

# Learning Bash with Rocky

## (English version)

---

A book from the Documentation Team

Version : 2025/12/13

*Rocky Documentation Team*

*Copyright © 2023 The Rocky Enterprise Software Foundation*

## Table of contents

---

1. Licence	4
2. Learning Bash with Rocky	5
2.1 Generalities	5
3. Bash - First script	7
3.1 My first script	7
4. Bash - Using Variables	10
4.1 Storing information for later use	10
4.2 Delete and lock variables	12
4.3 Use environment variables	13
4.4 Substitute commands	14
5. Bash - Data entry and manipulations	17
5.1 The read command	17
5.2 The cut command	19
5.3 The tr command	20
5.4 Extract the name and path of a file	20
5.5 Arguments of a script	21
5.5.1 The shift command	23
5.5.2 The set command	24
6. Bash - Check your knowledge	26
7. Bash - Tests	28
7.1 Testing the type of a file	30
7.2 Compare two files	32
7.3 Testing variables	32
7.4 Testing strings	32
7.5 Comparison of integer numbers	33
7.6 Combined tests	34
7.7 Numerical operations	34
7.8 The typeset command	35
7.9 The let command	35
8. Bash - Loops	37
8.1 The while conditional loop structure	37
8.2 The exit command	38
8.3 The break / continue commands	39
8.4 The true / false commands	39
8.5 The until conditional loop structure	40

8.6 The alternative choice structure select	40
8.7 The loop structure on a list of values for	42
9. Bash - Check your knowledge	44

## 1. Licence

---

RockyLinux offers Linux courseware for trainers or people wishing to learn how to administer a Linux system on their own.

RockyLinux materials are published under Creative Commons-BY-SA. This means you are free to share and transform the material, while respecting the author's rights.

**BY : Attribution.** You must cite the name of the original author.

**SA : Share Alike.**

- Creative Commons-BY-SA licence : <https://creativecommons.org/licenses/by-sa/4.0/>

The documents and their sources are freely downloadable from:

- <https://docs.rockylinux.org>
- <https://github.com/rocky-linux/documentation>

Our media sources are hosted at [github.com](https://github.com). You'll find the source code repository where the version of this document was created.

From these sources, you can generate your own personalized training material using [mkdocs](#). You will find instructions for generating your document [here](#).

How can I contribute to the documentation project?

You'll find all the information you need to join us on our [git project home page](#).

We wish you all a pleasant reading and hope you enjoy the content.

## 2. Learning Bash with Rocky

---

In this section, you will learn more about Bash scripting, an exercise that every administrator will have to perform one day or another.

### 2.1 Generalities

---

The shell is the command interpreter of Linux. It is a binary that is not part of the kernel, but forms an additional layer, hence its name "shell".

It parses the commands entered by the user and then executes them by the system.

There are several shells, all of which share some common features. The user is free to use the one that suits them best. Some examples are:

- the **Bourne-Again shell** (`bash`),
- the **Korn shell** (`ksh`),
- the **C shell** (`csh`),
- etc.

`bash` is present by default in most (all) Linux distributions. It is characterized by its practical and user-friendly features.

The shell is also a **basic programming language** which, thanks to some dedicated commands, allows:

- the use of **variables**,
- **conditional execution** of commands,
- the **repetition** of commands.

Shell scripts have the advantage that they can be created **quickly** and **reliably**, without **compiling** or installing additional commands. A shell script is just a text file without any embellishments (bold, italics, etc.).



#### Note

Although the shell is a "basic" programming language, it is still very powerful and sometimes faster than badly compiled code.

To write a shell script, you just have to put all the necessary commands in a single text file. By making this file executable the shell reads it sequentially, and executes the commands in it one by one. It is also possible to execute it by passing the name of the script as an argument to the bash binary.

When the shell encounters an error, it displays a message to identify the problem but continues to execute the script. But there are mechanisms to stop the execution of a script when an error occurs. Command-specific errors are also displayed on the screen or inside files.

What is a good script? It is:

- **reliable**: its operation is flawless even in case of misuse;
- **commented**: its code is annotated to facilitate the rereading and future evolution;
- **readable**: the code is indented appropriately, the commands are spaced out, ...
- **portable**: the code runs on any Linux system, dependency management, rights management, etc.

## 3. Bash - First script

---

In this chapter you will learn how to write your first script in bash.

---

**Objectives:** In this chapter you will learn how to:

- ✓ Write your first script in bash;
- ✓ Execute your first script;
- ✓ Specify which shell to use with the so-called shebang;

### ☒ linux, script, bash

**Knowledge:** ★

**Complexity:** ★

**Reading time:** 10 minutes

---

### 3.1 My first script

---

To start writing a shell script, it is convenient to use a text editor that supports syntax highlighting.

`vim`, for example, is a good tool for this.

The name of the script should respect some rules:

- no names of existing commands;
- only alphanumeric characters, i.e. no accented characters or spaces;
- extension `.sh` to indicate that it is a shell script.

 **Note**

The author uses the "\$" throughout these lessons to indicate the user's command-prompt.

```
#!/usr/bin/env bash
#
# Author : Rocky Documentation Team
```

```
# Date: March 2022
# Version 1.0.0: Displays the text "Hello world!"
#
# Displays a text on the screen :
echo "Hello world!"
```

To be able to run this script, as an argument to bash:

```
$ bash hello-world.sh
Hello world !
```

Or, more simply, after having given it the right to execute:

```
$ chmod u+x ./hello-world.sh
$ ./hello-world.sh
Hello world !
```

#### Note

To execute the script, it needs to be called with `./` before its name when you are in the directory where the script resides. If not in that directory, you will need to call it with the entire path to the script, OR place it in a directory that is within your PATH environment variable: (Examples: `/usr/local/sbin`, `/usr/local/bin`, etc.) The interpreter will refuse to execute a script present in the current directory without indicating a path (here with `./` before it).

The `chmod` command is to be passed only once on a newly created script.

The first line to be written in any script is to indicate the name of the shell binary to be used to execute it. If you want to use the `ksh` shell or the interpreted language `python`, you would replace the line:

```
#!/usr/bin/env bash
```

with :

```
#!/usr/bin/env ksh
```

or with :

```
#!/usr/bin/env python
```

This first line is called the `shebang`. It starts with the characters `#!` followed by the path to the binary of the command interpreter to use.

#### About the shebang

You may have encountered the "shebang" in a script that you've looked at that does not contain the "env" section and simply contains the interpreter to use. (Example: `#!/bin/bash`). The author's method is considered to be the recommended and proper way to format the "shebang".

Why is the author's method recommended? Because it increases the portability of the script. If for some reason the interpreter lived in an entirely different directory, the interpreter would **still** be found if you used the author's method.

Throughout the writing process, you should think about proofreading the script, using comments in particular:

- a general presentation, at the beginning, to indicate the purpose of the script, its author, its version, its use, etc.
- during the text to help understand the actions.

Comments can be placed on a separate line or at the end of a line containing a command.

Example:

```
# This program displays the date
date # This line is the line that displays the date!
```

## 4. Bash - Using Variables

---

In this chapter you will learn how to use variables in your bash scripts.

---

**Objectives:** In this chapter you will learn how to:

- ✓ Store information for later use;
- ✓ Delete and lock variables;
- ✓ Use environment variables;
- ✓ Substitute commands;

**☒ linux, script, bash, variable**

**Knowledge:** ★★

**Complexity:** ★

**Reading time:** 10 minutes

---

### 4.1 Storing information for later use

---

As in any programming language, the shell script uses variables. They are used to store information in memory to be reused as needed during the script.

A variable is created when it receives its content. It remains valid until the end of the execution of the script or at the explicit request of the script author. Since the script is executed sequentially from start to finish, it is impossible to call a variable before it is created.

The content of a variable can be changed during the script, as the variable continues to exist until the script ends. If the content is deleted, the variable remains active but contains nothing.

The notion of a variable type in a shell script is possible but is very rarely used. The content of a variable is always a character or a string.

```

#!/usr/bin/env bash

#
# Author : Rocky Documentation Team
# Date: March 2022
# Version 1.0.0: Save in /root the files passwd, shadow, group, and gshadow
#

# Global variables
FILE1=/etc/passwd
FILE2=/etc/shadow
FILE3=/etc/group
FILE4=/etc/gshadow

# Destination folder
DESTINATION=/root

# Clear the screen
clear

# Launch the backup
echo "Starting the backup of $FILE1, $FILE2, $FILE3, $FILE4 to $DESTINATION"

cp $FILE1 $FILE2 $FILE3 $FILE4 $DESTINATION

echo "Backup ended!"

```

This script makes use of variables. The name of a variable must start with a letter but can contain any sequence of letters or numbers. Except for the underscore "`_`", special characters cannot be used.

By convention, variables created by a user have a name in lower case. This name must be chosen with care so as not to be too evasive or too complicated. However, a variable can be named with upper case letters, as in this case, if it is a global variable that should not be modified by the program.

The character `=` assigns content to a variable:

```

variable=value
rep_name="/home"

```

There is no space before or after the `=` sign.

Once the variable is created, it can be used by prefixing it with a dollar \$.

```
file=file_name
touch $file
```

It is strongly recommended to protect variables with quotes, as in this example below:

```
file=file name
touch $file
touch "$file"
```

As the content of the variable contains a space, the first `touch` will create 2 files while the second `touch` will create a file whose name will contain a space.

To isolate the name of the variable from the rest of the text, you must use quotes or braces:

```
file=file_name
touch "$file"1
touch ${file}1
```

### **The systematic use of braces is recommended.**

The use of apostrophes inhibits the interpretation of special characters.

```
message="Hello"
echo "This is the content of the variable message: $message"
Here is the content of the variable message: Hello
echo 'Here is the content of the variable message: $message'
Here is the content of the variable message: $message
```

---

## 4.2 Delete and lock variables

The `unset` command allows for the deletion of a variable.

Example:

```
name="NAME"
firstname="Firstname"
echo "$name $firstname"
```

```
NAME Firstname
unset firstname
echo "$name $firstname"
NAME
```

The `readonly` or `typeset -r` command locks a variable.

Example:

```
name="NAME"
readonly name
name="OTHER NAME"
bash: name: read-only variable
unset name
bash: name: read-only variable
```

#### Note

A `set -u` at the beginning of the script will stop the execution of the script if undeclared variables are used.

## 4.3 Use environment variables

**Environment variables** and **system variables** are variables used by the system for its operation. By convention these are named with capital letters.

Like all variables, they can be displayed when a script is executed. Even if this is strongly discouraged, they can also be modified.

The `env` command displays all the environment variables used.

The `set` command displays all used system variables.

Among the dozens of environment variables, several are of interest to be used in a shell script:

Variables	Description
<code>HOSTNAME</code>	Host name of the machine.
<code>USER</code> , <code>USERNAME</code> and <code>LOGNAME</code>	Name of the user connected to the session.
<code>PATH</code>	Path to find the commands.
<code>PWD</code>	Current directory, updated each time the <code>cd</code> command is executed.
<code>HOME</code>	Login directory.
<code>\$\$</code>	Process id of the script execution.
<code>\$?</code>	Return code of the last command executed.

The `export` command allows you to export a variable.

A variable is only valid in the environment of the shell script process. In order for the **child processes** of the script to know the variables and their contents, they must be exported.

The modification of a variable exported in a child process cannot be traced back to the parent process.

 **Note**

Without any option, the `export` command displays the name and values of the exported variables in the environment.

## 4.4 Substitute commands

It is possible to store the result of a command in a variable.

 **Note**

This operation is only valid for commands that return a message at the end of their execution.

The syntax for sub-executing a command is as follows:

```
variable=`command`  
variable=$(command) # Preferred syntax
```

Example:

```
day=`date +%d`
homedir=$(pwd)
```

With everything we've just seen, our backup script might look like this:

```
#!/usr/bin/env bash

#
# Author : Rocky Documentation Team
# Date: March 2022
# Version 1.0.0: Save in /root the files passwd, shadow, group, and gshadow
# Version 1.0.1: Adding what we learned about variables
#

# Global variables
FILE1=/etc/passwd
FILE2=/etc/shadow
FILE3=/etc/group
FILE4=/etc/gshadow

# Destination folder
DESTINATION=/root

## Readonly variables
readonly FILE1 FILE2 FILE3 FILE4 DESTINATION

# A folder name with the day's number
dir="backup-$(date +%j)"

# Clear the screen
clear

# Launch the backup
echo *****
echo "      Backup Script - Backup on ${HOSTNAME} "
echo *****
echo "The backup will be made in the folder ${dir}."
echo "Creating the directory..."
mkdir -p ${DESTINATION}/${dir}

echo "Starting the backup of ${FILE1}, ${FILE2}, ${FILE3}, ${FILE4} to ${DESTINATION}/${dir}:"

cp ${FILE1} ${FILE2} ${FILE3} ${FILE4} ${DESTINATION}/${dir}
echo "Backup ended!"
```

```
# The backup is noted in the system event log:  
logger "Backup of system files by ${USER} on ${HOSTNAME} in the folder ${DESTINATION}/${dir}."
```

Running our backup script:

```
sudo ./backup.sh
```

will give us:

```
*****  
Backup Script - Backup on desktop  
*****  
The backup will be made in the folder backup-088.  
Creating the directory...  
Starting the backup of /etc/passwd, /etc/shadow, /etc/group, /etc/gshadow to /  
root/backup-088:  
Backup ended!
```

## 5. Bash - Data entry and manipulations

---

In this chapter you will learn how to make your scripts interact with users and manipulate the data.

---

**Objectives:** In this chapter you will learn how to:

- ✓ read input from a user;
- ✓ manipulate data entries;
- ✓ use arguments inside a script;
- ✓ manage positional variables;

### ☒ linux, script, bash, variable

**Knowledge:** ★★

**Complexity:** ★★

**Reading time:** 10 minutes

---

Depending on the script's purpose, it might need information either at launch or during execution. This info, not predetermined during script writing, can come from files, user input, or be passed as arguments when entering the script command, similar to many Linux commands.

### 5.1 The `read` command

---

The `read` command allows you to enter a character string and store it in a variable.

Syntax of the `read` command:

```
read [-n X] [-p] [-s] [variable]
```

The first example below, prompts you for two variable inputs: "name" and "firstname", but since there is no prompt, you would have to know ahead of time that this was the case. In the case of this particular entry, each variable input

would be separated by a space. The second example prompts for the variable "name" with the prompt text included:

```
read name firstname
read -p "Please type your name: " name
```

Option	Functionality
-p	Displays a prompt message.
-n	Limits the number of characters to be entered.
-s	Hides the input.

When using the `-n` option, the shell automatically validates the input after the specified number of characters. The user does not have to press the `Enter ↵` key.

```
read -n5 name
```

The `read` command allows you to interrupt the execution of the script while the user enters information. The user's input is broken down into words assigned to one or more predefined variables. The words are strings of characters separated by the field separator.

The end of the input is determined by pressing the `Enter ↵` key.

Once the input is validated, each word will be stored in the predefined variable.

The division of the words is defined by the field separator character. This separator is stored in the system variable `IFS` (**Internal Field Separator**).

```
set | grep IFS
IFS=$' \t\n'
```

By default, the IFS contains the space, tab and line feed.

When used without specifying a variable, this command simply pauses the script. The script continues its execution when the input is validated.

This is used to pause a script when debugging or to prompt the user to press `Enter ↵` to continue.

```
echo -n "Press [ENTER] to continue..."  
read
```

## 5.2 The `cut` command

The `cut` command allows you to isolate a column in a file or in a stream.

Syntax of the `cut` command:

```
cut [-cx] [-dy] [-fz] file
```

Example of use of the `cut` command:

```
cut -d: -f1 /etc/passwd
```

Option	Observation
<code>-c</code>	Specifies the sequence numbers of the characters to be selected.
<code>-d</code>	Specifies the field separator.
<code>-f</code>	Specifies the order number of the columns to select.

The main benefit of this command will be its association with a stream, for example the `grep` command and the `|` pipe.

- The `grep` command works "vertically" (isolation of one line from all the lines in the file).
- The combination of the two commands allows for the **isolation of a specific field in the file**.

Example:

```
grep "^root:" /etc/passwd | cut -d: -f3  
0
```

 **Note**

Configuration files with a single structure using the same field separator are ideal targets for this combination of commands.

## 5.3 The `tr` command

---

The `tr` command allows you to convert a string.

Syntax of the `tr` command:

```
tr [-cqd] string1 string2
```

Option	Observation
<code>-c</code>	All characters not specified in the first string are converted to the characters of the second string.
<code>-d</code>	Deletes the specified character.
<code>-s</code>	Reduce the specified character to a single unit.

An example of using the `tr` command follows. If you use `grep` to return root's `passwd` file entry, you would get this:

```
grep root /etc/passwd
```

returns:

```
root:x:0:0:root:/root:/bin/bash
```

Now let's use `tr` command and the reduce the "o's" in the line:

```
grep root /etc/passwd | tr -s "o"
```

which returns this:

```
rot:x:0:0:rot:/rot:/bin/bash
```

## 5.4 Extract the name and path of a file

---

The `basename` command allows you to extract the name of the file from a path.

The `dirname` command allows you to extract the parent path of a file.

Examples:

```
echo $FILE=/usr/bin/passwd  
basename $FILE
```

Which would result in "passwd"

```
dirname $FILE
```

Which would result in: "/usr/bin"

## 5.5 Arguments of a script

---

The request to enter information with the `read` command interrupts the execution of the script as long as the user does not enter any information.

This method, although very user-friendly, has its limits if the script is scheduled to run at night. To overcome this problem, it is possible to inject the desired information via arguments.

Many Linux commands work on this principle.

This way of doing things has the advantage that once the script is executed, it will not need any human intervention to finish.

Its major disadvantage is that the user will have to be warned about the syntax of the script to avoid errors.

The arguments are filled in when the script command is entered. They are separated by a space.

```
./script argument1 argument2
```

Once executed, the script saves the entered arguments in predefined variables: `positional variables`.

These variables can be used in the script like any other variable, except that they cannot be assigned.

- Unused positional variables exist but are empty.
- Positional variables are always defined in the same way:

Variable	Observation
\$0	contains the name of the script as entered.
\$1 to \$9	contain the values of the 1st to 9th argument
\${x}	contains the value of the argument $x$ , greater than 9.
\$#	contains the number of arguments passed.
\$* or \$@	contains in one variable all the arguments passed.

Example:

```
#!/usr/bin/env bash
#
# Author : Damien dit LeDubar
# Date : september 2019
# Version 1.0.0 : Display the value of the positional arguments
# From 1 to 3

# The field separator will be "," or space
# Important to see the difference in $* and $@
IFS=", "

# Display a text on the screen:
echo "The number of arguments (\$#) = \$#"
echo "The name of the script (\$0) = \$0"
echo "The 1st argument (\$1) = \$1"
echo "The 2nd argument (\$2) = \$2"
echo "The 3rd argument (\$3) = \$3"
echo "All separated by IFS (\$*) = \$*"
echo "All without separation (\$@) = \$@"
```

This will give:

```
$ ./arguments.sh one two "tree four"
The number of arguments ($#) = 3
The name of the script ($0) = ./arguments.sh
The 1st argument ($1) = one
The 2nd argument ($2) = two
The 3rd argument ($3) = tree four
```

```
All separated by IFS      ($*) = one,two,tree four
All without separation   ($@) = one two tree four
```

### **⚠ Warning**

Beware of the difference between `$@` and `$*`. It is in the argument storage format:

- `$*` : Contains the arguments in the format `"$1 $2 $3 ..."`
- `$@` : Contains arguments in the format `"$1" "$2" "$3" ..."`

It is by modifying the `IFS` environment variable that the difference is visible.

## 5.5.1 The shift command

The shift command allows you to shift positional variables.

Let's modify our previous example to illustrate the impact of the shift command on positional variables:

```
#!/usr/bin/env bash
#
# Author : Damien dit LeDubar
# Date : september 2019
# Version 1.0.0 : Display the value of the positional arguments
# From 1 to 3

# The field separator will be "," or space
# Important to see the difference in $* and $@
IFS=", "

# Display a text on the screen:
echo "The number of arguments (\$#) = \$#"
echo "The 1st argument      (\$1) = \$1"
echo "The 2nd argument      (\$2) = \$2"
echo "The 3rd argument      (\$3) = \$3"
echo "All separated by IFS  (\$*) = \$*"
echo "All without separation (\$@) = \$@"

shift 2
echo ""
echo "----- SHIFT 2 -----"
echo ""

echo "The number of arguments (\$#) = \$#"
echo "The 1st argument      (\$1) = \$1"
echo "The 2nd argument      (\$2) = \$2"
echo "The 3rd argument      (\$3) = \$3"
```

```
echo "All separated by IFS      (\$*) = $*"
echo "All without separation   (\$@) = $@"
```

This will give:

```
./arguments.sh one two "tree four"
The number of arguments ($#) = 3
The 1st argument      ($1) = one
The 2nd argument      ($2) = two
The 3rd argument      ($3) = tree four
All separated by IFS      ($*) = one,two,tree four
All without separation   ($@) = one two tree four

----- SHIFT 2 -----
The number of arguments ($#) = 1
The 1st argument      ($1) = tree four
The 2nd argument      ($2) =
The 3rd argument      ($3) =
All separated by IFS      ($*) = tree four
All without separation   ($@) = tree four
```

As you can see, the `shift` command has shifted the place of the arguments "to the left", removing the first 2.

#### Warning

When using the `shift` command, the `$#` and `$*` variables are modified accordingly.

## 5.5.2 The `set` command

The `set` command splits a string into positional variables.

Syntax of the `set` command:

```
set [value] [$variable]
```

Example:

```
$ set one two three
$ echo $1 $2 $3 $#
one two three 3
$ variable="four five six"
```

```
$ set $variable  
$ echo $1 $2 $3 $#  
four five six 3
```

You can now use positional variables as seen before.

## 6. Bash - Check your knowledge

---

✓ Among these 4 shells, which one does not exist:

Bash

Ksh

Tsh

Csh

✓ What is the correct syntax to assign a content to a variable:

variable:=value

variable := value

variable = value

variable=value

✓ How to store the return of a command in a variable:

file=\$(ls)

file=ls``

file:=\$ls

file = \$(ls)

file=\${ls}

✓ The read command allows you to read the contents of a file:

True

False

✓ Which of the following is the correct syntax for the command `cut` :

- `cut -f: -D1 /etc/passwd`
- `cut -d: -f1 /etc/passwd`
- `cut -d1 -f: /etc/passwd`
- `cut -c ":" -f 3 /etc/passwd`

✓ Which command is used to shift positional variables:

- `left`
- `shift`
- `set`
- `array`

✓ Which command transforms a string into positional variables:

- `left`
- `shift`
- `set`
- `array`

## 7. Bash - Tests

---

**Objectives:** In this chapter you will learn how to:

- ✓ work with the return code;
- ✓ test files and compare them;
- ✓ test variables, strings and integers;
- ✓ perform an operation with numeric integers;

### ☒ linux, script, bash, variable

**Knowledge:** ★★

**Complexity:** ★★★

**Reading time:** 10 minutes

Upon completion, all commands executed by the shell return a **return code** (also called **status** or **exit code**).

- If the command ran correctly, the convention is that the status code will be **zero**.
- If the command encountered a problem during its execution, its status code will have a **non-zero value**. There are many reasons for this: lack of access rights, missing file, incorrect input, etc.

You should refer to the manual of the `man command` to know the different values of the return code provided by the developers.

The return code is not visible directly, but is stored in a special variable: `$?`.

```
mkdir directory
echo $?
0
```

```
mkdir /directory
mkdir: unable to create directory
```

```
echo $?
1
```

```
command_that_does_not_exist
command_that_does_not_exist: command not found
echo $?
127
```

### Note

The display of the contents of the `$?` variable with the `echo` command is done immediately after the command you want to evaluate because this variable is updated after each execution of a command, a command line or a script.

### Tip

Since the value of `$?` changes after each command execution, it is better to put its value in a variable that will be used afterwards, for a test or to display a message.

```
ls no_file
ls: cannot access 'no_file': No such file or directory
result=$?
echo $?
0
echo $result
2
```

It is also possible to create return codes in a script. To do so, you just need to add a numeric argument to the `exit` command.

```
bash # to avoid being disconnected after the "exit 2
exit 123
echo $?
123
```

In addition to the correct execution of a command, the shell offers the possibility to run tests on many patterns:

- **Files:** existence, type, rights, comparison;
- **Strings:** length, comparison;
- **Numeric integers:** value, comparison.

The result of the test:

- `$?=0` : the test was correctly executed and is true;
- `$?=1` : the test was correctly executed and is false;
- `$?=2` : the test was not correctly executed.

## 7.1 Testing the type of a file

---

Syntax of the `test` command for a file:

```
test [-d|-e|-f|-L] file
```

or:

```
[ -d|-e|-f|-L file ]
```



### Note

Note that there is a space after the `[` and before the `]`.

## Options of the test command on files:

Option	Observation
-e	Tests if the file exists
-f	Tests if the file exists and is of normal type
-d	Checks if the file exists and is of type directory
-L	Checks if the file exists and is of type symbolic link
-b	Checks if the file exists and is of special type block mode
-c	Checks if the file exists and is of special type character mode
-p	Checks if the file exists and is of type named pipe (FIFO)
-S	Checks if the file exists and is of type socket
-t	Checks if the file exists and is of type terminal
-r	Checks if the file exists and is readable
-w	Checks if the file exists and is writable
-x	Checks if the file exists and is executable
-g	Checks if the file exists and has a set SGID
-u	Checks if the file exists and has a set SUID
-s	Tests if the file exists and is non-empty (size > 0 bytes)

## Example:

```
test -e /etc/passwd
echo $?
0
[ -w /etc/passwd ]
echo $?
1
```

An internal command to some shells (including bash) that is more modern, and provides more features than the external command `test`, has been created.

```
[[ -s /etc/passwd ]]
echo $?
1
```



### Note

We will therefore use the internal command for the rest of this chapter.

## 7.2 Compare two files

---

It is also possible to compare two files:

```
[[ file1 -nt|-ot|-ef file2 ]]
```

Option	Observation
-nt	Tests if the first file is newer than the second
-ot	Tests if the first file is older than the second
-ef	Tests if the first file is a physical link of the second

## 7.3 Testing variables

---

It is possible to test variables:

```
[[ -z|-n $variable ]]
```

Option	Observation
-z	Tests if the variable is empty
-n	Tests if the variable is not empty

## 7.4 Testing strings

---

It is also possible to compare two strings:

```
[[ string1 =| !=|<|> string2 ]]
```

Example:

```
[[ "$var" = "Rocky rocks!" ]]
echo $?
0
```

Option	Observation
=	Tests if the first string is equal to the second
!=	Tests if the first string is different from the second one
<	Tests if the first string is before the second in ASCII order
>	Tests if the first string is after the second in ASCII order

## 7.5 Comparison of integer numbers

Syntax for testing integers:

```
[[ "num1" -eq|-ne|-gt|-lt "num2" ]]
```

Example:

```
var=1
[[ "$var" -eq "1" ]]
echo $?
0
```

```
var=2
[[ "$var" -eq "1" ]]
echo $?
1
```

Option	Observation
-eq	Test if the first number is equal to the second
-ne	Test if the first number is different from the second
-gt	Test if the first number is greater than the second
-lt	Test if the first number is less than the second

 **Note**

Since numeric values are treated by the shell as regular characters (or strings), a test on a character can return the same result whether it is treated as a numeric or not.

```
test "1" = "1"
echo $?
0
test "1" -eq "1"
echo $?
0
```

But the result of the test will not have the same meaning:

- In the first case, it will mean that the two characters have the same value in the ASCII table.
- In the second case, it will mean that the two numbers are equal.

## 7.6 Combined tests

The combination of tests allows you to perform several tests in one command. It is possible to test the same argument (file, string or numeric) several times or different arguments.

```
[ option1 argument1 [-a|-o] option2 argument 2 ]
```

```
ls -lad /etc
drwxr-xr-x 142 root root 12288 sept. 20 09:25 /etc
[ -d /etc -a -x /etc ]
echo $?
0
```

Option	Observation
-a	AND: The test will be true if all patterns are true.
-o	OR: The test will be true if at least one pattern is true.

With the internal command, it is better to use this syntax:

```
[[ -d "/etc" && -x "/etc" ]]
```

Tests can be grouped with parentheses ( ) to give them priority.

```
(TEST1 -a TEST2) -a TEST3
```

The ! character is used to perform the reverse test of the one requested by the option:

```
test -e /file # true if file exists
! test -e /file # true if file does not exist
```

## 7.7 Numerical operations

The `expr` command performs an operation with numeric integers.

```
expr num1 [+|-|[*|/] [%] num2
```

## Example:

```
expr 2 + 2
4
```

### ⚠ Warning

Be careful to surround the operation sign with a space. You will get an error message if you forget. In the case of a multiplication, the wildcard character \* is preceded by \ to avoid a wrong interpretation.

Option	Observation
+	Addition
-	Subtraction
\*	Multiplication
/	Division quotient
%	Modulo of the division

## 7.8 The typeset command

The `typeset -i` command declares a variable as an integer.

## Example:

```
typeset -i var1
var1=1+1
var2=1+1
echo $var1
2
echo $var2
1+1
```

## 7.9 The let command

The `let` command tests if a character is numeric.

## Example:

```
var1="10"
var2="AA"
let $var1
echo $?
```

```
0
let $var2
echo $?
1
```

### ⚠ Warning

The `let` command does not return a consistent return code when it evaluates the numeric `0`.

```
let 0
echo $?
1
```

The `let` command also allows you to perform mathematical operations:

```
let var=5+5
echo $var
10
```

`let` can be substituted by `$(( ))`.

```
echo $((5+2))
7
echo $((5*2))
10
var=$((5*3))
echo $var
15
```

## 8. Bash - Loops

---

**Objectives:** In this chapter you will learn how to:

✓ use loops;

☒ linux, script, bash, loops

**Knowledge:** ★★

**Complexity:** ★★★

**Reading time:** 20 minutes

---

The bash shell allows for the use of **loops**. These structures allow for the execution of a **block of commands several times** (from 0 to infinity) according to a statically defined value, dynamically or on condition:

- `while`
- `until`
- `for`
- `select`

Whatever the loop used, the commands to be repeated are placed **between the words** `do` and `done`.

### 8.1 The while conditional loop structure

---

The `while / do / done` structure evaluates the command placed after `while`.

If this command is true (`$? = 0`), the commands placed between `do` and `done` are executed. The script then returns to the beginning to evaluate the command again.

When the evaluated command is false (`$? != 0`), the shell resumes the execution of the script at the first command after `done`.

Syntax of the conditional loop structure `while`:

```
while command
do
    command if $? = 0
done
```

Example using the `while` conditional structure:

```
while [[ -e /etc/passwd ]]
do
    echo "The file exists"
done
```

If the evaluated command does not vary, the loop will be infinite and the shell will never execute the commands placed after the script. This can be intentional, but it can also be an error. So you have to be **very careful with the commands that manage the loop and find a way to get out of it.**

To get out of a `while` loop, you have to make sure that the command being evaluated is no longer true, which is not always possible.

There are commands that allow you to change the behavior of a loop:

- `exit`
- `break`
- `continue`

## 8.2 The exit command

---

The `exit` command ends the execution of the script.

Syntax of the `exit` command :

```
exit [n]
```

Example using the `exit` command :

```
bash # to avoid being disconnected after the "exit 1
exit 1
```

```
echo $?
1
```

The `exit` command ends the script immediately. It is possible to specify the return code of the script by giving it as an argument (from `0` to `255`). If no argument is given, the return code of the last command of the script will be passed to the `$?` variable.

### 8.3 The `break` / `continue` commands

---

The `break` command allows you to interrupt the loop by going to the first command after `done`.

The `continue` command allows you to restart the loop by going back to the first command after `done`.

```
while [[ -d / ]]
do
    echo "Do you want to continue? (yes/no)"
    read ans
    [[ $ans = "yes" ]] && continue
    [[ $ans = "no" ]] && break
done
```

### 8.4 The `true` / `false` commands

---

The `true` command always returns `true` while the `false` command always returns `false`.

```
true
echo $?
0
false
echo $?
1
```

Used as a condition of a loop, they allow for either an execution of an infinite loop or the deactivation of this loop.

## Example:

```
while true
do
    echo "Do you want to continue? (yes/no)"
    read ans
    [[ $ans = "yes" ]] && continue
    [[ $ans = "no" ]] && break
done
```

## 8.5 The `until` conditional loop structure

---

The `until` / `do` / `done` structure evaluates the command placed after `until`.

If this command is false (`$? != 0`), the commands placed between `do` and `done` are executed. The script then returns to the beginning to evaluate the command again.

When the evaluated command is true (`$? = 0`), the shell resumes the execution of the script at the first command after `done`.

Syntax of the conditional loop structure `until`:

```
until command
do
    command if $? != 0
done
```

Example of the use of the conditional structure `until`:

```
until [[ -e test_until ]]
do
    echo "The file does not exist"
    touch test_until
done
```

## 8.6 The alternative choice structure `select`

---

The structure `select` / `do` / `done` allows for the display of a menu with several choices and an input request.

Each item in the list has a numbered choice. When you enter a choice, the value chosen is assigned to the variable placed after `select` (created for this purpose).

It then executes the commands placed between `do` and `done` with this value.

- The variable `PS3` contains the invitation to enter the choice;
- The variable `REPLY` will return the number of the choice.

A `break` command is needed to exit the loop.

#### Note

The `select` structure is very useful for small and simple menus. To customize a more complete display, the `echo` and `read` commands must be used in a `while` loop.

Syntax of the conditional loop structure `select`:

```
PS3="Your choice:"
select variable in var1 var2 var3
do
    commands
done
```

Example of the use of the conditional structure `select`:

```
PS3="Your choice: "
select choice in coffee tea chocolate
do
    echo "You have chosen the $REPLY: $choice"
done
```

If this script is run, it shows something like this:

```
1) Coffee
2) Tea
3) Chocolate
Your choice : 2
You have chosen choice 2: Tea
Your choice:
```

## 8.7 The loop structure on a list of values `for`

The `for / do / done` structure assigns the first element of the list to the variable placed after `for` (created on this occasion). It then executes the commands placed between `do` and `done` with this value. The script then returns to the beginning to assign the next element of the list to the working variable. When the last element has been used, the shell resumes execution at the first command after `done`.

Syntax of the loop structure on list of values `for`:

```
for variable in list
do
    commands
done
```

Example of using the conditional structure `for`:

```
for file in /home /etc/passwd /root/fic.txt
do
    file $file
done
```

Any command producing a list of values can be placed after the `in` using a sub-execution.

- With the variable `IFS` containing `$' \t\n'`, the `for` loop will take **each word** of the result of this command as a list of elements to loop on.
- With the `IFS` variable containing `$'\t\n'` (i.e. without spaces), the `for` loop will take each line of the result of this command.

This can be the files in a directory. In this case, the variable will take as a value each of the words of the file names present:

```
for file in $(ls -d /tmp/*)
do
    echo $file
done
```

It can be a file. In this case, the variable will take as a value each word contained in the file browsed, from the beginning to the end:

```
cat my_file.txt
first line
second line
third line
for LINE in $(cat my_file.txt); do echo $LINE; done
first
line
second
line
third line
line
```

To read a file line by line, you must modify the value of the `IFS` environment variable.

```
IFS=$'\t\n'
for LINE in $(cat my_file.txt); do echo $LINE; done
first line
second line
third line
```

## 9. Bash - Check your knowledge

---

✓ Every order must return a return code at the end of its execution:

True

False

✓ A return code of 0 indicates an execution error:

True

False

✓ The return code is stored in the variable `$@`:

True

False

✓ The test command allows you to:

Test the type of a file

Test a variable

Compare numbers

Compare the content of 2 files

✓ The command `expr`:

Concatenates 2 strings of characters

Performs mathematical operations

Display text on the screen

✓ Does the syntax of the conditional structure below seem correct to you? Explain why.

```
if command
    command if $?=0
else
```

```
command if $?!=0
fi
```

True

False

✓ What does the following syntax mean: \${variable:=value}

Displays a replacement value if the variable is empty

Display a replacement value if the variable is not empty

Assigns a new value to the variable if it is empty

✓ Does the syntax of the conditional alternative structure below seem correct to you? Explain why.

```
case $variable in
  value1)
    commands if $variable = value1
  value2)
    commands if $variable = value2
  *)
    commands for all values of $variable != of value1 and value2
    ;;
esac
```

True

False

✓ Which of the following is not a structure for looping?

while

until

loop

for

<https://docs.rockylinux.org/>