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.

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.

```
#SUM
a = 7
b = 5
a + b
```

```
     12
```

```
#SUBTRACTION
a - b
```

```
     2
```

```
#MULTIPLICATION
a * b
```

```
     35
```

```
#DIVISION
a / b
```

```
     1.4
```

```
#QUOTIENT
a // b
```

```
     1
```

```
#REST
a % b
```

```
     2
```

```
#EXPONENTIAL
a ** b

     16807
```

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:

```
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.

```
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.

```
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.

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

```
     int
```

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

```
     str
```

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.

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

```
     '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.

```
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.

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

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

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

```
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..

```
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 wil
print(out)
```

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

INPUT AND OUTPUT MANAGEMENT

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

```
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.

```
x = input()
```

```
5
```

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.

```
#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.
```

```
37
```

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

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

```
[5, 7]
```

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

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

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

```
[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.

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

```
[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.

```
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.

```
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.

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

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

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

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

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

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

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

```
#multiplication of lists
a*3
```

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

```
#length lists
len(a)
```

```
    3
```

```
#I can check whether an item is present in a list by means of the in and not in commands t
6 in a
```

```
    False
```

```
#I can convert string to list
stringa = "ciao"
list(stringa)
```

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

METHODS LISTS

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

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

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

```
spesa.remove("pasta") #removes an element
spesa
```

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

```
spesa.sort() #sort alphabetically or numerically
spesa
```

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

```
spesa.sort(reverse=True) #inverse order
spesa
```

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

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

```
1
```

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

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

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.

```
tuple=(1, 2, 3, 4)
tuple
```

```
(1, 2, 3, 4)
```

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.

```
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,

```
dizionario["chiave1"]
```

```
    12
```

```
dizionario[2]
```

```
    'numero intero'
```

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

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

```
    True
```

```
12 in dizionario
```

```
    False
```

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

```
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
```

```
    dict_items([('ciao', 'hello'), ('cane', 'dog'), ('gatto', 'cat'), ('bello', 'beautif
```

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

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

```
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:

```
dizionario.keys() #print the list of keys
```

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

```
dizionario.values()  #prints the list of values
```

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

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

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

```
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.

```
dizionario.get("cane","non trovata")
```

```
'dog'
```

```
dizionario.get("tree","non trovata")
```

```
'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

```
dizionario.setdefault("albero","tree")
dizionario.items()
```

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

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.

```
5 == 5 and 5 < 3
```

```
False
```

```
5 == 5 and 3 < 5
```

```
True
```

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

```
False
```

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

```
True
```

FLOW CONTROL: The if - elif - else operator

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

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

```
greater
```

THE CYCLIC STRUCTURES

THE WHILE CYCLE

The syntax used is as follows:

```
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:

```
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
```

```
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
```

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

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

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

```
     9
```

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

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

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.

```
def controllo_numerico():
  try:
    a = int(input('inserisci un numero: '))
    print(a)
  except ValueError :
    print('hai inserito un nome')

controllo_numerico()
```

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()

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:

```
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.

```
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.

```
a = 15

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

print (funzione())

     17
```

```
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.

```
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`".

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()`**

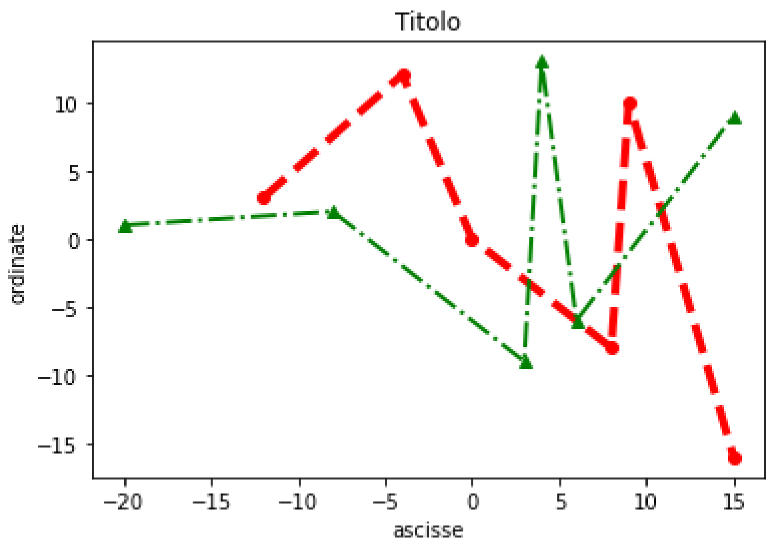
> useful for indicating some properties of the figure itself, such as the size with the figsize(width,height) subcommand

- **``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

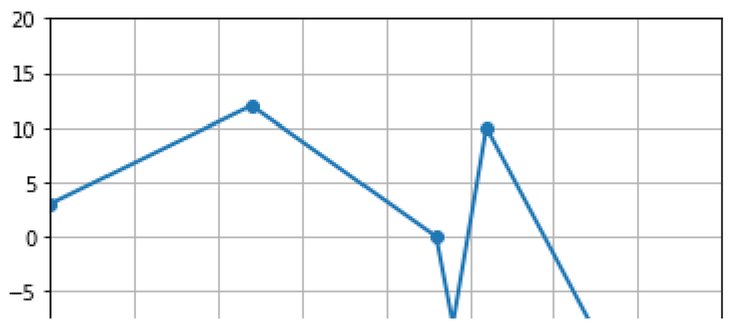
```
#@title ESEMPIO 1 (proprietà dei grafici, etichette e titolo)
import matplotlib.pyplot as plt
x1=[-12,-4,0,8,9,15]             #definizione di due liste randomiche che saranno
y1=[3,12,0,-8,10,-16]            #le ascisse e le ordinate
x2=[-20,-8,3,4,6,15]
y2=[1,2,-9,13,-6,9]
plt.plot(x1,y1,'ro--',lw=4)      #definisco il primo grafico con le rispettive proprietà
plt.plot(x2,y2,'g^-.',lw=2)      #definisco il secondo grafico con le rispettive propriet
plt.xlabel("ascisse")            #definisco il nome delle ascisse
plt.ylabel("ordinate")           #definisco il nome delle ordinate
plt.title("Titolo")              #definisco il titolo generale del plot
plt.show()
```

ESEMPIO 1 (proprietà dei grafici, etichette e titolo)



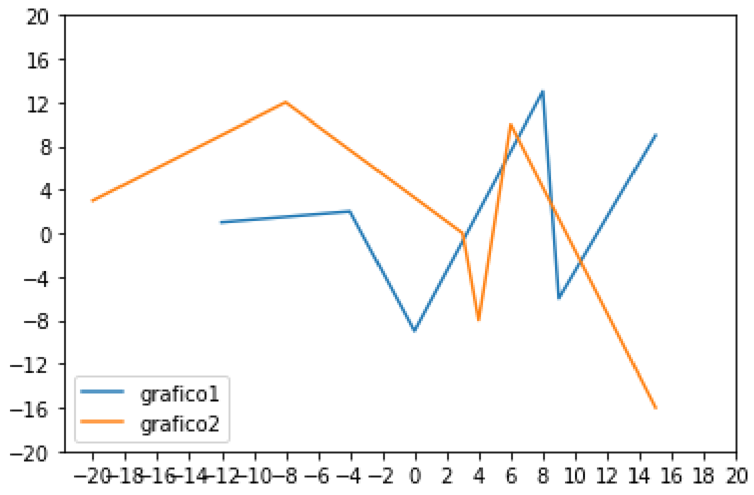
```
#@title ESEMPIO 2 (griglia e dimensioni assi)
import matplotlib.pyplot as plt
#non definisco di nuovo le liste, per semplicità uso quelle della cella precedente
plt.plot(x2,y1,'o-',lw=2)        #definisco il primo grafico
plt.grid()                        #inserisco la griglia
plt.axis([-20,20,-20,20])         #definisco la dimensione degli assi
plt.show()
```

ESEMPIO 2 (griglia e dimensioni assi)



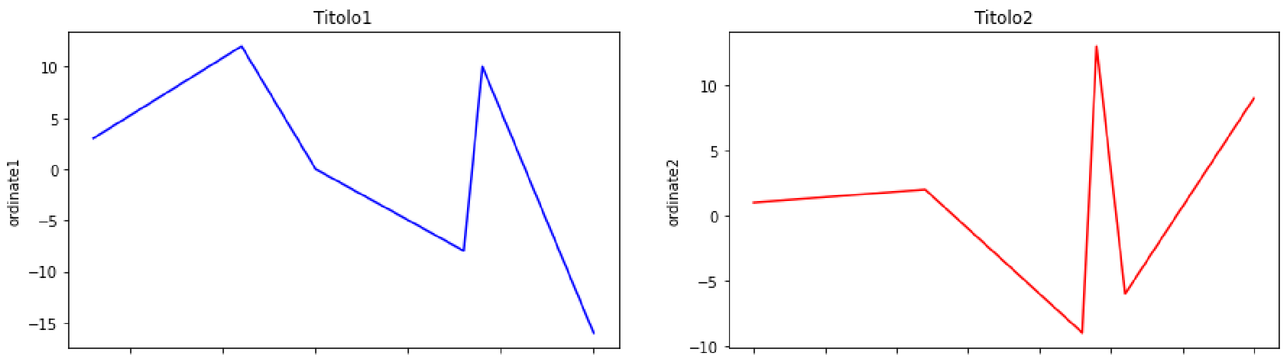
```
#@title ESEMPIO 3 (etichette grafici, legenda, indici assi)
import matplotlib.pyplot as plt
#non definisco di nuovo le liste, per semplicità uso quelle della prima cella
plt.plot(x1,y2,label='grafico1')      #definisco il primo grafico con le rispettive propri
plt.plot(x2,y1,label='grafico2')      #definisco il secondo grafico con le rispettive prop
plt.legend(loc='lower left')          #inserisco la legenda e definisco la sua posizione
plt.xticks([2*k for k in range(-10,11)])    #definisco i "tick" sull'asse x e ogni quanto
plt.yticks([4*k for k in range(-5,6)])      #definisco i "tick" sull'asse y e ogni quanto
plt.show()
```

ESEMPIO 3 (etichette grafici, legenda, indici assi)



```
#@title ESEMPIO 4 (subplot)
import matplotlib.pyplot as plt
plt.figure(figsize=(15,4))
plt.subplot(1,2,1)       #creo un subplot con 1 riga, 2 colonne e mi posiziono nella zona
plt.plot(x1,y1,'b')      #definisco il primo grafico
plt.xlabel("ascisse1")           #definisco il nome delle ascisse del primo plot
plt.ylabel("ordinate1")          #definisco il nome delle ordinate del primo plot
plt.title("Titolo1")             #definisco il titolo del primo plot
plt.subplot(1,2,2)        #richiamo il subplot creato prima e mi posiziono nella zona 2
plt.plot(x2,y2,'r')      #definisco il secondo grafico
plt.xlabel("ascisse2")           #definisco il nome delle ascisse del secondo plot
plt.ylabel("ordinate2")          #definisco il nome delle ordinate del secondo plot
plt.title("Titolo2")             #definisco il titolo del secondo plot
plt.show()
```

ESEMPIO 4 (subplot)



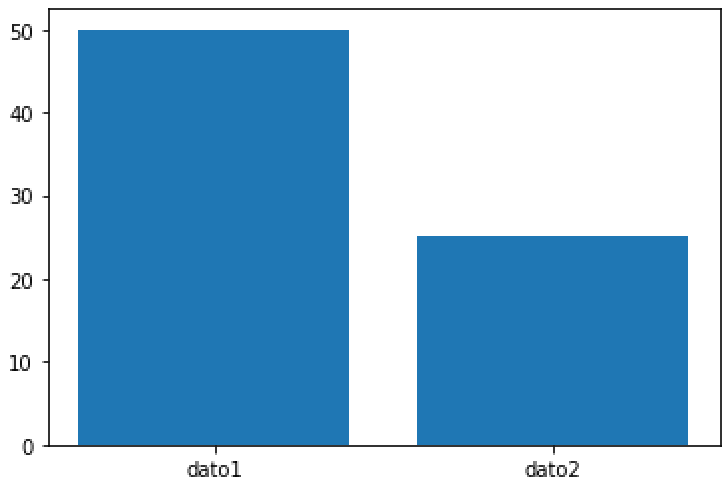
OTHER TYPES OF GRAPHS

- **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

```
#@title ESEMPIO GRAFICO A BARRE
a=["dato1","dato2"]
b=[50,25]
plt.bar(a,b)
plt.show()
```

ESEMPIO GRAFICO A BARRE



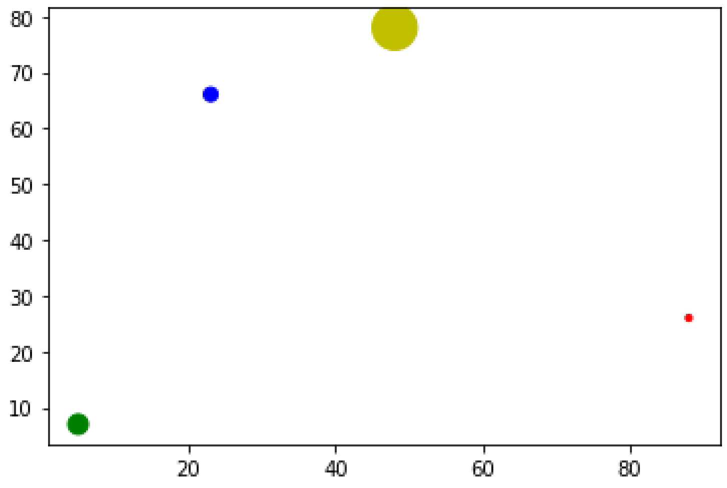
- **DIPERSION GRAPH**

> **`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.)

```
#@title ESEMPIO GRAFICO A DISPERSIONE
dizionario={'a' : [88,23,5,48],
            'b':[26,66,7,78],
            'colori':['r','b','g','y'],
            'dimensioni':[10,50,100,500]}
plt.scatter('a','b',c='colori',s='dimensioni',data=diz
plt.show()
```

ESEMPIO GRAFICO A DISPERSIONE



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

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

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

Tipologie di parametri

0.05

k: 4.5

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

n: 3

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.

```
#@title Distribuzione gaussiana al variare di μ e σ: variando il parametro mu si osserva la I

import scipy.stats as stats           # modulo necessario per plottare la distrubuzione ga
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)
```

Distribuzione gaussiana al variare di μ e σ: variando il parametro mu si osserva la traslazione della curva arancione, variando sigma come si modifica la curva verde. La funzione blu viene utilizzata come riferimento.

mu: -1.3

sigma: 10.9

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-f7c98e6bcbd8> in <module>()
      2 
      3 import scipy.stats as stats           # modulo necessario per plottare la 
distrubuzione gaussiana
----> 4 x = np.arange(-40, 40, 0.1)
      5 mu = -1.3 #@param {type:"slider", min:-5, max:5, step:0.1}
      6 sigma = 10.9 #@param {type:"slider", min:5, max:15, step:0.1}

NameError: name 'np' is not defined
```

SEARCH STACK OVERFLOW

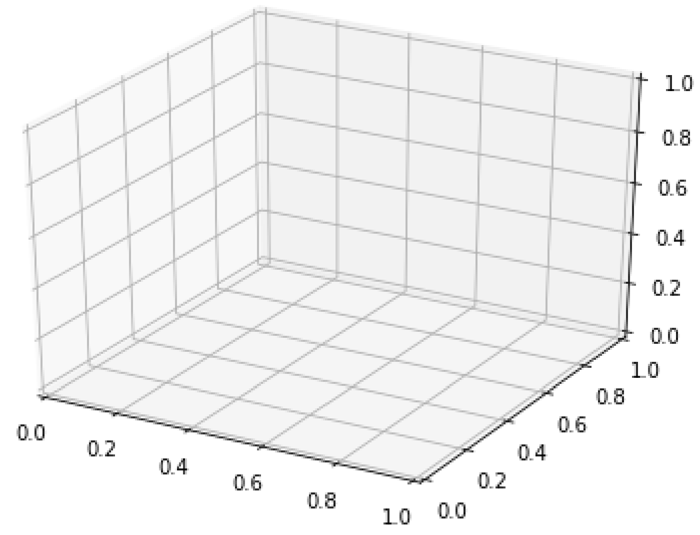
REALIZATION OF 3D MODELS

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

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from matplotlib import cm                  # modulo importato per realizzare alcune mappe
```

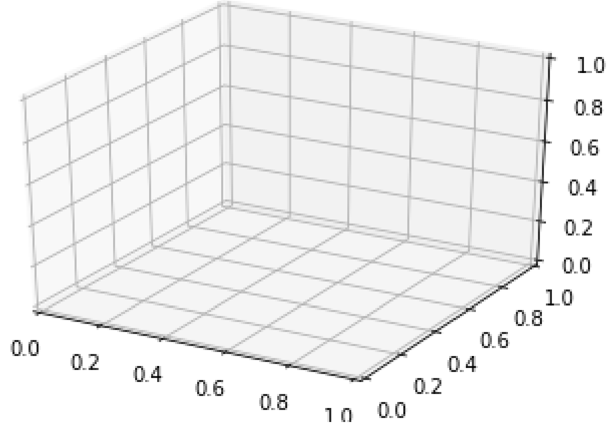
The first step is to create the figure, you can set its desired dimensions by passing it as a parameter to the **plt.figure()** command. After that to go from 2D to 3D you need to add the z-axis using one of Python's projections which is the 3D one

```
fig = plt.figure(figsize=(7,5))              # per esempio 7 (orizzontale) e 5 (vertical
ax = fig.add_subplot(111, projection='3d')     # si utilizza il comando add_subplot e si p
                                             # il primo parametro imposta la lunghezza c
```



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

```
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

```
#RAPPRESENTAZIONE DI UN PUNTO
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.scatter(3,5,7)                                # si passano come parametri le tre coordir
ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.scatter(2,4,82, marker='x')
ax.grid(False)                                   # permette di eliminare la griglia dal plc
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-877918de8012> in <module>()
      1 #RAPPRESENTAZIONE DI UN PUNTO
----> 2 fig = plt.figure(figsize=(10,5))
      3 ax = fig.add_subplot(1, 2, 1, projection='3d')
      4 ax.scatter(3,5,7)                                # si passano come parametri
le tre coordinate che individuano il punto
      5 ax = fig.add_subplot(1, 2, 2, projection='3d')

NameError: name 'plt' is not defined
```

SEARCH STACK OVERFLOW

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

```
# GRAFICO SCATTER
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)                  # passo come parametri dei vettori
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-5ec892053c9b> in <module>()
      1 # GRAFICO SCATTER
----> 2 x = np.random.randint(0,100,500)
      3 y = np.random.randint(0,100,500)
      4 z = np.random.randint(0,100,500)
      5 ax = plt.axes(projection='3d')

NameError: name 'np' is not defined
```

SEARCH STACK OVERFLOW

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

```
# GRAFICO A LINEA CONTINUA
ax = plt.axes(projection='3d')
x_data = np.linspace(-4*np.pi,4*np.pi,50)   # genera un vettore di valori (vedi appendice
y_data = np.linspace(-4*np.pi,4*np.pi,50)
z = x_data**2 + y_data**2                   # genera la funzione z da rappresentare
ax.plot(x_data,y_data,z)                    # permette di visualizzare la funzione passanc
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-eb392f5a7af9> in <module>()
      1 # GRAFICO A LINEA CONTINUA
----> 2 ax = plt.axes(projection='3d')
      3 x_data = np.linspace(-4*np.pi,4*np.pi,50)   # genera un vettore di valori
(vedi appendice per maggiori dettagli)
      4 y_data = np.linspace(-4*np.pi,4*np.pi,50)
      5 z = x_data**2 + y_data**2                   # genera la funzione z da
rappresentare

NameError: name 'plt' is not defined
```

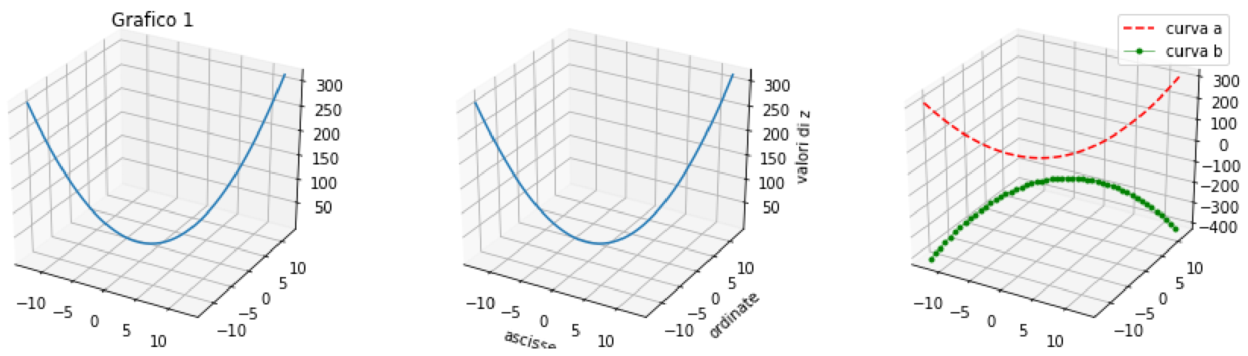
SEARCH STACK OVERFLOW

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

```
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

# PERSONALIZZAZIONE GRAFICI                    # add_subplot permette di realizzare
ax = fig.add_subplot(1, 3, 1, projection='3d')       # si passano come parametri (n° righe
ax.plot(x_data,y_data,z1)
ax.set_title('Grafico 1')                      # permette di inserire un titolo

ax = fig.add_subplot(1, 3, 2, projection='3d')
ax.plot(x_data,y_data,z1)
ax.set_xlabel('ascisse')                       # permette di etichettare gli assi
ax.set_ylabel('ordinate')
ax.set_zlabel('valori di z')

ax = fig.add_subplot(1, 3, 3, projection='3d')
ax.plot(x_data,y_data,z1, 'r--', label='curva a')
ax.plot(x_data,y_data,z2, 'g', marker='.', lw=0.5, label='curva b')    # permette di persc
ax.legend()                                                            # permette di inser
fig.tight_layout                                                       # precedente plot ]
#per tight_layout vedi appendice                                       # label='nome curva
```

```
<bound method Figure.tight_layout of <Figure size 1080x288 with 3 Axes>>
```

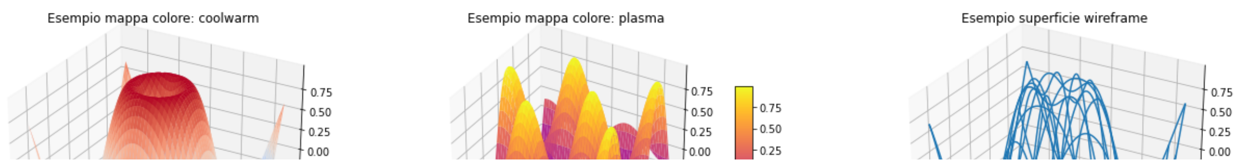


REPRESENT FUNCTIONS AS SURFACES

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

```
# GRAFICO DI SUPERFICIE
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)       # comando meshgrid() necessario per convertire i due vettor
                               # se non si utilizza python potrebbe restituire un messaggi
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,)          # con cmap= si può impostare una macca
ax.set_title('Esempio mappa colore: coolwarm')

ax = fig.add_subplot(1,3,2,projection='3d')
surf = ax.plot_surface(X,Y,Z2,cmap="plasma")
ax.set_title('Esempio mappa colore: plasma')
fig.colorbar(surf, shrink=0.5, aspect=10)           # con colorbar() è possibile inserire
                                                    # primo parametro il nome della variat
                                                    # aspect rappresenta lo spessore della
                                                    # shrink la sua altezza
ax= fig.add_subplot(1,3,3, projection='3d')
ax.plot_wireframe(X, Y, Z1, rstride=10, cstride=10)   #plot_wireframe() permette di ottene
ax.set_title('Esempio superficie wireframe')
```

```
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.

```
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])     # necessario collocare le coordinate x e y gene

mu = np.array([0.0, 0.0])                  # definisce il valore di μ

sigma = np.array([.5, .5])                 # definisce il valore di σ
covariance = np.diag(sigma**2)             # calcola la covarianza

z = multivariate_normal.pdf(xy, mean=mu, cov=covariance)
z = z.reshape(x.shape)                    # necessario per ricollocare z in una matrice

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

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

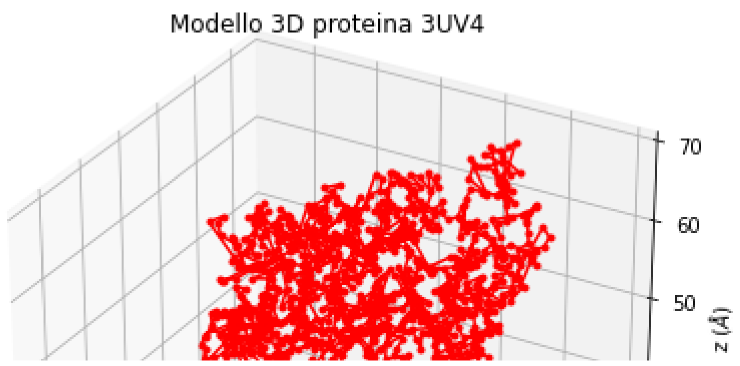
VISUALIZZAZIONE DELLE PROTEINE IN PYTHON (MODELLI 3D)

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

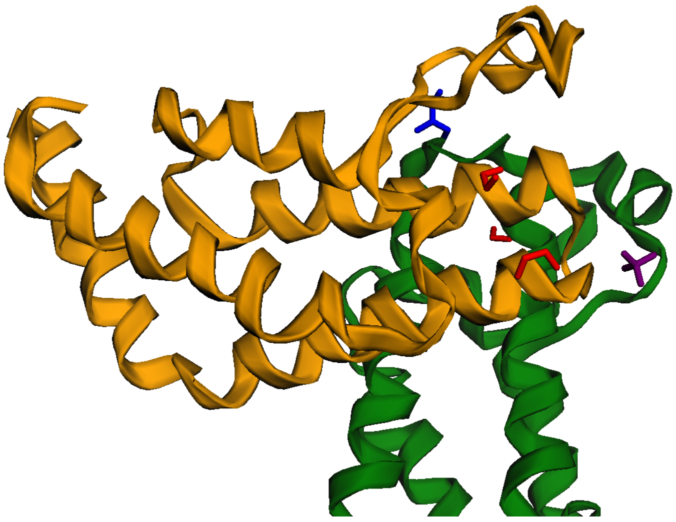
```
import numpy as np
import matplotlib.pyplot as plt
import fileinput

! wget https://files.rcsb.org/download/3UV4.pdb
coords = list()       #crea una lista chiamata coords all'interno della quale inseriremo l

#acquisizione dei dati
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])]) # permette
                                                                                # con appe
                                                                                # atomo ac
fileinput.close()

# rappresentazione grafico
coords=np.vstack(coords)                    # il comando vstack permette di trasferire i c
fig = plt.figure(figsize=(6,6))             #permette di creare una variabile figure in cu
ax = fig.add_subplot(projection='3d')       #la funzione add_subplot permette di aggiunger
ax.plot(coords[:,0], coords[:,1], coords[:,2],'r', marker='.')  # genera il grafico. si pa
ax.set_title('Modello 3D proteina 3UV4')                        # permette di iserire il t
ax.set_xlabel('x ($\AA$)')                  # permette di denominare gli assi (x y z) e in
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:

```
!pip install py3Dmol
import py3Dmol        # è un modulo che utilizzato per creare visualizzazioni di sistemi m
! wget https://files.rcsb.org/download/3UV4.pdb

view=py3Dmol.view()    # permette di richiamare la funzione in modo più rapido digitando s
view.addModel(open('3UV4.pdb', 'r').read(),'pdb')    # addModel permette di leggere il fil
view.setBackgroundColor('white')                     # permette di impostare il colore del
view.zoomTo()                                        # permette di visualizzare tutte le s
view.setStyle({'chain':'A'},{'cartoon': {'color':'green'}})          # il comando setSty
view.setStyle({'chain':'B'},{'cartoon': {'color':'orange'}})        # selezionare il col
view.setStyle({'resn':'PO4'},{'stick': {'color':'purple'}})
view.setStyle({'resn':'GOL'},{'stick': {'color':'blue'}})
view.setStyle({'resn':'EDO'},{'stick': {'color':'red'}})

# la proteina rappresentata è composta da due catene principali, catena A (verde) e catena
# e da tre ligandi:
# gruppo fosfato (PO4), viola
# glicerolo (GOL), blu
# 1,2-etandiolo (EDO), rosso
```



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.

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

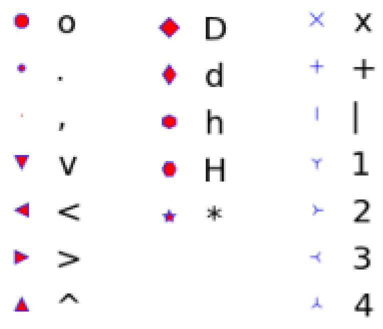
- 'm'=magenta

(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)


o
.
,
v
<
>
^
D
d
h
H
*
x
+
|
1
2
3
4

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:

```
import numpy as np
x=np.linspace(-4,4,50)         # inserisce i 50 valori equidistanti compresi tra -4 e 4
y=np.arange(0,50,0.5)          # inserisce i valori da 0 a 50 con passo di 0.5
z=np.random.randint(0,100,50)  # inserisce 50 numeri interi random presi tra 0 e 100
```

USO DELLE FUNZIONI TRIGONOMETRICHE O DI QUELLA ESPONENZIALE

```
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

```
from google.colab import files   #è un modo diverso di importare un modulo rispetto a quar
                                 # import google.colab.files (seguito da un "as nome" se s
uploaded = files.upload()        # eseguo la funzione per caricare un 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

EXERCISES

EXERCISE 1

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.

```
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]     #inizializzo vettore vuoto da riempire
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)
```

```
2 di fiori
fante di cuori
donna di picche
6 di quadri
re di picche
8 di cuori
```

```
5 di picche
7 di fiori
donna di cuori
10 di picche
['10 di picche', '2 di fiori', '5 di picche', '6 di quadri', '7 di fiori', '8 di cuo
```

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.

```
classe={"alice":[7,6,9,5],"andrea":[8,10,9,7],"carlo":[6,4,5,9],"fabio":[6,7,5,10],"sara":
print ("Inserire il nome del nuovo studente")
studente=input()
voti_studenti=[0,0,0,0]  #inizializzo vettore vuoto da riempire
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")
```

```
Inserire il nome del nuovo studente
```

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.

```
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("inserire la soglia di passaggio: "))  ##imponiamo che il dato in input s

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("il candidato ha passato il test con voto ",maggiori[i])

for i in range(len(minori)):
  print("il candidato non ha passato il test con voto ",minori[i])

n=0
soglia2=int(input("inserire la soglia di eccellenza: "))

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

print("il numero di persone sopra la soglia di eccellenza è ", 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

```
def rima(parola, elemento):
  if parola[-3:]==elemento[-3:]:
    trovato=True
    return trovato


parole=["scale","pale","male","letterale","mano","umano","catamarano","soglia","foglia","s
rime=[]
parola=str(input("Inserire la parola desiderata :"))
for elemento in parole:
  if rima(parola, elemento)==True:
    rime.append(elemento)

print (rime)
```

EXERCISE 1

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)

```
#@title SOLUZIONE ESERCIZIO 1
```

```
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("Attenione il massimo della ascissa deve essere maggiore del minimo. Cambia i valo
```

SOLUZIONE ESERCIZIO 1

Xmax: 10.6

Xmin: -3

EXERCISE 2

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

```
5#@title SOLUZIONE ESERCIZIO 2
entrate={'mesi':['GEN','FEB','MAR','APR','MAG','GIU','LUG','AGO','SET','OTT','NOV','DIC'],
         'valori':[0,0,0,0,0,0,0,0,0,0,0,0],
         'colori':['g','g','g','g','g','g','g','g','g','g','g','g']
}
n=0
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()
```

SOLUZIONE ESERCIZIO 2

EXERCISE 3

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)

Soluzione es_3

```
#@title Soluzione es_3
# DEFINIZIONE DELLE COORDINATE
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

# RAPPRESENTAZIONE GRAFICA
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 4

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

Soluzione es_4

```
#@title Soluzione es_4
! wget https://files.rcsb.org/download/7U4B.pdb
coords = list()

#acquisizione dei dati
```

```
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])]) # permette
                                                                               # con appe
                                                                               # atomo ac
fileinput.close()
coords=np.vstack(coords)

fig = plt.figure(figsize=(12,6))    #permette di creare una variabile figure in cui inseri
ax = fig.add_subplot(1, 2, 1, projection='3d')      #la funzione add_subplot permette di a
ax.plot(coords[:,0], coords[:,1], coords[:,2],'g', marker='.')  # genera il grafico. si pa
ax.set_title('Modello 3D proteina 3UV4')                        # permette di iserire il t
ax.set_xlabel('x ($\AA$)')            # permette di denominare gli assi (x y z) e inseri
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'}})           # il comando setSt
view.setStyle({'chain':'B'},{'cartoon': {'color':'orange'}})
view.setStyle({'chain':'C'},{'cartoon': {'color':'green'}})         # selezionare il colo
view.setStyle({'resn':'SO4'},{'stick': {'color':'red'}})
view.setStyle({'resn':'NW3'},{'stick': {'color':'blue'}})
```

```
**EXERCISE 5**

> We create a program that performs mathemat
operations using a function.
We also perform a series of checks that allo
handle any typing errors.
```

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.

```
#definiamo la funzione matematica
def funzioneMat():
        x= input('Inserisci il primo numero: ')
        print(' ')
        y= input('Inserisci il secondo numero: ')
        print(' ')
        p1=x.isnumeric() #restituiscono False se uno dei due numeri non e un decimale ma u
        p2=y.isnumeric() #lo sfrutto per fare un while di controllo e non fare operazioni
        while p1==False or p2==False :  #ciclo di controllo per i numeri inseriti
                print('inserisci di nuovo i due numeri perche hai inserito una parola in un
                x = input('1)-')
                y = input('2)-')
                if x.isnumeric()==True and y.isnumeric()==True:  #quando finalmente i due
                        x = int(x)  #essendo il comando di input una funazione che legge l
                        y = int(y)
                        print(' ')
                        print('la somma tra i due numeri è: ',x+y)
                        print('la differenza tra i due numeri è: ',x-y)
                        print('la moltiplicazione tra i due numeri è: ',x*y)
                        print('l\'elevamento a potenza è: ',x**y)
                        print('il resto della divisione è: ',x%y)
                        break  #esco dal ciclo while con questo comando una volta ottenuto



funzioneMat() #richiamo la funzione
```

```
Inserisci il primo numero: ciao

Inserisci il secondo numero: 3

inserisci di nuvo i due numeri perche hai inserito una parola in uno dei due valori

1)-3
2)-3

la somma tra i due numeri è:  6
la differenza tra i due numeri è:  0
la moltiplicazione tra i due numeri è:  9
l'elevamento a potenza è:  27
il resto della divisione è:  0
```

✓ 0 s data/ora di completamento: 22:35 ● ✕