Politecnico di Torino

BACHELOR DEGREE IN BIOMEDICAL ENGINEERING

INTERNSHIP REPORT

THESIS SUPERVISOR

Prof. M. A. Deriu

REFERENTS

Marcello Miceli

Marco Cannariato

Lorenzo Pallantte

Eric Zizzi

STUDENTS

Gabriella Marino Gammazza

Giulia Moscone

Elisa Bruatto

Giuseppe Armao

Ilaria Pulito

Valeria Saponaro

Davide Tkalez

Michele Catena Cardillo

ACCADEMIC YEAR 2021-2022

Index

Introduction

The internship we undertook in March is called "**Basic linux for molecular multiscale modeling**" and addresses a topic that, as the name suggests, focuses on the analysis of nanoscale biological tissues. In particular we focus on modeling atomistic scale samples and studying them.

We have focused our attention on the use of **software for multiscale modeling**; more specifically, we have had the opportunity to see how the tools provided by computer science allow us to implement processes that would take a long time to obtain fundamental information for biological research.

We learned how to use **Linux**, as well as the **Bash and Python languages**.

The final goal was the writing of the following tutorial, in which the processes that led to the knowledge of these modern software and languages for modeling multiscale samples are folded step by step.

The following tutorial provides an initial section to assist you during the **installation** of **Linux** on your computer. The ultimate goal is to get you to display a Linux terminal on your screen. The next chapter introduces the **Bash language**, including the basic commands and the most commonly used programming structures. This is followed by a chapter on making **Bash scripts**. The last section is devoted to the use of the **Python language** and, in particular, its modules dedicated to graphical representation.

A common thread throughout the tutorial will be the **analysis** and **modeling of proteins**. The use of the Bash language is aimed, in fact, at the analysis of .pdb files and that of Python at the graphical representation and modeling of the protein whose data are collected in the analyzed pdb file

SECTION 1

LINUX INSTALLATION

Regarding the installation of Linux on one's own pc, several paths can be followed. The first option is to install Linux as the only operating system. This is a radical solution; all data saved on the pc will be lost. In case you do not want to change the operating system, you can follow one of the following paths:

1) installation of a **virtual machine**

2) installation of a **subsystem on pc with Windows 10** system (WSL10)

3) installation of a **subsystem** on pc with **Windows 11** system (WSL11)

In the following sections you can find step-by-step instructions to follow for each of these three options

1.1 INSTALLATION VIRTUAL MACHINE

For detailed procedures follow the guide provided. Below we provide a brief summary of the necessary steps accompanied by some useful tips and links to solve any problems encountered during the installation process.

1. Download virtualbox from the link : www.virtualbox.org
2. Download the .exe file and proceed by clicking on 'next' without changing the basic settings
3. Download the .ova file at the link https://drive.google.com/file1-CLJIgcSw2BByQQNs1in326oyij2ofvz/view?usp=sharing

Tips:

- If you have difficulty downloading the file on the first attempt, try again several times until the goal is achieved. In fact, it is possible that you may have to repeat the procedure several times if the download starts but stops after a few minutes, signaling an error. It is recommended that you make at least 3-4 attempts..
- **Warning** : to start the download, do not press on the bar where 'zipper extractor' appears. , but press on the download icon in the upper right corner

1. Open the VirtualMachine and follow the instructions to import the file. Tip : change, if necessary, the CPU and RAM parameters, it is recommended to enter respectively (**CPU : 2 / RAM : 2048 MB**). These parameters can also be changed later.

2. Click on import (the file may take about ten minutes to load).

- **WARNING**: If the icon described in the help in the upper left corner of the virtualbox does not appear, or if the session ended warning appears, use the instructions for virtualizing the VM , specified at the following the link: https://www.aranzulla.it/come-attivare-la-virtualizzazione-nel-bios-1231556.html#:~:text=Ad%20every%20mode%2C%20every%20there%C3%B2,the%20Enable%2FActive%20option.

OTHER PROCEDURES *It is possible that once opened, the VM screen appears blurry and enlarged, you need to fix the scale and resolution settings as described in the guide (if you do not see the menu indicated in the guide , go to **Settings > Screen > Scale Factor > 100%**)

- Activating the **Virtualization of the BIOS** (Caution: the procedure to be followed may vary slightly from pc to pc). Press the **win+I** keys > select **edit pc settings** > **update and restore** > **restore** > **restart now** > **troubleshooting** > **advanced options** > **UFI firmware settings** > reboot > press the **f10** key > select **system configuration** (top) > **virtualization technology** > **enable**.
- For guest addition , in case the guide procedure does not work, follow the instructions below. Go to google and type **Oracle > download > VM virtualbox > Vbox guest addition**. Then go back to virtualBox , go to settings , go to storage , select 'VBoxGuestAddition.iso'. Select the diskette next to 'optical drive'. Select the item ' choose a disk file...' and import the Vbox guest addition download.
- In conclusion go to **Devices > insert guest addition CD image > press 'ok' > press 'run' > enter password 'student'** , if required. Finally press 'enter' when prompted by the terminal.

1.2 INSTALLATION LINUX SUBSYSTEM WINDOWS 10

- To begin, you must open the command prompt or Windows PowerShell by running them in administrator mode (right-click >run as administrator).
- Type `wsl --install`, press enter and wait for the installation to complete.
- Finish the installation, open the search bar and type `enable or disable Windows features'. Alternatively, type Windows+R and in the window that opens type "optionalfeatures."
- In the window that appears you need to check:
  - Virtual Machine Platform
  - Windows Hypervisor Platform
  - Windows Subsystem for Linux
- Press okay when finished and restart the computer.
- Upon reboot Ubuntu will open and proceed with installing the necessary files, wait a few minutes for it to finish.
  - If at the end of the installation appears on the screen `"error 0x80370102 the virtual machine could not be started because a required features is not installed"` you should:
    - first check that CPU virtualization is enabled. Press ctrl+alt+delete, open task manager >performance and check next to "virtualization" is "enabled". In case it is not, look at appendix 1.
    - if it still does not work it could be a problem related to the version of wsl you are using. Open the command prompt or Windows PowerShell and type `wsl --set-default-version 1` and then `wsl --install -d Ubuntu`.
- At the end of the installation you will be prompted for a username and password. Use a lowercase username to avoid any problems. It is normal for nothing to appear on the screen while typing the password (it is actually read correctly anyway).
- Typing "wsl" in the search bar will open the default distribution (to see what it is just type `wsl -l -v` at the command prompt).

1.3 INSTALLATION LINUX SUBSYSTEM WINDOWS 10

In windows 11 through the wsl subsystem it is possbile to run linux programs with their own GUI. Windows 11, as well as several recent versions of Win10, make use of WSL 2: this is the second redesigned version of WSL1 that runs a full Linux kernel in a Hyper-V hypervisor for better compatibility. When you enable this feature Windows 11 downloads a Linux kernel created by Microsoft that runs in the background, while, Windows Update keeps the kernel up-to-date. Eventually, you can also use your own custom Linux kernel.

Installation process.¶

Activation via command line.¶

1. Open the windows terminal or command prompt (cmd) **as administrator** (right-click on the taskbar start button)
2. Type at the prompt `wsl -install` to enable the windows subsystem for linux.

   2.2. Having completed this step those with a windows 10 system, but also on windows 11 following a different installation method from the command line, you need to type in the search bar "*optionalfeatures*" which will take you to the "*Windows Features*" screen where the following should be checked:

- Virtual Machine Platform
- Windows Hyper-V platform.
- Windows subsystem for Linux.

   After this step press OK and restart the computer

1. After the installation of wsl is complete, you need to restart the computer
2. Upon restarting the computer you can run the command `wsl -l -v` to find out what Linux distrubutions are already installed (You may have automatically installed the latest version of Ubuntu, as happened in our case, alternatively if there are no distributions already installed or if you prefer to install different ones see the following points)
3. To view all available distributions run the command `wsl -l -o`
4. To install a new distribution run the command `wsl -install -d NAME`, where NAME should be replaced with one of the distrubutions of your choice that appear as indicated in the previous point under the NAME column
5. You can install a variety of distributions, and afterwards you must reboot the computer. Once restarted you can start the distribution directly from the windows start menu

Enable WSL alternatively.

You can also enable the Windows subsystem for Linux through the operating system interface, without using the command line. The steps to be performed in this case are:

1. See section 2.2 of the previous method
2. Install the desired Linux distribution by downloading it directly from the Microsoft Store. Install it as you would any other application. Once the installation is finished, you can start it via the windows Start menu

Linux programs with a graphical user interface.

One of the main advantages of the WSL subsystem for windows 11 is precisely that it can natively run Linux programs with a graphical interface (which is not possible with WSL for windows 10, except by also installing a graphical server). To do this one must:

1. Boot the Linux distribution we chose earlier (If this is the first boot, a user name and password will be required)

2. Make sure you have the latest version of the distribution and possibly upgrade it via the command `sudo apt update && sudo apt upgrade -y`

3. Now you can then install any Linux program with a GUI via the command `sudo apt install NAME -y`, where NAME should be replaced with the name of the desired program. Some of the most common and widely used are: gimp (digital image processing software); gedit (text editor); nautilus (file management software).

4. Once installed simply type the name of the program into the Linux window and its window will open (Note that the terminal may show warning (Warning) messages or actual errors (Error): this is all completely normal and does not indicate a malfunction).

SECTION 2

2.1 INTRODUCTION TO LINUX

In this part of the project, we start talking about the basis of the language you have to know and use while operating on a Linux shell. We created a tutorial meant for students that are starting to have a first approach with a Linux-powered system, basing our choices in the making on the difficulties and the needs we had ourselves when trying to learn how to use it.

Our tutorial starts with an overview of the most important and commonly utilized commands, in particular the ones that are specific for starting to learn how to move between directories and basic file or directory manipulation, with a special focus on .pdb files manipulation. In fact, the aim of this tutorial is to be used in a teaching that sets the basis for utilizing Linux systems in 3D molecular modelling, in particular for studying proteins and their behavior in complex systems.

For each command we showed, we explained the most common uses and made examples about the most common applications, such as specific integrated functions that allow the user to utilize them with a large variety of possible declinations. Of course we could not have been able to explain all of the possible functions that can be used for every command, because they are a large number and the examples would have been endless, but we chose the ones that we tought could have been the most useful for a beginner, and other ones that appeared the most useful for making the students understands how the command works and how it can be utilized in different kind of applications.

To make it easier for the students to get familiar with the language, we provided some exercises in the end of the tutorial, that are structured to force them to use every command we talk about and to really question their functioning, which is a necessary passage to really understand the power and the features of any of them.

For each exercise, we also provided its commented solution using a code block that simulates a Linux shell and can be executed directly in the Colab environment, allowing the students to clearly see which results those codes actually give in a real Linux environment. The students can also use the simulated shell they find in the end of the notebook to personally try to elaborate the solutions of the exercises, or also to practice individually to get to know the environment and the language better. We think this could be particularly useful for those students who have difficulties in having a linux envirnoment available on their computers, or who simply haven't installed the VirtualMachine or the Linux Subsistem yet.

Here is a list of the most basic Bash language commands that allow you to begin interfacing with a Linux shell, particularly to start moving through folders and manipulating files.

2.2 LINUX COMMANDS

2.2.1 COMMANDS TO NAVIGATE THROUGH FOLDERS AND FILES

- **cd**

  Command used to move to another folder.

  example: `cd /home/user/name_directory`

- **cd ..**

Command used to return to the top folder.

Using `cd ../..` for example, goes up two folders, and so on.

- **ls**

  Lists all items (files and folders) in the folder you are in.

  `ls [options] [target]`

  SPECIAL CASES >

  ls /folder

  List items in the specified folder

  ls -l

  Show more information about the files and folders listed

  ls -a

  It also shows the hidden files present in the folder

- **pwd**

  Print the file or directory path on the screen.

- **mv**

  Rename the folder or file

  example: `mv filenamenew`

- **cp**

  Copy files or folders from one directory to another.

  example: `cp filename destination` (if you are in the directory where the file is).

- **mkdir**

  Creates a new folder in the current directory

  example: `mkdir foldername`

  example: `mkdir /home/newfolder` (creates the folder with the given path even if you are not in the directory where you want to create it)

- **touch**

  Creates new file whose name and extension are given.

  example: `touch file.txt`

2.2.2 COMMANDS FOR PRINTING TO SCREEN OR FILE.

- **echo**

  Print the given text to the terminal or to a file.

  example: `echo "TEXT"` (Print to screen)

  example: `echo "TEXT">file.txt` (creates new file)

  example: `echo "TEXT">>file.txt` (adds into existing file)

  It is also possible to print the contents of a variable or the outcome of a command in a file or variable.

  example: `echo $variable>>file.txt` (prints the contents of the variable in the file)

  example: `echo $(pdw)>variable` (prints the file path in the variable)

- **cat**

  Prints the contents of a file on the screen.

  example: `cat file.txt`

- **head / tail**

  Prints the first (head) or last (tail) 10 lines of the file.

  example: `head file.txt` > **SPECIAL CASES** >

  head -n 50 filename

  Print the first 50 lines of the file

  tail -n 20 filename

  Print the last 20 lines of the file

  tail -n +20 filename | head -n 11

  Print the lines between 20 and 30 (11 elements)

2.2.3 COMMANDS TO OPERATE ON FILE

- **grep**

  Isolates rows in a file containing a word, number, pattern.

  example: `grep "WORD" file.txt`.

  example: `grep "WORD" file.txt> file2.txt` (prints lines containing WORD from file.txt into file2.txt).

- **uniq**

  Finds repeated lines and keeps only one copy

  example: `uniq file.txt> file2.txt` (prints the file with the non-repeated lines in file2.txt).

- **sort**

  Sorts the rows of a file in numerical and then alphabetical order. Numeric has priority over alphabetical

  example: `sort input.txt>order.txt` (sorts the input file into the destination file).

- **cut**

  Print specific parts of each line of a file (columns or characters).

  > **SPECIAL CASES** >

  cut -c1-5 filename

  the `-c` command allows you to print precise characters from each line, in this case characters 1 through 5.

  cut -d: -f3 filename

  The `-d` command allows you to tell the system which, within the file, is the character separating the columns, so that it can treat it as divided into such and not by single character (in this example, the columns are separated by the character ":"). The `-f` command allows you to indicate which column to select (in this example the third).

  cut -d: -f3 --complement filename

  The `--complement` command allows you to "reverse" the action of the previous command: column 3 will be the only one not selected.

- **wc**

  Count the words, rows, or elements indicated by the options.

  > **SPECIAL CASES** >

wc -l filename

Count the lines in the file.

wc -w filename

Count the words.

wc -m filename

Count the characters.

wc -c filename

Count bytes.

ALTERNATIVE SYNTAX >

```
cat file.txt | wc -option
```

- **awk**

  It is a very versatile command used in file scripting. An example of its use is as follows, but being a true scripting language its uses are many.

  The structure of an awk script will be of the type:

  ```
  pattern {action}
  pattern {action}
  ```

  A more detailed discussion of the command is explained later in the dedicated section.

2.2.4 OTHER USEFUL COMMANDS.

- **--help**

  Allows you to have the terminal provide an explanation of how a command works.

  example: `echo --help` or `echo -h` (print explanation of echo command).

- **clear**

  Clears the screen while staying in the folder you are in.

- **sudo**

  Allows you to run a command as an administrator, so with higher level permissions. Prompts the user for a password.

  example: `sudo nomecommand`

2.3 THE AWK COMMAND.

AWK can be defined as a generic filter for text files. It processes one line at a time of the text file, performing different actions depending on whether the line meets certain conditions or contains certain patterns. It then reads line by line and applies the rules defined by the programmer to each.

SUMMARY:

- Everything that is to be read in the specific language of AWK must be inserted between ' '
- Actions in awk are enclosed in curly brackets { }
- Multiple instructions are executed in the order they appear and must be separated by ;

- The pattern consists of the expression enclosed between / /

Below we provide some examples of possible uses of the AWK command to perform various operations

EXAMPLE 1: Extract from a file only the rows that contain a certain word

In [ ]:
```
%%bash
cat elenco.txt | awk '/MARIO/ { print }'   #print on the screen only the lines
                                           #that contain the word 'MARIO'

# Alternatively, you can also use
awk '/MARIO/ { print }' file1 file2 file3   #Receives the files to be parsed
are
                                           #passed in standard input
# or
cat elenco.txt | awk '/MARIO/'
```

In case you want to make longer programs that cannot be executed directly from the command line, you can save the program to a file and specify to AWK the name of the file to be executed.

In [ ]:
```
%%bash
echo '/MARIO/ { print $0 }' > cerca-mario.awk    # variable $0 contains
                                                  # the entire line
cat elenco.txt | awk -f cerca-mario.awk          # the '-f' option allows you
to
                                                  # work on the files
```

Also widely used with AWK are the BEGIN and END commands, which allow an instruction to be executed at the beginning or end of the program, respectively.

In [ ]:
```
%%bash
awk 'BEGIN { print "Start Processing." }; {print $1 }; END { print "End
Processing." }'
```

EXAMPLE 2: Count the number of rows in a file (such as wc -l)

In [ ]:
```
%%bash
awk 'END  { print NR }'  # counts the number of rows entered as input
# or
awk 'END  { print NR }' elenco.txt     # count the rows of the inserted file
                                        #from command line
```

EXAMPLE 3: extract an input column (like the cut, but in some cases the cut may not work).

In [ ]:
```
%%bash
df -k | awk '{ print $4 }' elenco.txt #print column 4 of the supplied file
                                       #in input
```

EXAMPLE 4: Totalizing a column in input.

In [ ]:

```
%%bash
awk -v ncol=3 '{ sum += $ncol } END { print sum }'    # sum all the values
                                                      #of column 3 and at the
                                                      #end prints the sum
                                                      #total
```

EXAMPLE 5: Looking for the maximum value of a column.

In [ ]:

```
%%bash

awk '{ if ($1 > max) { max = $1} } END { print max }' #if the value acquired
                                                      #in input is greater than
                                                      #the maximum one until
                                                      #then met saves it
                                                      #in the variable max
                                                      #it eventually prints to
                                                      #video the maximum value
```

EXAMPLE 6: Print all rows whose x field ranges from m to n (generic values). For example, print the rows whose fourth field ranges from 31 to 33.

In [ ]:

```
%%bash

awk '$4 == 31, $4 == 33 { print $0 }'   #imposes the values of m=31 and n=33 of
                                        #the rows to be printed and then prints
                                        #all of the line ($0)
```

AWK allows two or more patterns to be combined using the logical AND operator (&&) and the OR operator (||)

EXAMPLE 7: Print rows whose third field is greater than 50 and fourth field less than 30

In [ ]:

```
%%bash

awk '$3 > 50 && $4 < 30 { print $1 }'    # imposes the two conditions.
                                          #( $3>50 e $4<30 )
                                         # if both are verified it prints
                                         # on the screen the first field
```

EXAMPLE 8: Print the file name and number of rows.

In [ ]:

```
%%bash
   # FILENAME and NR extract the filename and number of rows.

awk 'END { print "File", FILENAME, "contains", NR, "lines" }' elenco.txt
 # the strings to be printed should be inserted between ""
 # while the variables FILENAME and NR, which contain respectively.
 # the filename and the number of lines no.
```

Mathematical operations can also be performed more easily with AWK.

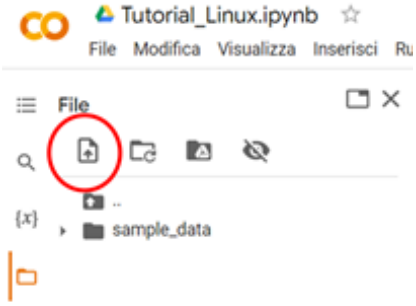
EXAMPLE 9: Calculate the sum of the values stored in the first field of each row.

In [ ]:

```
%%bash

awk '{ sum += $3 } END { printf "%d\n", sum}'  # printf allows you to have
                                               #more control over the format
                                               #of the output than print.
                                               # instruction in the first
                                               #parenthesis performs the sum
                                               # upon completion prints out on
                                               #the screen the result
```

2.4 PROPOSED EXERCISES

> To upload the file to be used for the exercises use the following procedure: On Colab, on the left bar select the folder icon. You will be faced with the following menu >



> Select the icon circled in red and choose the file to upload from your pc. Keep in mind that when the notebook is closed, the uploaded temporary files are deleted and must be uploaded again the next time you open it. > **WARNING**: To run the following cells, you must upload the 1yzb.pdb file to colab, as shown above.

You can download the .pdb file directly to your pc by clicking on this link
https://files.rcsb.org/download/1YZB.pdb

Alternatively, simply run the cell below to temporarily upload the file to colab

In [ ]:

```
#@title Run the cell to upload the file
!wget  https://files.rcsb.org/download/1YZB.pdb
```

```
--2022-06-03 13:08:15--  https://files.rcsb.org/download/1YZB.pdb
Resolving files.rcsb.org (files.rcsb.org)... 132.249.210.222
Connecting to files.rcsb.org (files.rcsb.org)|132.249.210.222|:443...
connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/octet-stream]
Saving to: ‘1YZB.pdb’

1YZB.pdb                [              <=>   ]   2.27M   571KB/s    in 3.9s

2022-06-03 13:08:19 (597 KB/s) - ‘1YZB.pdb’ saved [2384478]
```

2.4.1 EASY EXERCISES

EXERCISE 1

Create the folder "ese01," and create the file "example.txt" inside the folder. Write in the file a text of your choice, then add in the same file the path to the file itself (without deleting what the file already contains). Finally, print the contents of the file on the screen.

HINTS: mkdir, touch, echo, cat, pwd, >, >>

In [ ]:

```
#@title Exercise 1
%%bash

mkdir ese01  #create folder ese01 in the directory I am in.
touch exemple.txt    #create new file "esempio.txt"
echo Hello Pippo >>esempio.txt    #print the text in the existing file
echo $(pwd) >>esempio.txt    #print the file path in the file, without
                             #overwriting.
                             #The $ symbol is used to extract the contents of
the
                             # pwd command.
cat exemple.txt   #print on screen the file
```

EXERCISE 2

Create the folder "ese02" and copy the file 1yzb.pdb into it. Print the first and last lines of the file on the screen, then print them in the file "firstlast.pdb". Count the lines in both files.

HINTS: mkdir, head, tail, cat, wc, cp

In [ ]:

```
#@title Exercise 2
%%bash

mkdir /content/ese02
cd /content       # I move to the folder containing the file because
                    # in my case this is where the file I need is located.
cp 1YZB.pdb /content/ese02  #copy the file to the destination directory.
                            # ~/content/ese02
head -n 1 1YZB.pdb
head -n 1 1YZB.pdb >firstlast.pdb  # I extract the first line of the file
                                   #and print it into the newly created file
                                   #firstlast.pdb
tail -n 1 1YZB.pdb                    # I extract the last line
tail -n 1 1YZB.pdb >>firstlast.pdb   # print it in the same file
cat firstlast.pdb | wc -l  # count the lines in the file
                           # (with this command line, only the number of
                           #lines of the file is printed to the
                           #screen only the number of lines in the file,
                           #without repeating its name)
cat 1YZB.pdb | wc -l
```

EXERCISE 3

Create the directory /content/ese03 and move inside it. print the directory path to a folder called pathx. print the variable on the screen.

HINTS: mkdir, pwd, echo, cd

In [ ]:

```
#@title Exercise 3
%%bash

mkdir ese03
cd ese03
pathx=$(pwd)
echo $pathx
```

EXERCISE 4

Create the file file04.txt and print the variable pathx inside it. Finally, print on the screen the file

HINTS: cat, echo, pwd, touch

In [ ]:

```
#@title Exercise 4
%%bash

touch file04.txt
pathx=$(pwd)
echo $pathx >>file04.txt
cat file04.txt
```

EXERCISE 5

Create the directory /content/ese05 and copy the downloaded .pdb file to it. Extract the first 20 rows from the file, save them in the file first20.pdb. Finally save only the columns corresponding to "atom number" and "atom name" in the file simple.pdb.

HINTS: mkdir, cp, grep , head, awk, cat

In [ ]:

```
#@title Exercise 5
%%bash
mkdir /content/ese05
cd /content
cp 1YZB.pdb /content/ese05

grep "^ATOM" 1YZB.pdb | head -n 20 >first20.pdb  #extract only the lines that
                                                 #contain the word ATOM in the
                                                 #first column
                                                #(The symbol ^ indicates
                                                #precisely that
                                                #ATOM must be in the first
                                                #position to be
                                                #extracted
                                                #with head I extract the first
                                                #20 lines and I
                                                #print them into the new file
awk '{print $2 , $3}' first20.pdb > simple.pdb   #print columns 2 and 3
                                                  #from the file first20.pdb
```

```
                                                    # into the
                                                    #new file simple.pdb
cat simple.pdb
```

EXERCISE 6

Working on the .pdb file previously copied into the ese05 folder, print its contents to the screen. After cleaning up the window, print its first 50 lines on the screen, then print them in the file first50.pdb. When finished, verify that you have performed the step correctly by counting the lines in the new file and verifying that there are 50 of them.

HINTS: cat, clear, wc, head

In [ ]:

```
#@title Exercise 6
%%bash

cat 1YZB.pdb
#cd ~/ese05
clear
head -n 50 1YZB.pdb
head -n 50 1YZB.pdb >first50.pdb
cat first50.pdb | wc -l
```

2.4.2 DIFFICULT EXERCISES

EXERCISE 7

Extract only the lines beginning with ATOM from the file 1yzb.pdb, and save them to the fail atoms.pdb. Extract the first 10 lines of atoms.pdb and save them to the firts10.pdb file. Do the same with the last 10, saving them in last10.pdb. Print lines from 10 to 20 of the atoms.pdb file on the screen.

HINTS: grep, head, tail

In [ ]:

```
#@title Exercise 7
%%bash

cat 1YZB.pdb | grep ATOM >atoms.pdb
head -n 10 atoms.pdb >first10.pdb
tail -n 10 atoms.pdb >last10.pdb
tail -n +10 atoms.pdb | head -n 11
```

EXERCISE 8

From the file 1yzb.pdb extract only the carbon atoms and save them in a new file called atoms_c.pdb. Then count the number of carbon atoms. Create the file number_atoms.txt and write inside "There are "N" carbon atoms in the file 1yzb.pdb." Try other elements of your choice (e.g., oxygen, nitrogen, hydrogen) and add their data to the file number_atoms.txt, being careful not to delete what you wrote previously.

HINTS: awk, wc, touch, >, >>

In [ ]:

```
#@title Exercise 8
%%bash

awk '$12 ~ /C/ {print $0}' 1YZB.pdb >atomi_c.pdb     # As an alternative to
```

```
                                                    # the command
                                                   #grep, seen in the other
                                                   #exercises one can use
                                                   # an awk action
n_atomi=$(cat atomi_c.pdb | wc -l)   #save in the variable the number of rows
                                     #of the atoms_c file

echo In file 1YZB.pdb are present $n_atomi atoms of carbon >number_atoms.txt

# print in the file number_atoms.txt the given line of text, where I wrote
# $n_atoms will be printed the contents of the variable n_atoms.

awk '$12 ~ /H/ {print $0}' 1YZB.pdb >atomi_h.pdb
n_atomi=$(cat atomi_h.pdb | wc -l)   #print in the variable the number of
                                     #lines of the file atoms_h.pdb.

echo In file 1YZB.pdb are present $n_atomi atomi of hydrogen >>number_atoms.txt
# pay attention to >> which (unlike >) is used to add text
# to the file without deleting what is already there

cat number_atoms.txt
```

EXERCISE 9

Extract the ATOM lines from the file 1yzb.pdb and save them in the file "atoms.pdb". Then extract from that file the Serine residues (marked SER in column 4) and transcribe them into the file "residuiser.pdb." Print the coordinates of all the atoms belonging to those residues in the file "xyzser.pdb," and finally count the SER residues and print their numbers on the screen.

HINTS: grep, wc, awk, uniq, cat, >

In [ ]:

```
#@title Exercise 9 extended version
%%bash


grep "^ATOM" 1YZB.pdb>atoms.pdb
awk '{print $3, $4, $7, $8, $9}' atoms.pdb>residui.pdb  #print columns 3,
                                                         #4, 7, 8, 9 of the
file
                                                         #atoms.pdb in the file
                                                         #residui.pdb
grep "SER" residui.pdb>residuiser.pdb
awk '{print $3, $4, $5}' residuiser.pdb>xyzser.pdb   #print columns 3,
                                                     #4, 5 of the file
                                                     #residuiser.pdb in
                                                     # the xyzser.pdb.pdb file.
grep "CA" residuiser.pdb>residuiser1.pdb  #extract only the lines with carbon
                                            #alpha, this ensures that I
                                            #extract only one for each residue.
cat residuiser1.pdb | wc -l
```

In [ ]:

```
#@title Exercise 9 contracted version
%%bash
```

```
grep "^ATOM" 1YZB.pdb | grep SER | grep CA >residuiser.pdb  #extract only the
                                                           #strings with
carbon
                                                           #alpha, this
                                                           #guarantee to
                                                           #extract only one
                                                           #for each residue
awk '{ print $7, $8, $9 }' residuiser.pdb > xyzser.pdb     # I use awk to save
                                                           #the columns
                                                           #containing the
                                                      #coordinates in the file
cat residuiser.pdb | wc -l                  # use wc to count rows
```

EXERCISE 10

Extract the REMARK columns of the 1yzb.pdb file into a "remark.pdb" file. Print on screen the rows from 20 to 40, then print on screen the rows with 500 in the second column.

HINTS: grep, head, tail, cat

In [ ]:

```
#@title Exercise 10
%%bash

grep "^REMARK" 1YZB.pdb >remark.pdb
tail -n +20 remark.pdb | head -n 21   #print lines 20 through 40.
                                      #(21 rows in total)
grep "500" remark.pdb >remark1.pdb
cat remark1.pdb
```

EXERCISE 11

Create a new file called new.pdb containing the first 100 atoms of the file 1yzb.pdb . Extract and print on the screen all the x-coordinates of the atoms belonging to the new file.

HINTS: cat, grep, head, awk

In [ ]:

```
#@title Exercise 11
%%bash

cat 1YZB.pdb | grep ^ATOM >atoms.pdb
head -n 100 atoms.pdb >new.pdb
awk '{ print "The coordinate x is : ", $7}' new.pdb
```

EXERCISE 12

Create a new file called extracts.pdb consisting of the first 10 lines, lines 100 to 110 and the last 10 lines of atoms.pdb. Extract only the lines of atoms.pdb with the x-coordinate less than 1.5 and save them in the file x_below.pdb. Finally extract from atoms.pdb only the columns containing the x, y and z coordinates and the atom number, and save them in simple.pdb.

HINTS: cat, grep, head, tail, cut, awk

In [ ]:

```
#@title Exercise 12
%%bash

cat 1YZB.pdb | grep ATOM >atoms.pdb
head -n 10 atoms.pdb >extracted.pdb
tail -n +100 atoms.pdb | head -n 11 >>extracted.pdb
tail -n 10 atoms.pdb >>extracted.pdb

awk '$7 < 1.5 { print $0 }' atoms.pdb >x_below.pdb

awk '{ print $2, $7, $8, $9 }' atoms.pdb >simple.pdb
cat x_below.pdb
cat simple.pdb
```

EXERCISE 13

Create a file called file1.pdb containing only atoms having the x coordinate between 10.3 and 10.9 and two others called file2.pdb and file3.pdb containing only atoms having the y and z coordinates between the same values, respectively. At this point merge the three files into one called filefinale.pdb .

In [ ]:

```
#@title Exercise 13
%%bash
cat 1YZB.pdb | grep ^ATOM >atoms.pdb
awk '$7 >= 10.3 && $7 <= 10.9 { print $0 }' atoms.pdb >file1.pdb
awk '$8 >= 10.3 && $8 <= 10.9 { print $0 }' atoms.pdb >file2.pdb
awk '$9 >= 10.3 && $9 <= 10.9 { print $0 }' atoms.pdb >file3.pdb
echo "Atoms with x between 10.3 and 10.9 inclusive are."
cat file1.pdb
```

EXERCISE 14

Count the number of alanine residues contained in the file atoms.pdb (labeled ALA). Count how many trajectories are contained in 1yzb.pdb (marked by the word MODEL). Create a sequence.txt file containing all the numbers from 1 to 1000.

HINTS: cat, grep, wc, uniq, sort, seq

In [ ]:

```
#@title Exercise 14 contracted form
%%bash

grep ALA atomi.pdb | grep CA | sort | uniq -w 20 | wc -l
grep -v REMARK 1YZB.pdb | grep MODEL | wc -l
seq 1 1000 >sequence.txt
```

In [ ]:

```
#@title Exercise 14 extended form
%%bash
grep "ATOM" 1YZB.pdb>atoms.pdb
grep "ALA" atoms.pdb>residuiala.pdb

grep "CA" residuiala.pdb>residuiala1.pdb  #extract only the lines with carbon
                                          #alpha, this ensures that I
                                          #extract only one for each residue.
cat residuiala1.pdb | wc -l
```

```
grep -v "REMARK" 1yzb.pdb>noremark.pdb
grep "MODEL" noremark.pdb>model.pdb
cat model.pdb | wc –l
seq 1 1000 >sequence.txt
cat sequence.txt
```

2.5 USEFUL COLAB CELLS

Here is a useful cell for temporarily loading any callable files in the code

In [ ]:

```
from google.colab import files
files.upload()
```

Here at hand is a cell that simulates a full-fledged linux terminal

In [ ]:

```
!pip install google-colab-shell
from google_colab_shell import getshell
getshell()
```

SECTION 3

Based on what we saw in the previous section of the tutorial, we will now address a new topic: the implementation of a **script** in bash

3.1 SCRIPTING IN BASH

A **BASH SCRIPT** is nothing more than a file containing a series of instructions that are executed in order from first to last. Once the script has been made, you just have to launch it from the command line in order to execute all the instructions contained within it.

It is particularly useful when very repetitive operations must be performed. It is faster than executing all the individual commands one by one, plus it reduces the risk of making mistakes.

There are several ways to make a bash script.

Example 1:

In [ ]:

```
%%bash
vi filename.sh    # the command allows you to create a script within the
                  # which you can insert all the instructions that you
                  # want to execute

chmod +x filename.sh     # it's the command needed to activate the script. If it
                         # is not executed, upon invoking the script, from the
                         # command line the error
                         # "-bash: ./filename:Permission denied" will appear

./filename.sh        # ./ is the command that allows the script to be executed.
```

Example 2:

In [ ]:

```
%%bash
cat > filename.sh     # the command 'cat >' followed by the filename, allows you
                      # to create a script, that is not executable immediately
                      # but saved in the filename.sh file.
                      # In order to exit the script mode, it is necessary to
                      # press Ctrl+D.
chmod u+x filename.sh    # it's used to make the script executable
sudo chmod 777 filename.sh  # this command also allows you to make
                            # the script executable, but to use it, it is
                            # necessary that in the first line of the script
                            # appear ' #!/bin/bash ', plus, being a
                            # sudo command will prompt you for
                            # the password in order to execute it
bash filename.sh     # 'bash' is the command to run the script.
                     # You can also, alternatively, use the command
                     # ' ./ ' to run the script
```

Another very useful command in scripting is **nano**. It allows you to access and edit the script file, being able to move freely back and forth between lines and various commands.

In [ ]:
```
%%bash
nano filename.sh   # to be able to exit the script once the
                   # changes press Ctrl+X then press Y to save the
               # changes or N otherwise finally press the key
               # Enter once you exit the script it is again executable
```

On the terminal will then open the text editor with the contents of the file, the first line of a script is dedicated to the interpreter to be used (The interpreter is the operating system program that reads, interprets the script commands and executes them), so in this case it will have to be written `#!/bin/sh`. I can then enter the commands that will then be executed. When I have finished writing, I save the changes to the file with CTRL+O and exit the editor with CTRL+X. Now then, as in the previous cases, I must make the script executable (`chmod +x filename.sh`), and run it (`./filename.sh`)

Some very useful structures for making scripts: **IF STATEMENTS** and **LOOPS**.

3.2 IF STATEMENTS

The **if** command can be very useful within a script because it allows a series of statements to be made only if a certain condition is met.

In [ ]:
```
# example of syntax of an if statements
%%bash

n=12
if [ n -gt 10 ]                  # the condition to be checked must be
                                 # placed in brackets [], taking care to
                                 # respecting the spaces
do
echo n è maggiore di 10          # instruction to be executed if the
condition
                                 # is verified

else echo n è minore di 10       # instruction to be executed if the
condition
                                 # is not verified

fi                               # it is necessary to remember to close the
if
                                 # with the command 'fi'
```

The following are some comparison operators that are used to impose the conditions of if statements, but also of while and until loops.

Comparing integers:

In [ ]:
```
if [ "$a" -eq "$b" ]                                # -eq --> is equal to
if [ "$a" -ne "$b" ]                                # -ne --> is different
from
if [ "$a" -gt "$b" ] or if (( "$a" > "$b" )) # -gt --> is greater than
```

```
if [ "$a" -ge "$b" ] or if (( "$a" >= "$b" ))# -ge --> is greater o equal to
if [ "$a" -lt "$b" ] or if (( "$a" < "$b" )) # -lt --> is smaller than
if [ "$a" -le "$b" ] or (( "$a" <= "$b" ))# -le --> is smaller or equal to
```

Comparing strings:

In [ ]:

```
if [ "$a" = "$b"]  or   if [ "$a" == "$b"]         # equal to
if [ "$a" != "$b"]                                 # different from
```

3.3 LOOPS

The **FOR** loop can be used to perform repetitive operations, such as parsing all files within a folder.

In [ ]:

```
# for loop syntax example
%%bash

for file in $(ls)         # what follows the 'for' indicates for which elements
                          # to execute the instructions contained
                          # within the loop
do
....                      # after the 'do' should be inserted the commands that
                          # have to be on the variables in the list

done                      # indicates the end of the cycle
```

The **WHILE** loop can be used to carry out a series of instructions until a certain condition turns out to be true. The moment the condition defined by the 'while' is no longer true, it exits the loop directly and proceeds with subsequent commands

In [ ]:

```
# while loop syntax example
%%bash

a=0
while [ "$a" -lt 5 ]      # the 'while' is followed by the condition between []
do
((a++))                   # after 'do' you have to insert the instructions that
                          # have to be executed within the loop

done                      # ends the loop
echo $a
```

3.4 COMMAND *READ* IN BASH ON COLAB

The **read** command, as we have seen, can be used to take input data provided directly by the user and save it in a specific variable.

Example:

In [ ]:

```
%%bash
echo Insert two numbers:
read -r a b
```

If you tried running this code on your linux terminal, you would see that it works correctly and saves the two numbers you entered in the variables a and b.

Running the cell on colab, instead, things don't work as well. This is because bash commands are executed in a subshell on colab, so this interactive mode doesn't work.

In order to take advantage of the **read** command anyway, you must first create a script inside a cell and then run it in a separate cell. (see exercise 2)

It works in the same way if you want to pass parameters directly from the command line, on colab you must write a script and then run it in a separate cell (**!bash nomefile.sh parameter_1 parameter_2**) (See exercise 1).

WARNING: the syntax of the read command is slightly different in this mode.

Here's how to do it:

In [ ]:
```
%%bash
echo "#!/bin/bash
read -p 'insert a number ' var
echo You have inserted \$var " > file.sh

# With the command #!/bin/bash the script is created.
# All commands contained in the script must be put between ""
# At the end of the script you use > filename.sh to save it inside a script
file
# which can be called at any time from another cell.
```

In [ ]:
```
!bash file.sh
```

The **!bash filename.sh** command allows you to launch the previously defined and executed script.

3.5 FILE .PDB

A PDB file describes the three-dimensional structure of a protein contained in the Protein Data Bank. It also contains a whole set of secondary information inserted at the beginning as a header ("HEADER"). This part includes information about the authors of the research that determined the structure of the protein, some experimental observations or even the list of amino acids of which it is composed.

The main as well as the most extensive part, however, remains the list of atoms of which the protein is composed. For each atom a whole range of information is given including:

- the spatial coordinates (x, y, z)
- the name and number of the residue to which they belong.
- the respective chain (useful in the case of protein with a quaternary structure)
- a temperature factor (describing their vibration)

- the name of the atom, which is very useful for studying specific atoms such as carbon α (denoted by CA)

3.6 SCRIPT EXERCISES

The following are some exercises that you can perform to apply what was explained in the first part of the tutorial.

WARNING: In order to run the cells, simply run the cell below to temporarily upload the necessary files and folders to colab

In [ ]:
```
#@title **Run this cell in order to load the files needed to perform subsequent exercises**
%%bash
! wget https://files.rcsb.org/download/1YZB.pdb
! wget https://files.rcsb.org/download/7MCI.pdb
! wget https://files.rcsb.org/download/7WN4.pdb
! wget https://files.rcsb.org/download/7TZ4.pdb
mkdir /content/student
```

EXERCISE 1

Create a script that when started receives from the command line the name of a file to take the atoms from, the name of another file to save them to, the name of a coordinate, and reorders the atoms in the chosen file according to the descending order of the chosen coordinate, printing them to the chosen file.

HINTS: sort (-k, -nr), $N (for the corresponding data entered at program execution)

In [ ]:
```
#EXERCISE 1
#@title solution ex1
%%bash
echo "#!/bin/bash
case \$3 in   # I use a switch case (instead of nested ifs )
              # to handle better in the case of
              # multiple possible choices
x)
coordinate=7;; # based on the chosen coordinate I take the
                # number of the

y)              # corresponding column
coordinate=8;;
z)
coordinate=9;;
esac

sort -k\$coordinate -nr \$1 > \$2 " > es_1.sh
               # sort the chosen file ($1)
               # and save it to the second chosen file ($2).
```

In [ ]:
```
!bash es_1.sh 1YZB.pdb test.pdb x
```

In [ ]:

```
! cat test.pdb    # print the test.pdb file to verify that
                  # the atoms have been reordered properly
```

EXERCISE 2 (script)

Make a script that

- reads all the .pdb files in a folder and prints all their names on the screen.
- receives as input the name of one of them and the name of a residue *print the total number of residuals in the file and the number of the residual chosen

HINTS: for, ls, read, if

In [ ]:

```
#EXERCISE 2
#@ title solution ex2
%%bash
echo "#!/bin/bash
echo The files in the folder are:
for file in \$(ls *.pdb)
do
    echo \$file
done
read -p 'Choose one of the files and type the name: ' filename
# read -r filename (if you're not working on colab you can just use this
comman)
read -p 'Choose the residue of which you want to know how many there are and
         type its abbreviation (all in uppercase): ' res_name
# read -r res_name
for file in \$(ls)
do
  if [ \$file = \$filename ]
  then
    res_tot=\$(grep ^ATOM \$filename | grep CA | sort | uniq -w 20 | wc -l )
    res_choice=\$(cat \$file| grep \$res_name| grep CA| sort| uniq -w 20| wc -
l)
  fi
done
echo There are a total of \$res_tot residuals in the \$filename file,
#***
     \$res_choice of \$res_name " > es_2.sh


#TIPS: alternative code to check if a file is .pdb
#typefile=$(file -b $i) (N.B. $i contains the names of files in a directory)
#echo $typefile | cut -b 1-17 >typefile.txt
#typefilefinal=$(head -n 1 tipofile.txt)
#if [ "$typefilefinal" = "Protein Data Bank" ]
```

In [ ]:

```
!bash es_2.sh
```

EXERCISE 3 (script)

Make a script that

- receives in imput the name of a folder.
- read all the .pdb files in a folder.
- copy into a file called "list_protein.txt" the name of the file and the title of the protein contained within it (The title from the protein is contained within the file)
- count how many proteins have been written to the file and print on the screen the words "In the chosen folder there are N proteins and they are: " with the list of files with their titles following

HINTS: for, ls, if, cat, echo

In [ ]:

```
#EXERCISE 3
#@title solution ex3
%%bash
>list_proteins.txt
for file in $(ls)
do filename=${file##*/}
  extention=${filename##*.}
  if [ $extention = pdb ]
  then
    title=$( cat $file | grep TITLE)
    echo $file: $title >>list_proteins.txt
  fi
done
proteins_num=$(cat list_proteins.txt | wc -l)
echo In the folder there are $proteins_num proteins and they are:
cat list_proteins.txt
```

EXERCISE 4 (script)

Make a script that allows you to:

- read all the files in a directory.
- copy all .pdb files to the /content/student directory.
- count the number of HIS residues in each .pdb file and create a reshis.csv file in which the filename (< filename >) and the number (< n_residues >) of HIS residues counted must appear for each line.
- update the minres.stat file with the phrase: "< filename > has the lowest number of histidine residues equal to < rmin >

HINTS: for, ls, if, grep, echo, cd, cat, sort, uniq

Is it possible to avoid reading all the files in the folder since then only the .pdb is selected? Maybe directly reading only the files with the extension of interest (**ls** options)

In [ ]:

```
#EXERCISE4
#@title solution ex4
%%bash
rmin=10000 #initialize the variable
> hisres.csv
```

```
# It allows to move to the folder whose files you want to read
dir_source=/content
dir_destination=/content/student
cd $dir_destination # I move to the directory where I will later go to work
                    # on the files

# I make a for loop to perform the same operations on all files in the
directory
for file in $(ls $dir_source/*.pdb) # using ls *.pdb you can avoid reading all
                                    # files, but select, already in the for,
                                    # only the .pdb files
do
 filename=${file##*/}
 cp $dir_source/$filename $dir_destination #copies the file to the new
directory
 n_residues=$(grep ^ATOM $filename| grep HIS| grep CA| sort| uniq -w 20 | wc -
l)
  # selects only HIS rows then those with CA, reorder and delete identical
rows,
  # count residuals

  echo $filename , $n_residues >>hisres.csv
      if [ $n_residues -le $rmin ] # checks if the number of residuals is less
                                   # than those in the previous file
      then
         rmin=$n_residues
         echo $filename has the lowest number of histidine residues
              equal to $rmin > minres.csv
      fi
 done

    cat hisres.csv #print the contents of the two generated files on the screen
    echo
    cat minres.csv
```

EXERCISE 5 (script)

Make a script that

- allows you to read the files in a folder whose directory is provided in input by the user.
- count the number of ARG residues and save in the argres.csv file the name of each file with the corresponding number of residues putting them in order, from the file that contains the least residues to the one that contains the most.
- save in a second file xyz.pdb the coordinates of the atoms that have a value of z>100. (In the file must appear, for each line, the number of the atom, its name, the residue and its coordinates)

Ex. 18 CA ARG -43.986 -34.605 1.456

HINTS: read, echo, for, if grep, awk, sort, cat

*Executing this cell will prompt you to enter the path to the folder whose files you want to analyze. In order to proceed with running the program on colab you will need to type **/content**. If you run the code on your linux terminal instead, you will need to enter the full path to the folder on your computer.*

In [ ]:

```
#EXERCISE 5
#@title solution ex5
%%bash
echo "#!/bin/bash
read -p 'Enter the path of the directory you want to access: ' dir_source
# this command allows you to capture as a variable an input entered by the user

cd \$dir_source

>argres.csv
>argres.pdb
>xyz.pdb
# these commands are only used for emptying the files in case the scripts have
# already been run, to avoid shuffling the data

for file in \$(ls \$dir_source/*.pdb) # in this way the loop read only the .pdb
                                     # files in the folder
do
 filename=\${file##*/}
 n_residues=\$(grep ^ATOM \$filename| grep ARG| grep CA|sort| uniq -w 20| wc -
l)
 echo \$filename , \$n_residues >>argres.pdb
 grep ^ATOM \$filename | grep ARG | grep CA | sort | uniq -w 20 >arg.pdb
 awk ' \$9>100 {print \$2 , \$3 , \$4 , \$7 , \$8 , \$9} ' arg.pdb >>xyz.pdb
     # allows selecting only atoms bound to an ARG residue that are
     # in a position with z>100

 sort -g argres.pdb > argres.csv # allows you to sort the files
done

cat argres.csv
echo
cat xyz.pdb " > es_5.sh
```

In [ ]:

```
!bash es_5.sh
```

EXERCISE 6 (script)

Make a script that allows you to:

- Ask the user which folder to open and print on the screen all the files in the folder.
- Ask the user to enter the name of the file to be parsed and the remainder to be counted
- Verify that the file name has been entered in the correct format and count the number of residuals of the chosen type
- Print on the screen " The < filename > contains < n_residues > of < res_name> "

HINTS: echo, read, cd, while, grep

*Executing this cell will prompt you to enter the path to the folder whose files you want to analyze. In order to proceed with running the program on colab you will need to type **/content**. If you run the code on your linux terminal instead, you will need to enter the full path to the folder on your computer.*

In [ ]:

```
#@ title solution ex6
%%bash
echo "#!/bin/bash
read -p 'Enter the path of the directory you want to access: ' directory
# allows you to acquire a varialbile entered as input by the user

cd \$directory

read -p 'Enter the name of the file to be parsed: ' filename
read -p 'Enter the name of the residue that has to be counted,
         using the identifying three-digit abbreviation: ' res_name

# example of a possible control that should be done
# while/do cycle --> executes the contents of do until the condition [] is met.
while [ \$(expr length "\$res_name") -ne 3 ]
# the 'expr length' command allows reading the number of characters in a stript
# It is different from the 'wc' command which reads only rows as input

do
echo The name of the residue was not entered correctly
read -p ' !!!Enter the name of the residue that has to be counted,
         using the three-digit identifying abbreviation: ' res_name
done

n_residues=\$(grep ATOM \$filename | grep \$res_name |sort| uniq -w 20 | wc -l)
echo In \$filename there are \$n_residues residues of \$res_name " > es_6.sh
```

In [ ]:

```
!bash es_6.sh
```

SECTION 4

PYTHON LANGUAGE TUTORIAL

Python is a "high-level," object-oriented programming language suitable for, among other uses, developing distributed applications, scripting, numerical computation, and system testing.

4.1 BASIC ALGEBRA

To perform basic mathematical calculations in python, classical operators are used.

Below you can run some extremely basic code boxes to test different operations.

In [ ]:
```
#SUM
a = 7
b = 5
a + b
```

Out[ ]:
```
12
```

In [ ]:
```
#SUBTRACTION
a - b
```

Out[ ]:
```
2
```

In [ ]:
```
#MULTIPLICATION
a * b
```

Out[ ]:
```
35
```

In [ ]:
```
#DIVISION
a / b
```

Out[ ]:
```
1.4
```

In [ ]:
```
#QUOTIENT
a // b
```

Out[ ]:
```
1
```

In [ ]:

```
#REST
a % b
```

Out[ ]:
```
2
```

In [ ]:
```
#EXPONENTIAL
a ** b
```

Out[ ]:
```
16807
```

4.2 TYPES OF DATA

Variables used in python can be of various types. The most common types are:

- **int** : Variable that contains an integer
- **float** : Variable that contains a floating point number.
- **str** : Variable of type string, which can contain letters and numbers
- **bool** : Variable that can take two values, namely "true," "false," or "none"

To assign a variable a value, the following syntax is used:

In [ ]:
```
var = 10
stringa = "ciao"
```

When naming variables, there are some simple rules: only letters, numbers and underscores can be used, but a variable cannot be given a name that begins with a number. Notice below how the code returns error if these rules are not followed.

In [ ]:
```
1x = 2
```

It is important to keep in mind that in python it is possible to vary the type of a variable simply by assigning it a new value. Notice how in the code box below the variable c is initially an integer (the "type" function confirms this for us), while reassigning it a decimal value automatically becomes a float variable.

In [ ]:
```
c = 4
d=type(c)
print (d)
c = 4.5
d=type(c)
print (d)
```

```
<class 'int'>
<class 'float'>
```

It is also possible to explicitly cast variables by converting them from one type to another, using the commands displayed in the box below.

In [ ]:

```
stringa = "34"
s = int(stringa)
type(s)
```

Out[ ]:

```
int
```

In [ ]:

```
e = 24
e1 = str(24)
type(e1)
```

Out[ ]:

```
str
```

4.2.1 USE AND OPERATION WITH STRINGS IN PYTHON

SUM BETWEEN STRINGS

If you use the + operator between two strings, they will be concatenated, as visible in the following code.

In [ ]:

```
s1 = "what do I eat"
s2 = " for breakfast?"
s1 + s2
```

Out[ ]:

```
'what do I eat for breakfast?'
```

STRING FORMATTING

The % operator is also known as string formatter or string interpolation operator. This is used in the following way.

In [ ]:

```
nome = "mario"
eta = 32
print ("%s is %d years old." % (nome, eta))
```

```
mario is 32 years old.
```

Formatting in the string must include, in parentheses, a dictionary key, inserted immediately after the "%" character.

The main types of conversion are:

%d signed decimal integer

%f decimal in floating point

%s string

If a period "." followed by a number is inserted, this indicates precision.

In [ ]:
```
a = 4
b = 68
print("The relationship between a and b is:%.2f " % (a/b))
```

```
The relationship between a and b is:0.06
```

4.2.2 FORMATTING BY POSITION

The format() function allows the formatting of data into a string; specifically, I insert placeholders enclosed in curly brackets. To each placeholder I assign a numeric or alphanumeric parameter to identify it. With the format() method, I reprint it

In [ ]:
```
stringa="is it bigger {0} or {1}?"
print(stringa.format("Roma", "Milano"))
```

```
is it bigger Roma or Milano?
```

One can proceed equally with *f-strings*, as follows..

In [ ]:
```
nome = "mario"
anni = 23
altezza = 1.75
out = f"My name is {nome}, i'm  {anni} years old and i'm tall {altezza}. In 30
years I will be  {anni+30} years old"
print(out)
```

```
My name is mario, i'm  23 years old and i'm tall 1.75. In 30 years I will be
53 years old
```

4.2.3 INPUT AND OUTPUT MANAGEMENT

The command for screen printing in python is "print," which is used as follows.

In [ ]:
```
print ("hi")
a = 2
print ("i have ",a," apples")
```

```
hi
i have  2  apples
```

To receive a value as input instead, the "input" command is used.

In [ ]:
```
x = input()
```

```
5
```

4.2.4 LISTS

Lists are a data type that contains within it a set of data items that are the same or different in nature, sorted according to the list's own index. This is essential for retrieving a specific element of the list itself. Index numbering starts from 0 and it is also possible to number starting from the last element, which will have index -1 in that case.

In [ ]:
```
#example lists
lista=[2, 5, "cisao", 37]
lista[1] #recall the element in position 1 starting from the left
lista[-1] #recall the last element using inverted numbering.
```

Out[ ]:
```
37
```

It is also possible to retrieve a portion of the list by entering index ends separated by colons.

In [ ]:
```
primi=[2, 3, 5, 7, 11, 13, 17]
primi[2:4]
```

Out[ ]:
```
[5, 7]
```

In [ ]:
```
primi[:3] #if you want to start from the first item in the list.
```

Out[ ]:
```
[2, 3, 5]
```

In [ ]:
```
primi[2:] #recall from element with index 2 to the end
```

Out[ ]:
```
[5, 7, 11, 13, 17]
```

Lists can be created from the range function using the support of list function, which allows elements to be converted to lists.

In [ ]:
```
lista_numerica=list(range(99, 120)) #inserts into the list all the numbers from
99 to 120
lista_numerica
```

Out[ ]:
```
[99,
 100,
 101,
 102,
 103,
 104,
 105,
 106,
```

```
 107,
 108,
 109,
 110,
 111,
 112,
 113,
 114,
 115,
 116,
 117,
 118,
 119]
```

It is possible to use the for loop to iterate the list elements.

In [ ]:

```
for primo in primi:
  print(primo)
```

```
2
3
5
7
11
13
17
```

It is possible to create sublists within the list itself and retrieve them when needed for the entire list or even a single sublist item.

In [ ]:

```
elenco=["ciao", 24, [1, 2, 3, 5], 5]
print(elenco[2])
print(elenco[2][0]) #I am calling up the first element of the sublist
```

```
[1, 2, 3, 5]
1
```

You can edit a list item or delete it by the command.

In [ ]:

```
elenco=["ciao", 24, [1, 2, 3, 5], 5]
elenco[1] = "buongiorno"
elenco
```

Out[ ]:

```
['ciao', 'buongiorno', [1, 2, 3, 5], 5]
```

In [ ]:

```
del elenco[3]
elenco
```

Out[ ]:

```
['ciao', 'buongiorno', [1, 2, 3, 5]]
```

You can do several types of operations with lists very similar to what you can do with strings.

In [ ]:
```
#sum of 2 lists
a=[1, 2, 3]
b=[4, 5, 6]
a+b
```

Out[ ]:
```
[1, 2, 3, 4, 5, 6]
```

In [ ]:
```
#multiplication of lists
a*3
```

Out[ ]:
```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In [ ]:
```
#length lists
len(a)
```

Out[ ]:
```
3
```

In [ ]:
```
#I can check whether an item is present in a list by means of the in and not in
commands that return true or false
6 in a
```

Out[ ]:
```
False
```

In [ ]:
```
#I can convert string to list
stringa = "ciao"
list(stringa)
```

Out[ ]:
```
['c', 'i', 'a', 'o']
```

METHODS LISTS

There are several preset methods or functions that can be used with lists. For example:

In [ ]:
```
spesa=["uova", "latte", "pasta", "cereali"]
spesa.append("carne") #add an element to the list
spesa
```

Out[ ]:
```
['uova', 'latte', 'pasta', 'cereali', 'carne']
```

In [ ]:
```
spesa.remove("pasta") #removes an element
spesa
```

Out[ ]:
```
['uova', 'latte', 'cereali', 'carne']
```

In [ ]:
```
spesa.sort() #sort alphabetically or numerically
spesa
```

Out[ ]:
```
['carne', 'cereali', 'latte', 'uova']
```

In [ ]:
```
spesa.sort(reverse=True) #inverse order
spesa
```

Out[ ]:
```
['uova', 'latte', 'cereali', 'carne']
```

In [ ]:
```
numeri = [23, 45, 45, 90, 32]
numeri.index(45) #returns me the index of the element
```

Out[ ]:
```
1
```

In [ ]:
```
spesa.insert(1, "yogurt") #I insert at position 1 another item to the list
spesa
```

Out[ ]:
```
['uova', 'yogurt', 'latte', 'cereali', 'carne']
```

4.2.5 TUPLE

Tuples are another data type that contain a series of elements (like lists), defined using round brackets. They are very similar to lists with the difference that once created they cannot be modified. They are used when one wants to obtain an immutable list or when iteration speed must be taken into account: tuples, in fact, are much faster.

In [ ]:
```
tuple=(1, 2, 3, 4)
tuple
```

Out[ ]:
```
(1, 2, 3, 4)
```

4.2.6 THE DICTIONARIES

In Phyton, dictionaries are groups of data pairs, each of which consists of a "key" and a "value." To make an analogy with lists, the keys represent the indexes, while the values correspond to the

contents of the variable placed at the position marked by the index. An important difference, however, lies in the fact that keys can consist of any type of data, except for lists or other dictionaries, while values have no limitations on the type of data from which they can consist. Below is an example of initializing a dictionary, whose elements consist of disparate data types.

In [ ]:
```
dizionario = {"chiave1":12, "chiave2":"ciao", 2 :"numero intero",
"lista":[1,3,2,4,5]}
```

To retrieve a dictionary item, it is necessary to look it up using the content of the key, unlike lists, which required that the position (index) be used.

In [ ]:
```
dizionario["chiave1"]
```

Out[ ]:
```
12
```

In [ ]:
```
dizionario[2]
```

Out[ ]:
```
'numero intero'
```

With dictionaries, it is also possible to search for a key within the group, as shown below.

In [ ]:
```
"chiave2" in dizionario  #you can see how you can search in this way for keys,
not values
```

Out[ ]:
```
True
```

In [ ]:
```
12 in dizionario
```

Out[ ]:
```
False
```

It is also possible to insert or delete items from the dictionary with very simple commands.

In [ ]:
```
dizionario={"ciao":"hello","cane":"dog","gatto":"cat","bello":"beautiful"}
dizionario["cibo"] = "food"
dizionario.items()

#The ".items" command used in this example will be explained later, for now
suffice it to know that we need it to print the pairs contained in the on-
screen dictionary.
```

Out[ ]:
```
dict_items([('ciao', 'hello'), ('cane', 'dog'), ('gatto', 'cat'), ('bello',
'beautiful'), ('cibo', 'food')])
```

In [ ]:
```
del dizionario["bello"]
dizionario.items()
```

Out[ ]:
```
dict_items([('ciao', 'hello'), ('cane', 'dog'), ('gatto', 'cat'), ('cibo',
'food')])
```

In [ ]:
```
lista = dizionario.values()  #inserts values into a list
for n in lista:
  print (n)
```

```
hello
dog
cat
food
tree
```

METHODS

There are some important commands, called methods, which are preset functions that you can use when programming in phyton. Here are some that are essential in using dictionaries:

In [ ]:
```
dizionario.keys() #print the list of keys
```

Out[ ]:
```
dict_keys(['ciao', 'cane', 'gatto', 'cibo'])
```

In [ ]:
```
dizionario.values()  #prints the list of values
```

Out[ ]:
```
dict_values(['hello', 'dog', 'cat', 'food'])
```

In [ ]:
```
dizionario.items()  #print key-value pairs, grouped in brackets and separated
by commas.
```

Out[ ]:
```
dict_items([('ciao', 'hello'), ('cane', 'dog'), ('gatto', 'cat'), ('cibo',
'food')])
```

In [ ]:
```
for n in dizionario.values():
    print (n)     #prints the values by themselves, one per line
```

```
hello
dog
cat
food
```

dictionary.get("value_to_search", "error_text") Allows to search for a key and print in case of its absence a chosen error message, all in a single, very concise line of code.

In [ ]:
```
dizionario.get("cane","non trovata")
```

Out[ ]:
```
'dog'
```

In [ ]:
```
dizionario.get("tree","non trovata")
```

Out[ ]:
```
'non trovata'
```

Dictionary.setdefault("index_to_search", "value_to_search") It searches for a precise pair, and if it doesn't find it, it adds it

In [ ]:
```
dizionario.setdefault("albero","tree")
dizionario.items()
```

Out[ ]:
```
dict_items([('ciao', 'hello'), ('cane', 'dog'), ('gatto', 'cat'), ('cibo',
'food'), ('albero', 'tree')])
```

4.3 OPERATORS OF COMPARISON and BOOLEAN LOGIC

Operators that can be used in python are the classical comparison operators:

- **==** : Used to check whether two variables have the same value
- **!=** : Used to check if two variables are different
- **<, <=, >, >=** : Compare two variables.

As for Boolean logic, we again have the classical operators:

- **and** : Operator that returns value "true" only if both conditions it concatenates are.
- **or** : Operator that returns "true" if at least one of the conditions it concatenates is.
- **not** : negates the condition to which it is applied.

In [ ]:
```
5 == 5 and 5 < 3
```

Out[ ]:
```
False
```

In [ ]:
```
5 == 5 and 3 < 5
```

Out[ ]:
```
True
```

In [ ]:

```
c = (5 == 5)
not c
```

Out[ ]:

```
False
```

In [ ]:

```
3 > 4 or 4 < 7
```

Out[ ]:

```
True
```

4.4 FLOW CONTROL: The if - elif - else operator

The syntax for the if-elif-else command in python is as follows:

In [ ]:

```
a = 5
if a > 4:
  print ("greater")
elif a == 4:
  print ("equal")
else:
  print ("less")
```

```
greater
```

4.5 THE CYCLIC STRUCTURES

4.5.1 THE WHILE CYCLE

The syntax used is as follows:

In [ ]:

```
i = 0
while i < 10:
  print (i)
  i += 1  #increment of counter i by 1
```

```
0
1
2
3
4
5
6
7
8
9
```

THE BREAK AND CONTINUE COMMANDS

The break command makes it possible to, once a condition is identified, interrupt a current cycle that would otherwise still be running. Otherwise, the continue command, once the condition is identified, skips the current teraction but continues the current cycle. Here next are two examples:

In [ ]:

```
a = 0
while a < 15:
  a += 1
  print ("ok")
  if a == 10:
    print ("out")
    break;
```

```
ok
ok
ok
ok
ok
ok
ok
ok
ok
ok
out
```

In [ ]:

```
a = 0
while a < 15:
  a += 1
  if a == 5 or a == 10:
    print ("skip")
    continue
  print (a)
```

```
1
2
3
4
skip
6
7
8
9
skip
11
12
13
14
15
```

4.5.2 THE FOR LOOP

The syntax used is as follows:

for number in range (m):

(with m equal to the number before which the loop should stop). With this default syntax, the loop will start at 0 and have step 1

or

for number in range(start, end, step):

This way you specify start and step of your choice

In [ ]:

```
for n in range(11):
  print (n)
```

```
0
1
2
3
4
5
6
7
8
9
10
```

In [ ]:

```
for n in range(2,18,2):
  print (n)
```

```
2
4
6
8
10
12
14
16
```

4.6 HANDLE ERRORS IN PYTHON

During the execution of our programs in the Python language we often come across, especially when we are novices, errors or that category of errors referred to as '*exceptions*', which can be unpleasant as well as result in crashing the program and instantaneously interrupting the execution of the code.
A simple solution to handle errors might be to use the if/elif or else functions, which are called when the user does not correctly execute what is required by the program.

Example:
Let's write a few lines of code that will allow us to do an integer check and identify the various types of errors that can occur using the functions mentioned above:

A 'more structured alternative for handling errors and 'exceptions' might be to use the try and except functions, however to use these functions we need to create a function in the text editor to be called whenever we do not want Python's own error code to appear on the screen.

In [ ]:

```
def controllo_numerico():
  try:
    a = int(input('inserisci un numero: '))
```

```
    print(a)
  except ValueError :
    print('hai inserito un nome')

controllo_numerico()
```

4.7 FILE MANAGEMENT IN PYTHON

Files can be managed by Python in read and write operations (similar to other programming languages) through the use of several functions. The crucial functions in this type of operation are open and close, these use a simple syntax thus allowing files to be easily opened and closed in read and write operations.

Syntax example:

with open(" FILE PATH ",'file open mode ') as filename:
pass
or
filename = open('FILE PATH','file open mode')

You must indicate the file path instead of FILE PATH and the mode of opening the file. The file opening modes are: r,w,x,t,a,b,+ etc...

Which represent respectively:

- r : Read mode
- w : Write mode (the file is overwritten)
- x : File creation mode (if the file already exists, an error is returned)
- a : Open file write mode (without overwriting the file but queuing the contents)
- t : Opens a file in text mode.
- b : Opens a file in binary mode
- +: Opens the file in both read and write mode

filename.close() : Function to close the open file.

To actually write to the file you must use the write and read functions: nomefile.read() namefile.write()

4.8 FUNCTIONS DEFINATION

Functions are a tool that allows us to group a set of instructions that perform a specific task. Functions accept arguments (or parameters) as input, process them, and output a result. After defining a function, it is possible to execute it by simply calling it and providing it with the necessary parameters as input, specific to different situations. This method allows functions to be called only where necessary, making the code more orderly.

Function syntax:
def function_name(var1,var2,var3....):

- function is introduced by the def keyword;
- after def the function name is to be defined, in this case function_name;
- after the function name the list of parameters passed to the function is specified in round brackets;

- after the list of parameters there are the colon (:) which introduces an indented block of code (everything that is indented belongs to the function to exit the function it is necessary to remove the indentation);
- the code block can contain several instructions and returns.

To call a function you must write the name of the function (function_name in this case) and put in parentheses the global variables to be passed to the function (N.b. the number of global variables must be equal to the number of local variables defined in the function).

Example code:

In [ ]:
```
def termometro(T):
  if T >= 30:
    print('Fa molto caldo!!!')
  elif T<30 and T>15:
    print('C\'è una temperatura media')
  else:
    print('Fa freddo amico!')

t=int(input('Inserisci la temperatura in gradi Celsius: '))
termometro(t)
```

```
Inserisci la temperatura in gradi Celsius 6
Fa freddo amico!
```

We created a thermometer function in which based on the temperature value entered by the user tells us what the weather conditions are like, based on a temperature range we set.

Similar to other programming languages, Python provides a number of functions already in its library installed with the initial language installation, which can be used simply by calling them.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import fileinput
```

DIFFERENCE BETWEEN GLOBAL AND LOCAL VARIABLES

Variables can be defined in two different environments: the global environment, which belongs to the entire program, or the local environment, which belongs to a single function. While global variables are visible from any function in the program, local variables can only be seen and used within the function in which they are defined. However, there are also limitations regarding the use of global variables in the local environment. In fact, it is necessary, if you want to perform operations on global variables such as changing their value from within a function, to call them further with the "global" command, in order to highlight their origin. On the other hand, it is always possible to read and copy their contents without this precaution, so you can copy their value into a local variable and handle the latter freely, as long as you are inside the function.

In [ ]:
```
a = 15
```

```
def funzione():
  global a
  a += 2
  return (a)

print (funzione())
```

```
17
```

In [ ]:
```
a = 15

def funzione():
  b = a
  b += 2
  return (b)

print (funzione())
```

```
17
```

If you want to use a local variable in the global environment instead, the only possible way is to have the function return the desired value to the main program.

In [ ]:
```
def funzione():
  a = 15
  return (a)

c = funzione() + 2
print (c)
```

```
17
```

INSTALL MODULES IN PYTHON

A key aspect of python is the ability to install third-party modules. They are uploaded to "python package index". Usually these modules are installed from the terminal (via windows cmd). The modules are then installed via the script called pip (found in the python installation package). You must then move to the default python folder (on windows it is "C:\Users\NAME_UTENTE\AppData\Local\Programs\Python\Python310\Scripts") via the cd command. It is then possible to install the desired modules by typing "`pip install NOME_MODULE`".

4.9 GRAPHICS IN PYTHON

MATPLOTLIB.PYPLOT

We will now discuss the MATPLOT module and in particular the pyplot subpackage, which is very useful for graphing. We start by installing the module with

```
pip install matplotlib
```

(N.B. in colab cells you will not need to install the module first, just import it when needed)

Now then before we start using it we need to import the module into the shell, via the command

```
import matplotlib.pyplot as plt
```

(in this case in addition to importing the module and its respective submodule I also gave it a name, plt, so that it would be easier to call, but this is not a strictly necessary step)

Another very important module is numpy, which is very useful for performing many mathematical functions (this command will often be used during this tutorial)

So let's look at the main commands for displaying a graph

BASIC COMMANDS

- **`plt.plot()`**

  the plot function expects in imput at least one list of numeric values which it will read as ordinates of points (and as abscissae it will automatically take numbers from zero to N-1). Otherwise, it is possible to indicate 2 lists of numbers where the first indicates the abscissae and the second the ordinates. It is possible to represent several graphs in the same plot by repeating the command several times. Then there are subcommands of plt.plot:

  - `plt.plot(x,y,'ro--')`: As the third argument of the function, it is possible to give directions for customizing the lines of the plot: by doing so, it is possible to change the color of the line, the style (whether continuous, dashed, etc.) and whether to indicate the points and with what symbol. (See appendix for all the possibilities.) *`plt.plot(x,y,lw=N)`: allows you to indicate the size of the line with a value ranging from 0 (line absent) to potentially infinite
  - `plt.plot(x,y,label='name')`:allows you to give a name to the line, a name that will then be given in the legend (see command `plt.legend()`)

- **`plt.show()`**

  Should be given after a plot to actually show the graph

- **`plt.xlabel() / plt.ylable()`**

  Receives in imput a text that will be the names of the abscissae / ordinates respectively

- **`plt.title()`**

  Gets in imput a text that will be the title of the graph

- **`plt.grid()`**

  To set the grid in the graph.

- **`plt.axis([x_min,x_max,y_min,y_max])`**

  allows the extremes of the axis of the graph to be changed.

- **`plt.xticks / plt.yticks`**

  allows you to change the values given in the graphs, receive as input a list of values (N.B. can be used as another way to define axis extremes)

- **`plt.legend()`**

  allows you to insert a legend into the graph, and with the loc subcommand you can select where to insert it (very useful if you want to represent multiple graphs in the same plot)

- **`plt.figure()`**

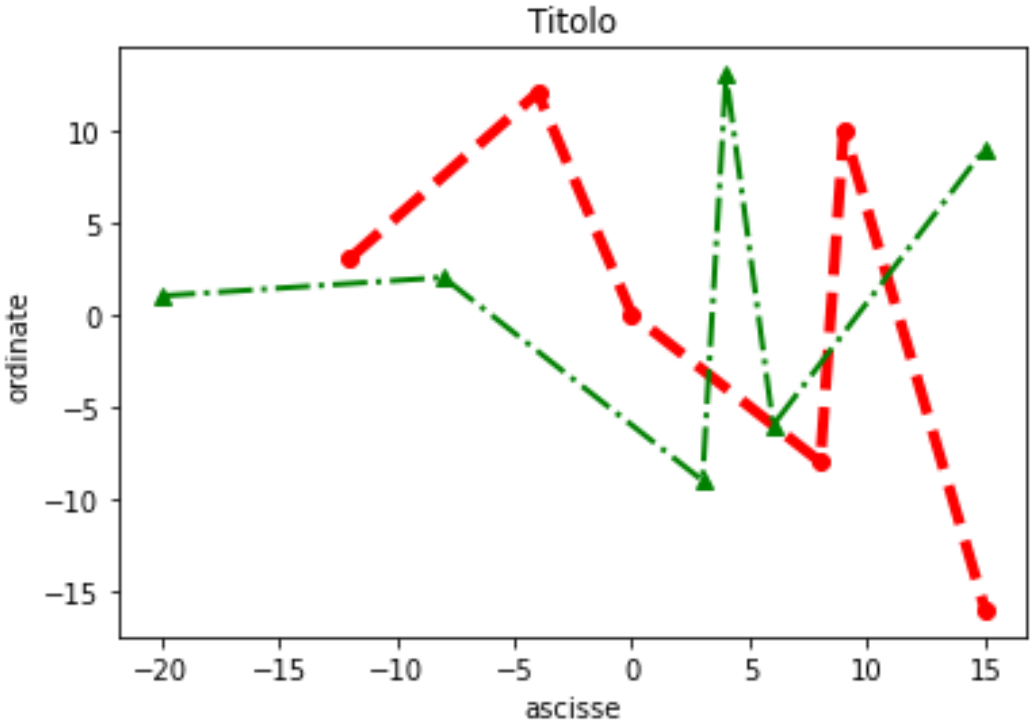
useful for indicating some properties of the figure itself, such as the size with the figsize(width,height) subcommand

- **`plt.subplot(rows, columns, position in grid)**

  Divides the image into multiple plots that will go to define a grid, the program receives as input the number of rows into which to divide the space, the number of columns, and the position I am going to work on at that moment (going to number from 1 in ascending order starting from top left). After that after each plt.subplot(n, n, n) a plt.plot will have to be inserted to indicate the graph that is to be represented in that area

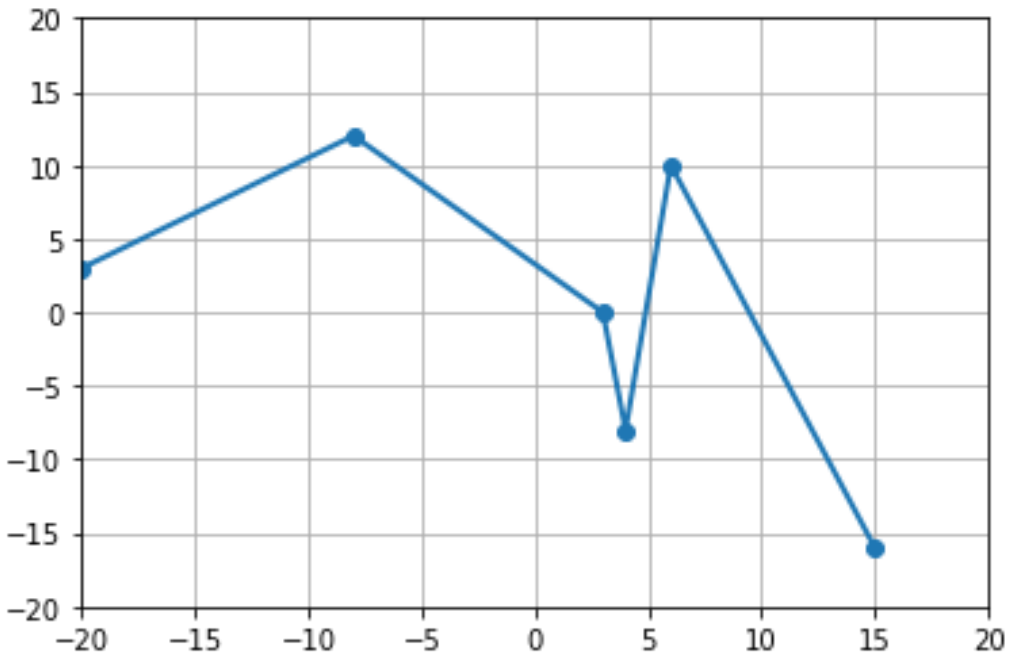
In [ ]:

```
#@title EXAMPLE 1 (graph properties, labels and title).
import matplotlib.pyplot as plt
x1=[-12,-4,0,8,9,15] #define two random lists that will be
y1=[3,12,0,-8,10,-16] #the abscissae and the ordinates
x2=[-20,-8,3,4,6,15]
y2=[1,2,-9,13,-6,9]
plt.plot(x1,y1,'ro--',lw=4) #define the first plot with respective properties
plt.plot(x2,y2,'g^-.',lw=2) #define the second graph with respective properties
plt.xlabel("abscissae") #define the name of the abscissae
plt.ylabel("ordinates") #define the name of the ordinates
plt.title("Title") #define the general title of the plot
plt.show()
```



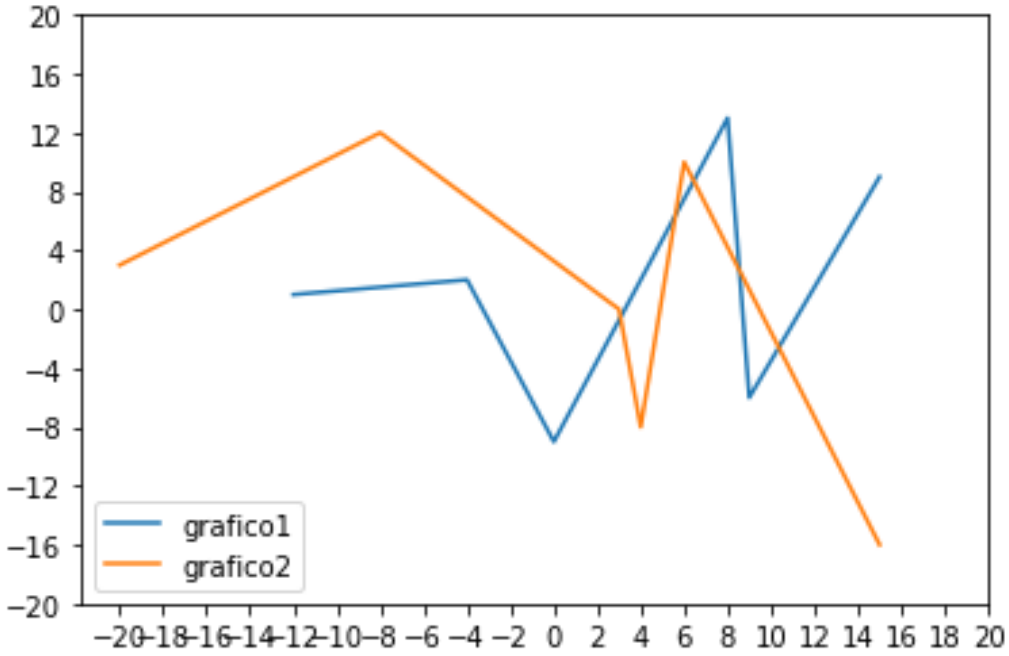
In [ ]:

```
#@title EXAMPLE 2 (grid and axis dimensions).
import matplotlib.pyplot as plt
#I don't define lists again,for simplicity I use the ones from the previous
cell
plt.plot(x2,y1,'o-',lw=2) #define the first plot
plt.grid() #insert the grid
plt.axis([-20,20,-20,20]) #define the size of the axes
plt.show()
```



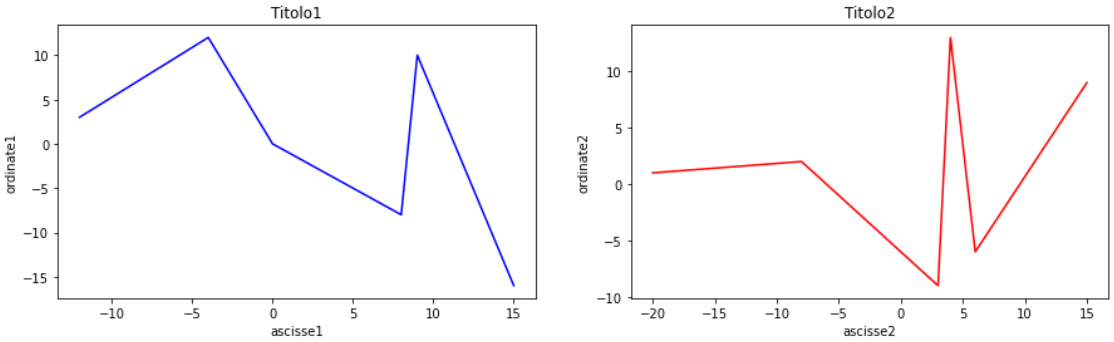
In [ ]:

```
#@title EXAMPLE 3 (graph labels, legend, axis indexes).
import matplotlib.pyplot as plt
# I don't define lists again, for simplicity I use the ones in the first cell
plt.plot(x1,y2,label='plot1') # defines the first plot with the respective
                              # properties
plt.plot(x2,y1,label='graph2') # define the second graph with respective
                               # properties
plt.legend(loc='lower left') # inserts the legend and defines its position
plt.xticks([2*k for k in range(-10,11)]) # defines the 'ticks' on the x-axis
and
                                         # how often to put them
plt.yticks([4*k for k in range(-5,6)])   # defines the "ticks" on the y-axis
and
                                         # how often to put them
plt.show()
```



In [ ]:

```
#@title EXAMPLE 4 (subplot).
import matplotlib.pyplot as plt
plt.figure(figsize=(15,4))
plt.subplot(1,2,1) # create a subplot with 1 row, 2 columns and place in area 1
plt.plot(x1,y1,'b') # define the first plot
plt.xlabel("abscissa1") # define the name of the abscissa of the first plot
plt.ylabel("ordinates1") # define the name of the ordinates of the first plot
plt.title("Title1") # define the title of the first plot
plt.subplot(1,2,2) # recall the subplot created and place myself in zone 2
plt.plot(x2,y2,'r') # define the second plot
plt.xlabel('abscissa2') # define the name of the abscissae of the second plot
plt.ylabel("ordinates2") # define the name of the ordinates of the second plot
plt.title("Title2") #define the title of the second plot
plt.show()
```



OTHER TYPES OF GRAPHS

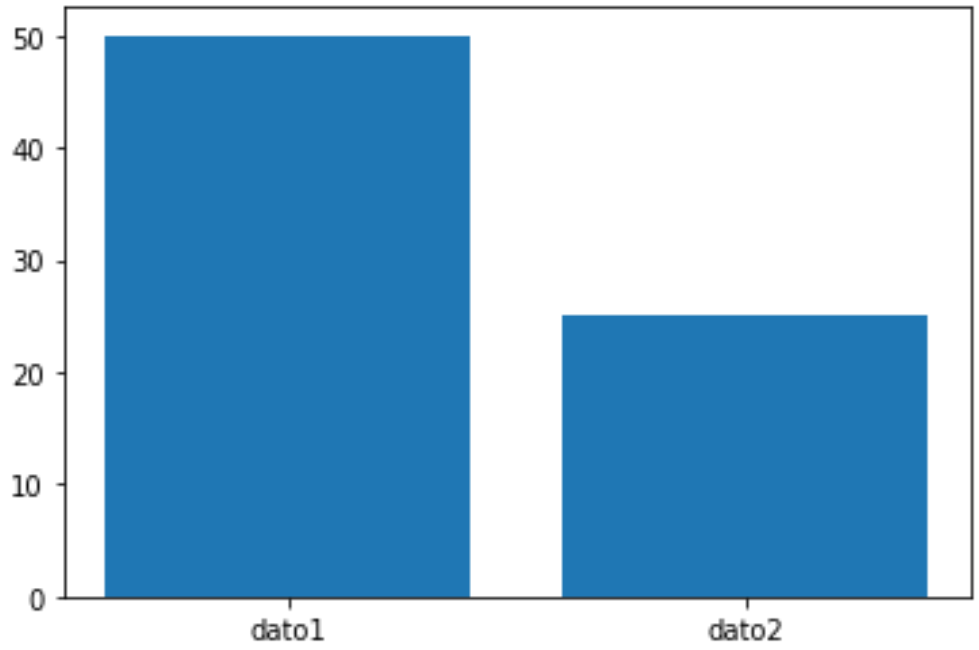
- **BAR GRAPH**

  `plt.bar()`

  Allows you to create bar graphs, as always it receives in input two lists, with the special feature that the first one may be a list of strings and not perforza of numbers

In [ ]:

```
#@title GRAPHIC BAR EXAMPLE.
a=["given1", "given2"]
b=[50,25]
plt.bar(a,b)
plt.show()
```



- **DIPERSION GRAPH**

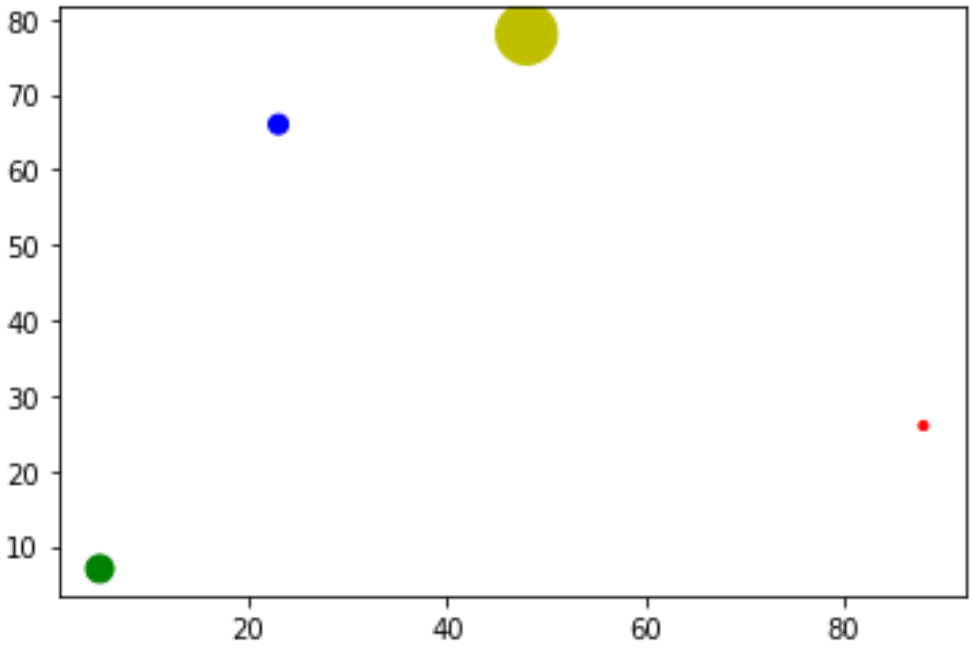
  `plt.scatter()`

  Graph that rasppresents one quantity relative to another.

  Data, in addition to the classic way with numerical lists, can be defined by a dictionary. For each pair of values on the graph, a point is represented. The values in the dictionary are usually numeric lists. The lists are passed to the function by their key. The command data=dictionary_name is used to indicate the reference dictionary. In addition, a whole series of parameters such as color (c=[...]) or size (s=[...]) can be defined on a point-by-point basis. (Lists of colors or sizes can also be defined via the dictionary.)

In [ ]:

```
#@title DISPERSION GRAPH EXAMPLE.
dictionary={'a' :[88,23,5,48],
            'b':[26,66,7,78],
            'colors':['r','b','g','y'],
            'dimensioni':[10,50,100,500]}
plt.scatter('a','b',c='colors',s='size',data=dictionary)
plt.show()
```



USE OF PARAMETERS TO INTERACTIVELY MODIFY THE PLOT

It is possible to set the variables represented on the plot as parameters that can be modified in order to visulize, in a more immediate way, how the function transforms as the parameters change.

To define a parameter, you can use the syntax: p = value_chosen #@param

It is also possible to choose the type of parameter to be set

1. parameter type **'slider'**: shows a bar characterized by a maximum and a minimum value. The parameter can take any value between the extremes (minimum reslot defined by the step)
2. parameter of type **'number'**: shows the value initially set but modifiable by the user
3. parameter of type **'integer'**: the parameter can take only integer values

In [ ]:

```
#@title Tipologie di parametri
# parametro slider
s = 0.05 #@param {type:"slider", min:-2, max:2, step:0.05}

# paramtro number
k = 4.5 #@param {type:"number"}

# parametro integer
n = 3 #@param {type:"integer"}
```

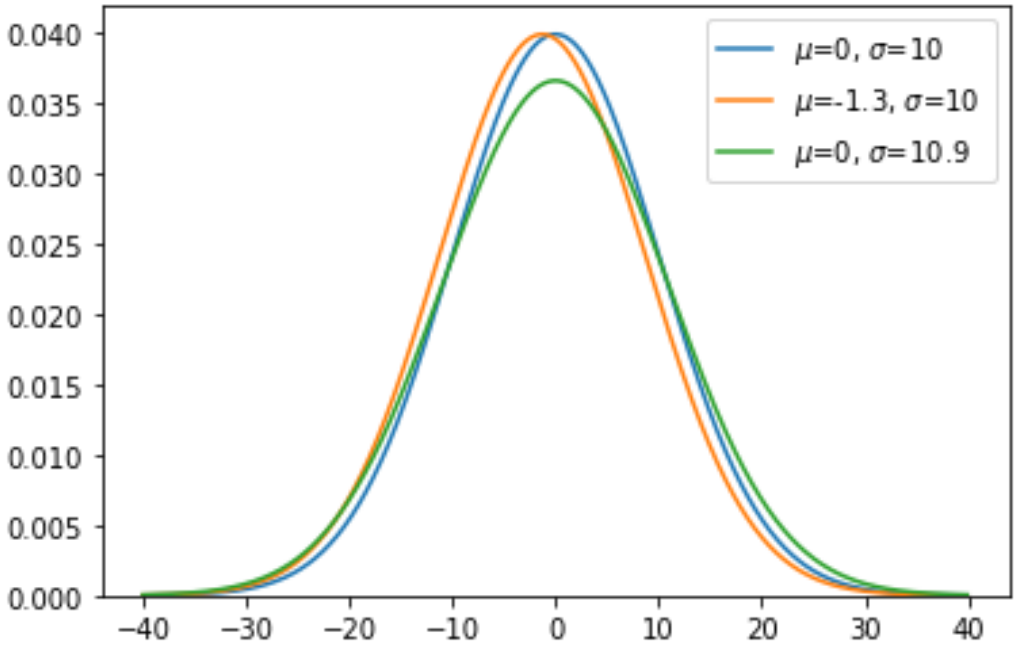
REPRESENTATION OF A GAUSSIAN DISTRIBUTION

It is possible to apply what we have seen about setting variable parameters to evaluate the performance of a function.

For the Gaussian distribution in particular, there is, within the scipy.stats package, the stats module that contains the function **stats.norm.pdf()** to which it is sufficient to pass as parameters (x, μ, σ), where x are the points whose graph you want to plot, to obtain the distribution.

In [ ]:
```
#@title Gaussian distribution as μ and σ change: varyinig mu and sigma

import scipy.stats as stats # module needed to plot Gaussian distrubution
x = np.arange(-40, 40, 0.1)
mu = -1.3 #@param {type: "slider", min:-5, max:5, step:0.1}
sigma = 10.9 #@param {type: "slider", min:5, max:15, step:0.1}

plt.plot(x, stats.norm.pdf(x, 0, 10), label="$\mu$=0, $\sigma$=10")
plt.plot(x, stats.norm.pdf(x, mu, 10), label="$\mu$={0}, $\sigma$=10
".format(mu))
plt.plot(x, stats.norm.pdf(x, 0, sigma), label="$\mu$=0,
$\sigma$={0}".format(sigma))
plt.legend()
plt.ylim(bottom=0)
```

Out[ ]:
```
(0.0, 0.04188852835306775)
```



4.10 REALIZATION OF 3D MODELS

To make three-dimensional graphs in Python you can take advantage of the modules already seen for 2D representation (**matplotlib.pyplot** and **numpy**) or, alternatively also import the **mplot3d** module

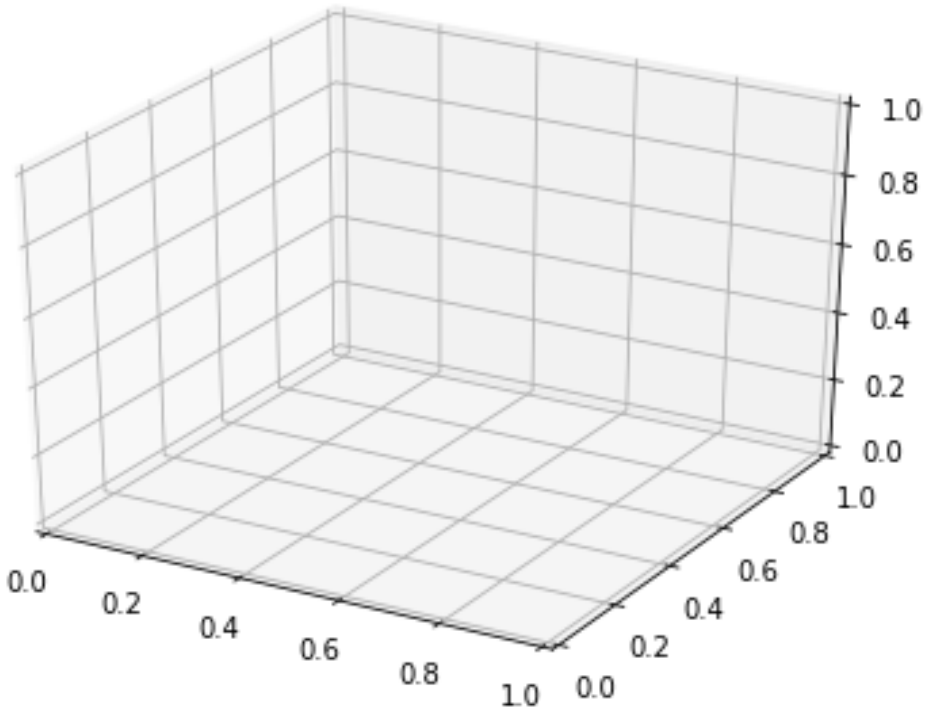
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```
from mpl_toolkits import mplot3d
from matplotlib import cm           # Imported module to make some color maps
```

The first step is to create the figure, you can set its desired dimensions by passing it as a parameter to the **plt.figure()** command. After that to go from 2D to 3D you need to add the z-axis using one of Python's projections which is the 3D one

In [ ]:

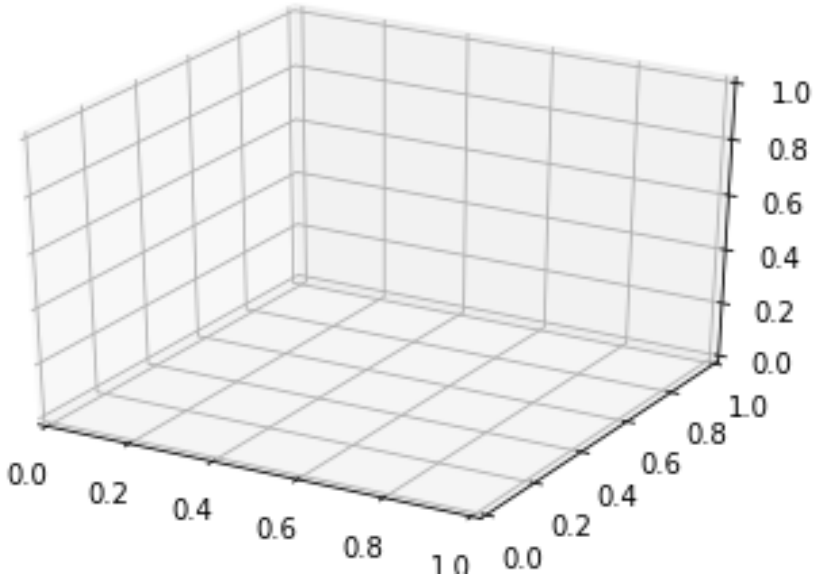
```
fig = plt.figure(figsize=(7,5)) # for example 7 (horizontal) and 5 (vertical)
                                # are the dimensions set for the figure
ax = fig.add_subplot(111, projection='3d') # you use the add_subplot command
and
                                # pass projection='3d' as the parameter
                                # the first parameter sets the length of the
axes
```



Alternatively, it is also possible to use only a single command

In [ ]:

```
ax = plt.axes(projection='3d')
```

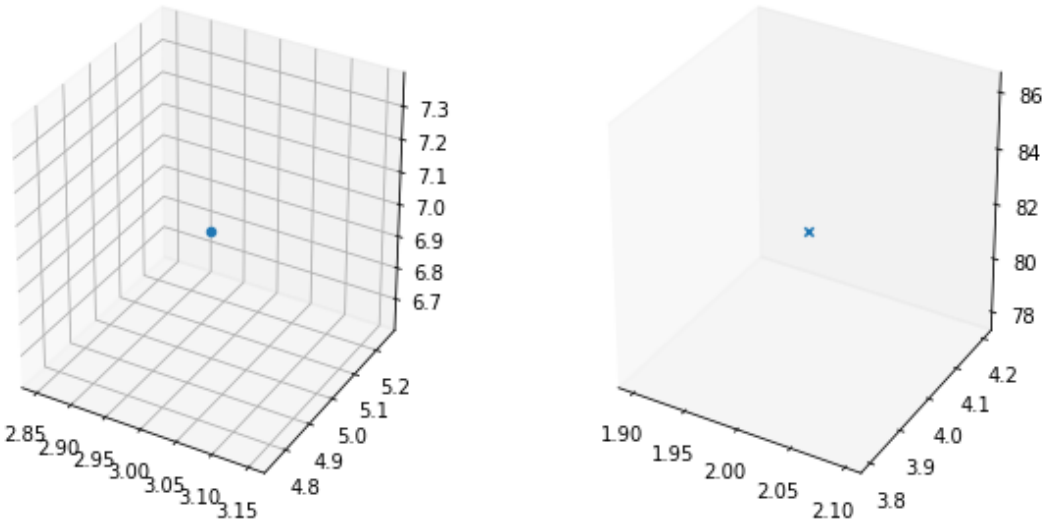


WARNING: you use the ax. object to add any other plots to the figure

With the **scatter()** command, any point can be displayed on the graph

In [ ]:

```
#REPRESENTATION OF A POINT
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.scatter(3,5,7) # you pass as parameters the three coordinates that locate
the
                  # point
ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.scatter(2,4,82, marker='x')
ax.grid(False) # allows the grid to be removed from the plot
```



To make **SCATTER** graphs, it is sufficient to pass vectors as parameters to the **scatter()** function, instead of the coordinates of a point synoglo

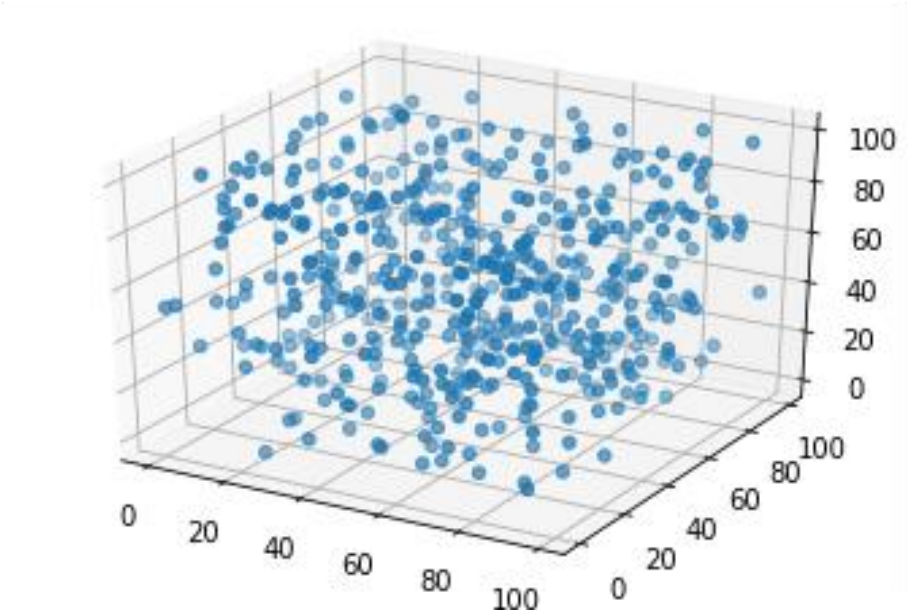
In [ ]:

```
# SCATTER PLOT
x = np.random.randint(0,100,500)
```

```
y = np.random.randint(0,100,500)
z = np.random.randint(0,100,500)
ax = plt.axes(projection='3d')
ax.scatter(x,y,z) # pass as parameters of vectors
```

Out[ ]:

```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7ff953bb1c50>
```



To make a **CONTINUOUS LINE** graph, which passes through a set of data points, the **plot()** function is used. This also makes it possible to represent the trend of functions in space

In [ ]:

```
# CONTINUOUS LINE GRAPH
ax = plt.axes(projection='3d')
x_data = np.linspace(-4*np.pi,4*np.pi,50) # generate a vector of values
                                          # (see appendix for details)
y_data = np.linspace(-4*np.pi,4*np.pi,50)
z = x_data**2 + y_data**2 # generates the z function to represent
ax.plot(x_data,y_data,z) # allows the function to be displayed by passing the
                         # two coordinate vectors as parameters
```

Out[ ]:

```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x7ff952351b90>]
```

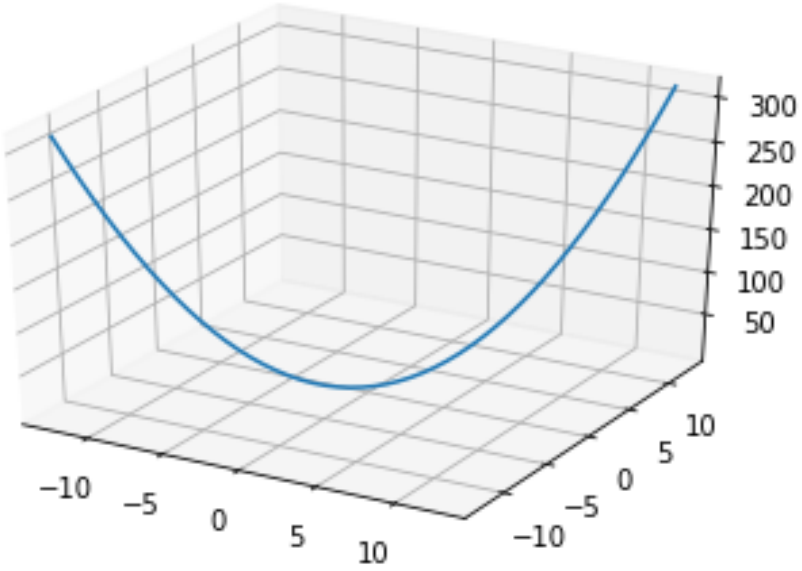


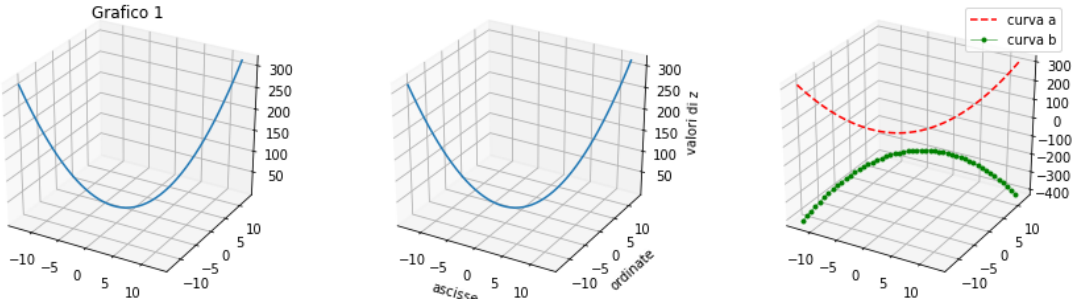
CHART CUSTOMIZATION

It is possible to customize the plot by changing a number of parameters and adding title, labels and legends. With the **add_subplot()** command, multiple graphs can be inserted into the same plot

In [ ]:

```
fig = plt.figure(figsize=(15,4))
x_data = np.linspace(-4*np.pi,4*np.pi,50)
y_data = np.linspace(-4*np.pi,4*np.pi,50)
z1 = x_data**2 + y_data**2
z2 = - x_data**2 - y_data**2 -100

# PERSONALIZING GRAPHS # add_subplot allows us to make multiple graphs, as if
# they were placed in a grid whose dimensions we can set
ax = fig.add_subplot(1, 3, 1, projection='3d') # we pass as parameters
                     # (n° rows, n° columns, position in which to put the
graph)
ax.plot(x_data,y_data,z1)
ax.set_title('Graph 1') # allow to insert a title

ax = fig.add_subplot(1, 3, 2, projection='3d')
ax.plot(x_data,y_data,z1)
ax.set_xlabel('abscissa') # allows axes to be labeled
ax.set_ylabel('ordinates')
ax.set_zlabel('values of z')

ax = fig.add_subplot(1, 3, 3, projection='3d')
ax.plot(x_data,y_data,z1, 'r--', label='curve a')
ax.plot(x_data,y_data,z2, 'g', marker='.', lw=0.5, label='curve b')
# allows to customize color, stroke and thickness of curve

ax.legend() # allows you to enter the legend, but you need to enter in the
fig.tight_layout # previous plot the label that identifies the curve
#for tight_layout see appendix # label='curve name'
```

Out[ ]:
```
<bound method Figure.tight_layout of <Figure size 1080x288 with 3 Axes>>
```

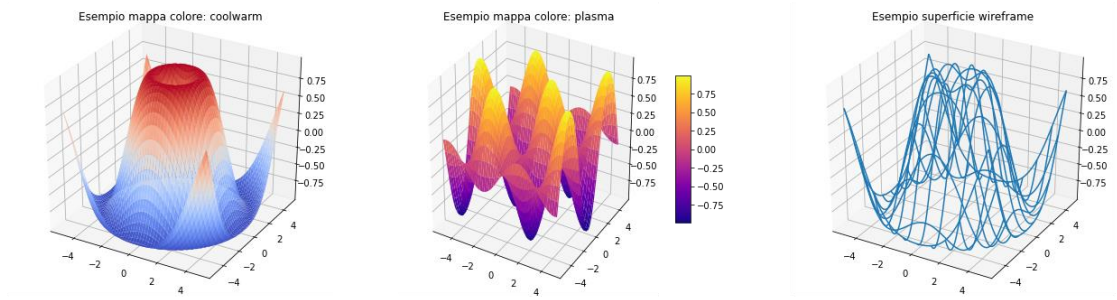


REPRESENT FUNCTIONS AS SURFACES

In Python it is also possible to represent surfaces graphically, through the function **plot_surface()** and customize them through color maps and the inclusion of legends

In [ ]:
```
# SURFACE GRAPH
fig = plt.figure(figsize=(22,6))

X = np.arange(-5,5,0.1)
Y = np.arange(-5,5,0.1)
X, Y = np.meshgrid(X, Y) # meshgrid() command needed to convert the two vectors
                         # X and Y into a coordinate grid
                         # if not using python may return an error message
R = np.sqrt(X**2 + Y**2)
Z1 = np.sin(R)
Z2 = np.sin(X) * np.cos(Y)

ax = fig.add_subplot(1,3,1,projection='3d')
ax.plot_surface(X,Y,Z1,cmap=cm.coolwarm,) # with cmap= you can set a color
mache
                                          # within the plot_surface
ax.set_title('Example color map: coolwarm')

ax = fig.add_subplot(1,3,2,projection='3d')
surf = ax.plot_surface(X,Y,Z2,cmap='plasma')
ax.set_title('Example color map: plasma')
fig.colorbar(surf, shrink=0.5, aspect=10) # with colorbar() you can insert a
                                         # color bar, you need to pass it as
the
              # first parameter the name of the variable with which the surface
              # plot was saved aspect represents the thickness of the bar
              # shrink its height
ax= fig.add_subplot(1,3,3, projection='3d')
ax.plot_wireframe(X, Y, Z1, rstride=10, cstride=10) #plot_wireframe() allows
you
                                  # to get a "wire" representation of the surface
ax.set_title('Example wireframe surface')
```

Out[ ]:
```
Text(0.5, 0.92, 'Esempio superficie wireframe')
```



REPRESENTATION OF A GAUSSIAN DISTRUBUTION IN SPACE

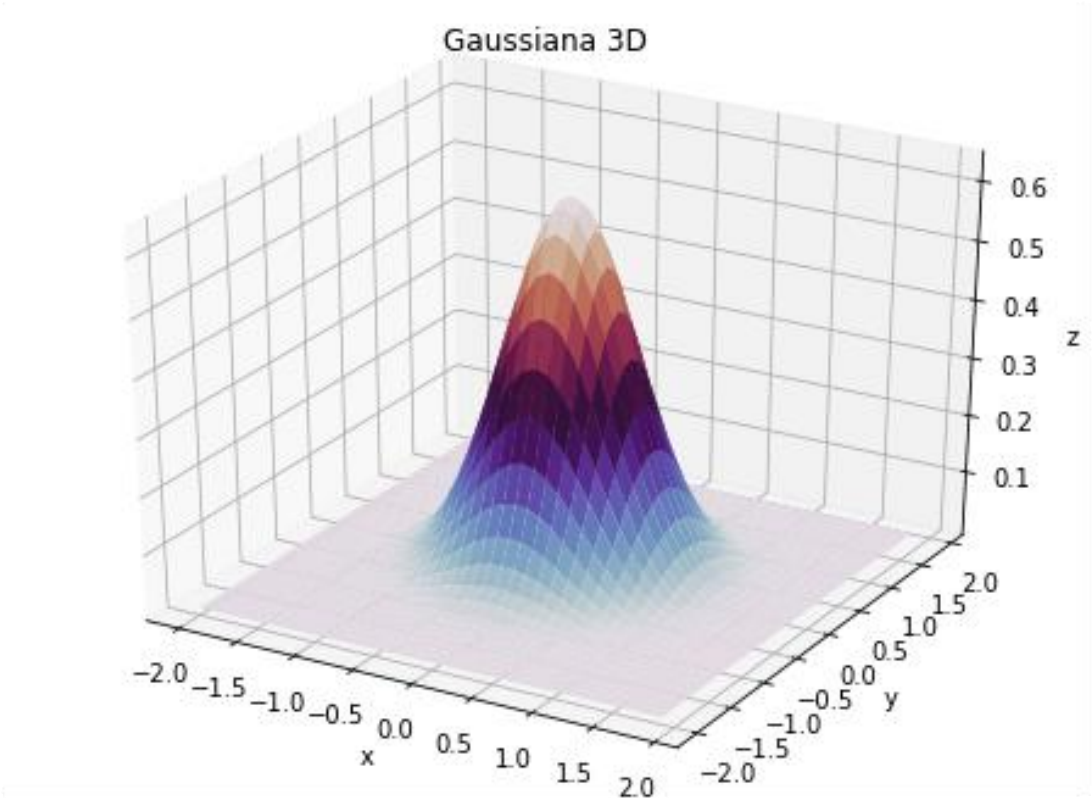
It is also possible to represent a Gaussian distribution in a 3D graph. One can use, for example, the function **multivariate_normal.pdf()**which requires as parameters: (vertical array containing (x,y), mean=μ, cov=covariance), of which μ and covariance are obviously associated with the Gaussian we want to plot.

In [ ]:

```
from scipy.stats import multivariate_normal
x = np.arange(-2, 2, 0.1)
y = np.arange(-2, 2, 0.1)
x, y = np.meshgrid(x, y)
xy = np.column_stack([x.flat, y.flat]) # need to place the generated x and y
                              # coordinates in a vertical array there pairs (x,y)

mu = np.array([0.0, 0.0]) # define the value of μ

sigma = np.array([.5, .5]) # defines the value of σ
covariance = np.diag(sigma**2) # calculates the covariance

z = multivariate_normal.pdf(xy, mean=mu, cov=covariance)
z = z.reshape(x.shape) # needed to relocate z into a matrix

fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(projection='3d')
ax.plot_surface(x,y,z, cmap='twilight')
ax.set_title('3D Gaussian')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("z")
```

Out[ ]:

```
Text(0.5, 0, 'z')
```



VISUALIZZAZIONE DELLE PROTEINE IN PYTHON (MODELLI 3D)

Sfruttando quanto visto finora, possiamo rivivere un modello tridimensionale di una proteina, partendo da un file .pdb come quelli che abbiamo analizzato nel capitolo precedente del tutorial, quello relativo al linguaggio bash.

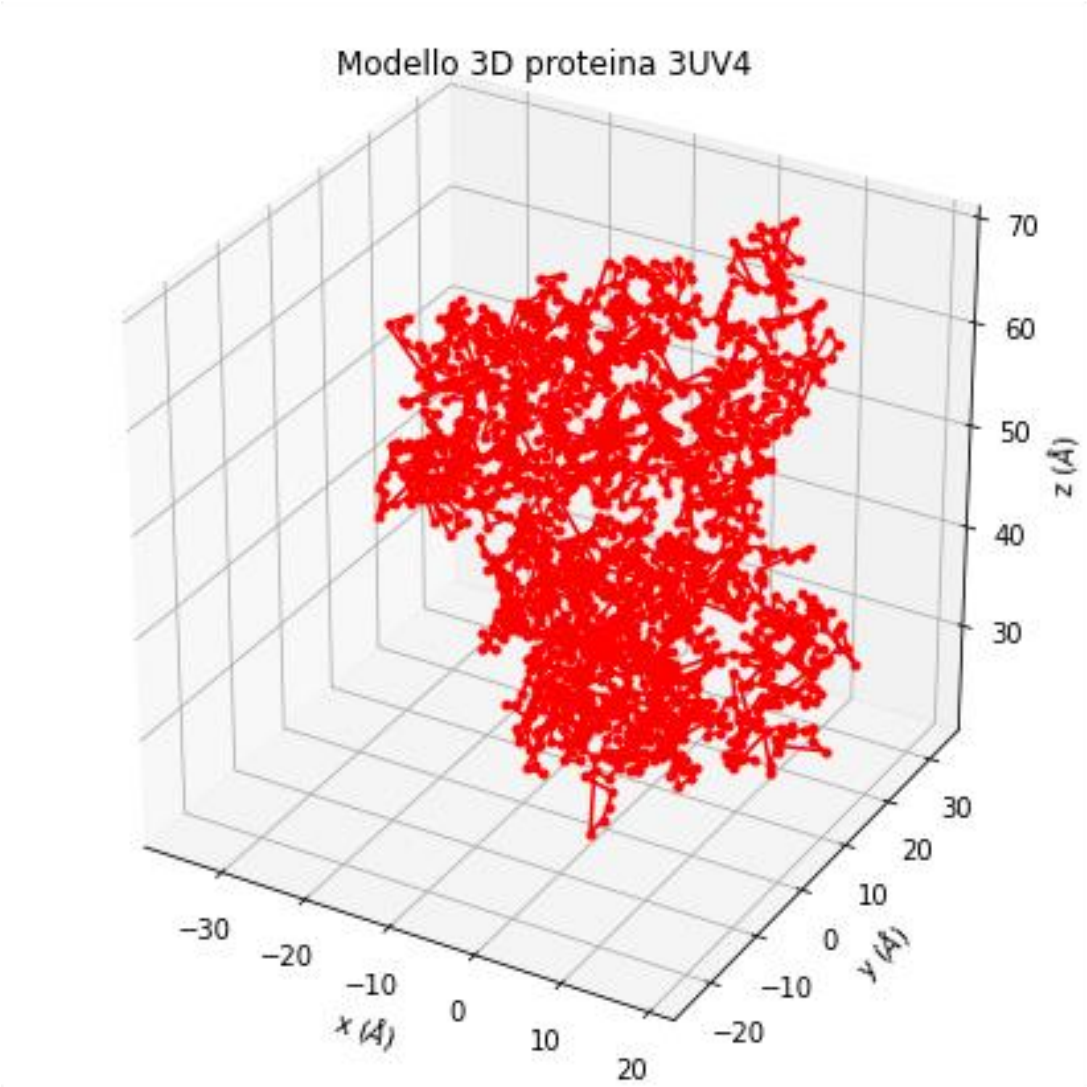
Per farlo, dobbiamo importare il file .pdb della proteina che vogliamo rappresentare (!wget) e acquisirne i dati leggendoli riga per riga attraverso un ciclo for e salvando le coordinate di ogni punto

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
import fileinput

! wget https://files.rcsb.org/download/3UV4.pdb
coords = list() # creates a list called coords within which we will enter the
                # coordinates of each point

#acquisition of data
for line in fileinput.input('3UV4.pdb'):
  if line.startswith('ATOM'):
    coords.append([float(line[26:38]), float(line[38:46]), float(line[46:54])))
    # allows the coordinates of each atom to be saved in the coords file
    # with append the coordinates of a new
    # atom with each iteration of the file
fileinput.close()

# graph representation
coords=np.vstack(coords) # the vstack command allows data to be transferred
```

```
into
                          # a vertical array
fig = plt.figure(figsize=(6,6)) #allows you to create a figure variable in
which
                               #to insert the graph and set the size of the
graph
ax = fig.add_subplot(projection='3d') #the add_subplot function allows you to
                                     #add a third axis as a subplot --> 3d ratio
ax.plot(coords[:,0], coords[:,1], coords[:,2],'r', marker='.')
# generate the plot. you pass as parameters the coordnates of each point

ax.set_title('3D protein model 3UV4') # allows to isert the title
ax.set_xlabel('x ($\AA$)')
# allows you to name the axes (x y z) and enter the unit of measurement of
# those axes (amstrong)
ax.set_ylabel('y ($\AA$)')
ax.set_zlabel('z ($\AA$)')
fig.tight_layout()
```

```
--2022-05-28 18:16:09--  https://files.rcsb.org/download/3UV4.pdb
Resolving files.rcsb.org (files.rcsb.org)... 128.6.158.70
Connecting to files.rcsb.org (files.rcsb.org)|128.6.158.70|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/octet-stream]
Saving to: ‘3UV4.pdb’

3UV4.pdb                [ <=>                ] 396.14K  --.-KB/s    in 0.04s

2022-05-28 18:16:09 (9.16 MB/s) - ‘3UV4.pdb’ saved [405648]
```



It is also possible to visualize proteins in other ways, such as by installing the **py3Dmol** module and downloading the .pdb file of the protein you want to represent with the following code:

In [ ]:

```
!pip install py3Dmol
import py3Dmol
! wget https://files.rcsb.org/download/3UV4.pdb
```

In [ ]:

```
view=py3Dmol.view() # allows the function to be called more quickly by just
                    # typing view()
view.addModel(open('3UV4.pdb', 'r').read(),'pdb') # addModel allows you to read
           # the pdb file of the two polypeptide chains that make up the
protein
view.setBackgroundColor('white') # allows to set the background color
view.zoomTo() # allows you to view all the structures
view.setStyle({'chain':'A'},{'cartoon': {'color':'green'}})
#the setStyle command allows you to customize the representation. It is
possible
# to select the colre of the represented element and the type of view
view.setStyle({'chain':'B'},{'cartoon': {'color':'orange'}})
# select the colre of the represented element and the type of view
```

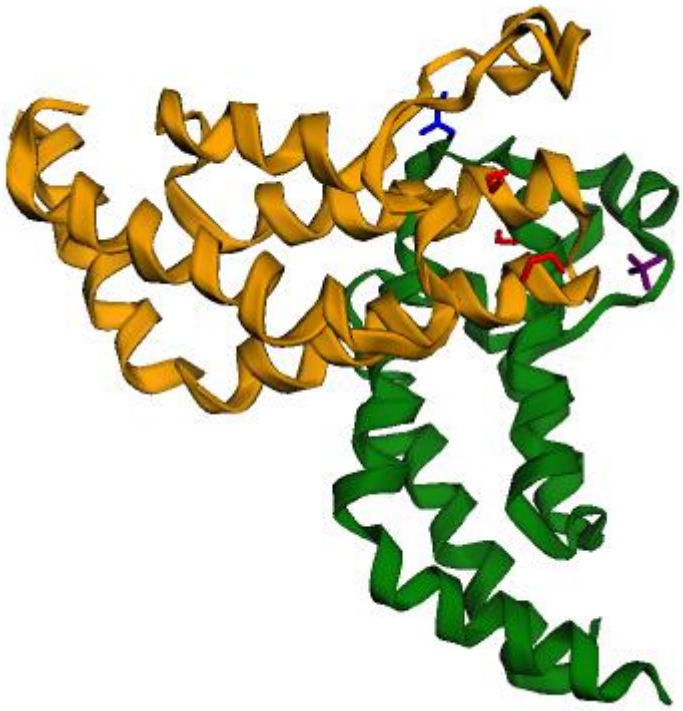
```
view.setStyle({'resn':'PO4'},{'stick': {'color':'purple'}})
view.setStyle({'resn':'GOL'},{'stick': {'color':'blue'}})
view.setStyle({'resn':'EDO'},{'stick': {'color':'red'}})

# the protein represented is composed of two main chains, chain A (green)
# and chain B (orange).
# and by three ligands:
# phosphate group (PO4), purple
# glycerol (GOL), blue
# 1,2-ethanediol (EDO), red
```

You appear to be running in JupyterLab (or JavaScript failed to load for some other reason). You need to install the 3dmol extension:

```
jupyter labextension install jupyterlab_3dmol
```

Out[ ]:

```
<py3Dmol.view at 0x7efc79dae790>
```



WARNING: In order to properly set the colors associated with each component of the protein for easier visualization, it is necessary to know the structure of the protein itself, how many chains it is composed of and which ligands. You can get this information, for example, from the site from which you download the .pdb file of the protein.

4.11 APPENDIX

LINE CUSTOMIZATION VIA PLT.PLOT.

As mentioned, it is possible to customize all the specifications of a line in a plot. You can use a more compact mode in which you indicate all the specifications in the third parameter, stating all attached the color of interest, style and type of marker for each point, for example with `plt.plot(x,y,'ro--')`.

Otherwise, a somewhat longer, but more easily interpreted script is possible, in which we indicate each feature by its parameter, thus with color='color', marker='symbol', linestyle='style'(the only difference is that in this case the color name will have to be written in full and not indicated by analetter). For example to have the same characteristics as in the previous example I will have to write:
`plt.plot(x,y,color='red',marker='o',linestyle='--')`

POSSIBLE COLORS

- 'b'=blue (default)
- 'r'=red
- 'g'=green
- 'c'=cyan
- 'y'=yellow
- 'w'=white
- 'm'=magenta

(N.B in the extended mode there is a possibility to select more colors, like black, blown, orange ec..)

POSSIBLE STYLES

- '-' = continuous line (default)
- '--' = dotted line
- '-.' = alternating dashed dotted line
- ':' = dotted line

POSSIBLE MARKERS (default no markers)

OTHER USEFUL MODULES

- numpy

During the creation of graphs, as a rule, a very useful module is NUMPY. It brings together a large collection of high-level mathematical functions for efficiently operating on the data that will later be used for graphs. Some examples of its use are to generate vectors and more simply to apply functions such as trigonometric

METHODS FOR GENERATING VECTORS:

In [ ]:

```
import numpy as np
x=np.linspace(-4,4,50) # inserts the 50 equidistant values between -4 and 4
y=np.arange(0,50,0.5) # inserts the values from 0 to 50 with step of 0.5
z=np.random.randint(0,100,50)#inserts 50 random integers taken between 0 and
100
```

USO DELLE FUNZIONI TRIGONOMETRICHE O DI QUELLA ESPONENZIALE

In [ ]:

```
import numpy as np
x = np.arange(-2*np.pi,2*np.pi,np.pi/20)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.exp(x)
```

- google.colab.files

  It is a submodule of Google.colab that can be useful for uploading files that will later be used

In [ ]:

```
from google.colab import files #it's a different way of importing a module than
# we have seen so far but you can always use the classic wording
# import google.colab.files (followed by an "as name" if you want to rename it)
uploaded = files.upload() # run the function to upload a file
```

MOST USEFUL COMMANDS FOR MAKING 3D MODELS:

- **plt.figure( )**

  allows you to create the figure in which to insert the graph

- **plot( )**

  allows you to represent the graph as a continuous line

- **plot_surface( )**

  allows you to represent a surface

- **scatter( )**

  allows individual points to be represented on the graph.

- **add_subplot( )**

  allows you to represent multiple graphs in the same plot and is needed to set up 3D projection

- **set( )**

  allows you to insert a number of options into the plot: set_xlabel(), set_title( ).

- **legend( )**

  allows you to enter the legend of the graph.

- **colorbar( )**

  allows you to enter a color bar describing a surface graph.

- **grid( )**

  in the grid(False) option allows you to remove the grid, set by default, from the graph

- **meshgrid( )**

  allows you to convert coordinate vectors to coordinate matrices.

- **tight_layout( )**

  allows automatic adjustment of subplot parameters to be compatible with 'figure' dimensions

4.12 EXERCISES

EXERCISE 1

Draw 10 cards from a deck of 52 (without wild cards). Place them in an appropriate list and tap them on the screen sorted by value.

In [ ]:

```
#@title solution ex1
import random
semi=["cuori", "fiori","quadri","picche"]
valori=["asso","2","3","4","5","6","7","8","9","10","fante","donna","re"]
i=0
estrazioni=[0,0,0,0,0,0,0,0,0,0]     #initialize empty vector to fill
while i<10:
  seme=random.choice(semi)
  valore=random.choice(valori)
  risultato= valore +" di "+ seme
  print(risultato)
  if risultato in estrazioni:
    continue
  else:
    estrazioni[i]= risultato
    i += 1


estrazioni.sort()
print(estrazioni)
```

EXERCISE 2

Create a dictionary containing the names of the students in a class as keys and the list of their grades as values. Enter a new student and his or her associated grades into the dictionary, taking name and grades as input. Look up the name of a pupil of your choice in the dictionary.

In [ ]:

```
#@title solution ex2
classe={"alice":[7,6,9,5],"andrea":[8,10,9,7],"carlo":[6,4,5,9],"fabio":[6,7,5,
10],"sara":[9,10,8,7]}
print ("Inserire il nome del nuovo studente")
studente=input()
voti_studenti=[0,0,0,0]  #initialize empty vector to fill
i=0
for i in range(4):
  print ("inserire il voto")
  voti_studenti[i]=input()


classe[studente]=voti_studenti
classe.items()

print("inserire il nome dell'alunno da cercare")
studente=input()
if studente in classe:
  print (classe[studente])
```

```
else:
  print("non lo conosco")
```

EXERCISE 3

Define a list of results obtained by candidates on an entrance test. Divide the list into two lists, those who passed the test and those who did not, asking as input the passing threshold. Count who scored above a certain threshold given as input.

In [ ]:

```
#@title solution ex3
risultati=[54.3, 56.7, 34.2, 76.4, 92.5, 24.6, 66.7, 71.3, 52.7, 33]
maggiori=[]
minori=[]
soglia=int(input("enter the passing threshold: "))
#let's set the input data to be taken as an integer and not as a string

for i in range(len(risultati)):
  if risultati[i]>=soglia:
    maggiori.append(risultati[i])
  else:
    minori.append(risultati[i])

maggiori.sort()
minori.sort(reverse=True)

for i in range(len(maggiori)):
  print("the candidate passed the test with grade ",maggiori[i])

for i in range(len(minori)):
  print("the candidate failed the test with grade ",minori[i])

n=0
soglia2=int(input("Enter the threshold of excellence: "))

for i in range(len(maggiori)):
  if maggiori[i]>=soglia2:
    n += 1

print("the number of people above the threshold of excellence is ", n)
```

EXERCISE 4

Write a program that, given a word as input, searches a predefined list of words for how many rhyme with the given word. Use a "rhyme" function

In [ ]:

```
#@title solution ex4
def rima(parola, elemento):
  if parola[-3:]==elemento[-3:]:
    trovato=True
    return trovato


parole=["scale","pale","male","letterale","mano","umano","catamarano","soglia",
        "foglia","sfoglia","pasta","basta","catasta"]
```

```
rime=[]
parola=str(input("Inserire la parola desiderata :"))
for elemento in parole:
  if rima(parola, elemento)==True:
    rime.append(elemento)

print (rime)
```

EXERCISE 5

We create a program that performs mathematical operations using a function. We also perform a series of checks that allow us to handle any typing errors.

In [ ]:

```
#@title solution ex5
#define math function
def functionMat():
        x= input('Enter the first number: ')
        print(' ')
        y= input('Enter the second number: ')
        print(' ')
        p1=x.isnumeric() #return False if one of the two numbers is not a
                         #decimal but a word
        p2=y.isnumeric() #I exploit this to do a while check and not do word
                         # operations with numbers but numbers with numbers
        while p1==False or p2==False : #check cycle for numbers entered
                print('enter the two numbers again because you have entered a
                       word in either or both values')
                x = input('1)-')
                y = input('2)-')
                if x.isnumeric()==True and y.isnumeric()==True: #when finally
                # the two numbers are actually decimals , then I proceed with
                # the operations
                  x = int(x) #since the input command is a function that
                        # reads strings , I will need to transform my numbers
                        # read as words into decimals
                  y = int(y)
                  print(' ')
                  print('the sum between the two numbers is: ',x+y)
                  print('the difference between the two numbers is: ',x-y)
                  print('the multiplication between the two numbers is: ',x*y)
                  print('the raising to power is: ',x**y)
                  print('the remainder of the division is: ',x%y)
                  break
              #exit the while loop with this command once I get what I wanted



functionMat() #recall the function
```

EXERCISE 6

Ceate a graph, with the **grid**, representing the sine function in an interval that is **chosen by the user** using **100 points** at all times. Use a **dashed line** in **green** color, indicating each point with a **circle**. Assign the **name to the axes** and name the graph with the **title "BREAST FUNCTION "**. Remember to check that the values of the range chosen by the user are possible.

(Hint: use the np.linespace function to create the list of x's, and the np.cos function to create the list of respective y's)

In [ ]:

```
#@title solution ex6
Xmax = 10.6 #@param {type:"slider", min:-50,max:50,step:0.2}
Xmin = -3 #@param {type:"slider", min:-50,max:50,step:0.2}
if Xmax > Xmin:
  x = np.linspace(Xmin,Xmax,100)
  y = np.sin(x)
  plt.plot(x,y,'go--')
  plt.ylabel("Ordinata")
  plt.xlabel('Ascissa')
  plt.title("FUNZIONE SENO")
  plt.grid()
  plt.show()
else:
  print("Warning: the maximum of the abscissa must be greater than the minimum.
        Change the values and re-run the cell")
```

EXERCISE 7

Create a graph that receives as input the annual revenues, divided by months, of a company and plots them on two graphs: a scatter plot and a bar graph. Represent the two graphs next to each other (in the same plot). Make the various points on the scatter plot proportional to the income of the respective month. Also in both graphs represent in green the months with income above 1500 and in red the others

In [ ]:

```
5#@title solution ex7
for chiave in entrate['mesi']:
 entrate['valori'][n]=int(input(f'inserisci le entrate del mese di
{chiave}\t'))
 if entrate['valori'][n] < 1500:
    entrate['colori'][n]='r'
  n+=1

plt.figure(figsize=(20,5))
plt.suptitle('INCASSI ANNUI')
plt.subplot(1,2,1)
plt.scatter('mesi','valori',data=entrate,s='valori',c='colori')
plt.subplot(1,2,2)
plt.bar(entrate['mesi'],entrate['valori'],color=(entrate['colori']))
plt.show()
```

EXERCISE 8

Make two side-by-side graphs depicting:

- the **z=sin(x)*y** curve (red) in a **3d graph** , with x and y two vectors chosen by you appropriately, entering as the **title** 'z-curve', specifying the name of the **axes** and the **legend** showing the z-function.
- the **surface** described by the same curve Z=sin(X)*Y , using the **color map** 'plasma' and entering the **color bar**. Use 'surface Z' as the title and name the axes.

Pay attention to the differences, in terms of **dimensions**, between a curve and a surface: the former requires one-dimensional parameters, the latter two-dimensional parameters (the **meshgrid()** command allows you to convert coordinate vectors to coordinate matrices)

In [ ]:

```
#@title solution ex8
# DEFINITION OF COORDINATES
x = np.arange(-5,5,0.1)
y = np.arange(-5,5,0.1)
z=np.sin(x)*y # curva

X, Y = np.meshgrid(x, y) # superficie
Z=np.sin(X)*Y

# GRAFIC REPRESENTATION
fig = plt.figure(figsize=(18,6))
ax = fig.add_subplot(1,2,1,projection='3d')
ax.plot(x,y,z, 'r', label=('z=sin(x)*y'))
ax.set_title('Curva z')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend()

ax = fig.add_subplot(1,2,2,projection='3d')
surf = ax.plot_surface(X,Y,Z, cmap='plasma')
ax.set_title('Superficie Z')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
fig.colorbar(surf, shrink=0.5, aspect=10)
```

EXERCISE 9

Make the **3d representation** of the **protein 7U4B**, either by simple 3D graph (plot function) or by exploiting **py3Dmol**.

Take care to make a complete graph (insert title and labels), and as for the model made with py3dmol, make sure that all structures of the protein are well visualized.

If possible represent the two graphs in the same plot to facilitate comparison between the 2 3D models

In [ ]:

```
#@title solution ex9
! wget https://files.rcsb.org/download/7U4B.pdb
coords = list()

#acquisition of data
for line in fileinput.input('7U4B.pdb'):
  if line.startswith('ATOM'):
    coords.append([float(line[26:38]), float(line[38:46]), float(line[46:54])))
    # allows the coordinates of each atom to be saved in the coords file
    # with append the coordinates of a new
    # atom with each iteration of the file
fileinput.close()
coords=np.vstack(coords)
```

```
fig = plt.figure(figsize=(12,6)) #allows you to create a figure variable
              #in which to insert the graph and set the size of the graph
ax = fig.add_subplot(1, 2, 1, projection='3d') #the add_subplot function
      # allows you to add as a subplot a third axis --> 3d ratio
ax.plot(coords[:,0], coords[:,1], coords[:,2],'g', marker='.')
# generates the plot. you pass as parameters the coordnates of each point
ax.set_title('3D model protein 3UV4')
# allows to isert the title
ax.set_xlabel('x ($\AA$)')           # allows you to name the axes (x y z)
          # and enter the unit of measurement of the axes themselves (amstrong)
ax.set_ylabel('y ($\AA$)')
ax.set_zlabel('z ($\AA$)')
fig.tight_layout()

view=py3Dmol.view()
view.addModel(open('7U4B.pdb', 'r').read(),'pdb')
view.setBackgroundColor('white')
view.zoomTo()
view.setStyle({'chain':'A'},{'cartoon': {'color':'purple'}})
# the setStyle command allows you to customize the representation. E'
# possible to select the color of the represented element and the type of
# view
view.setStyle({'chain':'B'},{'cartoon': {'color':'orange'}})
view.setStyle({'chain':'C'},{'cartoon': {'color':'green'}})

view.setStyle({'resn':'SO4'},{'stick': {'color':'red'}})
view.setStyle({'resn':'NW3'},{'stick': {'color':'blue'}})
```