a new variable and print it out while using different quotation marks.

#!/bin/bash

provider="Hostinger"

echo 'The best hosting provider is $provider'

echo "The best hosting provider is $provider"



As you can see, if the echo bash command is used with double quotation marks **““**, then the script will print out the actual value of a variable. Otherwise, if the single quotation marks **‘‘**are used, it will print out only the name of a variable.

**3. Sleep Command**

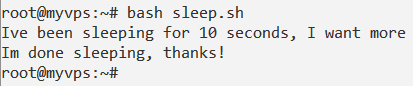
Sleep command halts all currently running bash scripts and puts the system to sleep. Start by creating a new bash script file:

nano sleep.sh

Then, paste in the following simple script:

#!/bin/bash

sleep 10 && echo “I’ve been sleeping **for** 10 seconds, I want more” && sleep 10 && echo “I’m done sleeping, thanks!”



The above example starts with a simple sleep bash command that will put your system to sleep for 10 seconds. After that, we combine the previously learned echo command with sleep – this way system will sleep for 10 seconds, then print out some words, sleep again, print out some words again and end its operation.

**Pro Tip**

A bash script can always be terminated by clicking **CTRL + C** without waiting for it to finish its operation.

**4. Wait Command**

**wait** is a built-in [**Linux command**](https://www.hostinger.com/tutorials/linux-commands) that waits for completion of running process. The wait command is used with a particular process id or job id.

Here’s how to create a **wait** bash script. Begin by creating a new bash file:

nano wait.sh

Paste in the following:

#!/bin/bash

wait 1234

echo “Done”

**Important!** If no job ID is provided, the wait command waits until all child background jobs are completed.

**5. Comments**

Users can easily add comments to their bash scripts with the **#**symbol. It is extra useful if you’ve got a lengthy script that needs explaining on some lines.

Begin by creating a new bash script:

nano comments.sh

Then paste in the following:

#!/bin/bash

# Define a variable named Hostinger

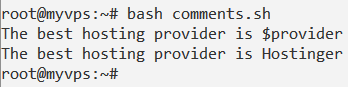
provider="Hostinger"

# Print out the following text

echo 'The best hosting provider is $provider'

# Print out the following text with $provider variable value

echo "The best hosting provider is $provider"



Keep in mind that bash comments are only visible on a text editor.

**6. Get User Input**

To take input from users, we’ll use the **read** bash command. First, create a new bash shell file:

nano read.sh

Then, fill it with the script below:

#!/bin/bash

echo "What is your age?"

read age

echo "Wow, you look younger than $age years old"

In the above example, an age value was entered by the user. The output was then printed via the echo command.

**7. Loops**

A loop is an essential tool in various programming languages. To put it simply, a [**bash loop**](https://www.hostinger.com/tutorials/bash-for-loop-guide-and-examples/) is a set of instructions that are repeated until a user-specified condition is reached. Start by creating a loop bash program:

nano whileloop.sh

Then paste in the following:

#!/bin/bash

n=0

**while** :

**do**

echo Countdown: $n

((n++))

done

This will work as a countdown to infinity until you press **CTRL + C** to stop the script.

Now that we’ve tested the while loop, we can move on to the for loop. Create a bash file for it:

nano forloop.sh

It should contain the script below:

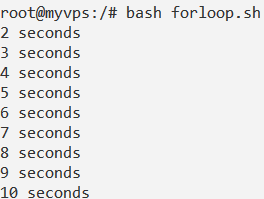
#!/bin/bash

**for** (( n=2; n<=10; n++ ))

**do**

echo "$n seconds"

done



The script prints out numbers from 2 to 10 while adding the**seconds** keyword to it.

**8. Create an Array**

A [**bash array**](https://www.hostinger.com/tutorials/bash-array) is a data structure designed to store information in an indexed way. It is extra useful if users need to store and retrieve thousands of pieces of data fast. What makes bash arrays special is that unlike any other programming language, they can store different types of elements. For example, you can use a bash array to store both strings and numbers.

Create a new file in the current directory:

nano array.sh

Combine the freshly learned **for** loop with a new indexed array:

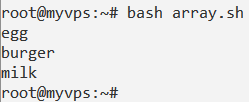
#!/bin/bash

# Create an indexed array

IndexedArray=(egg burger milk)

#Iterate over the array to get all the values

**for** i **in** "${IndexedArray[@]}";**do** echo "$i";done



The script iterates over the **IndexedArray** and prints out all the values.

**9. Conditional Statements**

The most popular and widely used conditional statement is **if**. Even though the if statement is easy to write and understand, it can be used in advanced shell scripts as well.

Begin with a new bash file:

nano if.sh

Paste the code below in it:

#!/bin/bash

salary=1000

expenses=800

#Check if salary and expenses are equal

**if** [ $salary == $expenses ];

**then**

echo "Salary and expenses are equal"

#Check if salary and expenses are not equal

elif [ $salary != $expenses ];

**then**

echo "Salary and expenses are not equal"

fi

This script creates two new variables and compares whether they are equal or not.

**10. Functions**

A bash function is a set of commands that can be reused numerous times throughout a bash script. Create a new file:

nano function.sh

Then, paste in the following code – it creates a simple Hello World function.

#!/bin/bash

hello () {

echo 'Hello World!'

}

hello

**11. Display String Length**

There are a couple of ways of counting string length in bash. We’ll talk about the simplest. Create a file named **stringlength.sh**:

nano stringlength.sh

Fill it with the following:

#!/bin/bash

# Create a new string

mystring="lets count the length of this string"

i=${#mystring}

echo "Length: $i"

Here, the **#**operator is used to get the length of the string variable.

**12. Extract String**

If users need to remove unnecessary parts from strings, they can use the Bash string extraction tools. Start by creating a new bash script:

nano extractstring.sh

The following script has 4 values, 3 of them being strings. In our example, we will extract only the number value. This can be done via the **cut** command. First, we instruct the command that each variable is separated by a comma by using the **-d**flag. Then we ask the cut command to extract the 5th value.

#!/bin/bash

cut -d , -f 5 <<< "Website,Domain,DNS,SMTP,5005"

In another example, we have a string that is mixed with some numbers. We will use expr substr commands to extract only the **Hostinger**text value.

#!/bin/bash

expr substr "458449Hostinger4132" 7 9

**13. Find and Replace String**

Another useful bash script for strings is **find and replace**. Create a file named **findreplace.sh**:

nano findreplace.sh

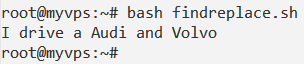
Then paste in the following bash script:

#!/bin/bash

first="I drive a BMW and Volvo"

second="Audi"

echo "${first/BMW/"$second"}"



The find and replace functionality doesn’t require any special commands, it can all be done with string manipulation.

**14. Concatenate Strings**

Concatenation is the term used for appending one string to the end of another string. Start by creating **concatenation.sh** file.

nano concatenation.sh

The most simple example would be the following:

#!/bin/bash

firststring="The secret is..."

secondstring="Bash"

thirdstring="$firststring$secondstring"

echo "$thirdstring"

The above script will connect the values of **firststring**and **secondstring** variables creating a whole new **thirdstring**.

A more advanced example would look like this:

#!/bin/bash

firststring="The secret is..."

firststring+="Bash"

echo "$firststring"

The script uses the **+=**operator to join the strings. With this method, you can concatenate strings with only one variable.

**15. Check if a Number is Even or Odd**

Odd and even numbers can be easily divided using the **if** statement and some simple math. Create a file named **evenoddnumbers.sh**:

nano evenoddnumbers.sh

The script uses the read command to read user input and divides it by 2. If the answer is 0, the number is even.

#!/bin/bash

read -p "Enter a number and I will check if its odd or even " mynumber

**if** [ $((mynumber%2)) -eq 0 ]

**then**

echo "Your number is even"

**else**

echo "Your number is odd."

fi

**16. Generate Factorial of Number**

The factorial of a number is the result of all positive descending integers. For example, the factorial of 5 would be 120:

5! = 5\*4\*3\*2\*1 = 120

Factorial scrips are very useful for users learning about recursion. Start by creating a **.sh** file executable:

factorial.sh

The following script will ask the user to enter a number they want to get the factorial of and use a **for loop** to calculate it.

#!/bin/bash

echo Enter the number you want to get factorial **for**

read mynumber

factorial=1

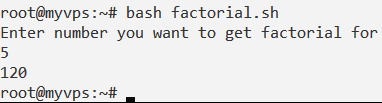
**for** ((i=1;i<=mynumber;i++))

**do**

factorial=$(($factorial\*$i))

done

echo $factorial



**17. Create Directories**

It is effortless to create directories in bash unless you need to create a lot of directories quickly. In the following example, we will use the bash script to create a set of directories with the same subdirectories in each.

First, create a file named **directories.sh**:

nano directories.sh

Then paste in the following code:

#!/bin/bash

mkdir -p {Math,English,Geography,Arts}/{notes,examresults,portfolio}

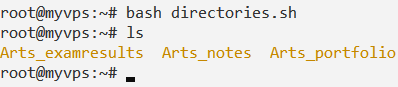
The script creates 4 main directories: **Math**, **English**, **Geography**, and **Arts**. The **Notes**, **examresults**, and **portfolio** subdirectories are also created inside each.

If you were to replace the **/** symbol in the middle with **\_**, the script would look like this:

#!/bin/bash

mkdir -p {Math,English,Geography,Arts}\_{notes,examresults,portfolio}

Here’s the output for it displaying a merge of the two directories:



**18. Read Files**

In order to read a file in bash, you will need to create a sample file first. Do so with the following command:

nano mysamplefile.txt

Fill it with some sample data:

Out of all scripting languages, bash is the most popular one. It allows programmers to run scripts effortlessly in a variety of Linux distros.

Then create the actual script file:

nano readfiles.sh

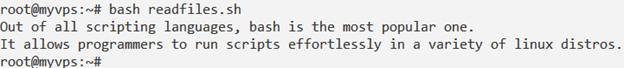
Fill it with the following lines:

#!/bin/bash

myvalue=`cat mysamplefile.txt`

echo "$myvalue"

Running the script results in this output:



**19. Print Files With Line Count**

We’ll print a file with its line count. Let’s create it first:

nano cars.txt

In our example, we will fill it with our favorite car brands:

Audi

BMW

Bentley

Maserati

Seat

Volvo

Save the file and create a new bash script:

nano printlines.sh

Then paste in the following code:

#!/bin/bash

myfile='cars.txt'

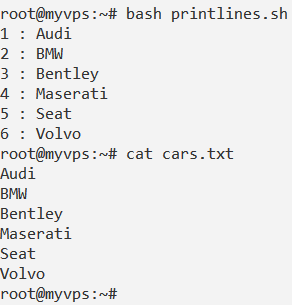
i=1

**while** read lines; **do**

echo "$i : $lines"

i=$((i+1))

done < $myfile



The file contents of **cars.txt** match the printout of the while loop script.

**20. Delete Files**

To delete an existing file, you can use an **if** statement to check if the file exists and instruct the bash script to remove it. Start by creating the bash script file:

nano deletefiles.sh

The following script will create a new file named **cars.txt**, and then – with the help of the if statement – check if it exists and delete it.

#!/bin/bash

myfile='cars.txt'

touch $myfile

**if** [ -f $myfile ]; **then**

rm cars.txt

echo "$myfile deleted"

fi

**21. Test if File Exists**

In order to check if a given file exists, users can perform conditional tests. In this case, we’ll usean**if**statement with a **-f** flag. The flag checks if a given file exists and is a regular file. Start by creating the script file:

nano exists.sh

Copy and paste the following script:

#!/bin/bash

MyFile=cars.txt

**if** [ -f "$MyFile" ]; **then**

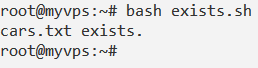
echo "$MyFile exists."

**else**

echo "$MyFile does not exist."

fi

Running the script results in the following output:



**22. Check Inodes and Disk Usage**

Inodes represent data units on a physical or virtual server. Each text file, video, folder, HTML file, or script is 1 inode. We’ll check how many inodes there are in a directory, as too many can cause the system to slow down significantly. Start by creating the bash script:

nano inodesdisk.sh

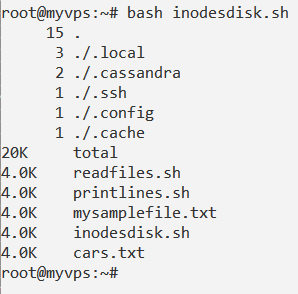
Paste in the following code – it will check inodes in descending order as well as show disk usage in a given directory:

#!/bin/bash

find . -printf "%h\n" | cut -d/ -f-2 | sort | uniq -c | sort -rn

du -shc \* | sort -rh

It will look something like this on the command line:



The given directory has 15 inodes and all files take up 20KB.

**23. Send Email Example**

It is possible to send mail via bash scripts as well. In order to do so, users first need a functional mail transport agent. On Ubuntu 20.04, the installation command will look like this:

sudo apt-get install mailutils

Once you’ve taken care of the mail transport agent installation, create a new bash script:

nano mail.sh

Here are its contents:

#!/bin/bash

Recipient="myawesomeinbox@domain.tld"

Mysubject="Regarding our talk"

Mymessage="Call me"

`mail -s $Mysubject $Recipients <<< $Mymessage`

**Important! The above script is meant for testing purposes only as it won’t work normally with services like Gmail. We recommend using PHPMailer instead.**

**24. Update Packages**

Keeping the system and all of its applications up to date is crucial. You can create a bash script to do it. Mind that this script requires root privileges. First, create the bash script file:

nano maintenance.sh

Fill it with these lines:

#!/bin/bash

apt-get update

apt-get upgrade

Make sure to preface the script with the sudo command when you run it:

sudo bash maintenance.sh

**Important!Apt package manager is used on Debian based distributions only. If you’re using a different distribution, make sure to update the command accordingly.**

**25. Show Server Information**

The following script will list a few important server metrics: system’s date, uptime as well as memory, and network usage statistics. We’ll start by creating a new file for it:

nano system.sh

Here’s the script for it:

#!/bin/bash

echo "Date"

date

echo "Uptime"

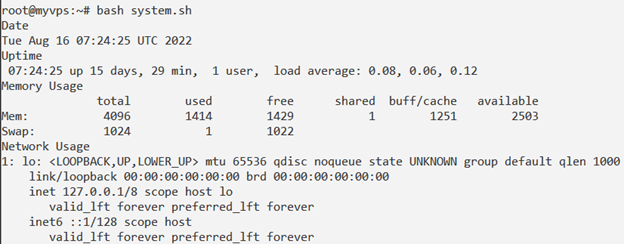
uptime

echo "Memory Usage"

free -m

echo "Network Usage"

ip a

****

# Bash Scripting Tutorial for Beginners: What It Is, How to Write One, and Script Examples

If you’ve ever used a Linux operating system like most [**virtual private servers**](https://www.hostinger.com/vps-hosting), you may have heard of bash. It’s a Unix shell that reads and executes various commands.

When you need to run several bash commands, you don’t have to execute them manually one at a time. Instead, it’s possible to create a script file that contains bash functions to run those commands.

It may sound complicated, but by learning its basics, you will understand the bash scripting language and find out how it can help your workflow.

This article will cover the process of bash scripting. We’ll go over everything from bash commands to running a bash program on a Linux terminal.

* [**What Is Bash?**](https://www.hostinger.com/tutorials/bash-scripting-tutorial#What_Is_Bash)
* [**Why Use Bash Scripting?**](https://www.hostinger.com/tutorials/bash-scripting-tutorial#Why_Use_Bash_Scripting)
* [**Get Familiar With Bash Commands**](https://www.hostinger.com/tutorials/bash-scripting-tutorial#Get_Familiar_With_Bash_Commands)
* [**Basic Bash Commands for Your First Bash Script**](https://www.hostinger.com/tutorials/bash-scripting-tutorial#Basic_Bash_Commands_for_Your_First_Bash_Script)
* [**4 Easy Functions to Try On Your First Bash Script**](https://www.hostinger.com/tutorials/bash-scripting-tutorial#4_Easy_Functions_to_Try_On_Your_First_Bash_Script)
* [**How to Run Bash Script**](https://www.hostinger.com/tutorials/bash-scripting-tutorial#How_to_Run_Bash_Script)
* [**Bash Tutorial FAQ**](https://www.hostinger.com/tutorials/bash-scripting-tutorial#Bash_Tutorial_FAQ)

## What Is Bash?

Bash, short for **Bourne-Again Shell**, is a Unix shell and a command language interpreter. It reads shell commands and interacts with the operating system to execute them.

To fully understand bash shell scripting, you need to know two concepts – **shell** and **scripting**.

Shell is a macro processor that uses commands to interact with the operating system. This means that it can retrieve, process, and store information on a computer.

Meanwhile, scripting is the process of compiling shell commands into a new file using a text editor.

When you write bash in a text editor, you’re compiling **bash commands** or**bash functions** – a set of commands that can be called numerous times by only using the function name. The text is then saved as an executable **bash script** file with the **.sh**extension.

## Why Use Bash Scripting?

Bash scripts can help with your workflow as they compile many lengthy commands into a single executable script file.

For example, if you have multiple commands that you have to run at a specific time interval, you can compile a bash script instead of typing and executing the commands manually one by one. You only need to execute the script file when it’s necessary.

Here are some other advantages of using bash scripts:

* **Well-structured commands**– structure the commands in a sequence so that every time you execute the script, it will run in the right order.
* **Task automation**– automate the script execution at any defined time using cron’s time-based scheduler.
* **Transparency**– people can check the content of a script since it’s in a readable text file. However, if you run the commands using another program written in a different programming language, such as C++, you’ll need to access the source code.
* **Transferable**– if you transfer a script to other Linux distributions, it’ll still work, providing that shell commands are available on that particular operating system.

#### Pro Tip

Linux has a bash shell command manual. It contains descriptions of all technical terms and standard shell variables. Type and execute the **man** bash command to display the manual on the terminal.

## Get Familiar With Bash Commands

Bash is available on almost all types of Unix-based operating systems and doesn’t require a separate installation. You will need a Linux command line, also known as the Linux terminal. It’s a program that contains the shell and lets you execute bash scripts.

Use this command to check the list of available shells on your Unix operating system:

cat /etc/shells

The output should show a list like this:

/bin/bash

/bin/sh

/bin/tcsh

/bin/csh

Each bash shell script needs to start with **#!**followed by the absolute path to the bash interpreter. To view the path, enter this command:

which bash

It should produce the following output:

/bin/bash

This is the standard path to the bash interpreter on most Unix operating systems. To let the shell know that it should run commands using the bash interpreter, start the script with this line:

#!/bin/bash

**Important!** If you want to run bash scripts on a virtual private server, connect to it via an SSH client.

The next step is to write and compile the commands in a **.sh** file using a text editor. You will need a Unix text editor such as [**VIM**](https://www.vim.org/) or [**GNU Nano**](https://www.nano-editor.org/). In this tutorial, we’ll use the **Nano**text editor to create the file by inputting this command:

nano function.sh

This will open a new **.sh** file for you to edit. Begin by writing **#!/bin/bash** followed by bash commands.

**Important!** Once you’re done using the Nano text editor, press **Ctrl+X** to close it, then press **Y** and **Enter** to save the changes.

## Basic Bash Commands for Your First Bash Script

In order to successfully create your first bash script, you need to understand the essential bash commands. They are the main elements of a script, and you must know what they do and how to write them properly.

There are a lot of bash commands on Linux. To start things off, we’ll cover seven basic ones.

### 1. Comments

Comments feature a description on certain lines in the script. The terminal doesn’t parse comments during execution, so they won’t affect the output.

There are two ways to add comments to a script. The first method is by typing **#** at the beginning of a single-line comment.

#!/bin/bash

#Command below prints a Hello World text

echo “Hello, world!”

The second method is by using **:** followed by **‘**. This method works for multiple-line comments.

#!/bin/bash

read a

: ‘

The following commands prints

Hello, world!

‘

echo “Hello, World!”

### 2. Variables

Variables are symbols that represent a character, strings of characters, or numbers. You only need to type the variable name in a command line to use the defined strings or numbers.

To assign a variable, type the variable name and the string value like here:

testvar=“This is a test variable”

In this case, **testvar**will is the variable name, and **This is a test variable**is the string value. When assigning a variable, we recommend using a variable name that’s easy to remember and represents its value.

To read the variable value in the command line, use the **$**symbol before the variable name. Take a look at the example below:

#!/bin/bash

testvar=“This is a test variable”

echo $testvar

The second command line uses **echo** to print out the value of **testvar**. The output of that script will be:

This is a test variable

Let’s take a look at how you can enter a string value by using the **read** command and make the script compare two string values from different variables:

#!/bin/bash  
echo “Enter a number”   
read a #The user input in this command line will be stored as variable a  
b=50 #The value of variable b  
if [[$a -eq $b]]  
then  
echo “Same number”  
else  
echo “Different number”  
fi

The output of that script should be as follows:

Enter a number  
20  
Different number

Note that **line 3** is the value that becomes variable **a**.

The script compares the variable **a** with the value **20** and the variable **b** with the value **50**. Since the values are different, the script prints out **Different number**.

However, if the user inputs **50**, this will be the output:

Enter a number  
50  
Same number

This example also uses conditional statements, which we will discuss later.

### 3. echo Command

**echo** is a well-known command used in many programming languages. There are various options you can use with echo to print the output on the terminal.

The first and most common use of echo is to output standard text:

#!/bin/bash  
echo “Hello, world!”

The output of that command is **Hello, World!**By default, when using the echo command like this, the terminal will input a new line underneath that. If you want to echo an output without a new line, you can do so by using **-n**.

#!/bin/bash  
echo -n “Hello, world!”

Use the option **\n**to introduce a line break into the output. To enable the **backslash (\)**, you need to include **-e**.

#!/bin/bash  
echo -e “Hello, \nworld!”

The output of that command will look like this:

Hello,  
world!

The option **\t**adds a horizontal tab space:

#!/bin/bash  
echo -e “\tHello, world!”

This command’s output will indent the text to the right:

Hello, world!

You can also combine several options. For example, combine **\n**and **\t**to break the text into lines and indent it to the right:

#!/bin/bash  
echo -e “\n\tHello, \n\tworld!”

The output of that command will look like this:

Hello,

world!

### 4. Functions

A function compiles a set of commands into a group. If you need to execute the command again, simply write the function instead of the whole set of commands.

There are several ways of writing functions.

The first way is by starting with the function name and following it with parentheses and brackets:

function\_name () {  
first command  
second command  
}

Or, if you want to write it in a single line:

function\_name () { first command; second command; }

The second method to write a function is using the reserved word **function**followed by the function name. This eliminates the need for parentheses:

function function\_name {  
first command  
second command  
}

This method also has a single-line version:

function function\_name { first command; second command; }

For example, we can write two functions with multiple **echo** commands:

#!/bin/bash  
hello\_world () {  
echo “Hello, World!”  
echo “This is a test function”  
}  
print\_message () {  
echo “Let’s learn bash programming”  
echo “Enjoy this tutorial”  
}

Note that writing the functions as in the example above only defines them and doesn’t execute the contained commands. To execute a function, enter its name into the command line.

Now, let’s use the two examples above in a complete bash function, including its execution:

#!/bin/bash  
#Define a hello world function  
hello\_world () {  
echo “Hello, World!”  
echo “This is a test function”  
}  
#Define a print message function  
print\_message () {  
echo “Let’s learn bash programming”  
echo “Enjoy this tutorial”  
}  
#Execute the hello world function  
hello\_world  
#Execute the print message function  
print\_message

This is the output of the script above:

Hello, World!  
This is a test function  
Let’s learn bash programming  
Enjoy this tutorial

### 5. Loops

[**Loop bash commands**](https://www.hostinger.com/tutorials/bash-for-loop-guide-and-examples/) are useful if you want to execute commands multiple times. There are three types of them you can run in bash – **for**, **while**, and **until**.

The **for**loop runs the command for a list of items:

#!/bin/bash  
for item in [list]  
do  
[commands]  
done

The following example uses a **for**loop to print all the days of the week:

#!/bin/bash  
for days in Monday Tuesday Wednesday Thursday Friday Saturday Sunday  
do  
echo “Day: $days”  
done

On **line 2**, **“days”** automatically becomes a variable, with the values being the day names that follow. Then, in the **echo** command, we use the **$** symbol to call the variable values.

The output of that script will be as follows:

Day: Monday  
Day: Tuesday  
Day: Wednesday  
Day: Thursday  
Day: Friday  
Day: Saturday  
Day: Sunday

Notice that even with just one command line in the loop script, it prints out seven **echo**outputs.

The next type of loop is **while**. The script will evaluate a condition. If the condition is **true**, it will keep executing the commands until the output no longer meets the defined condition.

#!/bin/bash  
while [condition]  
do  
[commands]  
done

Let’s take a look at a simple example that involves a variable and increment operator, denoted as **((++))**:

#!/bin/bash  
i=0  
while [ $i -le 5 ]  
do   
echo $i  
((i++))  
done

The variable starts with a **0**value, and the increment operator will increase it by one. The condition set is less than or equal to five, so the command will keep iterating until the output reaches five. The output of that script will be as follows:

0  
1  
2  
3  
4  
5

The last type of loop, **until**, is the opposite of **while**. It will iterate the command until the condition becomes true.

If we want the same output as the **while** example above using the **until** loop, we can write the script like this:

#!/bin/bash  
i=0  
until [ $i -gt 5 ]  
do   
echo $i  
((i++))  
done

Now, this command will iterate until the output value reaches five. The output will be the same as our example with the **while** loop:

0  
1  
2  
3  
4  
5

### 6. Conditional Statements

Many programming languages, including bash, use conditional statements like **if**, **then**, and **else**for decision-making. They execute commands and print out outputs depending on the conditions.

The **if**statement is followed by a conditional expression. After that, it’s followed by **then**and the command to define the output of the condition. The script will execute the command if the condition expressed in the **if**statement is true.

However, if you want to execute a different command if the condition is false, add an **else**statement to the script and follow it with the command.

Let’s take a look at simple **if**, **then**, and **else**statements. Before the statement, we will include a variable so the user can input a value:

#!/bin/bash  
echo “Enter a number”  
read num  
if [[$num -gt 10]]  
then  
echo “The number is greater than 10”  
else  
echo “The number is not greater than 10”

### 7. Reading and Writing Files

There are several methods to read a file, with the **cat**command being the most popular one. Note that this command reads the **whole** file content.

To read the content line by line, use the **read**command and a loop. Before writing a script to read a file, make sure that the file exists first.

In the following example, we have a **to-do.txt**file that contains a to-do list:

Reply email  
Finish report  
Call clients  
Team evaluation

We’ll use the **cat** and **read**commands in our bash function to read and print the content of the **to-do.txt**file. The first part of the script will use the **cat**command, while the second part will use the **read** command in a loop.

#!/bin/bash  
echo “Reading the file using cat command”  
content=’cat to-do.txt’  
echo $content  
echo “Reading the file using read command and loop”  
filename=’to-do.txt’  
while read line  
do  
echo $line  
done<$filename

The output of the script will be as follows:

Reading the file using cat command  
Reply email Finish report Call clients Team evaluation  
Reading the file using read command and loop  
Reply email  
Finish report  
Call clients  
Team evaluation

To write a command output into a file, use the redirection operators, represented with the **>**and **>>**symbols, and follow them with the file name:

output > filename  
output >> filename

Be careful when choosing the operator. If the file exists, the **>** operator will overwrite its content with a zero-length string. It means you’ll lose the existing file content. If the inputted file name doesn’t exist, it will create it.

The **>>**operator, on the other hand, will add the output to the given file.

Here’s a simple redirection to write the output into a text file:

echo “Hello, world!” >> hello\_world.txt

Redirection also works with the **read** command to write any user input. This script example will add the input value into the **name.txt** file:

#!/bin/bash  
echo “Enter your name”  
read Name  
echo $Name >> name.txt

Because the script redirects the variable output into the file, you won’t see any output printed. To see the output by printing the file content, add the following command line to read the file:

echo ‘cat name.txt’

Make sure that you have the permission to read and write in the file to prevent the **permission denied**error. If you want to add the output to existing files, make sure to type in the correct file names.

## 4 Easy Functions to Try On Your First Bash Script

Now that we know some bash commands, we’ll look at more basic bash function examples for your first script.

As mentioned earlier, when you want to write a bash script file, use the nano filename.sh command to create and open a **.sh**file and start writing your bash functions. Don’t forget to exit and save the file when you’re done.

### Start with a Simple echo Function

Let’s start with a simple echo function. Start by defining the function name followed by the echo command on the next line, just like in the example below:

#!/bin/bash

testfunction () {

echo “My first function”

}

testfunction

Save this script in **testFunction.sh**. When you execute the script on the command line, you should see the following output:

My first function

Keep in mind that if you swap the position of the function definition with the function call, it will result in an error. Let’s see the example below:

#!/bin/bash  
testfunction  
testfunction(){  
echo “My first function”  
}

This snippet won’t work. It calls the function in the second command line and defines the function later. In this case, the interpreter can’t find the function when it executes the script, resulting in a **command not found**error.

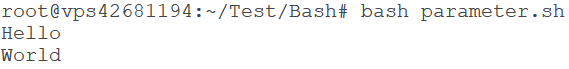


### Use a Few Parameters

Bash functions accept any number of parameters. The example below accepts two parameters:

#!/bin/bash  
testfunction () {  
echo $1  
echo $2  
}  
testfunction “Hello” “World”

**$1**represents the first argument while**$2** represents the second argument in the function execution line. As we used **“Hello”**and **“World”** for the arguments, the output will look like this:



You can also use the command line arguments and perform bash functions. One such example is shown below:

#!/bin/bash  
addition () {  
sum=$(($1+$2))  
return $sum  
}  
read -p "Enter a number: " int1  
read -p "Enter a number: " int2  
addition $int1 $int2  
echo "The result is : " $?

The addition is assigned in a variable **sum**, and this is returned from the function. Bash functions always return one single value. User input is taken by using **read** for both numbers. Finally, the result is printed using **$?** which stores the return value **$sum** from the function.

### Create Directories and Change Paths

Now, let’s look at another function example where we first create a directory and then change the path to point to a new location. This function will contain **mkdir**and **cd**[**Linux commands**](https://www.hostinger.com/tutorials/linux-commands) to create a new directory and change the current directory:

#!/bin/bash

sampleFunction () {

mkdir -p $1

cd $1

}

sampleFunction myDir

Function will read the first argument and create a directory with that name. After you execute the script, check the present working path using the **pwd** command on the terminal. You’ll see that you are currently within the newly created **myDir**.

### Combine Loops and Conditionals

Loops and conditional statements are also popular in bash scripting. We’ll look at a few instances of using both in the same script:

#!/bin/bash  
isvalid=true  
count=1  
while [ $isvalid ]  
do  
echo $count  
if [ $count -eq 5 ];  
then  
break  
fi  
((count++))  
done

The example above uses **while**and **if** statements. This executes the **while** loop five times after checking the conditional statement.

The output of this script will be:

1  
2  
3  
4  
5

The **for** loop can increment and decrement the counters. An example of a **for** loop is shown below:

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

The output of this for loop should be:

10 9 8 7 6 5 4 3 2 1

With **if** statements, we can also define **else if**by using**elif**statement:

#!/bin/bash  
echo "Enter a valid number"  
read n  
if [ $n -eq 101 ];  
then  
echo "This is the first number"  
elif [ $n -eq 510 ];  
then  
echo "This is the second number"  
elif [ $n -eq 999 ];  
then  
echo "This is the third number"  
else  
echo "No numbers over here"  
fi

We can also write that script using the **case**statement. In the **case**statements, **;;** represents a case break, so if the variable value meets any of the conditions, it jumps to the end of the script:

#!/bin/bash  
echo "Enter a valid number"  
read n  
case $n in  
101)  
echo "This is the first number" ;;  
510)  
echo "This is the second number" ;;  
999)  
echo "This is the third number" ;;  
\*)  
echo "No numbers over here" ;;  
esac

## How to Run Bash Script

Now that we have written a bash script, let’s learn how to run it from the terminal. There are three methods to do it – using the **bash**command, using the **./**command, and running the script from a different directory.

### Using the Bash Command

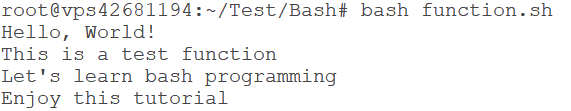
The first method is by using the **bash** command from the appropriate directory. For example, you may have a **function.sh**bash script containing simple echo functions in the **Test/Bash**directory. You have to open the directory first by using this command:

cd Test/Bash

Then, execute the following **bash**command to run the bash script:

bash function.sh

You should see the output like this:



If you try to run the script without the **bash**command, you’ll receive a **command not found**error message.



### Using the ./ Command

You can run a bash script without the **bash**command. However, you have to set the file to have the **execute**permission using the following command from the appropriate directory:

chmod +x function.sh

This command modifies the file permissions so that everyone can execute the file. Once you’ve done that, execute the bash script by using this command:

./function.sh

If you don’t set the permission correctly, the terminal will print a **Permission denied**error message:



Like the **bash**command, you’ll also get a **command not found**error if you don’t use **./**in your command.

### Run the Script From a Different Directory

Another alternative is to run the script from a different directory. After you create the bash script, use the **pwd**command to find your current directory. Once you know the path, you can run the script from any directory. For example, use the following command to run **function.sh** from the home directory:

bash Test/Bash/function.sh

#### Pro Tip

Use the **cd** command to go to the home directory straight away regardless of the directory you are in.

## Conclusion

Bash reads shell commands and interacts with the operating system to execute them. The great aspect of bash is that you can compile many bash commands and functions into a single executable script, helping you streamline your workflow.

To create a bash script file, you need to understand the various shell commands and their proper syntax. In this tutorial, we’ve covered seven basic commands:

* Comments
* Variables
* Echo
* Functions
* Loops
* Conditional statements
* Reading and writing files

However, there is much more to learn if you want to be able to utilize the full potential of bash. Practice with the examples we have provided and continue exploring bash so you can write better and more efficient scripts.