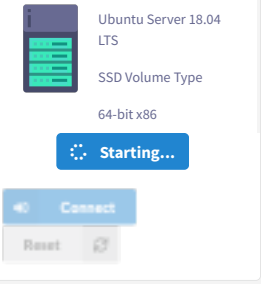
Bash and Linux - Bash language (EN)

60 minutes Normal



DataScientest • com

Introduction to Linux

Bash

Bash is a scripting language installed by default on Linux systems. It is the language used in the console: `echo` is the for example the Bash equivalent to Python `print`. But first we need to talk about executable files.

Executing files

To execute a file, we need a program to execute. There are two ways of specifying this.

First we can simply specify the program outside of the file.

> Open a file called `my_python_script.py` and paste the following lines in it

```
import os

print('Hello ! This is the content of the "/" directory:')

for file in os.listdir('/'):
    print('\t', file)

print('And that is all for the moment !')
```

> Save the file and run it with the following command

```
python3 my_python_script.py
```

Using this, Python will read the content of the file and execute it: we only need reading rights. To make it an executable, we need to specify a **shebang** meaning a string that specifies which program to use to run it. For Python, this is going to be `#!/usr/bin/python3`

> Open the `my_python_script.py` and paste this **shebang** at the top of the file

Your file can now be executed without using the syntax used before but simply `./my_python_script.py` if we have the right permission.

> Grant the current user execution rights and execute the file

Show / Hide solution

```
3
4  # executing the script
5  ./my_python_script.py
```

> Using the `echo` and `ls` functions, create a `my_bash_script.sh` executable file that mimics the behavior of the previous file. The shebang for `bash` is `#!/usr/bin/bash`

Show / Hide solution

```
1  #!/usr/bin/bash
2  echo Hello ! This is the content of the "/"
   directory:
3  ls /
4  echo And that is all for the moment !
```

> Follow the same step as before to make use executable to the current user and execute it

Show / Hide solution

```
1  # changing rights
2  chmod u+x my_bash_script.sh
3
4  # executing the script
5  ./my_bash_script.sh
6
```

Comments

As you may have guessed from our previous codes, comments are introduced using `#`.

> Run the following command

```
1  # this will do nothing ...
2  # ... and it did nothing !
```

Variables

To define a variable, we can use `=`. But to access the content of the variable, we need to use `$` before the name of the variable.

> Run the following command

```
1  # defining a variable
2  my_variable=1
3
4  # trying but failing to print its content
   echo my_variable
5
6  # printing its content
7  echo $my_variable
8
9  # redefining the variable
10 my_variable=2
11
12 # printing its content
13 echo $my_variable
14
```

Note that you cannot let a space between or after the `=` sign

```
my_variable = 1
my_variable =1
my_variable= 1
```

You will get 3 `command not found` errors.

Data types

We can use different data types:

- integer
- floats
- strings
- arrays
- ...

To define a string, we can ticks: `'` or `"`. Their behaviour is different:

> Run the following commands

```

my_variable=1

echo 'the content of my variable is $my_variable'
echo "the content of my variable is $my_variable"
```

In the second case, the content of the variable is computed.

To define an array, we can use different ways. The first way is to add parenthesis:

```
my_array=(hello world)
```

This is the equivalent of Python `my_array = ['hello', 'world']`. We can also define the elements within an array by giving directly their index:

```
my_array=()
my_array[0]=hello
my_array[1]=world
```

Note that indexes are not forced to follow each other:

```
my_array[2]=or
my_array[4]=hello
my_array[1000]=world
```

To access data within the array, we can use:

```
# printing the first element of the array
echo ${my_array[0]}

# printing a non existent element of the array
echo ${my_array[10]}


# printing the whole content of the array
echo ${my_array[*]}

# printing the indexes of the array
echo ${!my_array[*]}

# printing the number of elements in the array
echo ${#my_array[*]}
```

Numerical operations

To use numerical operations, we can use the `let` command. For example:

```
let "c=b"
let "d = a + b * c"
echo $d
```

Mathematical operations are quite similar as other programming languages: `+`, `-`, `*`, `/`, `**`, ...

Conditions

To create conditions, we can use a `if ... then ... fi` block. For example:

```
firstname="Daniel"
if [ $firstname = "Daniel" ]
then
echo "Hi Daniel !"
fi
```

And we can also add an `else` statement:

```
firstname="Daniel"
if [ $firstname = "Daniel" ]
then
echo "Hi Daniel !"
else
echo "Hello" + $firstname +"!"
fi
```

And `elif` statements:

```
firstname="Daniel"
if [ $firstname = "Daniel" ]
then
echo "Hi Daniel !"
elif [ $firstname = "Diane" ]
then
echo "Good morning Diane"
else
echo "Hello" + $firstname +"!"
fi
```

There are a lot of different ways to create conditions:

- `$var1 = $var2` checks if two strings are equal
- `$var1 != $var2` cheks if two strings are different
- `-z $variable` check if an array is empty
- `-n $variable` checks if an array is not empty
- `$var1 -eq $var2` checks if numerical values are equal
- `$var1 -ne $var2` checks if numerical values are not equal
- `$var1 -gt $var2` checks `var1 > var2`
- `$var1 -lt $var2` checks `var1 < var2`
- `$var1 -ge $var2` checks `var1 >= var2`
- `$var1 -le $var2` checks `var1 <= var2`

To combine multiple conditions, you can use `&&` for an `AND` association and `||` for an `OR` association.

```
firstname="Diane"
name="Datascientest"
if [ firstname="Daniel" ] && [
name="Datascientest" ]
then
echo "Hi Daniel Data"
else
echo "Hello" + $firstname + $name
fi
```

While loops

To create `while` loops we can use the following syntax:

```
let "i=i+1"
done
echo $i
```

For loops

To create a `for` loop, we can use the following syntax:

```
for x in '1st iteration' '2nd iteration' '3rd iteration'
do
echo $x
done
```

or if an array `my_array` is defined:

```
for x in ${my_array[*]}
do
echo $x
done
```

You can also use the `seq` function to generate a list of integers:

> Run the following command

```
seq 3 22
```

For example, we could try:

```
for i in `seq 3 22`
do
echo $i
done
```

Note the use of ` to perform the evaluation.

Functions

We can define functions in two different ways:

```
my_function () {
echo "Let's do something here"
}
```

```
function my_function {
echo "Let's do something here"
}
```

> Run the following command to define a function and call it

```
# defining the function
function my_function {
echo "Let's do something here"
}

# calling the function
my_function
```

Arguments are not specified while defining a function but can be access through `$` followed by the nunber of the argument, starting from 1 !!!

> Run the following command

```
echo "First argument"
echo $1
echo "Second argument"
echo $2
}

# calling the function
my_function "Daniel" "24"

```

Exercise

To practice, we are going to define a script that defines a function and run it on an array.

The function should compute the factorial value of a number: a Python equivalent would be:

```

def factorial(x):
    result = 1
    while x > 0:
        result = result * x
        x = x - 1
    print(result)


```

This function should be run for every number from 1 to 10 with a step of 2.

> Implement this code

Show / Hide solution

```
#!/usr/bin/sh

function factorial {
    let result=1
    let x=$1
    while [ $x -gt 0 ]
        do
        let result=result*x
        let x=x-1
        done
    echo $result
    }

my_index=`seq 1 5`

for i in ${my_index[*]}
    do
    let number=i*2
    echo $number
    factorial number
done

```

In this part, we have seen how to perform simple actions with bash, from defining variables to defining functions. There is of course a lot to learn about this language but these are the basics that we may have to use in the next lessons.

Validate