BY BOBBY ILIEV

Introduction to Bash Scripting

FOR DEVELOPERS

- **This version was published on Feb 01 2021**

This is an open-source introduction to Bash scripting guide that will help you learn the basics of Bash scripting and start writing awesome Bash scripts that will help you automate your daily SysOps, DevOps, and Dev tasks. No matter if you are a DevOps/SysOps engineer, developer, or just a Linux enthusiast, you can use Bash scripts to combine different Linux commands and automate tedious and repetitive daily tasks so that you can focus on more productive and fun things.

The guide is suitable for anyone working as a developer, system administrator, or a DevOps engineer and wants to learn the basics of Bash scripting.

The first 13 chapters would be purely focused on getting some solid Bash scripting foundations, then the rest of the chapters would give you some real-life examples and scripts.

My name is Bobby Iliev, and I have been working as a Linux DevOps Engineer since 2014. I am an avid Linux lover and supporter of the open-source movement philosophy. I am always doing that which I cannot do in order that I may learn how to do it, and I believe in sharing knowledge.

I think it's essential always to keep professional and surround yourself with good people, work hard, and be nice to everyone. You have to perform at a consistently higher level than others. That's the mark of a true professional.

For more information, please visit my blog at https://bobbyiliev.com, follow me on Twitter @bobbyiliev_ and YouTube.

This book is made possible thanks to these fantastic companies!

The Streaming Database for Real-time Analytics.

Materialize is a reactive database that delivers incremental view updates. Materialize helps developers easily build with streaming data using standard SQL.

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale.

It provides highly available, secure, and scalable compute, storage, and networking solutions that help developers build great software faster.

Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available.

For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

If you are new to DigitalOcean, you can get a free $100 credit and spin up your own servers via this referral link here:

Free $100 Credit For DigitalOcean

The DevDojo is a resource to learn all things web development and web design. Learn on your lunch break or wake up and enjoy a cup of coffee with us to learn something new.

Join this developer community, and we can all learn together, build together, and grow together.

Join DevDojo

For more information, please visit https://www.devdojo.com or follow @thedevdojo on Twitter.

This ebook was generated by Ibis developed by Mohamed Said.

Ibis is a PHP tool that helps you write eBooks in markdown.

The cover for this ebook was created with Canva.com.

If you ever need to create a graphic, poster, invitation, logo, presentation – or anything that looks good — give Canva a go.

MIT License

Welcome to this Bash basics training guide! In this **bash crash course**, you will learn the **Bash basics** so you could start writing your own Bash scripts and automate your daily tasks.

Bash is a Unix shell and command language. It is widely available on various operating systems, and it is also the default command interpreter on most Linux systems.

Bash stands for Bourne-Again SHell. As with other shells, you can use Bash interactively directly in your terminal, and also, you can use Bash like any other programming language to write scripts. This book will help you learn the basics of Bash scripting including Bash Variables, User Input, Comments, Arguments, Arrays, Conditional Expressions, Conditionals, Loops, Functions, Debugging, and testing.

In order to write Bash scripts, you just need a UNIX terminal and a text editor like Sublime Text, VS Code, or a terminal-based editor like vim or nano.

Let's start by creating a new file with a `.sh` extension. As an example, we could create a file called `devdojo.sh`.

To create that file, you can use the `touch` command:

```
touch devdojo.sh
```

Or you can use your text editor instead:

```
nano devdojo.sh
```

In order to execute/run a bash script file with the bash shell interpreter, the first line of a script file must indicate the absolute path to the bash executable:

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

This is also called a Shebang.

All that the shebang does is to instruct the operating system to run the script with the `/bin/bash` executable.

Once we have our `devdojo.sh` file created and we've specified the bash shebang on the very first line, we are ready to create our first `Hello World` bash script.

To do that, open the `devdojo.sh` file again and add the following after the `#!/bin/bash` line:

```
#!/bin/bash

echo "Hello World!"
```

Save the file and exit.

After that make the script executable by running:

```
chmod +x devdojo.sh
```

After that execute the file:

```
./devdojo.sh
```

You will see a "Hello World" message on the screen.

Another way to run the script would be:

```
bash devdojo.sh
```

As bash can be used interactively, you could run the following command directly in your terminal and you would get the same result:

```
echo "Hello DevDojo!"
```

Putting a script together is useful once you have to combine multiple commands together.

As in any other programming language, you can use variables in Bash Scripting as well. However, there are no data types, and a variable in Bash can contain numbers as well as characters.

To assign a value to a variable, all you need to do is use the `=` sign:

```
name="DevDojo"
```

> **Notice:** as an important note, you can not have spaces before and after the `=` sign.

After that, to access the variable, you have to use the `$` and reference it as shown below:

```
echo $name
```

Wrapping the variable name between curly brackets is not required, but is considered a good practice, and I would advise you to use them whenever you can:

```
echo ${name}
```

The above code would output: `DevDojo` as this is the value of our `name` variable.

Next, let's update our `devdojo.sh` script and include a variable in it.

Again, you can open the file `devdojo.sh` with your favorite text editor, I'm using nano here to open the file:

```
nano devdojo.sh
```

Adding our `name` variable here in the file, with a welcome message. Our file now looks like this:

```
#!/bin/bash

name="DevDojo"

echo "Hi there $name"
```

Save it and run the file using the command below:

```
./devdojo.sh
```

You would see the following output on your screen:

```
Hi there DevDojo
```

Here is a rundown of the script written in the file:

- `#!/bin/bash` - At first, we specified our shebang.
- `name=DevDojo` - Then, we defined a variable called `name` and assigned a value to it.
- `echo "Hi there $name"` - Finally, we output the content of the variable on the screen as a welcome message by using `echo`

You can also add multiple variables in the file as shown below:

```
#!/bin/bash

name="DevDojo"
greeting="Hello"

echo "$greeting $name"
```

Save the file and run it again:

```
./devdojo.sh
```

You would see the following output on your screen:

```
Hello DevDojo
```

Note that you don't necessarily need to add semicolon `;` at the end of each line. It works both ways, a bit like other programming language such as JavaScript!

With the previous script, we defined a variable, and we output the value of the variable on the screen with the `echo $name`.

Now let's go ahead and ask the user for input instead. To do that again, open the file with your favorite text editor and update the script as follows:

```
#!/bin/bash

echo "What is your name?"
read name

echo "Hi there $name"
echo "Welcome to DevDojo!"
```

The above will prompt the user for input and then store that input as a string/text in a variable.

We can then use the variable and print a message back to them.

The output of the above script would be:

- First run the script:

```
./devdojo.sh
```

- Then, you would be prompted to enter your name:

```
What is your name?
Bobby
```

- Once you've typed your name, just hit enter, and you will get the following output:

```
Hi there Bobby
Welcome to DevDojo!
```

To reduce the code, we could change the first `echo` statement with the `read -p`, the `read` command used with `-p` flag will print a message before prompting the user for their input:

```
#!/bin/bash

read -p "What is your name? " name

echo "Hi there $name"
echo "Welcome to DevDojo!"
```

Make sure to test this out yourself as well!

As with any other programming language, you can add comments to your script. Comments are used to leave yourself notes through your code.

To do that in Bash, you need to add the # symbol at the beginning of the line. Comments will never be rendered on the screen.

Here is an example of a comment:

```
# This is a comment and will not be rendered on the screen
```

Let's go ahead and add some comments to our script:

```
#!/bin/bash

# Ask the user for their name

read -p "What is your name? " name

# Greet the user
echo "Hi there $name"
echo "Welcome to DevDojo!"
```

Comments are a great way to describe some of the more complex functionality directly in your scripts so that other people could find their way around your code with ease.

You can pass arguments to your shell script when you execute it. To pass an argument, you just need to write it right after the name of your script. For example:

```
./devdojo.com your_argument
```

In the script, we can then use `$1` in order to reference the first argument that we specified.

If we pass a second argument, it would be available as `$2` and so on.

Let's create a short script called `arguments.sh` as an example:

```
#!/bin/bash

echo "Argument one is $1"
echo "Argument two is $2"
echo "Argument three is $3"
```

Save the file and make it executable:

```
chmod +x arguments.sh
```

Then run the file and pass **3** arguments:

```
./arguments.sh dog cat bird
```

The output that you would get would be:

```
Argument one is dog
Argument two is cat
Argument three is bird
```

To reference all arguments, you can use `$@`:

```
#!/bin/bash

echo "All arguments: $@"
```

If you run the script again:

```
./arguments.sh dog cat bird
```

You will get the following output:

```
All arguments: dog cat bird
```

Another thing that you need to keep in mind is that `$0` is used to reference the script itself.

This is an excellent way to create self destruct the file if you need to or just get the name of the script.

For example, let's create a script that prints out the name of the file and deletes the file after that:

```
#!/bin/bash

echo "The name of the file is: $0 and it is going to be self-deleted."

rm -f $0
```

You need to be careful with the self deletion and ensure that you have your script backed up before you self-delete it.

If you have ever done any programming, you are probably already familiar with arrays.

But just in case you are not a developer, the main thing that you need to know is that unlike variables, arrays can hold several values under one name.

You can initialize an array by assigning values devided by space and enclosed in `()`. Example:

```
my_array=("value 1" "value 2" "value 3" "value 4")
```

To access the elements in the array, you need to reference them by their numeric index.

> **Notice:** keep in mind that you need to use curly brackets.

- Access a single element, this would output: `value 2`

```
echo ${my_array[1]}
```

- This would return the last element: `value 4`

```
echo ${my_array[-1]}
```

- As with command line arguments using `@` will return all arguments in the array, as follows: `value 1 value 2 value 3 value 4`

```
echo ${my_array[@]}
```

- Prepending the array with a hash sign (#) would output the total number of elements in the array, in our case it is 4:

```
echo ${#my_array[@]}
```

Make sure to test this and practice it at your end with different values.

Let's review the following example of slicing in a string in Bash:

```
#!/bin/bash

letters=( "A""B""C""D""E" )
echo ${letters[@]}
```

This command will print all the elements of an array.

Output:

```
$ ABCDE
```

Lets see a few more examples:

- Example 1

```
#!/bin/bash

letters=( "A""B""C""D""E" )
b=${letters:0:2}
echo "${b}"
```

This command wil print array from starting index 0 to 2 where 2 is exclusive.

```
$ AB
```

- Example 2

```
#!/bin/bash

letters=( "A""B""C""D""E" )
b=${letters::5}
echo "${b}"
```

This command will print from base index 0 to 5, where 5 is exclusive and starting index is default set to 0 .

```
$ ABCDE
```

- Example 3

```
#!/bin/bash

letters=( "A""B""C""D""E" )
b=${letters:3}
echo "${b}"
```

This command will print from starting index 3 to end of array inclusive .

```
$ DE
```

In computer science, conditional statements, conditional expressions, and conditional constructs are features of a programming language, which perform different computations or actions depending on whether a programmer-specified boolean condition evaluates to true or false.

In Bash, conditional expressions are used by the `[[` compound command and the `[` built-in commands to test file attributes and perform string and arithmetic comparisons.

Here is a list of the most popular Bash conditional expressions. You do not have to memorize them by heart. You can simply refer back to this list whenever you need it!

- True if file exists.

```
[[ -a ${file} ]]
```

- True if file exists and is a block special file.

```
[[ -b ${file} ]]
```

- True if file exists and is a character special file.

```
[[ -c ${file} ]]
```

- True if file exists and is a directory.

```
[[ -d ${file} ]]
```

- True if file exists.

```
[[ -e ${file} ]]
```

- True if file exists and is a regular file.

```
[[ -f ${file} ]]
```

- True if file exists and is a symbolic link.

```
[[ -h ${file} ]]
```

- True if file exists and is readable.

```
[[ -r ${file} ]]
```

- True if file exists and has a size greater than zero.

```
[[ -s ${file} ]]
```

- True if file exists and is writable.

```
[[ -w ${file} ]]
```

- True if file exists and is executable.

```
[[ -x ${file} ]]
```

- True if file exists and is a symbolic link.

```
[[ -L ${file} ]]
```

- True if the shell variable varname is set (has been assigned a value).

```
[[ -v ${varname} ]]
```

True if the length of the string is zero.

```
[[ -z ${string} ]]
```

True if the length of the string is non-zero.

```
[[ -n ${string} ]]
```

- True if the strings are equal. `=` should be used with the test command for POSIX conformance. When used with the `[[` command, this performs pattern matching as described above (Compound Commands).

```
[[ ${string1} == ${string2} ]]
```

- True if the strings are not equal.

```
[[ ${string1} != ${string2} ]]
```

- True if string1 sorts before string2 lexicographically.

```
[[ ${string1} < ${string2} ]]
```

- True if string1 sorts after string2 lexicographically.

```
[[ ${string1} > ${string2} ]]
```

- Returns true if the numbers are **equal**

```
[[ ${arg1} -eq ${arg2} ]]
```

- Returns true if the numbers are **not equal**

```
[[ ${arg1} -ne ${arg2} ]]
```

- Returns true if arg1 is **less than** arg2

```
[[ ${arg1} -lt ${arg2} ]]
```

- Returns true if arg1 is **less than or equal** arg2

```
[[ ${arg1} -le ${arg2} ]]
```

- Returns true if arg1 is **greater than** arg2

```
[[ ${arg1} -gt ${arg2} ]]
```

- Returns true if arg1 is **greater than or equal** arg2

```
[[ ${arg1} -ge ${arg2} ]]
```

As a side note, arg1 and arg2 may be positive or negative integers.

As with other programming languages you can use `AND` & `OR` conditions: