(python) Programming for APC

Lecture 1 : Terminal usage and the shell

Tommaso Ronconi

MA PER SEGUIR VIRTUTE E CANOSCENZA

SISSA

Outline for today

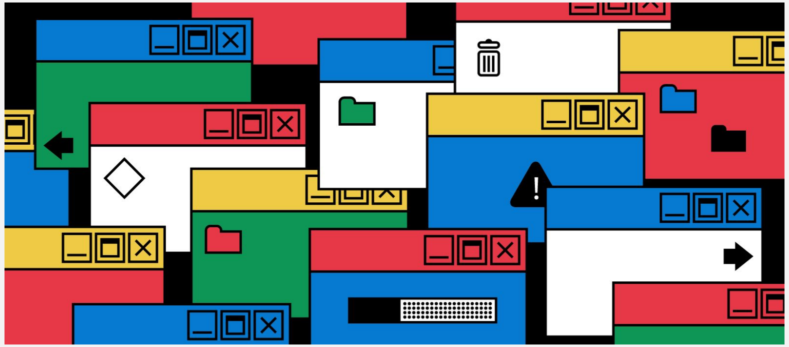
Hic sunt leones: the Command Line Interface

- ➔ We are going to figure out what a **terminal** is and how the **shell** works
- ➔ **Bash**, aka your best friend in front of the black(-ish) screen

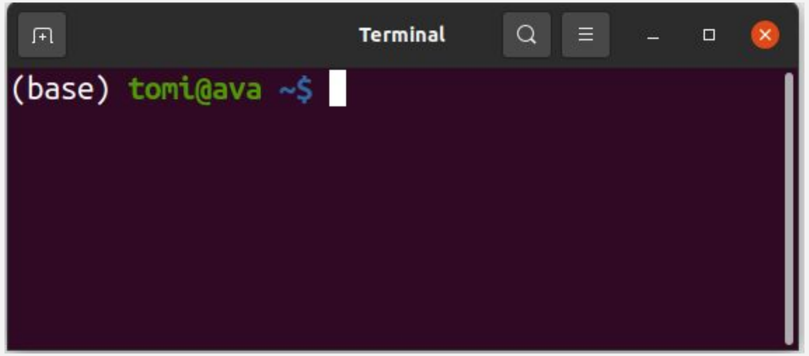
$ BASH survival kit _

GUI vs CLI

GUI: Graphical User Interface



CLI: Command Line Interface



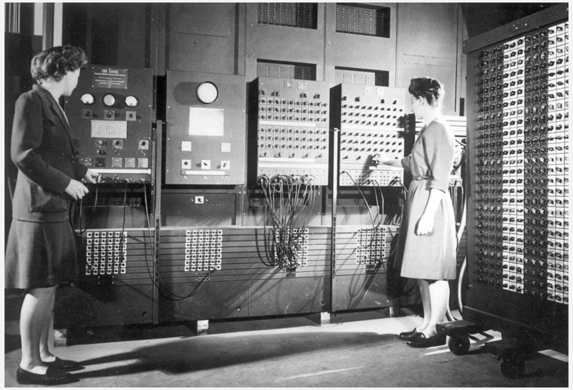
```
(base) tomi@ava ~$ 
```

PROS:

- user friendly very easy to use
- prevents the user from severely damaging the system

CONS:

- very little flexibility, few actions are possible
- adds overhead to the execution



PROS:

- enables development of very specific applications
- allows you to do basically everything to the system

CONS:

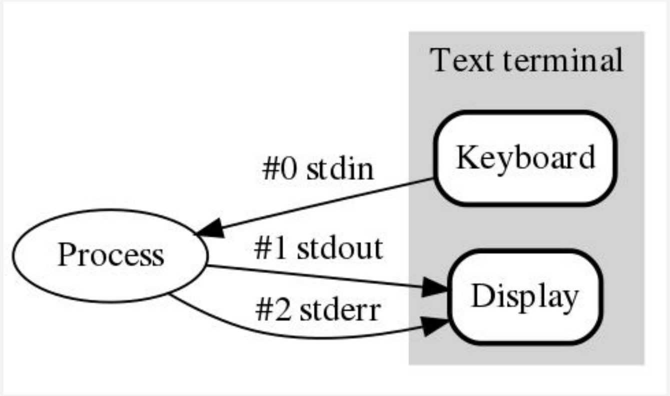
- requires knowing your moves
- allows you to do basically everything to the system

What is a terminal?

real terminals are not a thing anymore, what you have on your laptop is a **terminal emulator**

- Allows to access the command line interface to a computer
- It's the most powerful tool you have on your machine
- Sooner or later (most likely sooner) you'll need to use it
- In research, your applications do not have a GUI and if they do, you will end up in bottlenecks sooner or later

STDIN: what you type on screen, collects **inputs**
STDOUT: what is printed on screen in case of **success**
STDERR: what is printed on screen in case of **failure**



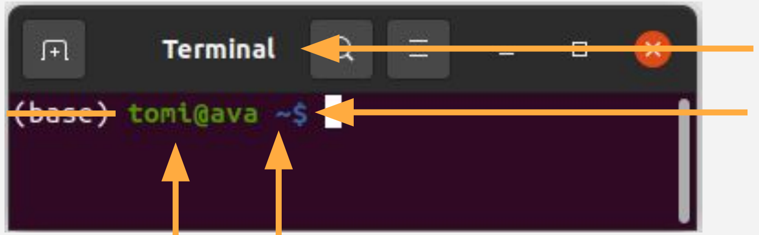
The Shell and Bash

Set BASH shell:
`chsh -s /bin/bash`

➔ The CLI has its own language, this language is **the shell**

➔ Different **"flavours"** (**bash**, tcsh, zsh, ksh, …)

[they differ from some aspects but the things we'll see today should work anyways]



```
(base) tomi@ava ~$
```

Hello, we are inside the CLI

$ means I am using the **Bourne Again SHell** (a.k.a. BASH)

My position is "**~**" == my home directory

I am user "tomi" connected to the "ava" machine

➔ Each and every behaviour of the shell is defined in some file located somewhere in the **filesystem**

e.g.1: `(base) tomi@ava ~$` is the **Prompt String** defined through the variable **PS1**

e.g.2: `~/.bashrc` is the **bash Run-Commands** file, defines user's customizations (note that on some system these customization might be accessible from a `.bashprofile` instead, this doesn't change things much)

Hands on - let’s move around the system

- ➔ Open a terminal (Ctrl+Alt+T in Linux)
- ➔ Look around:
  - list files: **`ls`**
  - show disk usage: **`du`**
  - show sub-directory structure: **`tree`**

- ➔ Let’s go check our **`.bashrc`** :
  - what is it? **`stat ~/.bashrc`**

    `stat -c "File:%n, Size: %s, Type: %F" ~/.bashrc`
  - open it with a text editor!

- ➔ And make something useful:
  - alias the above formatted command to **`fstat`**
  - let’s secure our remove command: **`rm -i`**/**`-I`**
  - save+close: test it!

RTFM:
read the f#!*ing manual

```
$ man [command]
```

Black box of death:

What is the difference between
`rm -rf */`
and
`rm -rf * /`
??
(DO NOT EVER USE THE 2nd!!!)

Hands on - Pipelines, Batches and Scripts

➔ A **PIPELINE** is a sequence of commands, the **output of the previous** command is the **input of the following** one. In bash this can be done at the command line with the **pipe symbol** '|'

```
$ du -h /usr/lib | tail
```

we can also redirect the output of a command somewhere else:

```
$ du -h /usr/lib > /tmp/lib_list.txt
```

➔ **BATCH vs SCRIPT** both are sequences of commands, necessary to complete some task

- Batch: a list of commands to be executed in sequence
- Script: commands + conditionals + cycles, a bit more sophisticated
  - the shebang (`!#`)
  - execute it (`source`)
  - make it executable (`chmod`)

https://github.com/TommasoRonconi/hitchhikers_guide/tree/main/exercise0_bash

Hands on - Variables

➔ **VARIABLE ASSIGNMENT**

```
$ var1=5
$ var2=hello
```

note that there are
NO SPACES before/after `'='`

➔ **so this is wrong:**

```
$ var3= world
Command 'world' not found
```

➔ **ACCESS A VARIABLE**

```
$ echo $var2
hello
```

note usage of the **'$' symbol!**

ENVIRONMENT VARIABLES
are defined for **every process** in your system
examples:
`PWD, PATH, PS1, HOME,` …
➔ check them with `printenv`
define one:

```
$ export WORKDIR=/home/tomi/work
```

➔ check it doesn't already exist:

```
$ printenv | awk '/WORKDIR/{print $0}'
```

Special variables exist
`$$, $?, $@,` …
try executing these 2 commands:

```
$ ls
$ echo $?
```

BASH Variables do not have a type!

in general everything is a string
but you can do arithmetics with them
as long as they are made of **digits only**

Hands on - Arrays: collections of variables

- An array is defined within brackets, a space to separate the elements:

  ```
  $ array1=( 42 hello "daje" )
  ```

  note 1: no spaces before/after the "`=`"
  note 2: arrays are indexed from `0` to `N-1`

- Iteration requires a combination of squared and curly brackets:

  ```
  $ echo ${array1[0]} ${array1[2]}
  42 daje
  ```

- You have special methods to perform special operations:
  - get all the elements of an array with the "`@`" character:

    ```
    $ echo ${array1[@]}
    42 hello daje
    ```

  - get the array size with the "`#`" character:

    ```
    $ echo ${#array1[@]}
    3
    ```

Create a list of numbers with given spacing using the "seq" command:

syntax: `seq [first [increment]] last`

```
$ array3=$(seq 0 2 10)
$ echo ${array3[@]}
0 2 4 6 8 10
```

Hands on - Loops & Conditionals

FOR-LOOPS

```
$ for i in {0..10..1}; do echo $i; done

$ for i in $( seq 0 1 10 ); do echo $i; done
```

IF-STATEMENT

```
#! /bin/bash

if [[ <condition> ]]
then
     <do something>
elif [[ <second condition> ]]
then
     <do something else>
else
     <notice there is no "then">
     <and no condition to verify>
fi
```

Anyways it's better to try this directly on the terminal

(if we have time)

… and BTW at this link you can find a complete list of all the conditional operations you can perform in bash

Recap on brackets

`()` **round brackets:** are used to define arrays

```
$ array=(1 2 3 4) # defines an array
```

`[]` **squared brackets:** are used in conditionals, what's inside will return **true** or **false**

```
$ if [[ -f ~/.bashrc ]]; then echo ok!; fi
```

note that it is though recommended to use the double squared brackets `[[ … ]]`

squared brackets can also be used to access the elements of an array if combined with ..

`{}` **curly brackets:** are used to access variables

```
$ echo ${array[1]} # access the 2nd element of array
2
$ echo $PWD # access an environment variable
/home/tomi
```

`$()` **dollar sign followed by round brackets:** executes a bash command within the round brackets

```
$ echo $( ${array[3]} + 38 ) # you can also assign the value to a variable
42
```

Some useful Bash commands

The very basic “what’s going on?”-kit

`man`	man-pages of command
`ls`	List files
`find`	find something
`which`	where a command is located
`head/tail`	show first/last lines of file
`du`	disk usage
`top`	who is using CPU and RAM?
`jobs`	list user processes in this shell
`kill[all]`	un-politely shut-down smth

Move around the filesystem

`cd`	change directory
`pushd/popd`	add/remove dirs on a stack

Create and remove stuff

`mkdir`	make an empty directory
`touch`	create an empty file
`cp`	copy with renaming
`mv`	move and/or rename
`rsync`	like “`cp`” but better
`rm`	remove something **FOREVER**

A little above “basic”

`cat/paste`	concatenate (by-row/column)
`diff`	see differences between files
`stat`	infos about some file
`type`	infos about some command
`wc -l`	number of lines in a file

Text editing for pros

`grep`	better tool to search stuff (string/files)
`sed`	stream editor to modify text strings
`awk`	the best text processing language ever

Concatenate commands

`;`	(semicolon) execute after
`|`	(pipe) redirect stdout to stdin
`&`	(ampersand) execute both
`&&/||`	execute if exit status 0/not 0

Be god. God’s called `root`

`sudo`	emulate god (run a command as root)
`su`	become god (login as root)

And that’s all folks! (for today)