# 03_Bash

January 18, 2025

## 1 Lab03 Bash

### 1.0.1 Beyond "clicking", Shell as a necessity for proficent data analysis and computing

"Clicking" is for amatours, so stop doing that. When dealing with advanced computing and analysis tasks, it is needed to operate by means of shell commands and scripting. That is the case for several puroposes: file system managment, low level operations, advanced local and remote configurations, access and management of remote resources, etc. Several languages exists (all rather similar to each other), we will review BASH, a default **interpreter** on many GNU/Linux systems.

### 1.0.2 To start: using bash on MacOS

The MacOS application is `Terminal`, which used BASH until Catilina, then ZSH (almost the same).

With MacOS the default is ZSH but if you type `bash`you enter a BASH shell temporarily. To exit the temporary shell, type `exit`. If unsure if the commands for BASH are the same in ZSH, you can always switch to BASH. Type `echo $SHELL` to check which principal shell are you using (since default is zsh, the output will always be `/bin/zsh`) whereas type `echo $0` to check which temporary shell you are on.

Example

```
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % bash #from now

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(myenv) %n@%m %1~ %# echo $SHELL #running this still gives zsh...
/bin/zsh
(myenv) %n@%m %1~ %# echo $0 # but running this gives the current temporary shell
bash
(myenv) %n@%m %1~ %# exit
exit
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % echo $0 # runni
/bin/zsh
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % history > enter
```

## 1.1 Bash Shell Scripting

**Bash**

- **Definition**: Bash (Bourne-Again SHell) is a specific implementation of a **shell**, derived from the Bourne shell (`sh`) with additional features.
- **Key Characteristics**:
  - It is both a **command interpreter** (you type commands interactively) and a scripting language (you write commands in a script file and execute it).
  - Bash is widely used as the default shell on many Linux distributions and macOS (before zsh became the default).
  - Supports advanced features like:
    * **Filename globbing** (e.g., `*.txt` matches all `.txt` files).
    * **Piping** (e.g., `ls | grep "pattern"` to pass output from one command to another).
    * **Control structures**: `if`, `for`, `while`, etc., for scripting.
    * **Command substitution**: Embedding commands within commands (e.g., `$(date)`).
- **Think of Bash as**: A powerful tool for executing commands, automating tasks, and managing systems.

**Shell**

- **Definition**: A shell is a **general term** for any command-line interface (CLI) that allows a user to interact with the operating system.
- **Key Characteristics**:
  - It's a **macro processor**, meaning it processes and interprets text commands into actions for the operating system.
  - Can be interactive (you type commands and get responses immediately) or non-interactive (running pre-written scripts).
  - Several different shell implementations exist, including:
    * **Bourne shell (`sh`)**: The original Unix shell.
    * **C shell (`csh`)**: Known for C-like syntax.
    * **Z shell (`zsh`)**: Modern shell with many usability features.
    * **Bash (`bash`)**: A popular descendant of `sh` with extended features.
- **Think of Shell as**: The umbrella term for any CLI interface, of which Bash is a prominent example.

### 1.1.1 About this notebook

In the following we will explore the basic functionalities of bash, scrating the surface of an incredibly rich application. As for python, the web extensively provides documentation for all you'll ever need to do; the same and more do the LLM (e.g. GPT or Claude). The command `man` is anyway available for all the commands (most of which has also the `-help` option available).

Note that Jupyter allows execution of bash commands (and scripts) within a python notebook or by means of a dedicated emulator of a Terminal.

The code exposed in the following is however meant to be copied and tried on your computer shell. Scripts are meant to be edited on dedicated files, by means of your preferred text editor.

## 1.2 1: Navigating/exploring the file system

- `whoami`: who am I? I.e. which account am I using?

- `sudo -l`: am I a superuser? sudo stands for "Super-User-DO-(something)"
- `hostname` : what's the name of the computer
- `pwd`: present working directory. pwd returns the content of the global variable $PWD, see later

- `cd ..`: go to the directory above
- `cd -`: get back to previous directory
- `cd $HOME`: go to home directory
- `mkdir test`: make a new directory
- `mkdir -p tmp/foo`: the -p option stands for "parents". It allows the creation of parent directories as well if not existing already.
- `rm -r tmp/foo`:vdelete that last directory, "-r" recursive option to go into subfolders and delete all content
- `du -h`: check the amount of data in your current directory
- `df -h`: check the memory usage for the main folders
- `touch tmp_file`: create a new, empty file
- `ls -latrh`: check content of directory -l -> list, -a -> include hidden directories, -t -> time ordered, -r -> reversed, h -> size in Bytes "'

### 1.2.1 Anatomy of "ls -l" (list) output

The output presents 7 fields, standing for:

1. Permissions ("r": read, "w": write , "x": execute , "-": no permission)
2. Number of hardlinks
3. File owner
4. File group
5. File size
6. Modification time
7. Filename

The first string of the output contains the permissions. It shall be intepreted as: `File type - Owner permissions - Group permissions - Everyone permissions`

Example:

```
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash % ls -l
total 96
-rw-r--r--   1 miriamzara  staff  26657 Dec 27 10:42 03_Bash.ipynb
-rw-r--r--   1 miriamzara  staff   8121 Nov  7 11:55 03ex_Bash.ipynb
-rw-r--r--   1 miriamzara  staff   4405 Nov  7 09:41 LCP_22-23_students.csv
```

```
-rwxr-xr-x   1 miriamzara  staff    403 Nov  6 11:52 my_script.sh
drwxr-xr-x  24 miriamzara  staff    768 Nov 20 11:29 students
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash %
```

## 1.3 2: Files management

**Changing permissions**:

Issues with file access permissions happen all the time, you better learn how to set the permissions otherwise you will get into trouble in the future. The command for changing permissions is `chmod`, which stands for "change mode".

Example: allow "others" to read and write tmp_file: `chmod o+rw tmp_file`

**Appending to and overwriting a .txt file**:

```
touch tmp_file # create new empty temporary file
echo "Add some other text to a file" > tmp_file # overwrite the content
more tmp_file  #check the content of tmp_file
echo "Append text to a file" >> tmp_file  # append some other text
more tmp_file #read again
```

**Finding a file**

The general syntax is `find /path/to/dir -name "filename"`

Example:

```
find . -name "tmp_file" # find in current dir
find . -name "tmp*" # find in current dir with wildcard matching
```

**Copy files**

(absolute path can be specified of course)

```
cp tmp_file ./tmp_file_copy
```

**Remove files**

```
rm tmp_file_copy
```

## 1.4 3: Regular Expressions (RegExp)

Regular expressions are a very powerful tool for string manipulation, text find&replace, etc. Its syntax -although not obvious and complex- is used in several programming languages and is worth learning. This is one of the cases where consulting the manual(s) regularly is rather unavoidable. In the following will we use regexp with `grep` and `sed`, two basics commands you **must** know.

### 1.4.1 `grep` Command:

The grep command is used to search for specific patterns in files or input. It *prints* lines that match the given pattern.

```
grep [options] pattern [file] #basic syntax
grep "^#" data.csv #prints the lines starting with character # in file data.csv
```

### 1.4.2 sedCommand:

The sed command is used to perform text transformation (replacement, deletion, in-place edit, line range edit). sed stands for "Stream EDitor".

```
sed [options] 'command' file #basic syntax
sed 's/apple/orange/' file.txt #Local replacement: replaces the first occurrence of 'apple' in
sed 's/apple/orange/g' file.txt  #Global replacement: replaces all occurrences of 'apple' in t
sed -i 's/apple/orange/g' file.txt # In-place Global replacement: does not create a new file w
sed '/error/d' file.txt #Deletion: delete all the lines containing the word 'error'.
```

### 1.4.3 Example: reading and manipulating some measurement data

The file is data.csv, containing some data from a lab experiment.

**Basic grep commands:**

- `grep "sensor" data.csv`: print the line containing the word "sensor"
- `grep "^#" data.csv`: print all the lines starting with "#" (metadata)
- `grep -v "^#" data.csv`: print all the lines NOT starting with "#" (data payload)
- `grep -c -v "^#" data.csv`: count these lines

**Looking for specific syntax (with wildcard matching)**** Suppose you know the sensor name is encoded as X[lowecase letter][uppercase letter].v[decimal digit].

To print the line containing it: `grep -E 'X[a-z][A-Z]\.v[0-9]' data.csv`

Take care as the searches are case-sensitive: `grep -E 'X[a-z][a-z]\.v[0-9]' data.csv` returns nothing.

**Printing specific parts of a line** You need `grep` together with `cut`, whose syntax is `cut -field number -delimiter character`.

Example:

```
grep "started" data.csv | cut -f7 -d " "
```

Get the field number 7 (`-f7`), specify that fields are separated by " ".

```
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash % grep "started" data.csv
'# Recording started on Oct 29 15:00:00
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash % grep "started" data.csv| cut -f6 -d " "
29
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash % grep "started" data.csv| cut -f7 -d " "
15:00:00
```

**Basic `sed` commands: Substitute** Basic syntax is: `sed "[substitution expression]" filename`. By default, substitution is only applied to the first occurrence on each line.

`sed "s/Oct/Nov/" data.csv` substitute Oct with Nov

**Replacement** A more complex example: add ".0" to all single digit number

```
sed "s/\b[0-9]\{1\}\b/&.0/" data.csv
```

- `\b[0-9]\{1\}\b`: this is a word bound. Matches any single digit from 0 to 9 and limits the match to exactly one digit.
- `/&.0/`: replace with the entire match found (&) followed by ".0". Without &, it would not append .0, but instead would replace the matched number with .0.

### 1.5 4: Variables

Variables can be global or local. The latter are set in the usual intuitive manner, the former requires the `export` command. **Variable are accessed by preposing the $ character.** Global variables are used extensively by applications you don't have/don't want to have control of, thus do not mess with them unless you know what you are doing. **Global variables typically use capital letters**, thus for your own variable is best to use lower case letters. The command `env` list all the global variables currently set. An example of global variable is `PATH`.

```
# the whole list of global variables
env

# what is your PATH?
echo $PATH

# set your own global variable?
export MY_GLOBAL_VARIABLE=pippo
echo $MY_GLOBAL_VARIABLE

# set your local variable
variable=7
echo $variable # use tab for auto-completion
```

---

**What is the `PATH` Variable?** The `PATH` variable is an environment variable in Unix-like operating systems (such as Linux and macOS) that specifies a list of directories the system searches for executable files when a user runs a command.

**How it Works:** When you type a command in the terminal (like `python`, `git`, or `ls`), the shell looks for the executable file corresponding to that command. Instead of requiring you to type the full path of the command every time, the shell checks the directories listed in the `PATH` variable to find the executable.

**Structure of `PATH`** - The `PATH` variable contains a colon-separated list of directories. - For example, if your `PATH` is:

`/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin`

The shell will search for the executable in `/usr/local/bin`, then `/usr/bin`, then `/bin`, and so on, in the order listed. For example, if you run the command `python`, the system looks through the directories listed in the `PATH` variable until it finds an executable file named `python`. If it finds it in `/usr/bin/python`, it runs that file. To sum up, PATH allows you to run executables from any location without needing to specify their full paths.

## 1.6 5: Logging the Standard output and Standard error

Every time a command is executed, three possible outcomes might happen: 1) the command will produce an expected output 2) the command will generate an error 3)the command produces no output at all. It happens often that the need is there (e.g. for logging purposes) to store the output in separate files. The `>` notation is used to redirect stdout to a file whereas `2>` notation is used to redirect stderr and `&>` is used to redirect both stdout and stderr.

**Example:**

```
# redirecting standard error
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % ls -l ghost_vari
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % more err.txt
ls: ghost_variable: No such file or directory
err.txt (END)
#
# redirecting standard output
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % ls -l err.txt >
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % more log.txt
-rw-r--r--  1 miriamzara  staff  46 Nov  6 11:33 err.txt
log.txt (END)
```

## 1.7 6: Numeric and string comparison, operations with numbers

The following table summarizes how to compare numbers and strings.

| Description | Numeric Comparison | String Comparison |
|---|---|---|
| less then | -lt | < |
| greater equal | -gt | > |
| equal | -e | = |
| not equal | -ne | != |
| less or equal | -le | N.A. |
| greater or equal | -ge | N.A. |

**CAREFUL:** the result of a comparison is **1 if False and 0 if True**. The following is a trick to access directly the result of the comparison; those are used mainly in conditional statements

Example:

```
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash % a=1 #careful! do not insert spaces!
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash % b=2
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash % [ $a -lt $b ] # is a less than b?
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash % echo $? # to print the result of operat
0 #true
(myenv) miriamzara@MacBook-Pro-di-Miriam 03_Lab_Bash % [ $a -lt $b ]; echo $? #to perform and
0
```

Aritmetic operations can be done in several ways. Quite common is the usage of Arithmetic Expansion, thanks to its simplicity (trading off though with the complexity of the permitted operations:

in fact, it only operates with integer numbers. For floats you need to use another method). Other tool are there like `expr` or `let`, we will provide examples with `bc`, which is rather intuitive and powerful.

```
# Aritmetic Expansions
echo $(( 10*5 + 15 ))
echo $(( $a + $b*3 ))
echo $((15 % 5)) #division rest

# bc
echo '8.5 / 2.3' | bc
result=$(echo "3 * 4" | bc)
result=$(echo "scale=4; 10 / 3" | bc)
echo $result  # Output: 3.3333 four decimal digits
sqr=$(echo 'scale=6;sqrt(2)' | bc) # set the variable 'sqr' to the result of the following ope
# 'scale=6' sets the precision of the result as 6 decimal digits.
# output is 1.414213
echo $sqr
```

### 1.8 7: Conditional statements

As in every other language, conditional statements are of paramount importance and exploited exstensively also in bash. They best fit in a script (see later), but can be used also on the command line. The syntax is similar to the other languages

*`fi` is the reversed of `if`: it is used to close the if statement*

```
a=400
b=200

# an inline command
if [ $a -gt $b ]; then echo "$a is greater than $b! "; fi

# the following needs to go in a script
if [ $a -lt $b ]; then
    echo "$a is less than $b! "
else
    echo "$a is greater than $b! "
fi
```

### 1.9 8: Loops

same considerations as for conditional statements apply for loop cycles. There are several ways they can be implemented, we will review a few of them. Like in the if statement, the loop must be closed. This is done through command `done`.

```
# some for loops syntax
for i in 1 2 3; do echo $i; done

for i in {1..10}; do echo $i; done
```

```
for i in $(seq 1 2 20); do echo $i; done

for (( i=1; i<=5; i++ )); do echo $i; done

for i in `ls .`; do echo $i; done

for file in ./*; do if [ "${file}" == "./log.txt" ]; then break; fi; echo $file; done

# while (and until) works too
counter=0
while [ $counter -lt 3 ]; do let counter+=1; echo $counter; done

# some for loops syntax
for i in 1 2 3; do echo $i; done

for i in {1..10}; do echo $i; done

for i in $(seq 1 2 20); do echo $i; done

for (( i=1; i<=5; i++ )); do echo $i; done

for i in `ls .`; do echo $i; done

for file in ./*; do if [ "${file}" == "./log.txt" ]; then break; fi; echo $file; done

# while (and until) works too
counter=0
while [ $counter -lt 3 ]; do let counter+=1; echo $counter; done
```

### 1.10 9: Scripting

All the instructions above (and many many more) can be put together into a script. An example follows:

```
#!/bin/bash

# checking if the user provided an input
if [ -z $1 ]
then
    echo "this script requires as input the name of the file to be created"
    exit
fi
.
# check if the file is there
if [ ! -f "./$1" ]
then
    echo "the file ./$1 does not exist! Setting a default value"
    file="newfile.txt"
```

```
else
    file=./$1
fi

touch $file

for (( i=1; i<=5; i++ ))
do
    echo "add line $i" >> $file
done
```

The initial statement `#!/bin/bash` tell the shell how to interpret the subsequent commands.

You can save those lines on a new file (e.g. `my_script.sh`) and try to run it. First you may want to make it executalbe

```
chmod +x my_script.sh

./my_script.sh output.txt
```

### 1.10.1 My try

See also "my_script_bis.sh" which simply accesses and prints inputs provided by user

```
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % nano my_script.s
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % chmod +x my_scri
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % ./my_script.sh o
the file ./output.txt does not exist! Setting a default value
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 % ./my_script.sh #
this script requires as input the name of the file to be created
(myenv) miriamzara@MacBook-Pro-di-Miriam LaboratoryOfComputationalPhysics_Y7 %
```

## 1.11 Documentation

As mentioned above, there is plenty of documentation available directly via `man` or on the internet. A rather important command is **`history` which list the set of commands provided in the current shell**. After an intense session of operations on the shell, **you may want to run it and pipe its stdout into a log file for later consultation**

### 1.11.1 AWK

awk is a powerful text processing tool in Unix-like operating systems that is used for pattern scanning and processing. It is primarily used for manipulating data in files or streams, performing operations on text such as filtering, extracting fields, and performing arithmetic operations. It is more powerful than `grep` and `sed`.

Basic syntax is:

```
awk 'pattern { action }' inputfile
```

Examples:

awk '{print}' file.txt awk '{print $1}' file.txt awk '{sum += $1} END {print sum}

Built-in Variables: - $0: Represents the entire line. - $1, $2, …: Represents individual fields in the current line. - NF: The number of fields in the current line. - NR: The number of the current record (line). - FS: The field separator (default is space). - OFS: The output field separator (default is space).