Published in Introduction into BASH

Mikhail Raevskiy Follow

Jul 23, 2020 · 11 min read ★ · Listen

Save

Bash Scripts — Part 1 — Getting Started

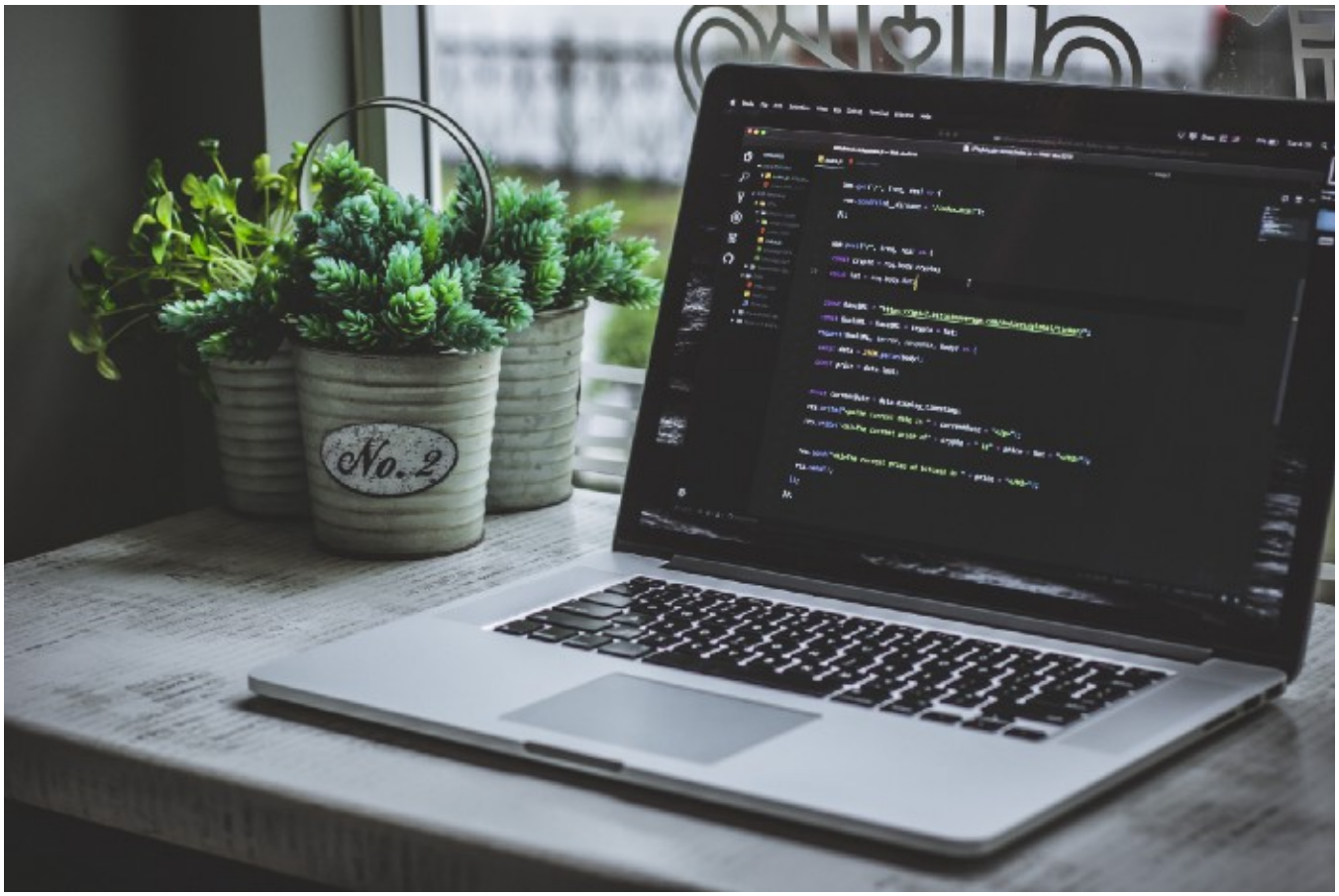

Photo by Safar Safarov on Unsplash

Command-line scripts are sets of the same commands that can be entered from the keyboard, assembled into files, and united by some common purpose. At the same time, the results of the work of teams can be either of independent value or serve as input data for other teams. Scripting is a powerful way to automate frequently performed actions. So, speaking of the command line, it allows you to execute several commands at one time, entering them separated by semicolons:

```
pwd ; whoami
```

In fact, if you've tried it in your terminal, your first bash script that uses two commands has already been written. It works like this. First, the command `pwd` displays information about the current working directory, then the command `whoami` displays information about the user under which you are logged in.

Using this approach, you can combine as many commands as you like on one line, the limitation is only in the maximum number of arguments that can be passed to the program. You can determine this limitation using the following command:

```
getconf ARG_MAX
```

Command-line is a great tool, but you have to enter commands every time you need them. What if you write a set of commands to a file and just call that file to execute them? In fact, the file we are talking about is called a command-line script.

How bash scripts work

Create an empty file using the command `touch` . In its first line, we need to indicate which shell we are going to use. We're interested in `bash` , so the first line of the file will


In other lines of this file, the hash character is used to denote comments that the shell does not process. However, the first line is a special case, here a hash followed by an exclamation point (this sequence is called shebang) and the path to `bash` , indicate to the system what the script was created for `bash` .

Shell commands are delimited with line feeds, comments are delimited with hash marks. This is how it looks:

```
#!/bin/bash
# This is a comment
pwd
whoami
```

Here, just as in the command line, you can write commands on one line, separated by semicolons. However, writing the commands on separate lines makes the file easier to read. The shell will process them anyway.

Setting script file permissions

Save the file with a name `myscript` and you are almost done creating the bash script. Now all that remains is to make this file executable, otherwise, trying to run it, you will encounter an error `Permission denied` .

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
bash: ./myscript: Permission denied
```

Let’s make the file executable:

```
chmod +x ./myscript
```


```
./myscript
```

After setting the permissions, everything works as it should:

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
/home/raevskym/bash_course
raevskym
```

Displaying messages

The command is used to output text to the Linux console `echo` . Let's take advantage of the knowledge of this fact and edit our script, adding explanations to the data that are displayed by the commands already in it:

```
#!/bin/bash
# our comment is here
echo "The current directory is:"
pwd
echo "The user logged in is:"
whoami
```

This is what happens after running the updated script:

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
The current directory is
/home/raevskym/bash_course
The used logged in is:
raevskym
```

Now we can display explanatory notes using the command `echo` . If you don't know how


Using variables

Variables allow you to store information in the script file, for example — the results of commands for use by other commands.

There is nothing wrong with executing individual commands without storing the results of their work, but the possibilities of this approach are very limited.

There are two types of variables that can be used in bash scripts:

- Environment Variables
- User variables

Environment Variables

Sometimes shell commands need to work with some system data. For example, here's how to display the home directory of the current user:

```
#!/bin/bash
# display user home
echo "Home for the current user is: $HOME"
```

Please note that we can use the system variable `$HOME` in double-quotes, this will not prevent the system from recognizing it. This is what happens if you run the above script.

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
Home for the current user is: /home/mraevsky
```

What if you want to display a dollar sign? Let's try this:


The system will detect the dollar sign in the quoted string and assume that we referenced a variable. The script will try to display the value of an undefined variable `$1`. This is not what we want. What to do?

Using the backslash control character before the dollar sign can help in this situation:

```
echo "I have \$1 in my pocket"
```

The script will now output exactly what is expected:

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
I have $1 in my pocket
```

User variables

In addition to environment variables, bash scripts allow you to set and use your own variables in the script. These variables hold their value until the script finishes executing.

As in the case of system variables, user variables can be accessed using the dollar sign:

```
#!/bin/bash
# testing variables
grade=5
person="Adam"
echo "$person is a good boy, he is in grade $grade"
```

Here's what happens after running such a script:

Command substitution

One of the most useful features of bash scripts is the ability to extract information from the output of commands and assign it to variables, which allows this information to be used anywhere in the script file.

This can be done in two ways.

- With the backtick “”
- With construction `$()`

Using the first approach, be careful not to use a single quote instead of the backtick. The command must be enclosed in two such icons:

```
mydir=`pwd`
```

In the second approach, the same thing is written like this:

```
mydir=$(pwd)
```

And the script, in the end, may look like this:

```
#!/bin/bash
mydir=$(pwd)
echo $mydir
```

During its operation, the output of the command `pwd` will be saved in a variable `mydir`, the contents of which, with the help of the command `echo`, will go to the console.


Mathematical operations

To perform mathematical operations in a script file, you can use a construction of the form `$((a+b))`:

```
#!/bin/bash
var1=$(( 5 + 5 ))
echo $var1
var2=$(( $var1 * 2 ))
echo $var2
```

We will get the following output:

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
10
20
```

If-then control construct

Some scenarios require control of the command execution flow. For example, if some value is more than five, you need to perform one action, otherwise — another. This is applicable in so many situations, and here the control construct will help us `if-then`. In its simplest form, it looks like this:

```
if <some-condition>
then
<some-commands-go-here>
fi
```


```
then
echo "It works"
fi
```

In this case, if the command is `pwd` completed successfully, the text "it works" will be displayed in the console.

Let's take advantage of our knowledge and write a more complex script. Let's say you need to find a certain user in `/etc/passwd`, and if you manage to find it, report that it exists.

```
#!/bin/bash
user=likegeeks
if grep $user /etc/passwd
then
echo "The user $user Exists"
fi
```

This is what happens after running this script:

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
raevskym:x:1000:1000:raevskym,,,:/home/raevskym:/bin/bash
The user raevskym Exists
```

Here we used the command `grep` to find a user in a file `/etc/passwd`. If the command `grep` is unfamiliar to you, its description can be found here.

In this example, if a user is found, the script will display an appropriate message. What if the user cannot be found? In this case, the script will simply complete execution without informing us of anything. I would like him to tell us about this too, so we will improve the code.


a failure, we will use the construction `if-then-else`. Here's how it works:

```
if <some-condition>
then
<some-commands>
else
<some-commands>
fi
```

If the first command returns zero, which means its successful execution, the condition will be true and the execution will not go down the branch `else`. Otherwise, if something other than zero is returned, which means failure, or a false result, the commands after it will be executed `else`.

Let's write a script like this:

```
#!/bin/bash
user=anotherUser
if grep $user /etc/passwd
then
echo "The user $user Exists"
else
echo "The user $user doesn’t exist"
fi
```

His execution went on a branch `else`.

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
The user anotherUser doesn't exist
```

Well, let’s move on and ask ourselves about more complex conditions. What if you need


```
if <condition-1>
then
<some-commands>
elif <condition-2>
then
<some-other-commands>
fi
```

If the first command returns zero, which indicates its successful execution, the commands in the first block will be executed `then`, otherwise, if the first condition turns out to be false, and if the second command returns zero, the second block of code will be executed.

```
#!/bin/bash
user=anotherUser
if grep $user /etc/passwd
then
echo "The user $user Exists"
elif ls /home
then
echo "The user doesn't exist but anyway there is a directory under
/home"
fi
```

In such a script, you can, for example, create a new user using a command `useradd` if the search did not return any results, or do something else useful.

Comparing numbers

Numeric values can be compared in scripts. Below is a list of the relevant commands.

> `n1 -eq n2` Returns true if `n1` equal `n2`.
>
> `n1 -ge n2` Returns true if `n1` greater or equal `n2`.


> `n1 -ne n2` Returns true if `n1` not equal `n2` .

Let's try one of the comparison operators as an example. Note that the expression is enclosed in square brackets.

```
#!/bin/bash
val1=6
if [ $val1 -gt 5 ]
then
echo "The test value $val1 is greater than 5"
else
echo "The test value $val1 is not greater than 5"
fi
```

This is what this command will output:

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
The test value is greater than 5
```

The value of the variable is `val1` greater than 5, as a result, the branch of `then` the comparison operator is executed and the corresponding message is displayed in the console.

String comparison

You can also compare string values in scripts. Comparison operators look pretty simple, but string comparison operations have certain peculiarities, which we will touch on below. Here is a list of operators.

> `str1 = str2` Tests strings for equality, returns true if strings are identical.
>
> `str1 != str2` Returns true if the strings are not identical.


Here's an example of comparing strings in a script:

```
#!/bin/bash
user ="likegeeks"
if [$user = $USER]
then
echo "The user $user  is the current logged in user"
fi
```

As a result of the script execution, we get the following.

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
The user raevskym is the current logged in user
```

Here is one feature worth mentioning about string comparison. Namely, the operators ">" and "<" must be escaped with a backslash, otherwise, the script will not work correctly, although no error messages will appear. The script interprets the ">" sign as an output redirection command.

This is how working with these operators looks like in code:

```
#!/bin/bash
val1=text
val2="another text"
if [ $val1 \> $val2 ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```


```
text is less than another text
```

Note that the script, although executed, generates a warning:

```
./myscript: line 5: [: too many arguments
```

To get rid of this warning, `$val2` enclose in double-quotes:

```
#!/bin/bash
val1=text
val2="another text"
if [ $val1 \> "$val2" ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```

Now everything works as it should.

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
text is greater than another text
```

Another feature of the “>” and “<” operators is how they work with upper and lower case characters. In order to understand this feature, let’s prepare a text file with the following content:

```
Raevskym
raevskym
```


```
sort myfile
```

It will sort the lines from the file like this:

```
raevskym
Raevskym
```

The command `sort`, by default, sorts lines in ascending order, that is, the lowercase letter in our example is smaller than the uppercase one. Now let's prepare a script that will compare the same strings:

```
#!/bin/bash
val1=Likegeeks
val2=likegeeks
if [ $val1 \> $val2 ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
```

If you run it, it turns out that the opposite is true — the lowercase letter is now larger than the uppercase one.

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
Raevskym is less than raevskym
raevskym@DESKTOP-JNF3L6H:~/bash_course$ sort ./myfile
raevskym
Raevskym
```


character codes.

The command `sort` , in turn, uses the sort order specified in the system language settings.

File checks

The commands below are probably the most commonly used in bash scripts. They allow you to check various conditions regarding files. Here is a list of these commands.

> `-d file` Checks if a file exists and is a directory.
> `-e file` Checks if the file exists.
> `-f file` Checks if a file exists and is a file.
> `-r file` Checks if the file exists and is readable.
> `-s file` ⊓ Checks if the file exists and if it is empty.
> `-w file` Checks if the file exists and is writable.
> `-x file` Checks if a file exists and is executable.
> `file1 -nt file2` Checks if newer `file1` than `file2` .
> `file1 -ot file2` Checks if it's older `file1` than `file2` .
> `-O file` Checks if the file exists and is owned by the current user.
> `-G file` Checks if the file exists and if its group ID matches the group ID of the current user.

These commands, as well as many others, discussed today, are easy to remember. Their names, being abbreviations from various words, directly indicate the checks they perform.

Let’s try one of the commands in practice:

```
#!/bin/bash
mydir=/home/likegeeks
```


```
else
echo "The $mydir directory does not exist"
fi
```

This script, for an existing directory, will display its contents.

```
raevskym@DESKTOP-JNF3L6H:~/bash_course$./myscript
The /home/raevskym/bash_course directory exists
bash_seminar_1.md  myfile  Images  Notes
```

We suppose you can experiment with the rest of the commands yourself, they all apply in the same way.

Outcome

Today we have covered how to get started writing bash scripts and covered some basic things. In fact, the topic of bash programming is huge.

Source: https://habr.com/ru/company/ruvds/