[PICTURE]

Bash Mastery: The Complete Guide to Bash Shell Scripting

Master Bash Shell Scripting to Automate Tasks, Save Time, and Boost Your Career. Practical Projects + All Code Included.

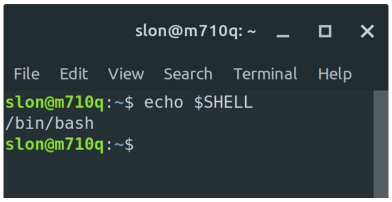
https://www.udemy.com/course/bash-mastery/

Table of Contents

1 How to build a bash script

1.1 Intro

Check that your default shell is BASH
CLI=> ***echo $SHELL***



```
slon@m710q:~$ echo $SHELL
/bin/bash
slon@m710q:~$
```

It should be bash. If it isn’t use next command
CLI=> ***chsh -s /bin/bash***

1.2 Shell vs Script

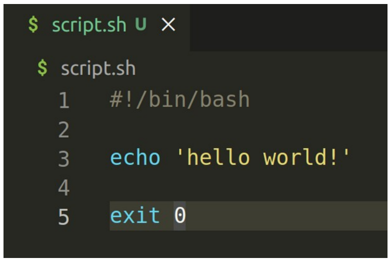
Shell is a program that interprets the commands that you type in your terminal and passes them to the OS.
The purpose of the shell is to make it more convenient for you to issue commands to your computer.

There are lots of different shells. But we are going to use BASH.
BASH = **B**ourne **A**gain **Sh**ell (it based on Bourne Shell – SH – that was created by Stephen Bourne in 1979).

BASH script is a file that contains commands for the BASH shell. Shell reads commands 1-by-1 and execute them.

1.4 Core components of BASH script

1. Each script should have the same first line:
#!/bin/bash
2. Some commands
3. Exit statement (not required)



```
#!/bin/bash

echo 'hello world!'

exit 0
```

Technically your script don’t need the file extension.
Your script should be executable.
CLI=> ***chmod +x script.sh***
Now you can run your script:
CLI=> ***./script.sh***

1.5 PRO components of BASH script

For comments you can use **#**
It might a good idea to add into your script info about your name, date of creation, short description and how to use it.

1.6 Setting up secure script permissions

To view file permissions you can use

```
CLI=> ls -l
- rwx rwx rwx 1 root root 40 Oct 17 15:39 script.sh
```

The first symbol shows a type. Dash (-) means that it is a file. (d) means that it is a directory.
Next 3 symbols show permissions for the owner of this file (read-wrire-execute).
Next 3 symbols show permissions for the owner group of this file (read-wrire-execute).
Next 3 symbols show permissions for the other users (read-wrire-execute).

Read – 4
Write – 2
Execute – 1

You can set permissions using 3-digit number

```
CLI=> chmod 711 script.sh
```

1.8 system PATH

You can check all the system paths using:

```
CLI=> echo $PATH
```

Main file with the settings is /home/slon/.profile
you can open it with any text editor (nano in my case):

```
CLI=> nano ~/.profile
```

You can add a link to your local folder with all the scripts

```
export PATH="$PATH:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/1"
```

Now you need to update that system parameters

```
CLI=> source $PATH
```

and check that your folder is a system path now

```
CLI=> echo $PATH
```

After all these actions you can execute any bash scripts (that are stored in that folder) from any folder

2 Variables and shell expansions

2.1 Intro

Variables allow us to store useful data under convenient names.

Shell expansions are very powerful features that allow us to retrieve data, process command output and perform calculations.

2.2 User-defined variables and shell expansions

Parameter is any entity that stores values. Shell parameters are used to store and reference useful data that we can use in our scripts.

There are 3 main types of shell parameters:

1. Variables
2. Positional parameters.
3. Special parameters.

Variable is a parameter that we can manually change.

```
#!/bin/bash
student="John"
echo "Hello ${student}!"
```

!don’t use any spaces next to equal sign

!naming convention – only lower case letters

!parameter expansion it’s when shell insert value of your variable/parameter (${student} OR just $student)

2.3 Shell/environment variables

Shell variables can be:

1. Bourne shell variables (were introduced in bourne shell in 1979) – 10 in total

PATH variable stores the list of folders that the shell will search for executable files to run as command names.

HOME variable stores the absolute path to the current user’s home directory.

USER variable stores the username of the current user.

HOSTNAME variable stores the name of the current computer.

HOSTTYPE variable stores the type of processor architecture the current computer is using.

PS1 variable stores the prompt string shown in the terminal before each command.

2. BASH shell variables (were introduced in more modern BASH shell) – about 95 in total

2.4 Parameter expansion tricks

Changing (actually value of the variable won’t be changed) text/string:
1. Change the first letter to Lower Case – ***echo ${student,}***
2. Convert string to Upper Lower – ***echo ${student,,}***
3. Change the first letter to Upper Case (caret symbol) – ***echo ${student^}***
4. Convert string to Upper Case – ***echo ${student^^}***
5. Define a length of the string – ***echo ${#student}***
6. Slicing – ***echo ${student:[OFSET]:[LENGTH]}***

```
phrase="Hello world!"
```

Slicing from the beginning

```
echo ${word:0:5} //Hello
echo ${word:3:5} //lo wo
echo ${word:7} //orld! - slicing all the symbols from 7th symbol
echo ${word:7:} //empty string
```

Slicing from the end

```
echo ${word: -5:3} //orl - !PUT THE SPACE BEFORE -5 !
echo ${word: -5} //orld!
```

2.6 Command substitution $(command)

Command substitution is a shell feature that allows you to grab the output of a command and use it inside another commands

```
#!/bin/bash
time=$(date +%H:%m:%S)
echo "Hello ${USER^^}! It’s $time right now!"
```

2.9 Arithmetic expansion

```
#1/bin/bash
echo $((4+2))
x=25
y=5
echo "$x + $y = $(($x+$y))"
echo "ORDER OF PRECEDENCE:"
echo "1. Everything inside brackets ()"
echo "BRACKETS: 3 * (2 + 4) = $((3 * (2 + 4)))"
echo "2. Exponentiation (**)"
echo "EXPONENTIATION: $x + $y ** 2 = $((x+y**2))"
echo "3. Division (/), Multiplication (*) and Modular (%)"
echo "DIVISION: 3 + $x / $y = $((3+x/y))"
echo "MULTIPLICATION: 5 + $x * $y = $((5+x*y))"
echo "MODULAR: ($x + 1) % $y = $(((x+1)%y))"
echo "4. Addition (+) and Substraction (-)"
echo "ADDITION: $x + $y = $((x+y))"
echo "SUBSTRACTION: $x - $y = $((x-y))"
```

!By default we can work with INTEGERS only!

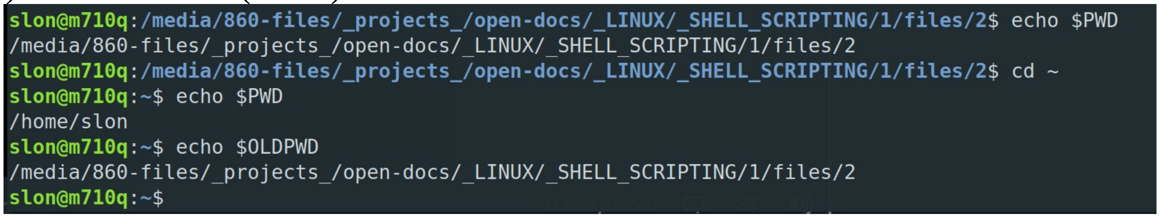
2.10 Decimal numbers with bc command

Bc is a tool that you can use inside your terminal to do some basic calculations.
You can pipe result of calculus to bc and setup how many symbols after dot you want to see
CLI=> ***echo “5/2” | bc*** //2
CLI=> ***echo "scale=2; 5/2" | bc*** //2.50
!In bc you need to use a^b instead of a**b!

2.12 Tilda expansion

Tilda (~) contains the path to the current user’s home directory
CLI=> ***echo ~*** //home/slon
You can print out the absolute path to the home directory of another user:
CLI=> ***echo ~anotheruser*** //home/anotheruser

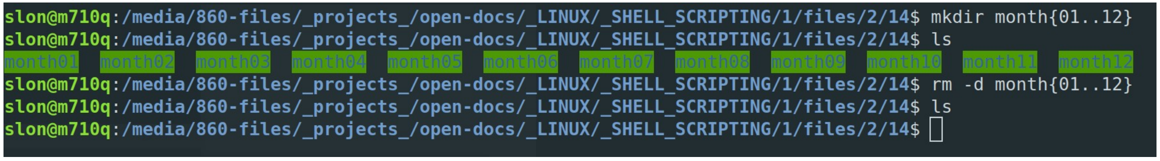
$PWD (OR ~+) and ***$OLDPWD*** (OR ~-)



```
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2$ echo $PWD
/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2$ cd ~
slon@m710q:~$ echo $PWD
/home/slon
slon@m710q:~$ echo $OLDPWD
/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2
slon@m710q:~$
```

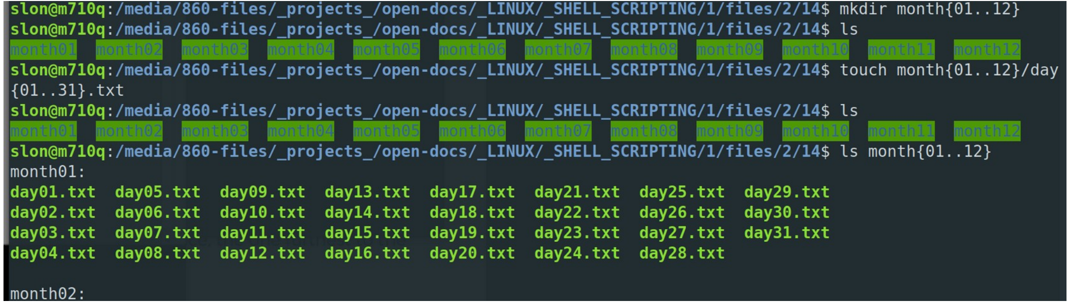
2.14 Brace expansion

String list
CLI=> ***echo {12,d,hello,true}*** //12 d hello true
Range list
CLI=> ***echo {1..10}*** //1 2 3 4 5 6 7 8 9 10
CLI=> ***echo {z..a}*** //z y x w v u t s r q p o n m l k j i h g f e d c b a
CLI=> ***echo {500..10000..2500}*** //500 3000 5500 8000
CLI=> ***echo month{1..12}*** //month1 month2 month3 month4 month5 month6 month7 month8 month9 month10 month11 month12
CLI=> ***echo month{01..12}*** //month01 month02 month03 month04 month05 month06 month07 month08 month09 month10 month11 month12
CLI=> ***echo month{01..12}***



```
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$ mkdir month{01..12}
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$ ls
month01  month02  month03  month04  month05  month06  month07  month08  month09  month10  month11  month12
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$ rm -d month{01..12}
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$ ls
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$
```

CLI=> ***mkdir month{01..12}*** //will create 12 new folders
CLI=> ***touch month{01..12}/day{01..31}.txt*** //will create 31 new text files in each folder
CLI=> ***ls month{01..12}*** //will print out content of each folder
CLI=> ***rm -dr month{01..12}*** //will delete all the folders with all the files
!You can use a string list the same way as an range list!



```
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$ mkdir month{01..12}
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$ ls
month01  month02  month03  month04  month05  month06  month07  month08  month09  month10  month11  month12
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$ touch month{01..12}/day
{01..31}.txt
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$ ls
month01  month02  month03  month04  month05  month06  month07  month08  month09  month10  month11  month12
slon@m710q:/media/860-files/_projects_/open-docs/_LINUX/_SHELL_SCRIPTING/1/files/2/14$ ls month{01..12}
month01:
day01.txt  day05.txt  day09.txt  day13.txt  day17.txt  day21.txt  day25.txt  day29.txt
day02.txt  day06.txt  day10.txt  day14.txt  day18.txt  day22.txt  day26.txt  day30.txt
day03.txt  day07.txt  day11.txt  day15.txt  day19.txt  day23.txt  day27.txt  day31.txt
day04.txt  day08.txt  day12.txt  day16.txt  day20.txt  day24.txt  day28.txt

month02:
```