

## 2 Introduzione

Python è un linguaggio di programmazione generalista noto per essere semplice da utilizzare per noi poveri umani, ovvero la fase di scrittura del codice è molto più leggera e scorrevole, rispetto ad esempio ad un codice in linguaggio C. In oltre, a differenza di altri linguaggi, esso è interpretato e non compilato; questo porta dei vantaggi, ad esempio se si verifica un errore a tempo di esecuzione la shell ci avverte indicandoci le righe di codice da noi scritte dove l'errore è avvenuto. In linguaggi compilati, come C, fortran o altri, il compilatore crea un file chiamato eseguibile dal quale però non può risalire al codice scritto da noi e quindi ciò che causa un errore a tempo di esecuzione (e.g. il famoso segmentation fault) è difficile da ritrovare. Ovviamente a causa della conservazione della massa, o si ha la botte piena o la moglie ubriaca, aut aut tertium non datur; nella fattispecie un esempio di svantaggio che possiede un linguaggio interpretato rispetto ad uno compilato è nelle prestazioni: Python è molto più lento di C o fortran, anche se un buon uso delle molte e vaste librerie che Python possiede può migliorare un po' le cose.



### 2.1 Notazioni

Nel seguito delle note saranno presenti codici in dei riquadri e, per completezza, dopo la riga [Output] viene presentato anche il risultato degli stessi nel caso ci fossero (i.e. ciò che viene stampato su shell).

### 2.2 I 4 (per ora) comandamenti dell'informatica

- Se funziona quanto basta non toccare che si guasta.
- RTFM: Read The Fucking Manual. La documentazione on-line è il miglior posto dove trovare risposte.
- Non dite che non funziona finché non avete provato a spegnere e riaccendere.
- Il computer fa esattamente quello che voi gli dite di fare non quello che volete che faccia.

## 3 Lezione Zero: Installazione

### 3.1 Installazione dell'ambiente: Pyzo

Il primo passo è procurarsi l'ambiente software tramite il quale è possibile scrivere, gestire e compilare il codice. La scelta su quale ambiente utilizzare è chiaramente arbitraria e soggetta al gusto del singolo. Un buon ambiente che si consiglia è Pyzo. Alla pagina <https://pyzo.org/start.html> è possibile trovare i link per scaricare l'opportuno installer a seconda del sistema operativo che si usa (quelli indicati sotto lo Step 1). Si faccia anche attenzione alla differenza tra gli installer per sistemi a 32 o 64 bit<sup>1</sup>. Nel caso in cui vi piaccia smanettare con i sistemi Linux, consigliamo come procedura alternativa (e più immediata) accedere al terminale e digitare i seguenti comandi:

```
1 $ sudo apt -get install python3 -pip python3 - pyqt5
2 $ sudo python3 -m pip install pyzo -- upgrade
3 $ pyzo
```

Tramite l'ultimo comando si accede alla schermata dell'ambiente Pyzo. A seconda della distribuzione che si utilizza potrebbe essere necessario utilizzare il comando yum al posto di apt-get, in particolare se utilizzate Fedora e derivati invece di Debian/Ubuntu.

### 3.2 Installazione dell'interprete: Anaconda

Ora che abbiamo l'ambiente bisogna munirsi di un interprete. Tra i tanti, si consiglia Anaconda, che porta in automatico tutti i pacchetti necessari per il lavoro scientifico. Esso è reperibile al seguente indirizzo: <https://www.anaconda.com/download/>. Allo scopo di mantenere la compatibilità con il sistema Pyzo si raccomanda di scaricare la versione corrispondente a Python 3 e non Python 2. Alternativamente è possibile procurarsi Miniconda, che è una versione ridotta e più leggera di Anaconda che arriva con molti meno pacchetti, ma occupa chiaramente meno spazio in memoria. Esso è reperibile al seguente indirizzo: <https://conda.io/miniconda.html>. È fortemente consigliato installare l'interprete nella cartella di default, in modo da rendere più semplice il lavoro di riconoscimento del programma da parte di Pyzo. Una volta installato l'interprete, aprendo Pyzo dovreste essere in grado di riconoscere sulla sinistra un editor di testo e sulla destra, uno sopra l'altro, una console per l'inserimento dei comandi e un file browser per accedere in modo più immediato alle cartelle del computer. Una volta aperto Pyzo, quest'ultimo dovrebbe riconoscere automaticamente l'interprete appena installato (Anaconda, Miniconda o altro) e potrebbe chiedervi di confermare questa scelta. Nel caso invece non riesca a trovare da solo l'interprete, magari perché installato in una cartella diversa da quella di default o perché ne avete installato più di una versione, bisogna selezionarlo manualmente tramite la procedura seguente. Dalla schermata principale di Pyzo, selezionate il menu "Shell" in alto, scegliendo quindi "Edit shell configurations". Nella finestra che viene aperta, selezionate dal menu a tendina del campo "exe" la versione di Python (ad esempio, anaconda3) che avete appena installato. Cliccate sul pulsante "Done" e poi riavviate Pyzo per terminare questa procedura. Se invece non vedete l'interprete appena installato tra le opzioni del menu a tendina, occorre specificare manualmente il percorso intero dove è stato installato l'interprete. Dato che sono stati registrati numerosi problemi nella ricerca del percorso da indicare per quanto riguarda Anaconda su Mac OS, di seguito è riportato un template del percorso dove avviene l'installazione di default, da indicare per intero.

```
1 /Users/nome_utente/opt/anaconda3/bin/python
```

oppure

```
1 /Users/nome_utente/anaconda3/bin/python
```

### 3.3 Installazione dei pacchetti

Python, come tanti altri linguaggi di programmazione, dispone di pacchetti di funzioni già pronte e direttamente utilizzabili da parte del programmatore. Anaconda contiene già tutti i pacchetti che ci serviranno, nel caso in cui abbiate optato per Miniconda, è probabile che abbiate bisogno di scaricare alcuni pacchetti aggiuntivi. L'operazione può essere effettuata accedendo alla console di Pyzo e digitando semplicemente:

```
1 install <nome_del_pacchetto >
```

oppure

```
1 pip install <nome_del_pacchetto >
```

Per essere sicuri che sia andato tutto bene provate a scrivere:

```
1 import <nome_del_pacchetto >
```

se non succede nulla siete apposto

---

<sup>1</sup>Al seguente link potete trovare informazioni per scoprire, nel caso in cui non lo sapeste, se l'architettura del vostro computer è a 32 o 64 bit: <https://support.microsoft.com/it-it/help/15056/windows-7-32-64-bit-faq>

## 4 The Zen of Python

Una volta installato pyzo, oppure python, aprite un terminale, che sia quello di pyzo, la shell di ubuntu (dopo aver digitato python3), o di anaconda, e scrivete:

```
1 >>> import this
```

ecco cosa uscirà:

The Zen of Python, by Tim Peters

Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one— and preferably only one —obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than \*right\* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea — let's do more of those!

## 5 Prima lezione

### 5.1 Funzione print

Se un macchina fosse senziente e gentile, quindi non un'intelligenza artificiale cresciuta su twitter, forse come prima cosa saluterebbe tutti e il modo per comunicare è la funzione print, che ci permette di stampare a schermo (sulla shell) delle informazione contenute nel codice. Vediamo quindi il più classico degli esempi:

```
1 print('Hello world!')
2
3 [Output]
4 Hello world!
```

Bene, se siete riuscire ad eseguire questo codice siete ufficialmente dei programmatori. Giusto per voler essere pedanti i numeri che vedete a sinistra non hanno un vero e proprio significato per l'esecuzione; il loro unico scopo è indicare all'utente a che riga si trova. Questa funzione può stampare sia valori che espressioni (in gergo stringhe) [capiremo tra poco cosa queste magiche cose che vengono stampate sono effettivamente]:

```
1 print('Hello world!')
2 print('42')
3
4 print('Hello world!', 42)
5 print('Adesso vado \n a capo')
6
7 [Output]
8 Hello world!
9 42
10 Hello world! 42
11 Adesso vado
12 a capo
```

### 5.2 Commenti

Come vedete dal codice precedente alla linea 4 ci sta un simbolo poco familiare a chi per la prima volta approccia la programmazione: "\n" chi era costui? Scritto così il codice, l'unico modo per capirlo è che voi eseguiate il codice e vedendo il risultato cerciate di risalire al significato. Ora chiaramente questa procedura è abbastanza sbrigativa ma se dovete capire diverse parti del codice, il quale magari impiega un tempo non trascurabile a darvi un risultato, beh diciamo che non è una bella vita. Al fine dunque di rendere fruibile sia ad altri o anche al voi stesso del futuro il codice è opportuno inserire i commenti, ovvero frasi che non vengono lette dall'interprete (o dal compilatore) che spiegano cosa voi stiate facendo; altrimenti vi ritroverete nella scomoda situazione in cui solo Dio saprebbe spiegarvi il funzionamento del vostro codice.

```
1 #per i commenti che occupano una singola linea di codice si usa il cancelletto
2 #stampo hello world
3 print('Hello world!')
4
5 """
6 per un commento di maggiori linee di codice
7
8 vanno usate tre virgole per racchiuderlo
9 """
10 '''
11 ma van bene
12
13 anche tre apici
14 '''
15 [Output]
16 Hello world!
```

Ricordate, un codice viene letto molte più volte rispetto a quanto viene scritto. Quindi commentate, sempre.

### 5.3 Variabili

Una variabile è un nome, un simbolo, che si dà ad un certo valore. In Python non è necessario definire le variabili prima di utilizzarle specificandone il tipo, come faremmo in C o fortran, esse si creano, o meglio si inizializzano, usando il comando di assegnazione '='. Facciamo un esempio con variabile numeriche:

```
1 numerointero= 13
2 numeroavirgolamobile= 13.
3
4 print('Numero intero:', numerointero, 'Numero in virgola mobile:', numeroavirgolamobile)
5 #oppure possiamo stampare in questo modo:
```

```

6 print(f'Numero intero: {numerointero}, Numero in virgola mobile: {numeroavirgolaabile}')
7
8 [Output]
9 Numero intero: 13 Numero in virgola mobile: 13.0
10 Numero intero: 13 Numero in virgola mobile: 13.0

```

Un'altra cosa molto fondamentale, oltre i commenti, per la fruibilità del codice è il modo di dare nomi alle variabili. Nel codice di sopra il nome delle variabili è alquanto esplicativo del loro significato, ed è in genere buona norma, appunto, dare nomi che siano intuitivi. Ora non vi dico che dovete chiamare una variabile: "momentoagolaresullasseZ", che ci vogliono tre anni solo a scriverla, che nel frattempo pure il protone inizia a decadere, ma di certo chiamarla "Lz" piuttosto che "a" o "pippo" è una strada che andrebbe perseguita. Ovviamente le variabili possono essere non solo numeri ma anche molto altro, e possiamo verificarne il tipo grazie alla funzione 'type()':

```

1 #inizializziamo delle variabili
2 n = 7
3 x = 7.
4 stringa = 'kebab'
5 lista = [1, 2., 'cane']
6 tupla = (42, 'balena')
7 dizionario = {'calza': 0, 'stampante': 0.5}
8
9 #stampiamole e stampiamone il tipo
10 print(n, type(n))
11 print(x, type(x))
12 print(stringa, type(stringa))
13 print(lista, type(lista))
14 print(tupla, type(tupla))
15 print(dizionario, type(dizionario))
16
17 [Output]
18 7 <class 'int'>
19 7.0 <class 'float'>
20 kebab <class 'str'>
21 [1, 2.0, 'cane'] <class 'list'>
22 (42, 'balena') <class 'tuple'>
23 {'calza': 0, 'stampante': 0.5} <class 'dict'>

```

Dizionari, liste, tuple, possono essere elementi molto utili. Giusto qualche info un po' anticipata tutti essi sono indicizzati, i primi due sono modificabili le tuple invece sono immutabili, come abbiamo visto non ci sono particolari problemi di casting, essi cioè possono contenere elementi di vario genere. I dizionari inoltre, utilizzando un sistema chiave valore possono essere utili per gestire alcuni output di codici lunghi o complessi; per esempio sono molto utili nella programmazione in parallelo, dove l'ordine si perde quindi un dizionario è un ottimo modo per tenere traccia di tutta l'esecuzione.

```

1 """
2 liste tuple e dizionari sono oggetti indicizzati che possono
3 ogni tipo di informazioni al loro interno
4 """
5 lista = [0, 1, 'lampada', [0, 23], (29, 11), {3:4, 'capra':'panca'}] # lista
6 tupla = (3, 2, "ruspa", [0, 0], (3, 9), {87:90})
7 dictz = {0:1, 1:[2, 3], 'astolfo':(2, 3), "diz":{1:2}}
8
9 # stampo tutto a schermo
10 print(lista)
11 print(tupla)
12 print(dictz)
13
14 # tramite l'indice accedo all'elemento della lista o della tupla
15 print(lista[3])
16 print(tupla[3])
17
18 # per il dizionario va invece usata la chiave
19 print(dictz['astolfo'])
20 print(dictz[1])
21
22 # modifico elemento all'indice zero nella lista e nel dizionario
23 lista[0] = (0, 1, 2, 3, 4, 5)
24 dictz[0] = "BUONA PASQUA"
25
26 print(lista)
27 print(dictz)
28
29 # se lo facessi con la tupla avrei un errore
30 tupla[0] = 1

```

```

31
32 [Output]
33 [0, 1, 'lampada', [0, 23], (29, 11), {3: 4, 'capra': 'panca'}]
34 (3, 2, 'ruspa', [0, 0], (3, 9), {87: 90})
35 {0: 1, 1: [2, 3], 'astolfo': (2, 3), 'diz': {1: 2}}
36 [0, 23]
37 [0, 0]
38 (2, 3)
39 [2, 3]
40 [(0, 1, 2, 3, 4, 5), 1, 'lampada', [0, 23], (29, 11), {3: 4, 'capra': 'panca'}]
41 {0: 'BUONA PASQUA', 1: [2, 3], 'astolfo': (2, 3), 'diz': {1: 2}}
42 Traceback (most recent call last):
43   File "/home/francesco/GitHub/4BLP/1 Prima Lezione/codiciL1/list_tuple_dict.py", line 30, in
    <module>
44     tupla[0] = 1
45 TypeError: 'tuple' object does not support item assignment

```

Concentriamoci attualmente su come fare le classiche operazioni matematiche tra variabili, numeriche ovviamente:

```

1 x1 = 3
2 y1 = 4
3
4 somma = x1 + y1
5 prodotto = x1*y1
6 differenza = x1 - y1
7 rapporto = x1/y1
8 potenza = x1**y1
9
10 print(somma, prodotto, differenza, rapporto, potenza)
11 [Output]
12 7 12 -1 0.75 81

```

E finqui tutto bene, tutto abbastanza normale e mi raccomando ricordate che l'elevamento a potenza si fa con doppio asterisco "\*\*". Vale la pena dilungarsi un attimo su una piccola questione: l'aritmetica in virgola mobile è intrinsecamente sbagliata poiché giustamente il computer ha uno spazio di memoria finita e quindi non può tenere infinite cifre decimali. A seconda del tipo della variabile ci si ferma ad un tot di cifre decimali, 8 per i float32, 16 per i float64; dove il numero che segue la parola float indica il numero di bit che il computer usa per scrivere il numero. Questa precisione può comunque essere cambiata grazie alla libreria: "mpmath"; la quale permette di settare una precisione arbitraria, divertitevi a scoprirla. Vediamo un classico esempio:

```

1 x = 0.1
2 y = 0.2
3 z = 0.3
4
5 print(x + y)
6 print(z)
7
8 [Output]
9 0.30000000000000004
10 0.3

```

Il precedente è solo uno tra i molti esempi che si potrebbero fare per far notare come l'aritmetica dei numeri in virgola mobile possa dare problemi. Come esercizio lasciato al lettore provate a vedere se è vero che  $a + (b + c) = (a + b) + c$  con  $a, b, c$  numeri in virgola mobile, probabilmente il computer non sarà d'accordo. Ai computer i numeri in virgola mobile, i numeri reali, non piacciono molto, preferiscono i numeri interi e quelli razionali (e ovviamente adorano le potenze di due, grazie codice binario):

```

1 #variabili
2 x = 0.1
3 y = 0.2
4 z = 0.3
5 #sommo le prime due
6 t = x + y
7
8 """
9 applico una funzione che mi fornisce
10 una tupla contenente due numeri interi
11 il cui rapporto restituisce il numero iniziale.
12 output del tipo: (numeratore, denominatore)
13 """
14 print(t.as_integer_ratio())
15 print(z.as_integer_ratio())
16
17 [Output]
18 (1351079888211149, 4503599627370496)
19 (5404319552844595, 18014398509481984)

```

Vediamo quindi che un numero reale è scritto in realtà, e altrimenti non si potrebbe fare, come numero razionale. Per quanto riguarda i numeri in virgola mobile, possiamo scegliere quante cifre dopo la virgola stampare, vediamo con un esempio:

```
1 #definiamo una variabile
2 c = 3.141592653589793
3
4 #stampa come intero
5 print('%d' %c)
6
7 #stampa come reale
8 print('%f' %c) #di default stampa solo prime 6 cifre
9 print(f'{c}') #di default stampa tutte le cifre
10
11 #per scegliere il numero di cifre, ad esempio sette cifre
12 print('%7f' %c)
13 print(f'{c:.7f}')
14
15 [Output]
16 3
17 3.141593
18 3.141592653589793
19 3.1415927
20 3.1415927
```

Notare che il computer esegue un arrotondamento. Inoltre abbiamo usato la lettera "f" perché vogliamo un numero decimale, se volessimo altri formati potremmo usare: "i" o "d" per gli interi, "o" per un numero in base otto, "x" per un numero esadecimale, "e" per un numero in notazione scientifica. Infine le lettere "c" e "s" indicano rispettivamente un singolo carattere e una stringa.

Una variabile può essere ridefinita e cambiare valore, addirittura cambiate tipo, il computer userà l'assegnazione più recente (attenzione mi raccomando che ci vuole poco che una cosa del genere fornisca errori):

```
1 #definiamo una variabile
2 x = 30
3
4 """
5
6 operazioni varie
7
8
9 """
10
11 #ridefiniamo la variabile
12 x = 18
13
14 print('x=', x)
15
16 """
17 E' possibile anche sovrascrivere una variabile
18 con un numero che dipende dal suo valore precedente:
19 """
20 x = x + 1 #incrementiamo di uno
21 #Oppure:
22 x += 1
23 print('x=', x)
24
25 x = x * 2 #moltiplichiamo per due
26 #Oppure:
27 x *= 2
28 print('x=', x)
29
30 [Output]
31 x= 18
32 x= 20
33 x= 80
```

Come si vede i vari comandi  $x = x \text{ operazione numero}$  possono essere abbreviati con  $x \text{ operazione} = \text{numero}$ .

## 5.4 Librerie

Le librerie sono luoghi mistici create dagli sviluppatori, esse contengono molte funzioni, costanti e strutture dati predefinite; in generale se volete fare qualcosa esisterà una libreria con una funzione che implementa quel qualcosa o che comunque vi può aiutare in maniera non indifferente (ogni tanto però è interessante andare a vedere cosa ci sta dietro, ma ne parleremo, non molto, ma in vari ambiti, più avanti). Prima di poter accedere

ai contenuti di una libreria, è necessario importarla. Per farlo, si usa il comando `import`. Solitamente è buona abitudine importare tutte le librerie che servono all'inizio del file. Ecco un paio di esempi:

```
1 import numpy
2
3 #per usare un contenuto di questa libreria basta scrivere numpy.contenuto
4 pigreco = numpy.pi
5 print(pigreco)
6
7 #Possiamo anche ribattezzare le librerie in questo modo
8 import numpy as np
9 #da ora all'interno del codice numpy si chiama np
10
11 eulero = np.e
12 print(eulero)
13
14 [Output]
15 3.141592653589793
16 2.718281828459045
```

```
1 import math
2
3 coseno=math.cos(0)
4 seno = math.sin(np.pi/2) #python usa di default gli angoli in radianti!!!
5 senosbagliato = math.sin(90)
6
7 print('Coseno di 0=', coseno, "\nSeno di pi/2=", seno, "\nSeno di 90=", senosbagliato)
8
9 #bisogna quindi stare attenti ad avere tutti gli angoli in radianti
10 angoloingradi = 45
11 #questa funzione converte gli angoli da gradi a radianti
12 angoloinradianti = math.radians(angoloingradi)
13
14 print("Angolo in gradi:", angoloingradi, "Angolo in radianti:", angoloinradianti)
15
16 [Output]
17 Coseno di 0= 1.0
18 Seno di pi/2= 1.0
19 Seno di 90= 0.8939966636005579
20 Angolo in gradi: 45 Angolo in radianti: 0.7853981633974483
```

Le due librerie qui usate contengono funzioni simili, ad esempio il seno è implementato sia in `numpy` che in `math`, cambia il fatto che `math` può calcolare il seno di un solo valore, mentre `numpy`, come vedremo, può calcolare il seno di una sequenza di elementi. Come sapere tutte le possibili funzioni contenute nelle librerie e come usarle? "Leggetevi il cazzo di manga" (n.d.r. Leggetevi la documentazione disponibile tranquillamente online). Altra cosa interessante è che essendo la documentazione scritta sul codice, esattamente come qui noi scriviamo i commenti, potete consultarla anche da shell. Su Pyzo vi basta scrivere sulla shell: `"nomefunzione?"`, mentre se usate una shell normale, stile quella di ubuntu dovete prima digitare `"python"` oppure `"python3"` sulla shell e vi si aprirà l'ambiente python dove potete importare i pacchetti come se scriveste codice e per vedere la documentazione vi basta fare `"help(nomefunzione)"`. Inoltre spesso si trova la sintassi:

```
1 from numpy import *
```

con `numpy` o con qualsiasi altra libreria. Questo ci permette di non dover scrivere ogni volta `"numpy."` o `"np."` davanti le funzioni che vogliamo utilizzare. Bisogna però stare attenti all'esistenza di funzioni con stesso nome ma che fanno cose diverse, come dicevamo prima la funzione `seno` ad esempio. In un contentesto in cui si importano sia `math` che `numpy` usando l'asterisco ci sarà un conflitto su che funzione usare; o meglio, verrà usata la funzione della libreria più recentemente importata. il che vuol dire che se scrivete:

```
1 from numpy import *
2 from math import *
```

la funzione `seno` chiamata sarà quella di `math` e quindi avrete un errore se il possibile input non divessere essere un numero ma un array.

## 5.5 Come leggere il Traceback

Quelli che abbiamo sopra sono esempi di errori che danno come risultato in genere l'interruzione del codice. Quindi quando la shell di Pyzo si tinge di rosso cremisi che nemmeno fosse appena finita una battaglia campale di Vlad figlio del drago voivoda di Valacchia è buona norma leggere attentamente il messaggio di errore, il traceback (che come suggerisce il nome va letto al contrario; da sotto verso sopra), e capire cosa si è sbagliato. Il traceback infatti è la catena di eventi che hanno portato all'errore. Analizziamo il caso di sopra.



```

1 Traceback (most recent call last):
2   File "<tmp 1>", line 5, in <module>
3     b = 1/a
4 ZeroDivisionError: division by zero

```

La prima riga ci fa capire che c'è stato un errore.

La seconda ci dice che nel file chiamato " < tmp1 > " (perché mi ero dimenticato di salvarlo ancora, sennò ci sarebbe il path del file) alla linea 5, nel codice eseguito (se fosse dentro una funzione ci sarebbe, oltre a questa, un'altra riga uguale con il nome della funzione al posto di < module >) è successo qualcosa.

La terza linea è la linea di codice a cui è avvenuto il misfatto.

La quarta ci dà due informazioni, separate dai due punti: il tipo di errore che è avvenuto e le informazioni riguardo ad esso.

Ora capisco che prima vi dico di leggerlo al contrario e poi qui ve lo descrivo nel normale ordine di lettura, ma è solo per farvi capire il significato delle varie linee di errore. Facciamo adesso un esempio un po' più complicato, leggendolo correttamente, in cui useremo funzioni quindi magari potete tornare a rileggerlo dopo se volete.

```

1 def err(c):
2     b = 1/c
3     return b
4
5 def run(c):
6     print(err(c))
7
8 if __name__ == "__main__":
9     run(0)
10
11 [Output]
12 Traceback (most recent call last):
13   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 9, in
14     <module>
15     run(0)
16   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 6, in
17     run
18     print(err(c))
19   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 2, in
20     err
21     b = 1/c
22 ZeroDivisionError: division by zero

```

Leggiamolo insieme, partendo dal basso abbiamo:

C'è stata una divisione per zero che ha causato l'errore di tipo ZeroDivision error alla linea 2 nella funzione "err" contenuta nel codice avente quel path (sta volta il codice era salvato).

L'errore è stato causato dalla chiamata della funzione "err" da parte della funzione "run" a linea 6, contenuta nello stesso file, hanno infatti stesso path.

Ma ancora prima l'errore è causato dalla chiamata della funzione "run" (che ha chiamato err, che ha prodotto l'errore) all'interno dello stesso codice, alla linea 9. Insomma fiera dell'est di Branduardi spostati proprio. Se proprio non capite che vi sta dicendo, buona norma è copiare la riga più bassa del traceback e incollarla su Google. Qualcuno avrà già avuto il vostro problema.

## 5.6 Un paio di trick

Abbiamo visto che Python non necessita di punti e virgola per delimitare una linea come in C. Tuttavia è possibile usarli con lo stesso scopo, quindi se volete mettere due comandi Python sulla stessa riga vi basta separarli con un ";" e verranno eseguiti come fossero due righe diverse. Inoltre possono essere usate sulla shell per nascondere l'output del comando, dato che come premete invio sulla shell la riga appena scritta viene interpretata, esattamente come su Mathematica. Sempre su shell invece se avete bisogno di fare dei calcoli veloci stile calcolatrice potete usare l'underscore per riferirvi al risultato precedente, come quando sulla calcolatrice premete il tasto Ans.

## 6 Seconda lezione

Ripetiamo tutti insieme: Python conta da zero.

### 6.1 Gli array

Un array unidimensionale è semplicemente una sequenza ordinata di numeri; è, in sostanza, un vettore e come tale si comporta. Utilizzeremo la libreria numpy. Per alcuni aspetti essi sono simili alle liste native di Python ma le differenze sono molte, in seguito ne vedremo alcune. Cominciamo con qualche esempio:

```
1 import numpy as np
2
3 #Creiamo un array di 5 elementi
4 array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0]) #scrivere 2.0 equivale a scrivere 2.
5
6 print(array1)
7
8 #per accedere a un singolo elemento dell'array basta fare come segue:
9 elem = array1[1]
10
11 #ATTENZIONE! Gli indici, per Python, partono da 0, non da 1!
12 print(elem)
13
14 [Output]
15 [ 1.  2.  4.  8. 16.]
16 2.0
```

ora, avendo creato il nostro array potremmo volendo aggiungere o togliere degli elementi:

```
1 import numpy as np
2
3 array1=np.array([1.0, 2.0, 4.0, 8.0, 16.0])
4
5 #Aggiungiamo ora un numero in una certa posizione dell'array:
6 array1 = np.insert(array1, 4, 18)
7 '''
8 abbiamo aggiunto il numero 18 in quarta posizione, la sintassi e' :
9 np.insert(array a cui vogliamo aggiungere un numero, posizione dove aggiungerlo, numero)
10 '''
11 print(array1)
12
13 #Per aggiungere elementi in fondo ad un array esiste anche il comando append della libreria
    numpy:
14 array2 = np.append(array1, -4.)
15 print(array2)
16 #Mentre per togliere un elemento basta indicare il suo indice alla funzione remove di numpy:
17 array2 = np.delete(array2, 0)
18 print(array2)
19
20 [Output]
21 [ 1.  2.  4.  8. 18. 16.]
22 [ 1.  2.  4.  8. 18. 16. -4.]
23 [ 2.  4.  8. 18. 16. -4.]
```

Analogamente ciò può essere fatto per le liste con le funzioni native, quindi non di numpy, append e pop. Più corretto sarebbe dire che esse sono dei metodi della classe che gestisce le liste, infatti come vediamo nel prossimo esempio la sintassi è leggermente diversa, ma non vale la pena complicarci troppo la vita.

```
1 # lista iniziale
2 lista = [1, 2, 3, 4]
3 print(lista)
4
5 #aggiungo un elemento in coda, quindi alla fine della lista
6 lista.append(42)
7 print(lista)
8
9 #rimuovo l'ultimo elemento
10 lista.pop()
11 print(lista)
12
13 [Output]
14 [1, 2, 3, 4]
15 [1, 2, 3, 4, 42]
16 [1, 2, 3, 4]
```

Altri metodi interessanti per le liste sono "index()", "remove()", "count()", "insert()", "reverse()", "extend()", "sort()"; divertitevi a scoprire cosa ciascuno fa.

## 6.2 Tipi di array

Come le variabili numeriche sopra anche gli array posseggono i tipi e qui viene la prima differenza con le liste, se ad un array di numeri provassimo ad aggiungere un elemento che sia una stringa avremmo un errore; questo perché ogni array di numpy ha un suo tipo ben definito, che viene fissato, implicitamente o esplicitamente, al momento della creazione. Possiamo sì creare un array di tipo misto ma con tale array non si potrebbero fare le classiche operazioni matematiche.

```
1 import numpy as np
2
3 array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])
4
5 tipoarray1 = array1.dtype
6 print(tipoarray1)
7
8 a = np.array([0, 1, 2])
9 #abbiamo scritto solo numeri interi => array di interi
10
11 b = np.array([0., 1., 2.])
12 #abbiamo scritto solo numeri con la virgola => array di numeri float
13
14 """
15 #nota: anche se si dice "numero con la virgola",
16 vanno scritti sempre col punto!
17 La virgola separa gli argomenti
18 """
19
20 c = np.array([0, 3.14, 'giallo'])
21 #quest'array e' misto. Ci sono sia numeri interi che float che stringhe
22
23
24 #ora invece il tipo viene definito in maniera esplicita:
25 d = np.array([0., 1., 2.], 'int')
26 e = np.array([0, 1, 2], 'float')
27
28 print(a, a.dtype)
29 print(b, b.dtype)
30 print(c, c.dtype)
31 print(d, d.dtype)
32 print(e, e.dtype)
33
34
35 [Output]
36 float64
37 [0 1 2] int32
38 [0. 1. 2.] float64
39 ['0' '3.14' 'giallo'] <U32
40 [0 1 2] int32
41 [0. 1. 2.] float64
```

## 6.3 Array predefiniti

Vediamo brevemente alcuni tipi di array già definiti e di uso comune:

```
1 import numpy as np
2
3 #array contenente tutti zero
4 arraydizeri_0 = np.zeros(3)#il numero specificato e' la lunghezza
5 arraydizeri_1 = np.zeros(3, 'int')
6
7 #array contenente tutti uno
8 arraydiuni_0 = np.ones(5)#il numero specificato e' la lunghezza
9 arraydiuni_1 = np.ones(5, 'int')
10
11 print(arraydizeri_0, arraydizeri_1)
12 print(arraydiuni_0, arraydiuni_1)
13
14 """
15 questo invece e' un array il cui primo elemento e' zero
16 e l'ultimo elemento e' 1, lungo 10 e i cui elementi sono
17 equispaziati in maniera lineare tra i due estremi
18 """
19 equi_lin = np.linspace(0, 1, 10)
20 print(equi_lin)
21
```

```

22
23 """
24 questo invece e' un array il cui primo elemento e' 10^1
25 e l'ultimo elemento e' 10^2, lungo 10 e i cui elementi sono
26 equispaziati in maniera logaritmica tra i due estremi
27 """
28 equi_log = np.logspace(1, 2, 10)
29 print(equi_log)
30
31 [Output]
32 [0.  0.  0.] [0 0 0]
33 [1.  1.  1.  1.  1.] [1 1 1 1 1]
34 [0.          0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
35  0.66666667 0.77777778 0.88888889 1.          ]
36 [ 10.          12.91549665  16.68100537  21.5443469   27.82559402
37  35.93813664  46.41588834  59.94842503  77.42636827 100.          ]

```

## 6.4 Operazioni con gli array

Vediamo ora un po' di cose che si possono fare con gli array:

```

1 import numpy as np
2
3 array1 = np.array([1.0, 2.0, 4.0, 8.0, 16.0])
4
5 primi_tre = array1[0:3]
6 print('primi_tre = ', primi_tre)
7 """
8 Questa sintassi seleziona gli elementi di array1
9 dall'indice 0 incluso all'indice 3 escluso.
10 Il risultato e' ancora un array.
11 """
12
13 esempio = array1[1:-1]
14 print(esempio)
15 esempio = array1[-2:5]
16 print(esempio)
17 #Questo metodo accetta anche valori negativi, con effetti curiosi
18
19
20 elementi_pari = array1[0::2]
21 print('elementi_pari = ', elementi_pari)
22 """
23 In questo esempio invece, usando invece due volte il simbolo :
24 intendiamo prendere solo gli elementi dall'indice 0 saltando di 2 in 2.
25 Il risultato e' un array dei soli elementi di indice pari
26 """
27
28 rewind = array1[::-1]
29 print('rewind = ', rewind)
30 """
31 Anche qui possiamo usare valori negativi.
32 In particolare questo ci permette di saltare "all'indietro"
33 e, ad esempio, di invertire l'ordine di un'array con un solo comando
34 """
35
36 [Output]
37 primi_tre =  [1.  2.  4.]
38 [2.  4.  8.]
39 [ 8. 16.]
40 elementi_pari =  [ 1.  4. 16.]
41 rewind =  [16.  8.  4.  2.  1.]

```

Benché strano python non rifiuta gli indici negativi semplicemente perché essi indicizzano l'array al contrario ma partendo da 1. Quindi "-1" indica l'ultimo elemento, "-2" il penultimo e così via. Grande comodità sono le operazioni matematiche che possono essere fatte direttamente senza considerare i singoli valori, o meglio Python ci pensa da sé a fare le operazioni elemento per elemento. Gli array devono avere la stessa dimensione altrimenti avremmo errore, infatti potrebbe esserci un elemento spaato.

```

1 import math
2 import numpy as np
3
4 v = np.array([4, 5, 6])
5 w = np.array([1.2, 3.4, 5.8])
6

```

```

7 #classiche operazioni
8 somma = v + w
9 sottr = v - w
10 molt = v * w
11 div = v / w
12
13 print(v, w)
14 print()
15 print(somma, sottr, molt, div)
16 print()
17 #altri esempi
18 print(v**2)
19 print(np.log10(w))
20
21 """
22 come dicevamo prima qui' otterremmo errore poiche'
23 math lavora solo con numeri o, volendo,
24 array unidimensionali lunghi uno
25 """
26 print(math.log10(w))
27
28 [Output]
29 [4 5 6] [1.2 3.4 5.8]
30
31 [ 5.2  8.4 11.8] [2.8 1.6 0.2] [ 4.8 17.  34.8] [3.33333333 1.47058824 1.03448276]
32
33 [16 25 36]
34 [0.07918125 0.53147892 0.76342799]
35 Traceback (most recent call last):
36   File "<tmp 1>", line 26, in <module>
37     print(math.log10(w))
38 TypeError: only size-1 arrays can be converted to Python scalars

```

Se provassimo le stesse con delle liste solo la somma non darebbe errore, ma il risultato non sarebbe comunque lo stesso che otteniamo con gli array. Anche moltiplicare un array o una lista per un numero intero produce risultati diversi se provate vi sarà facile capire perché si è specificato che il numero deve essere intero. Ai fisici piace dire che un vettore è un qualcosa che trasforma come un vettore, e con questi esempi potrete capire che una lista non "trasforma" come un vettore mentre un array di numpy sì. Per questo nella programmazione scientifica se ne fa largo uso.

## 6.5 Maschere

Vi sarete di certo accorti che per creare un array di numpy quel che facciamo è passare una lista alla funzione "np.array" e infatti prima avevamo visto un array che conteneva una stringa, e per cui tutti gli elementi erano diventate stringhe. Similmente è possibile fare un array di valori booleani, cioè vero e falso.

```

1 import numpy as np
2
3 # array che contiene valori logici, detti booleani
4 b = np.array([True, False])
5
6 print(b)
7
8 [Output]
9 [ True False]

```

E fin qui nulla di sorprendente si potrebbe dire. Possiamo fare però una cosa interessante:

```

1 import numpy as np
2
3 # array che contiene valori logici, detti booleani
4 b = np.array([True, False])
5
6 # normalissimo array numerico
7 x = np.array([32, 89])
8 y = x[b] # x in corrispondenza di indici di b
9 print(y)
10
11 [Output]
12 [32]

```

Vediamo che se passiamo come indice del nostro array un array di valori logici, otteniamo un nuovo array, il quale contiene solo gli elementi corrispondenti all'indice che coincide con i valori "True" all'interno dell'array logico. Abbiamo creato quella che si dice una maschera e può essere utili in molti casi, vediamo un altro esempio.

```

1 import numpy as np
2
3 # creo un array
4 x = np.array([5, 4, 2, 8, 3, 9, 7, 2, 6, 3, 9, 8])
5
6 # voglio selezionare solo gli elementi maggiori di una certa soglia
7 mask = x >= 4 # mask e' un array di booleani, secondo la condizione data
8 # mask vale True negli indici in cui il valore di x e' maggiore o uguale a 4
9
10 print(x)
11 print(mask)
12 print(x[mask])
13
14 [Output]
15 [5 4 2 8 3 9 7 2 6 3 9 8]
16 [ True  True False  True False  True  True False  True False  True  True]
17 [5 4 8 9 7 6 9 8]

```

Poi se i nostri dati fossero una funzione del tempo potremmo magari creare la maschera sul tempo e applicarla ai nostri dati, magari perchè vogliamo considerare solo un certo range. Finché gli array son lunghi uguali potete fare quello che volete.

## 6.6 Matrici

Se un array unidimensionale lungo  $n$  è un vettore ad  $n$  componenti allora un array bidimensionale sarà una matrice.

```

1 import numpy as np
2
3 #esiste la funzione apposita di numpy per scrivere matrici.
4 matrice1 = np.matrix('1 2; 3 4; 5 6')
5 #Si scrivono essenzialmente i vettori riga della matrice separati da ;
6
7 #equivalente a:
8 matrice2 = np.matrix([[1, 2], [3, 4], [5,6]])
9 # oppure anche: matrice2 = np.array([[1, 2], [3, 4], [5,6]])
10 print(matrice1)
11 print(matrice2)
12
13
14 matricedizeri = np.zeros((3, 2)) #tre righe, due colonne: matrice 3x2
15 print('Matrice di zeri:\n', matricedizeri, '\n')
16 matricediuni = np.ones((3,2))
17 print('Matrice di uni:\n', matricediuni, '\n')
18
19 [Output]
20 [[1 2]
21  [3 4]
22  [5 6]]
23 [[1 2]
24  [3 4]
25  [5 6]]
26 Matrice di zeri:
27 [[0. 0.]
28  [0. 0.]
29  [0. 0.]
30
31 Matrice di uni:
32 [[1. 1.]
33  [1. 1.]
34  [1. 1.]

```

E ovviamente anche qui possiamo fare le varie operazioni matematiche:

```

1 import numpy as np
2
3 matrice1 = np.matrix('1 2; 3 4; 5 6')
4 matricediuni = np.ones((3,2))
5
6 sommadimatrici = matrice1 + matricediuni
7 print('Somma di matrici:\n', sommadimatrici)
8
9 matrice3 = np.matrix('3 4 5; 6 7 8') #matrice 2x3
10 prodottodimatrici = matrice1 * matrice3 #matrice 3x(2x2)x3
11 # attenzione che non funziona se si usa np.array()
12 #alternativamente si potrebbe scrivere: prodottodimatrici = matrice1 @ matrice3

```

```

13 # che funziona sia con np.matrix che np.array
14 print('\nProdotto di matrici:\n', prodottodimatrici)
15
16 [Output]
17 Somma di matrici:
18 [[2. 3.]
19  [4. 5.]
20  [6. 7.]]
21
22 Prodotto di matrici:
23 [[15 18 21]
24  [33 40 47]
25  [51 62 73]]

```

Ci siamo fermati alle matrici, oggetti a due indici, ma volendo avremmo potuto creare oggetti a più indici (i famosi tensori, tanto sempre numeri sono) ad esempio "np.ones((3,3,3))" creerebbe un oggetto a tre indici di uni, che possiamo vedere ad esempio come un vettore a tre componenti, ciascuna delle quali è una matrice  $3 \times 3$ . Se questo vi sembra strano aspettate di fare meccanica quantistica e ne riparlamo.

Ora è importante far notare una cosa: l'operazione di assegnazione con gli array (o liste) è delicata:

```

1 import numpy as np
2
3 a = np.array([1, 2, 3, 4])
4 print(f"array iniziale: {a}, id: {id(a)}")
5
6 b = a
7 b[0] = 7
8
9 print(f"array iniziale: {a}, id: {id(a)}")
10 print(f"array finale : {b}, id: {id(b)}")
11
12 #usiamo ora copy invece che l'assegnazione
13
14 a = np.array([1, 2, 3, 4])
15 print(f"array iniziale: {a}, id: {id(a)}")
16
17 b = np.copy(a)
18 b[0] = 7
19
20 print(f"array iniziale: {a}, id: {id(a)}")
21 print(f"array finale : {b}, id: {id(b)}")
22
23 [Output]
24 array iniziale: [1 2 3 4], id: 2226551695088
25 array iniziale: [7 2 3 4], id: 2226551695088
26 array finale : [7 2 3 4], id: 2226551695088
27 array iniziale: [1 2 3 4], id: 2226573280912
28 array iniziale: [1 2 3 4], id: 2226573280912
29 array finale : [7 2 3 4], id: 2226578310224

```

Come vedete se usiamo l'operato di assegnazione anche l'array iniziale cambia poiché sia a che b sono riferiti allo stesso indirizzo di memoria, mentre usando la funzione "copy" ora il secondo array ha un diverso indirizzo e quindi il problema non si pone più.

## 6.7 Esercizi

Ora che grazie agli array abbiamo un po' più di carne al fuoco voglio proporvi un paio di esercizi da fare da voi per prendere familiarità con questi oggetti, o strutture dati volendo. In basso ci sarà anche la soluzione che gradirei non guardaste prima di aver fatto almeno un tentativo. Ovviamente per fare questi esercizi non dovete usare informazioni ulteriori a quelle apprese fin'ora.

1. Verificare che anche con le liste l'operazione di assegnazione fa coincidere gli indirizzi di memoria.
2. Creare la matrice identità  $I_{3 \times 3}$  e verificare che, dato un vettore  $v$  a vostra scelta,  $v^T I v$  sia uguale a "sum(v\*\*2)".
3. Creare una matrice  $L_{4 \times 4}$  tutta nulla ma la cui diagonale sia: "-1, 1, 1, 1" e vedere cosa succede facendo  $v^T L v$ .
4. Data una matrice di zeri  $4 \times 2$  (4 righe, 2 colonne), riempite la colonna di sinistra e poi invertite le colonne.
5. Dato un linspace (e.g. tra 0 e 20) creare due nuovi array con solo i numeri pari uno e solo i dispari l'altro.
6. Dato un array a vostro piacimento, creare una maschera, con una o più condizioni sempre a vostro gusto.
7. Creare una matrice che contenga sulle colonne le tabelline da 0 a 10 (non banale da fare in poche righe).

Ecco la mia soluzione di questi esercizi. non riporto l'output per brevità.

Primo:

```
1 a = [1, 2, 3, 4]
2 print(f"array iniziale: {a}, id: {id(a)}")
3
4 b = a
5 b[0] = 7
6
7 print(f"array iniziale: {a}, id: {id(a)}")
8 print(f"array finale : {b}, id: {id(b)}")
```

Secondo:

```
1 import numpy as np
2
3 I = np.zeros((3, 3))
4 idx = [0, 1, 2] # lista degli indici
5
6 I[idx, idx] = 1
7
8 v = np.array([4, 8, 2]) # vettore a caso
9 # calcolo prodotto scalare
10 print(v.T @ I @ v)
11 print(sum(v**2))
```

Terzo:

```
1 import numpy as np
2
3 L = np.zeros((4, 4))
4 idx = [0, 1, 2, 3] # lista degli indici
5
6 L[idx, idx] = [-1, 1, 1, 1]
7
8 v = np.array([4, 8, 2, 1]) # vettore a caso
9 print(v.T @ L @ v)
10 v = np.array([9, 0, 2, 1]) # vettore a caso
11 print(v.T @ L @ v)
```

Quarto:

```
1 import numpy as np
2
3 A = np.zeros((4, 2))
4 A[:, 0] = 1
5 print(A)
6 # faccio lo swap cambiando l'ordine degli indici
7 print(A[:, [1, 0]])
```

Quinto:

```
1 import numpy as np
2
3 x = np.linspace(0, 20, 21, dtype=int)
4
5 pari = x[x%2 == 0] # maschera per i pari
6 disp = x[x%2 == 1] # maschera per i dispari
7 print(pari)
8 print(disp)
```

Sesto:

```
1 import numpy as np
2
3 x = np.linspace(0, 20, 21, dtype=int)
4
5 # maschera con due condizioni
6 mask = (x > 5) & (x < 15)
7 # "&" per and mentre "|" per or
8 print(x[mask])
```

Settimo:

```
1 import numpy as np
2
3 x = np.linspace(0, 10, 11, dtype=int)
4 # trasformo il vettore in una matrice N x 1
5 y = x[:, None]
6 print(x)
7 print(y)
8 print(x*y)
```



## 7 Terza lezione

### 7.1 Le funzioni

Don't repeat yourself: è questa la logica delle funzioni. Le funzioni sono frammenti di codici, atti a ripetere sempre lo stesso tipo di operazioni con diversi valori dei parametri in input a seconda delle esigenze. Come al solito vediamo degli esempi:

```
1 def area(a, b):
2     """
3     restituisce l'area del rettangolo
4     di lati a e b
5     """
6     A = a*b #calcolo dell'area
7     return A
8
9 #chiamiamo la funzione e stampiamo subito il risultato
10 print(area(3, 4))
11 print(area(2, 5))
12
13 """
14 Se la funzione non restituisce nulla
15 ma esegue solo un pezzo di codice,
16 si parla propriamente di procedura
17 e il valore restituito e' None.
18 """
19 def procedura(a):
20     a = a+1
21
22 print(procedura(2))
23
24 """
25 Volendo si possono creare anche funzioni
26 che non hanno valori in ingresso:
27 """
28 def pigreco():
29     return 3.14
30 print(pigreco())
31
32 [Output]
33 12
34 10
35 None
36 3.14
```

Portiamo all'attenzione due fatti importanti:

- È fondamentale in Python che il corpo della funzione sia indentato, per seguire un raggruppamento logico del codice. Lo stesso vale per altri costrutti che vedremo tra poco. Insomma le parti indentate del codice devono essere logicamente connesse.
- Definendo degli argomenti per una funzione si creano delle variabili "locali", il cui nome non influenza tutto quello che c'è fuori dalla funzione stessa. Ad esempio, per la funzione area abbiamo definito una variabile "A", ma posso tranquillamente definire una nuova variabile "A" al di fuori della funzione e non avrei problemi di sovrascrittura.

Abbiamo visto che le funzioni possono prendere dei parametri o anche nessun parametro, quindi la domanda che sorge spontanea è: ne possono prendere infiniti? La risposta è sì ma prima di vederlo facciamo una piccola deviazione e parliamo delle istruzioni di controllo.

### 7.2 Istruzioni di controllo

Per istruzioni di controllo si intendono dei comandi che modificano il flusso di compilazione di un programma in base a determinati confronti e/o controlli su certe variabili. Ci sono casi in cui il computer deve fare cose diverse a seconda degli input o fare la stessa cosa un certo numero di volte fino a che una certa condizione sia o non sia soddisfatta.

#### 7.2.1 Espressioni condizionali: if, else, elif

Tramite l'istruzione if effettuiamo un confronto/controllo. Se il risultato è vero il programma esegue la porzione di codice immediatamente sotto-indentata. In caso contrario, l'istruzione else prende il controllo e il programma esegue la porzione di codice indentata sotto quest'ultima. Se l'istruzione else non è presente e il controllo

avvenuto con l'if risultasse falso, il programma semplicemente non fa niente. Vediamo il caso classico del valore assoluto:

```
1 def assoluto(x):
2     """
3     restituisce il valore assoluto di un numero
4     """
5     # se vero restituisci x
6     if x >= 0:
7         return x
8     #altrimenti restituisci -x
9     else:
10        return -x
11
12 print(assoluto(3))
13 print(assoluto(-3))
14
15 [Output]
16 3
17 3
```

È possibile aggiungere delle coppie if/else in cascata tramite il comando "elif", che è identico semanticamente a "else if"; per esempio:

```
1 def segno(x):
2     """
3     funzione per capire il segno di un numero
4     """
5     #se vero ....
6     if x > 0:
7         return 'Positivo'
8     #se invece ....
9     elif x == 0:
10        return 'Nullo'
11    #altrimenti ....
12    else:
13        return 'Negativo'
14
15 print(segno(5))
16 print(segno(0))
17 print(segno(-4))
18
19 [Output]
20 Positivo
21 Nullo
22 Negativo
```

### 7.2.2 Cicli: while, for

Partiamo con i cicli while: essi sono porzioni di codice che iterano le stesse operazioni fino a che una certa condizione risulta essere verificata:

```
1 def fattoriale(n):
2     """
3     Restituisce il fattoriale di un numero
4     """
5     R = 1
6     #finche' e' vero fai ...
7     while n > 1:
8         R *= n
9         n -= 1
10    return R
11
12 print(fattoriale(5))
13
14 [Output]
15 120
```

Un'accortezza da porre con i cicli while è verificare che effettivamente la condizione inserita si verifichi altrimenti il ciclo non si interrompe e va avanti per sempre, ed è molto molto tempo, della serie che fa prima a decadere il protone.

Passando ai cicli for invece essi ripetono una certa azione finché un contatore non raggiunge il massimo. Vediamo come implementare il fattoriale con questo ciclo:

```

1 def fattoriale(n):
2     """
3     restituisce il fattoriale di un numero
4     """
5     R = 1
6     #finche' i non arriva ad n fai ...
7     for i in range(1, n+1):
8         R = R*i
9     return R
10
11 print(fattoriale(5))
12
13 [Output]
14 120

```

Abbiamo quindi introdotto una variabile ausiliaria "i" utilizzata in questo contesto come contatore, cioè come variabile che tiene il conto del numero di cicli effettuati. Nel caso in esame, stiamo dicendo tramite l'istruzione for che la variabile "i" deve variare all'interno della lista  $\text{range}(1, n+1) = [1, 2, \dots, n]$ . Il programma effettua l'operazione  $R = R*i$  per tutti i valori possibili che i assume in questa lista, nell'ordine. Da notare il comando range che crea una lista sulla quale iterare, ma noi abbiamo visto già le liste e gli array e abbiamo visto che presentano alcune somiglianze, un'altra somiglianza da far vedere è che entrambi sono 'iterabili' e quindi possiamo iterarci sopra:

```

1 import numpy as np
2
3 def trova_pari(array):
4     """
5     restituisce un array contenente solo
6     i numeri pari dell'array di partenza
7     """
8     R = np.array([]) #array da riempire
9     #per ogni elemento in array fai ...
10    for elem in array:
11        if elem%2 == 0:
12            R = np.append(R, elem)
13    return R
14
15 a = np.array([i for i in range(0, 11)])
16 """
17 il precedente e' un modo piu' conciso di scrivere:
18 a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
19 """
20 print(a)
21 print(trova_pari(a))
22
23 [Output]
24 [ 0  1  2  3  4  5  6  7  8  9 10]
25 [ 0.  2.  4.  6.  8. 10.]

```

In questo esempio abbiamo utilizzato gli array ma si potrebbe senza problemi rifare tutto con le liste. Altri due comandi interessanti per quanto riguarda i cicli sono: enumerate e zip.

enumerate:

```

1 import numpy as np
2
3 #creiamo un array
4 array = np.linspace(0, 1, 5)
5
6 """
7 in questo modo posso iterare contemporaneamente
8 sia sugli indici sia sugli elementi dell'array
9 """
10 for index, elem in enumerate(array):
11     print(index, elem)
12
13 [Output]
14 0 0.0
15 1 0.25
16 2 0.5
17 3 0.75
18 4 1.0

```

zip:

```

1 import numpy as np
2

```

```

3 #creiamo tre un array
4 array1 = np.linspace(0, 1, 5)
5 array2 = np.linspace(1, 2, 5)
6 array3 = np.linspace(2, 3, 5)
7 """
8 in questo modo posso iterare contemporaneamente
9 sugli elementi di tutti gli array
10 """
11 for a1, a2, a3 in zip(array1, array2, array3):
12     print(a1, a2, a3)
13
14 [Output]
15 0.0 1.0 2.0
16 0.25 1.25 2.25
17 0.5 1.5 2.5
18 0.75 1.75 2.75
19 1.0 2.0 3.0

```

Anche qui come le funzioni è necessario indentare.

### 7.3 Ancora funzioni

Dopo questa digressione torniamo alle funzioni, abbiamo detto che una funzione può prendere infiniti argomenti, ma dal punto di vista pratico come lo implementiamo, in un modo semi decente? Una risposta sarebbe quella di passare alla funzione non delle singole variabili ma un array o una lista, cosa che si può fare tranquillamente, e lavorare poi all'interno della funzione con gli indici per utilizzare i vari elementi dell'array, o della lista, o ciclarci sopra. Un altro modo per farlo è usare: \*args (args è un nome di default, potremmo chiamarlo mimmo):

```

1 def molt(*numeri):
2     """
3     restituisce il prodotto di n numeri
4     """
5     R = 1
6     for numero in numeri:
7         R *= numero
8     return R
9
10 print(molt(2, 7, 10, 11, 42))
11 print(molt(5, 5))
12 print(molt(10, 10, 2))
13
14 [Output]
15 64680
16 25
17 200

```

L'esempio appena visto non è altro che la funzione fattoriale di prima leggermente modificata e che non prende più in input una sequenza crescente di numeri. I parametri vengono passati come una tupla e in questo caso il simbolo "\*" viene definito operatore di unpacking proprio perché "spacchetta" tutte le variabili che vengono passate alla funzione.

### 7.4 funzioni lambda

Esiste un particolare tipo di funzioni in python, le cosiddette funzioni lambda, sono fondamentalmente uguali alle funzioni dichiarate dall'utente ma devono avere una singola espressione, quindi niente calcoli complicati nel mezzo. La sintassi è: lambda argomenti : espressione. Sono in genere usate come anonime, magari da passare ad altre funzioni ad esempio se bisogna fittare con una retta, (vedere lezione successiva) posso usare curve\_fit così: curve\_fit(lambda x, m, q : x\*m + q, x, y, ...); ma ciò non vieta comunque di dargli un nome.

```

1 f = lambda x : 3*x
2 print(f(2))
3
4 h = lambda x, y, z : x*y + z
5 print(h(2, 3, 4))
6
7 [Output]
8 6
9 10

```

Ovviamente esse possono anche essere usate all'interno di una funzione magari come ciò che la funzione ritorna, il che vuol dire che la funzione restituirà un'altra funzione.

```

1 def g(x):
2     '''restituisce potenza x-esima di y
3     '''
4     return lambda y : y**x
5
6 G = g(3) # potenza cubica
7 print(G(4))
8
9 #=====
10 #=====
11
12 def m(f, y, x):
13     '''passo una funzione ad una funzione
14     '''
15     return f(y) - x
16
17 print(m(lambda x : x**2, 5, 1))
18
19 [Output]
20 64
21 24

```

## 7.5 Grafici

Fare un grafico è un modo pratico e comodo di visualizzare dei dati, qualsiasi sia la loro provenienza. Capita spesso che i dati siano su dei file (per i nostri scopi in genere file .txt o .csv ) e che i file siano organizzati a colonne:

```

1 #t[s] x[m]
2 1      1
3 2      4
4 3      9
5 4     16
6 5     25
7 6     36

```

Per leggerli:

```

1 import numpy as np
2
3 #Leggiamo da un file di testo classico
4 path = 'dati.txt'
5 dati1, dati2 = np.loadtxt(path, unpack=True)
6 """
7 unpack=True serve proprio a dire che vogliamo che
8 dati1 contenga la prima colonna e dati2 la seconda
9 La prima riga avendo il cancelletto verra' saltata
10 """
11
12 #se vogliamo invece che venga letto tutto come una matrice scriviamo:
13 path = 'dati.txt'
14 dati = np.loadtxt(path) # sarebbe unpack = False
15 #dati sara' nella fattispecie una matrice con due colonne e 6 righe
16
17
18 #leggere da file.csv
19 path = 'dati.csv'
20 dati1, dati2 = np.loadtxt(path, usecols=[0,1], skiprows=1, delimiter=',', unpack=True)
21 """
22 si capisce senza troppa fatica che usecols indica le colonne che vogliamo leggere
23 skiprows il numero di righe da saltare e delimiter indica il carattere che separa le colonne
24 """

```

Un interessante tipo di file sono i file con estensione ".npy" comodi se i dati sono di dimensioni elevate:

```

1 import time
2 import numpy as np
3
4 x = np.linspace(0, 1, int(2e6)) # dati
5
6 #=====
7 # file txt
8 #=====
9
10 # salviamo su file
11 start = time.time()

```

```

12 path = r'c:\Users\franc\Desktop\dati.txt'
13 f = open(path, 'w')
14 for i in x:
15     f.write(f'{i} \n')
16 f.close()
17 end = time.time() - start
18
19 print(f"tempo di scrittura txt: {end} s")
20
21 #legghiamo da file
22 start = time.time()
23 X = np.loadtxt(path, unpack=True)
24 end = time.time() - start
25
26 print(f"tempo di lettura txt: {end} s")
27
28 #=====
29 # file npy
30 #=====
31
32 # salviamo su file
33 start = time.time()
34 path = r'c:\Users\franc\Desktop\dati.npy'
35 np.save(path, x)
36 end = time.time() - start
37
38 print(f"tempo di scrittura npy: {end} s")
39
40 #legghiamo da file
41 start = time.time()
42 X = np.load(path, allow_pickle='TRUE')
43 end = time.time() - start
44
45 print(f"tempo di lettura npy: {end} s")
46
47 [Output]
48 tempo di scrittura txt: 5.610752582550049 s
49 tempo di lettura txt: 9.489601373672485 s
50 tempo di scrittura npy: 0.016022205352783203 s
51 tempo di lettura npy: 0.015637636184692383 s

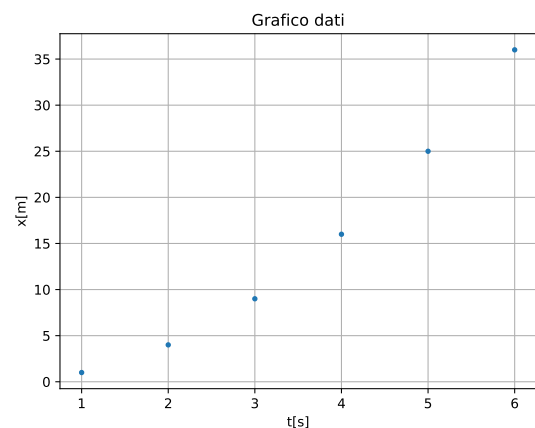
```

Abbiamo usato la libreria time per misurare il tempo, "time.time()" restituisce i numeri di secondi trascorsi dall'inizio dell'epoca unix (1 gennaio 1970). Come vediamo la differenza di tempi è abissale. Passiamo ora alla creazione di un grafico. Creare ora un grafico è semplice grazie all'utilizzo della libreria matplotlib:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #Leggiamo da un file di testo classico
5 path = 'dati.txt'
6 dati1, dati2 = np.loadtxt(path, unpack=True)
7
8 plt.figure(1) #creiamo la figura
9
10 #titolo
11 plt.title('Grafico dati')
12 #nomi degli assi
13 plt.xlabel('t[s]')
14 plt.ylabel('x[m]')
15 #plot dei dati
16 plt.plot(dati1, dati2, marker='.', linestyle='')
17 #aggiungiamo una griglia
18 plt.grid()
19 #comando per mostrare a schermo il grafico
20 plt.show()

```



Commentiamo un attimo quanto fatto: dopo aver letto i dati abbiamo fatto il grafico mettendo sull'asse delle ascisse la colonna del tempo e su quello delle ordinate la colonna dello spazio; se all'interno del comando "plt.plot(...)" scambiassimo l'ordine di dati1 e dati2 all'ora gli assi si invertirebbero, non avremmo più x(t) ma t(x). Inoltre il comando "marker='.'" sta a significare che il simbolo che rappresenta il dato deve essere un punto; mentre il comando "linestyle=''" significa che non vogliamo che i punti siano uniti da una linea (linestyle='-' dà una linea, linestyle='- -' dà una linea tratteggiata).

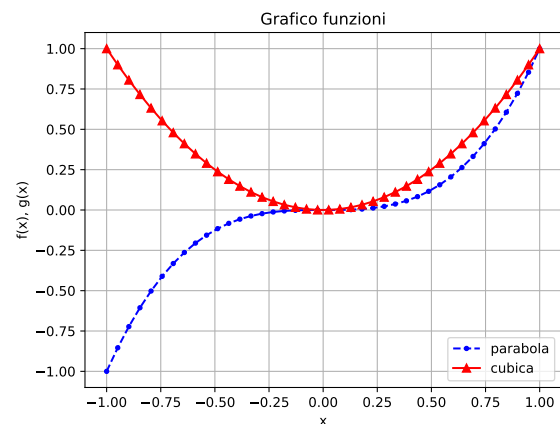
Se invece volessimo graficare una funzione o più definite da codice? Anche qui i comandi sono analoghi:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(x):
5     """
6     restituisce il cubo di un numero
7     """
8     return x**3
9
10 def g(x):
11     """
12     restituisce il quadrato di un numero
13     """
14     return x**2
15
16 #array di numeri equispaziati nel range [-1,1] usiamo:
17 x = np.linspace(-1, 1, 40)
18
19 plt.figure(1) #creiamo la figura
20
21 #titolo
22 plt.title('Grafico funzioni')
23 #nomi degli assi
24 plt.xlabel('x')
25 plt.ylabel('f(x), g(x)')
26 #plot dei dati
27 plt.plot(x, f(x), marker='.', linestyle='--', color='blue', label='parabola')
28 plt.plot(x, g(x), marker='^', linestyle='-', color='red', label='cubica')
29 #aggiungiamo una leggenda
30 plt.legend(loc='best')
31 #aggiungiamo una griglia
32 plt.grid()
33 #comando per mostrare a schermo il grafico
34 plt.show()

```

Notare che per distinguere le due funzioni oltre al "marker" e al "linestyle" abbiamo aggiunto il comando "color" per dare un colore e il comando "label" che assegna un'etichetta poi visibile nella legenda (loc='best' indica che Python la mette dove ritiene più consono, in modo che non rischi magari di coprire porzioni di grafico). Ovviamente è consigliata una lettura della documentazione per conoscere tutti gli altri comandi possibili per migliorare/abbellire il grafico da adde alle funzioni già presenti. Altre funzioni utili possono essere: "plt.axis(...)" che imposta il range da visualizzare su entrambi gli assi; il comando "plt.xscale(...)" che permette di fare i grafici con una scala, magari logaritmica o altro sull'asse x (analogo sarà sulle y mutatis mutandis).



Ultima menzione da fare sono gli istogrammi:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 plt.figure(1)
5 plt.title('grafico a barre')
6 plt.xlabel('valore')
7 plt.ylabel('conteggi')
8 # Sull'asse x utilizziamo un array di 10 punti equispaziati.
9 x = np.linspace(1,10,10)
10 # Sull'asse y abbiamo, ad esempio, il seguente set di dati:
11 y = np.array([2.54, 4.78, 1.13, 3.68, 5.79, 7.80, 5.4, 3.7, 9.0, 6.6])
12
13 # Il comando per la creazione dell'istogramma corrispondente e':
14 plt.bar(x, y, align='center')
15
16 plt.figure(2)
17 plt.title('istogramma di una distribuzione gaussiana')
18 plt.xlabel('x')
19 plt.ylabel('p(x)')

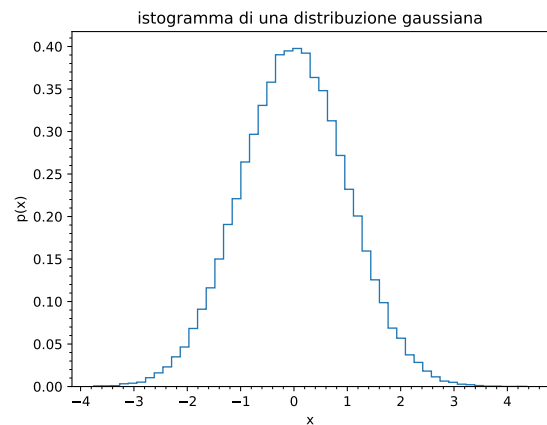
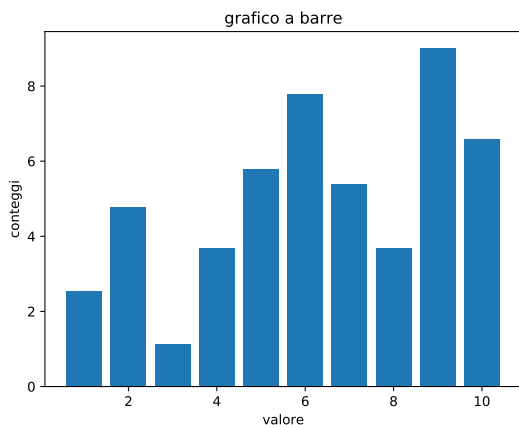
```

```

20 """
21 """
22 lista di numeri distribuiti gaussianamente con media 0 e varianza 1
23 si usa l'underscore nel for poiche' non serve usare
24 un'altra variabile. Avremmo potuto scrivere for i ...
25 ma la i non sarebbe comparsa da nessun'altra parte
26 sarebbe stato uno spreco
27 """
28 z = [np.random.normal(0, 1) for _ in range(int(1e5))]
29 plt.hist(z, bins=50, density=True, histtype='step')
30 plt.minorticks_on() # tick piccoli sugli assi
31
32 plt.show()

```

Piccolo appunto che bisogna fare, nel caso di "plt.hist()" bisogna stare attenti perché il numero di bin va scelto con cura (qui abbiamo scritto 50 sulla fiducia).



## 7.6 Standard input

È (a proposito, la È si fa premendo Alt+0200 sul tastierino, quantomeno su windows) inoltre possibile dare la codice che abbiamo scritto degli input da shell. Esistono due modi per farlo: l'uso della funzione "input()", oppure utilizzare "argparse" per dare al codice ciò che serve direttamente da linea di comando su shell, ad esempio se fossimo su una partizione linux senza un editor stile pyzo. Cominciamo con la prima possibilità:

```

1 """
2 Programma per calcolare il trinagolo di tartaglia
3 """
4 import numpy as np
5
6 # leggo da input un valore e lo rendo intero
7 # la stringa verra stampata su shell
8 n = int(input("Ordine del triangolo: "))
9 a = np.zeros((n, n), dtype=int) # matrice per i coefficienti
10
11 # calocolo i coefficienti del trinagolo
12 a[0,0] = 1
13 for i in range(1, n):
14     a[i, 0] = 1
15     for j in range(1, i):
16         a[i, j] = a[i-1, j-1] + a[i-1, j]
17     a[i,i] = 1
18
19 # stampo a schermo
20 for i in range(n):
21     for j in range(i+1):
22         # solo per fare la forma a piramide
23         if j == 0 : # non funziona con numeri a due cifre
24             print(*[" "]*(n-i),a[i, j], end='')
25         else:
26             print("",a[i, j], end='')
27     print()
28
29 [Output]
30 Ordine del triangolo: 5
31     1
32    1 1

```



```

33     1 2 1
34     1 3 3 1
35     1 4 6 4 1

```

Vediamo ora come usare argparse. Ora però il codice è più comodo eseguirlo su una shell, che sia quella di anaconda o quella della vostra distro linux è uguale. Le modifiche al codice sono veramente poche:

```

1 """
2 Programma per calcolare il trinagolo di tartaglia
3 """
4 import argparse
5 import numpy as np
6
7 description='Programma per calcolare il trinagolo di tartaglia leggendo le informazioni da
8   linea di comando'
9 # descrizione accessibile con -h su shell
10 parser = argparse.ArgumentParser(description=description)
11 parser.add_argument('dim', help='Dimensione della matrice, ovvero potenza del binomio')
12 args = parser.parse_args()
13 n = int(args.dim) # accedo alla variabile tramite il nome messo a linea 10
14
15 a = np.zeros((n, n), dtype=int) # matrice per i coefficienti
16
17 # calcolo i coefficienti del trinagolo
18 a[0,0] = 1
19 for i in range(1, n):
20     a[i, 0] = 1
21     for j in range(1, i):
22         a[i, j] = a[i-1, j-1] + a[i-1, j]
23     a[i,i] = 1
24
25 # stampo a schermo
26 for i in range(n):
27     for j in range(i+1):
28         # solo per fare la forma a piramide
29         if j == 0 : # non funziona con numeri a due cifre
30             print(*[" "]*(n-i),a[i, j], end='')
31         else:
32             print(" ",a[i, j], end='')
33     print()

```

Vi metto uno screen della shell per capire cosa è successo (un po' sgranata ma pazienza):

```

C:\Windows\System32\Windc (base) PS C:\Users\franc\Desktop\Nuova cartella\3 Terza Lezione\codicil3> python tartaglia_argparse.py -h
usage: tartaglia_argparse.py [-h] dim

Programma per calcolare il trinagolo di tartaglia leggendo le informazioni da linea di comando

positional arguments:
  dim                Dimensione della matrice, ovvero potenza del binomio

optional arguments:
  -h, --help          show this help message and exit
(base) PS C:\Users\franc\Desktop\Nuova cartella\3 Terza Lezione\codicil3> python tartaglia_argparse.py 5
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
(base) PS C:\Users\franc\Desktop\Nuova cartella\3 Terza Lezione\codicil3>

```

## 7.7 Prestazioni

Avevamo accennato al fatto che Python fosse lento ma che utilizzando le librerie si potesse un po' migliorare le prestazioni, vediamo un esempio:

```

1 import time
2 import numpy as np
3
4 #inizio a misurare il tempo
5 start = time.time()
6
7
8 a1 = 0 #variabile che conterra' il risultato
9 N = int(5e6) # numero di iterazioni da fare = 5 x 10**6
10
11 #faccio il conto a 'mano'
12 for i in range(N):

```

```

13     a1 += np.sqrt(i)
14
15 #finisco di misurare il tempo
16 end = time.time()-start
17
18 print(end)
19
20 #inizio a misurare il tempo
21 start = time.time()
22
23 #stesso conto ma fatto tramite le librerie di python
24 a2 = sum(np.sqrt(np.arange(N)))
25
26 #finisco di misurare il tempo
27 end = time.time()-start
28
29 #sperabilmente sara' minore del tempo impiegato prima
30 print(end)
31
32 [Output]
33 11.588378429412842
34 0.8475463390350342

```

Vediamo che quindi usando le funzioni di numpy, (`np.arange`) e le funzioni della libreria standard di Python (`sum`), è possibile fare lo stesso conto in un tempo molto minore che tramite un ciclo `for`. Questo perché le librerie non sono totalmente in Python ma in molta parte in C e/o fortran.

## 7.8 Gestione errori

È molto facile scrivere codice che produca errore, magari perché distrattamente ci siamo dimenticati qualcosa o magari qualcosa è stato implementato male. Esiste un costrutto che ci permette di gestire gli errori in maniera tranquilla diciamo. Facciamo un semplice esempio:

```

1 a = 0
2 b = 1/a
3 print(b)
4
5 [Output]
6 Traceback (most recent call last):
7   File "<tmp 1>", line 5, in <module>
8     b = 1/a
9 ZeroDivisionError: division by zero

```

Abbiamo fatto una cosa molto brutta, nemmeno Dio può dividere per zero (al più possiamo appellarci alla censura cosmica e mettere un'orizzonte a vestire la divisione per zero) e quindi il computer ci dà errore. Possiamo aggirare il problema, evitando così il second impact, in due modi diciamo:

### 7.8.1 Try e except

Possiamo utilizzare il costrutto `try except` dicendo al computer: prova a fare la divisione e, sia mai funziona, se questa però dà errore, e l'errore è `"ZeroDivisionError"` allora assegna a `b` un altro valore. In questo modo eventuali istruzioni presenti dopo vengono eseguite e il codice non si arresta.

```

1 a = 0
2 try :
3     b = 1/a
4 except ZeroDivisionError:
5     b = 1
6
7 print(b)
8
9 [Output]
10 1

```

Anche qui è fondamentale indentare il blocco delle istruzioni.

### 7.8.2 Raise Exception

Mettiamo il caso in cui ci siano operazioni da fare in cui il valore della variabile `"b"` è importante, quindi sarebbe meglio interrompere il flusso del codice perché con un dato valore il risultato finale sarebbe poco sensato. Si può fare il controllo del valore e sollevare un'eccezione per fermare il codice.

```

1 """
2 leggo un valore da shell
3 uso del comando try per evitare che venga letto
4 qualcosa che non sia un numero: e.g. una stringa
5 """
6 try:
7     b = int(input('scegliere un valore:'))
8 except ValueError:
9     print('hai digitato qualcosa diverso da un numero, per favore ridigitare')
10    b = int(input('scegliere un valore:'))
11
12 #se si sbaglia a digitare di nuovo il codice si arresta per ValueError
13
14 #controllo se e' possibile proseguire
15 if b > 7 :
16     #se vero si blocca il codice sollevando l'eccezione
17     messaggio_di_errore = 'il valore scelto risulta insensato in quanto nulla supera 7, misura
18     massima di ogni cosa'
19     raise Exception(messaggio_di_errore)
20
21 [Output]
22 8
23 Traceback (most recent call last):
24   File "<tmp 1>", line 18, in <module>
25     raise Exception(messaggio_di_errore)
26 Exception: il valore scelto risulta insensato in quanto nulla supera 7, misura massima di ogni
27     cosa

```

## 7.9 Esercizi

Anche qui voglio lasciarvi qualche esercizio per farvi prendere confidenza con quanto imparato finora:

1. Scrivere una funzione che dati lunghezza del lato e il numero degli stessi per un poligono regolare ne calcoli l'area "Area(l, n)" e calcolare Area(3, n) + Area(4, n) e confrontarla con Area(5, n) per diversi valori di n>=3.
2. Fare con un ciclo più plot su uno stesso grafico, dove la funzione deve dipendere dalla variabile su cui si cicla (e.g.  $x^i$  con x un array di un certo range e i la variabile del ciclo).
3. Stessa cosa di sopra ma ora ogni curva deve avere un colore e uno linestyle diverso e una legenda.
4. Creare una funzione che legga da input un numero intero con la condizione che esso sia maggiore di zero e che dia la possibilità di inserirlo nuovamente finché la condizione non è verificata.
5. Rifare il primo esercizio usando una funzione lambda.
6. Sovrapporre i plot di un istogramma e della funzione di distribuzione associata, a vostra scelta, e aggiungere al grafico tutte le bellurie del caso.
7. Scrivere una funzione che dato un array ne restituisca la media e la deviazione standard.

Formule utili:

Area poligoni regolari:

$$A = \frac{1}{2}(nl) \frac{l \cot(\pi/n)}{2}$$

media e deviazione standard:

$$\mu = \sum_i^n x_i \quad \sigma = \sqrt{\frac{1}{n(n-1)} \sum_i^n (x_i - \mu)^2}$$

Ecco la mia soluzione di questi esercizi. non riporto l'output per brevità.

Primo:

```

1 def Area(l, n):
2     '''
3     funzione per calcolare l'area di un poligono
4     regolare di lato l e numero lati n
5     '''
6     a = 1/2 * l/np.tan(np.pi/n) # apotema
7     p = n*l                      # perimetro
8     A = p*a/2                   # area
9     return A
10
11 l = [3, 4, 5]                  # dimensioni dei lati, (terna pitagorica)
12 n = np.arange(4, 12)          # numero di lati dei poligoni
13
14 A3, A4, A5 = [], [], []
15 # liste in cui ci saranno le aree dei poligoni di lato rispettivamente 3, 4, 5
16

```

```

17 for n_i in n:      # loop sul numero di lati
18     for l_i in l: # lup sulla dimesione
19
20         A0 = Area(l_i, n_i) # calcolo dell'area
21
22         if l_i == 3: # conservo a seconda dei lati
23             A3.append(A0)
24         elif l_i == 4:
25             A4.append(A0)
26         elif l_i == 5:
27             A5.append(A0)
28
29
30 for i in range(len(n)):
31     print(f'A3 + A4 = {A3[i]+A4[i]:.3f}')
32     print(f'    A5    = {A5[i]:.3f}')

```

Secondo:

```

1 power = [0.5, 1, 2]      # potenze
2 x = np.linspace(0, 1, 1000) # range sulle x
3
4 plt.figure(1)
5 for p in power:
6     plt.plot(x, x**p) # un plot alla volta sulla stessa figura
7
8 # bellurie
9 plt.grid()
10 plt.title("Esercizio 2")
11 plt.xlabel("x")
12 plt.ylabel("f(x)")
13 plt.show()

```

Terzo:

```

1 power = [0.5, 1, 2]      # potenze
2 color = ['k', 'r', 'b']  # colore di ogni curva
3 lnsty = ['-', '--', '-.'] # stile di ogni curva
4 label = [r'$\sqrt{x}$', r'$x$', r'$x^2$'] # nome della curva
5 x = np.linspace(0, 1, 1000)
6
7 plt.figure(1)
8 for p, c, ls, lb in zip(power, color, lnsty, label):
9     # un plot alla volta sulla stessa figura
10    plt.plot(x, x**p, c=c, linestyle=ls, label=lb)
11
12 #bellurie
13 plt.grid()
14 plt.title("Esercizio 3")
15 plt.xlabel("x")
16 plt.ylabel("f(x)")
17 plt.legend(loc='best')
18 plt.show()

```

Quarto:

```

1 def read():
2     '''
3     funzione che legge da input un numero con la condizione che esso
4     sia maggiore di zero e che dia la possibilita' di inserirlo
5     nuovamente finche' la condizione non e' verificata.
6     Volendo si puo' generalizzare il codice passando la condizione come input
7     '''
8
9     while True: # Il codice deve runnare finche' non inserisco un numero buono
10
11         try: # provo a leggere il numero e a renderlo intero
12             x = int(input("Iserisci un numero: "))
13
14         except ValueError: # se non riesco sollevo l'eccezione
15             print(f"Fra ti ho chiesto di mettere un numero") # messaggio di errore
16             continue # questo comando fa ripartire il ciclo da capo
17
18         if x > 0: # se la lettura e' andata a buon fine verifico la condizione
19             return x # se e' verificata ritorno il numero
20         else:
21             # altrimenti stampo un messaggio di errore
22             print("In numero inserito e' minore di zero, sceglierne un altro.")

```

```

23         continue # e faccio ripartire il ciclo da capo
24
25
26 x = read()
27 print(f"Il numero letto e': {x}")

```

Quinto:

```

1 Area = lambda l, n : (1/2 * 1/np.tan(np.pi/n))*(n*l)/2
2
3 l = [3, 4, 5] # dimensioni dei lati, (terna pitagorica)
4 n = np.arange(4, 12) # numero di lati dei poligoni
5
6 A3, A4, A5 = [], [], []
7 # liste in cui ci saranno le aree dei poligoni di lato rispettivamente 3, 4, 5
8
9 for n_i in n: # loop sul numero di lati
10     for l_i in l: # lup sulla dimensione
11
12         A0 = Area(l_i, n_i) # calcolo dell'area
13
14         if l_i == 3: # conservo a seconda dei lati
15             A3.append(A0)
16         elif l_i == 4:
17             A4.append(A0)
18         elif l_i == 5:
19             A5.append(A0)
20
21
22 for i in range(len(n)):
23     print(f'A3 + A4 = {A3[i]+A4[i]:.3f}')
24     print(f'    A5 = {A5[i]:.3f}')

```

Sesto:

```

1 # Gaussiana
2 m = 0
3 s = 1
4 z = [np.random.normal(m, s) for _ in range(int(1e5))]
5
6 # Plot dati
7 plt.figure(1)
8 plt.hist(z, bins=50, density=True, histtype='step', label='dati')
9 plt.grid()
10 plt.xlabel("x")
11 plt.ylabel("P(x)")
12 plt.title("Distribuzione gaussiana")
13
14 # Plot curva
15 x = np.linspace(-5*s + m, 5*s + m, 1000)
16 plt.plot(x, np.exp(-(x-m)**2 / (2*s**2))/np.sqrt(2*np.pi*s**2), 'b', label=f"N({m}, {s})")
17 plt.legend(loc='best')
18 plt.show()

```

Settimo:

```

1 def obs(x):
2     '''
3     funzione che dato un set di dati x in input
4     restituisce media e deviazione standard
5     '''
6     N = len(x)
7
8     media = sum(x)/N
9     varianza = sum((x - media)**2)/(N*(N-1))
10    dev_std = np.sqrt(varianza)
11
12    return media, dev_std
13
14 dati = np.array([0, 3, 5, 1, 5, 667, 2, 4, 9, 3, 33])
15
16 m, dm = obs(dati)
17 print(f"media = {m:.3f} +- {dm:.3f}")

```

## 8 Quarta lezione

### 8.1 Importare file Python

Abbiamo visto come utilizzare le librerie, tutto a partire dal comando `import`. Oltre alle librerie possiamo importare anche altri file Python scritti da noi, magari perchè in quel file è implementata una funzione che ci serve. Facciamo un esempio:

```
1 def f(x, n):
2     """
3     restituisce la potenza n-esima di un numero x
4     Parametri
5     -----
6     x, n : float
7
8     Return
9     -----
10    v : float
11        x**n
12    """
13
14    v = x**n
15
16    return v
17
18 if __name__ == '__main__':
19     #test
20     print(f(5, 2))
21
22 [Output]
23 25
```

Abbiamo questo codice che chiamiamo "elevamento.py" che ha implementato la funzione di elevamento a potenza e supponiamo di voler utilizzare questa funzione in un altro codice, possiamo farlo grazie ad `import`:

```
1 import elevamento
2
3 print(elevamento.f(3, 3))
4
5 [Output]
6 27
```

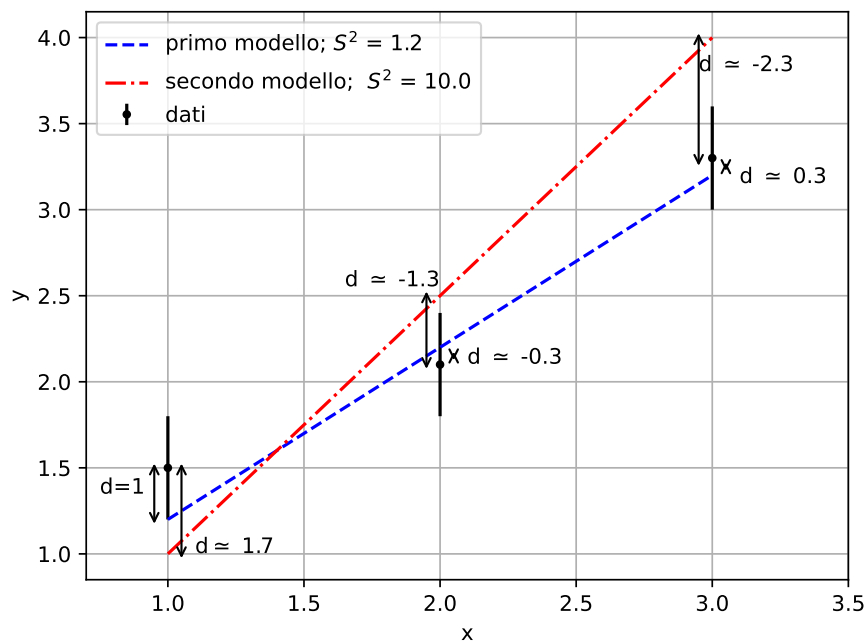
Notiamo nel codice iniziale la presenza dell' `if`, esso serve per far sì che tutto ciò che sia scritto sotto venga eseguito solo se il codice viene lanciato come 'main' appunto e non importato come modulo su un altro codice. In genere l'utilizzo di questa istruzione è buona norma quando si vuol scrivere un codice da importare altrove.

### 8.2 Fit

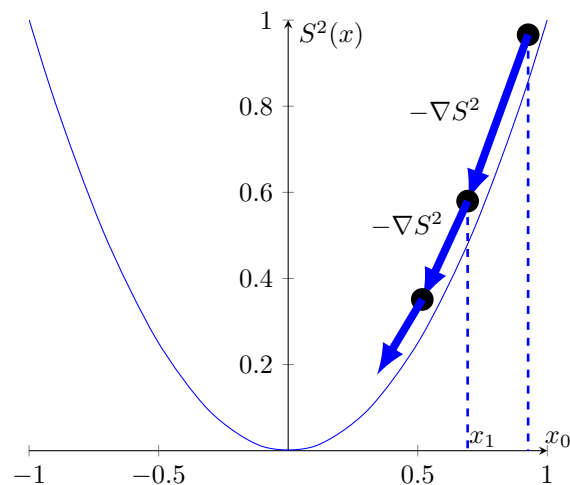
Nell'ambito della statistica un fit, cioè una regressione lineare o non che sia (dove la linearità è riferita ai parametri della funzione), è un metodo per trovare la funzione che meglio descrive l'andamento di alcuni dati. Nel caso di regressione lineare la procedura da eseguire non è troppo complicata, mentre per la regressione non lineare le cose si fanno parecchio complicate e si utilizzano algoritmi di ottimizzazione. Se noi abbiamo quindi un modello teorico che ci dice che un corpo cade con una legge oraria della forma  $y(t) = h_0 - \frac{1}{2}gt^2$ , grazie al fit possiamo trovare i valori dei parametri della legge oraria,  $h_0$  e  $g$ , che meglio adattano la curva ai dati (nella speranza che escano valori fisicamente sensati, dato che in genere i dati sono di origine sperimentale o simulativa). In ogni caso comunque l'idea di ciò che va fatto è trovare il minimo della seguente funzione:

$$S^2(\{\theta\}_j) = \sum_i \frac{(y_i - f(x_i; \{\theta\}_j))^2}{\sigma_{y_i}^2} \quad (1)$$

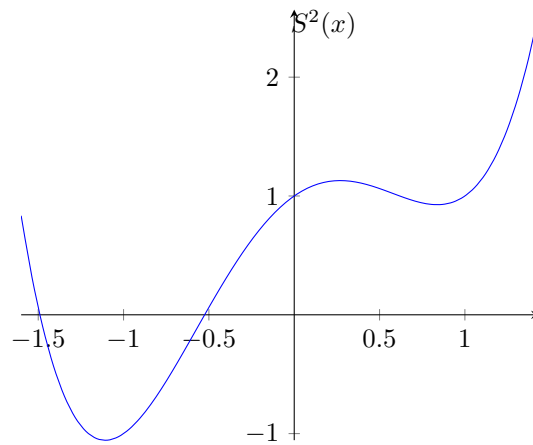
che nel caso in cui il termine dentro la somma sia distribuito in modo gaussiano allora la quantità  $S^2$  è distribuita come un chiquadro, e da qui si potrebbe fare tutta una discussione sulla significatività statistica di quello che andiamo a fare, che ovviamente noi non facciamo. Analizziamo un attimo questa formula:  $S^2$  è in linea di principio una funzione a molte variabili e che restituisce un numero reale. Il termine dentro la somma rappresenta la distanza tra valore del dato e valore della funzione in unità della barra d'errore del dato. Facciamo un esempio visivo per rendere più chiaro il concetto. Consideriamo giusto a titolo di esempio tre punti e due possibili rette che noi possiamo pensare che più o meno approssimino i dati.



Nel grafico  $d$  è proprio la distanza del modello dal dato in unità di barre di errore, e quindi la somma di tutte queste quantità elevate al quadrato è il valore di  $S^2$  che è riportato nella legenda. Notiamo quindi che effettivamente la retta che presenta un valore di  $S^2$  minore è quella che ad occhio meglio approssima i dati. Ora uno potrebbe pensare che quindi basta calcolare  $S^2$  su una griglia e vedere dove assume il valore più piccolo. Questo è un metodo abbastanza brute force e il linea di principio funziona, ma quello che in genere si fa è un po' diverso. In linea di principio per capire come funziona un'algoritmo di minimo basta pensare ad una pallina che cade in una ciotola. Precisiamo che si vuole fare tutta questa trattazione per far capire che la parte più delicata di questa procedura è scegliere quello che noi chiameremo nel codice "init" e che esso violentemente aggiusta o complica la nostra situazione. Consideriamo per semplicità, didattica e grafica, una funzione di una singola variabile.



Quel che noi facciamo è scegliere un  $x_0$ , (il nostro init) ed aggiornare questa posizione considerando la pendenza della funzione, che altro non sarebbe che la derivata della funzione che vogliamo considerare. Concedetemi, per maggiore generalità, si sostituire il termine derivata con il termine gradiente, indicato dal simbolo  $\nabla$ . Quindi quello che il codice fa è diciamo analogo ad una pallina che si muove sotto l'azione di un potenziale, che sarebbe  $S^2$  e la sua derivata, il suo gradiente, non è altro che la forza che la pallina sente. Facendo così troviamo una serie di  $x_i$  iterativamente, fino ad arrivare al minimo dove il gradiente, la forza esterna, è zero. Questo metodo è chiamato gradiente discendente. Finché abbiamo un solo minimo quindi va tutto bene. Lo troviamo senza problemi a prescindere da dove partiamo. Supponiamo ora una situazione più brutta:



In questo caso vediamo subito che abbiamo due punti in cui la derivata è nulla, quindi due minimi (questo sarebbe il caso di una regressione non lineare, a differenza di quella lineare di sopra, un solo minimo), ma quello a cui siamo interessati noi è il minimo assoluto. Se utilizzassimo il metodo precedente è facile vedere che se partiamo per esempio con  $x > 1$  ci incastriamo nel minimo locale. Le uniche zone buone sono soltanto quelle con  $x < 0$ . Vedete quindi che una piccola complicazione riduce di molto le nostre possibilità e dobbiamo quindi selezionare il nostro punto di partenza con delicatezza. Questo perché una volta arrivato al minimo locale la nostra "pallina" non ci arriva con una velocità come accadrebbe nella realtà e quindi non riesce a scavallare la collinetta. Fondamentalmente per migliorare la cosa dobbiamo spiegare al computer il concetto di inerzia e anche di attrito (se l'energia si conservasse la pallina oscillerebbe all'infinito e il codice non terminerebbe). Un esempio di ciò, chiamato gradiente discendente con momento, e anche di quanto visto sopra è disponibile in una delle appendici. Inoltre in questa stessa lezione andremo a vedere cosa fa effettivamente "curve\_fit" che è un pochino diverso. Un caso ancora peggiore lo vediamo adesso con un problema fisico, dove ora non abbiamo un semiasse da poter scegliere, ma solo una piccola e precisa zona, dovuto al fatto che ora non abbiamo due minimi ma molti di più. Prima di vedere il codice vediamo brevemente due grafici della quantità  $S^2$ , che con un po' di abuso di notazione chiamiamo chiquadro, nel caso di regressione lineare e non:

Chiquadro regressione lineare

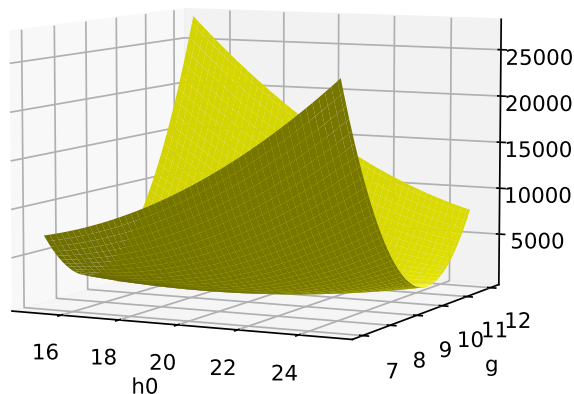


Figura 1: modello lineare  $y(t) = h_0 - \frac{1}{2}gt^2$ . Unico minimo, qualunque punto iniziale va bene.



## Chiquadro regressione non-lineare

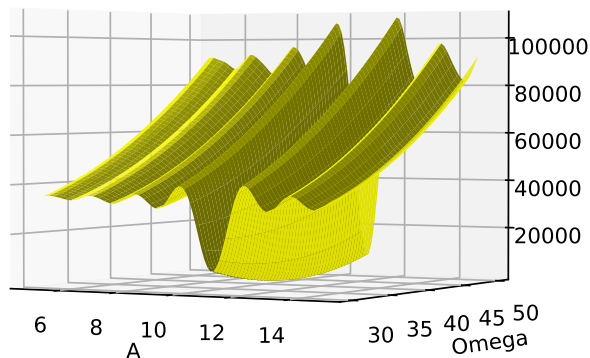


Figura 2: modello non lineare  $y(t) = A\cos(\omega t)$ . Tanti minimi locali bisogna stare attente a dove partire altrimenti l'algoritmo si blocca su soluzioni non fisiche. Solo una piccola regione va bene come valori iniziali.

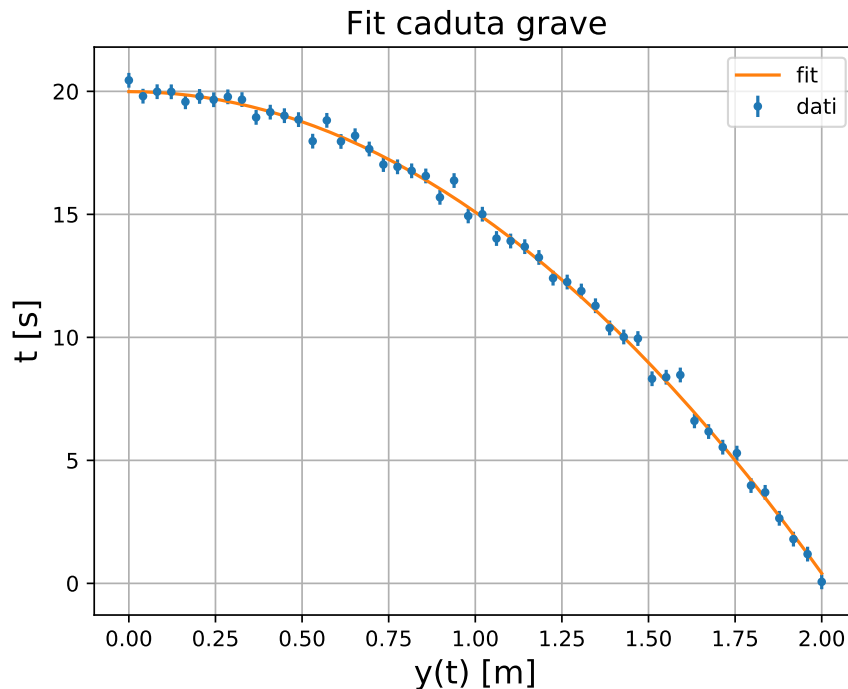
I codici per generare i grafici che abbiamo visto non sono riportati per brevità ma sono presenti nella cartella. Vediamo ora un semplice esempio di codice:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from scipy.optimize import curve_fit
4
5 def Legge_oraria(t, h0, g):
6     """
7     Restituisce la legge oraria di caduta
8     di un corpo che parte da altezza h0 e
9     con una velocità iniziale nulla
10    """
11    return h0 - 0.5*g*t**2
12
13 """
14 dati misurati:
15 xdata : fisicamente i tempi a cui osservo
16         la caduta del corpo non affetti da
17         errore
18 ydata : fisicamente la posizione del corpo
19         misurata a dati tempi xdata affetta
20         da errore
21 """
22
23 #misuro 50 tempi tra 0 e 2 secondi
24 xdata = np.linspace(0, 2, 50)
25
26 #legge di caduta del corpo
27 y = Legge_oraria(xdata, 20, 9.81)
28 rng = np.random.default_rng()
29 y_noise = 0.3 * rng.normal(size=xdata.size)
30 #dati misurati affetti da errore
31 ydata = y + y_noise
32 dydata = np.array(ydata.size*[0.3])
33
34 #funzione che mi permette di vedere anche le barre d'errore
35 plt.errorbar(xdata, ydata, dydata, fmt='.', label='dati')
36
37 #array dei valori che mi aspetto, circa, di ottenere
38 init = np.array([15, 10])
39 #eseguo il fit
```

```

40 popt, pcov = curve_fit(Legge_oraria, xdata, ydata, init, sigma=dydata, absolute_sigma=False)
41
42 h0, g = popt
43 dh0, dg = np.sqrt(pcov.diagonal())
44 print(f'Altezza iniziale h0 = {h0:.3f} +- {dh0:.3f}')
45 print(f"Accelerazione di gravita' g = {g:.3f} +- {dg:.3f}")
46
47 #grafico del fit
48 t = np.linspace(np.min(xdata), np.max(xdata), 1000)
49 plt.plot(t, Legge_oraria(t, *popt), label='fit')
50
51 plt.grid()
52 plt.title('Fit caduta grave', fontsize=15)
53 plt.xlabel('y(t) [m]', fontsize=15)
54 plt.ylabel('t [s]', fontsize=15)
55 plt.legend(loc='best')
56 plt.show()
57
58 [Output]
59 Altezza iniziale h0 = 19.988 +- 0.065
60 Accelerazione di gravita' g = 9.790 +- 0.071

```



L'utilizzo dell'array `init` ci aiuta a trovare il minimo assoluto in modo che il codice vada a cercare intorno a quei valori, evitando che il codice si incastri altrove; anche se in questo caso non era necessario in quanto regressione lineare, è comunque buona norma utilizzarlo. Provate a fittare il modello non lineare visto sopra e vi accorgerete come solo una piccola regione dei parametri conduca alla soluzione coretta e che basti spostarvi di poco per ottenere risultati poco sensati.

### 8.3 Dietro curve fit: Levenberg-Marquardt

Vogliamo ora provare ad andare dietro la libreria e vedere cosa fa effettivamente curve fit. Chiaramente i metodi di fit implementati sono molti e diversi, a seconda delle esigenze; per semplicità perciò andiamo a vedere quello che viene usato di default: Levenberg-Marquardt. Questo è un metodo iterativo, il che spiega la sensibilità ai valori iniziali, caratteristica di ogni metodo iterativo. Consideriamo la nostra funzione di fit  $f$  la quale dipende da una variabile indipendente e da un insieme di parametri  $\theta$ , il quale fondamentalmente è un vettore di  $\mathbb{R}^m$ . Possiamo espandere  $f$  in serie di Taylor intorno ad un valore dei nostri parametri:

$$f(x_i, \theta_j + \delta_j) \simeq f(x_i, \theta_j) + J_{ij}\delta_j \quad (2)$$

dove  $\delta_j$  è lo spostamento che viene fatto ad ogni passo dell'iterazione e  $J_{ij}$  è il gradiente di  $f$ , o jacobiano se volete:

$$J_{ij} = \frac{\partial f(x_i, \theta_j)}{\partial \theta_j} = \begin{bmatrix} \frac{\partial f(x_1, \theta_1)}{\partial \theta_1} & \dots & \frac{\partial f(x_1, \theta_m)}{\partial \theta_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x_n, \theta_1)}{\partial \theta_1} & \dots & \frac{\partial f(x_n, \theta_m)}{\partial \theta_m} \end{bmatrix} \quad (3)$$

Che è una matrice  $m \times n$  con  $m < n$  altrimenti il metodo non funziona e dobbiamo adottare altre strategie. Per trovare il valore di  $\delta$  espandiamo la (1):

$$\begin{aligned} S^2(\theta + \delta) &\simeq \sum_{i=1}^n \frac{(y_i - f(x_i, \theta) - J_{ij}\delta_j)^2}{\sigma_{y_i}^2} \\ &= (y - f(x, \theta) - J\delta)^T W (y - f(x, \theta) - J\delta) \\ &= (y - f(x, \theta))^T W (y - f(x, \theta)) - (y - f(x, \theta))^T W J\delta - (J\delta)^T W (y - f(x, \theta)) + (J\delta)^T W (J\delta) \\ &= (y - f(x, \theta))^T W (y - f(x, \theta)) - 2(y - f(x, \theta))^T W J\delta + \delta^T J^T W (J\delta) \end{aligned} \quad (4)$$

Dove  $W$  è tale che  $W_{ii} = 1/\sigma_{y_i}^2$  e derivando rispetto a  $\delta$  otteniamo il metodo di Gauss-Newton:

$$\frac{\partial S^2(\theta + \delta)}{\partial \delta} = -2(y - f(x, \theta))^T W J + 2\delta^T J^T W J = 0 \quad (5)$$

per cui facendo il trasposto a tutto otteniamo:

$$(J^T W J)\delta = J^T W (y - f(x, \theta)) \quad (6)$$

La quale si risolve per  $\delta$ . Per migliorare la convergenza del metodo si introduce un parametro di damping  $\lambda$  e l'equazione diventa:

$$(J^T W J - \lambda \text{diag}(J^T W J))\delta = J^T W (y - f(x, \theta)) \quad (7)$$

Il valore di  $\lambda$  viene cambiato a seconda se ci avviciniamo o meno alla soluzione giusta. Se ci stiamo avvicinando ne riduciamo il valore, andando verso il metodo di Gauss-Newton; mentre se ci allontaniamo ne aumentiamo il valore in modo che l'algoritmo si comporti più come un gradiente discendente (di cui in appendice ci sarà un esempio). La domanda è: come capiamo se ci stiamo avvicinando alla soluzione? Calcoliamo:

$$\begin{aligned} \rho(\delta) &= \frac{S^2(x, \theta) - S^2(x, \theta + \delta)}{|(y - f(x, \theta) - J\delta)^T W (y - f(x, \theta) - J\delta)|} \\ &= \frac{S^2(x, \theta) - S^2(x, \theta + \delta)}{|\delta^T (\lambda \text{diag}(J^T W J)\delta + J^T W (y - f(x, \theta)))|} \end{aligned} \quad (8)$$

se  $\rho(\delta) > \varepsilon_1$  la mossa è accettata e riduciamo  $\lambda$  senno rimaniamo nella vecchia posizione. Altra domanda a cui rispondere è: quando siamo arrivati a convergenza? definiamo:

$$R1 = \max(|J^T W (y - f(x, \theta))|) \quad (9)$$

$$R2 = \max(|\delta/\theta|) \quad (10)$$

$$R3 = |S^2(x, \theta)/(n - m) - 1| \quad (11)$$

Se una di queste quantità è minore di una certa tolleranza allora l'algoritmo termina. Rimane ora un'ultima domanda a cui rispondere e possiamo passare al codice. Dato che ci servono gli errori sui parametri di fit: come calcoliamo la matrice di covarianza? Basta calcolare:

$$\text{Cov} = (J^T W J)^{-1} \quad (12)$$

quindi gli errori saranno semplicemente la radice degli elementi sulla diagonale, e le altre entrate le correlazioni fra parametri.

Passiamo ora al codice:

```

1  """
2  the code performs a linear and non linear regression
3  Levenberg-Marquardt algorithm. You have to choose
4  some parameters delicately to make the result make sense
5  """
6
7  import numpy as np
8  import matplotlib.pyplot as plt
9
10
11 def lm_fit(func, x, y, x0, sigma=None, tol=1e-6, dense_output=False, absolute_sigma=False):
12     """
13     Implementation of Levenberg-Marquardt algorithm
14     for non-linear least squares. This algorithm interpolates
15     between the Gauss-Newton algorithm and the method of
16     gradient descent. It is iterative optimization algorithms
17     so finds only a local minimum. So you have to be careful
18     about the values you pass in x0
19
20     Parameters
21     -----
22     f : callable
23         fit function
24     x : 1darray
25         the independent variable where the data is measured.
26     y : 1darray
27         the dependent data, y <= f(x, {\theta})
28     x0 : 1darray
29         initial guess
30     sigma : None or 1darray
31         the uncertainty on y, if None sigma=np.ones(len(y))
32     tol : float
33         required tollerance, the algorithm stop if one of this quantities
34         R1 = np.max(abs(J.T @ W @ (y - func(x, *x0))))
35         R2 = np.max(abs(d/x0))
36         R3 = sum(((y - func(x, *x0))/dy)**2)/(N - M) - 1
37         is smaller than tol
38
39     dense_output : bool, optional dafult False
40         if true all iteration are returned
41     absolute_sigma : bool, optional dafult False
42         If True, sigma is used in an absolute sense and
43         the estimated parameter covariance pcov reflects
44         these absolute values.
45         pcov(absolute_sigma=False) = pcov(absolute_sigma=True) * chisq(popt)/(M-N)
46
47     Returns
48     -----
49     x0 : 1d array or ndarray
50         array solution
51     pcov : 2darray
52         The estimated covariance of popt
53     iter : int
54         number of iteration
55     """
56
57     iter = 0           #initialize iteration counter
58     h = 1e-7          #increment for derivatives
59     l = 1e-3          #damping factor
60     f = 10            #factor for update damping factor
61     M = len(x0)        #number of variable
62     N = len(x)         #number of data
63     s = np.zeros(M)    #auxiliary array for derivatives
64     J = np.zeros((N, M)) #gradient
65     #some trashold
66     eps_1 = 1e-1
67     eps_2 = tol
68     eps_3 = tol
69     eps_4 = tol
70
71     if sigma is None :    #error on data
72         W = np.diag(1/np.ones(N))
73         dy = np.ones(N)
74     else :
75         W = np.diag(1/sigma**2)
76         dy = sigma

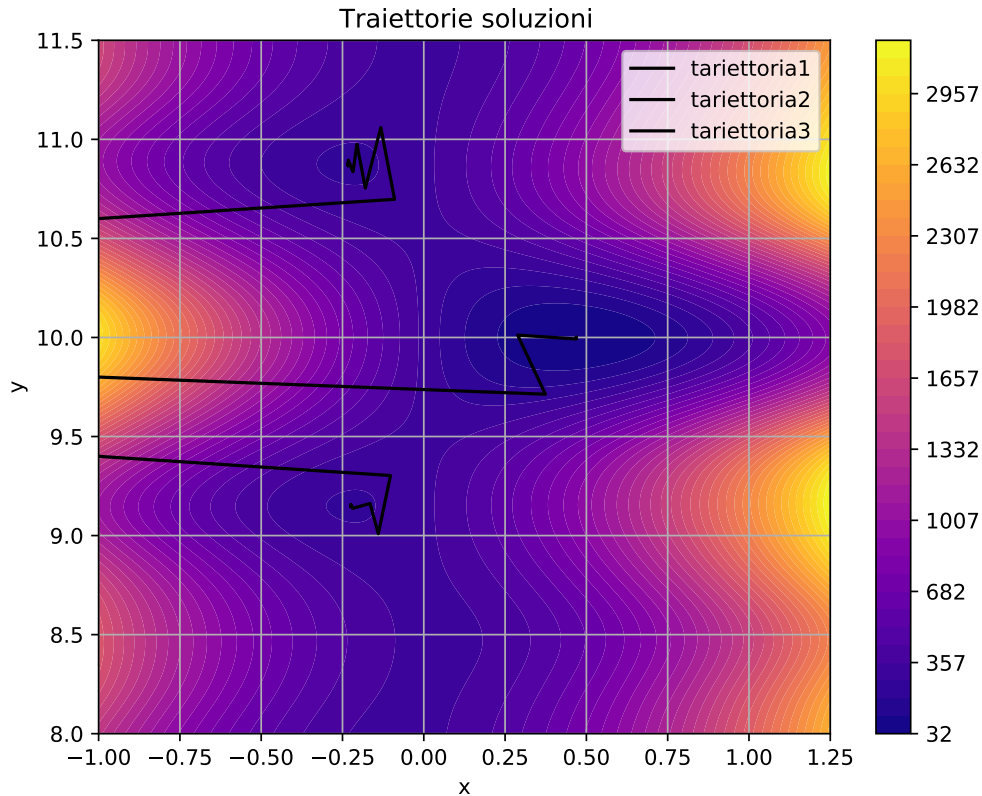
```

```

77
78
79 if dense_output:          #to store solution
80     X = []
81     X.append(x0)
82
83 while True:
84     #jacobian computation
85     for i in range(M):          #loop over variables
86         s[i] = 1                #we select one variable at a time
87         dz1 = x0 + s*h          #step forward
88         dz2 = x0 - s*h          #step backward
89         J[:,i] = (func(x, *dz1) - func(x, *dz2))/(2*h)    #derivative along z's direction
90         s[:] = 0                #reset to select the other
91     variables
92
93     JtJ = J.T @ W @ J           #matrix multiplication, JtJ is an MxM matrix
94     dia = np.eye(M)*np.diag(JtJ) #dia_ii = JtJ_ii ; dia_ij = 0
95     res = (y - func(x, *x0))     #residuals
96     b = J.T @ W @ res            #ordinate or dependent variable values of
97     system
98     d = np.linalg.solve(JtJ + l*dia, b) #system solution
99     x_n = x0 + d                 #solution at new time
100
101     # compute the metric
102     chisq_v = sum((res/dy)**2)
103     chisq_n = sum(((y - func(x, *x_n))/dy)**2)
104
105     rho = chisq_v - chisq_n
106     den = abs( d.T @ (l*np.diag(JtJ)*d + J.T @ W @ res))
107     rho = rho/den
108     # acceptance
109     if rho > eps_1 :             #if i'm closer to the solution
110         x0 = x_n                #update solution
111         l /= f                   #reduce damping factor
112     else:
113         l *= f                  #else magnify
114
115     # Convergence criteria
116     R1 = np.max(abs(J.T @ W @ (y - func(x, *x0))))
117     R2 = np.max(abs(d/x0))
118     R3 = abs(sum(((y - func(x, *x0))/dy)**2)/(N - M) - 1)
119
120     if R1 < eps_2 or R2 < eps_3 or R3 < eps_4:    #break condition
121         break
122
123     iter += 1
124
125     if dense_output:
126         X.append(x0)
127
128 #compute covariance matrix
129 pcov = np.linalg.inv(JtJ)
130
131 if not absolute_sigma:
132     s_sq = sum(((y - func(x, *x0))/dy)**2)/(N - M)
133     pcov = pcov * s_sq
134
135 if not dense_output:
136     return x0, pcov, iter
137 else :
138     X = np.array(X)
139     return X, pcov, iter

```

Il parametro `dense_output` è stato inserito per fare un plot interessante per far vedere la dipendenza dalle condizioni iniziali. Non riportiamo l'intero codice per non appesantire, la restante parte trattava solo di fare il plot delle curve di livello. In ogni caso è disponibile nell'apposita cartella il codice intero. Questo è il primo vero codice che fa qualcosa di molto complicato esso usa tutto quanto spiegato fin'ora e adesso è evidente l'importanza di mettere commenti, dare nomi sensati e rendere leggibile il codice. Bisogna ricordarsi che i codici in genere vengono scritti una volta ma letti tante volte quindi la chiarezza non va dosata con parsimonia.



Questo grafico rappresenta le curve di livello del modello non lineare  $y(t) = A \cos(\omega t)$  ed è facile vedere come partendo da condizioni diverse il fit si incastrì in minimo locali. L'asse y corrisponde a  $\omega$  mentre l'asse x corrisponde ad  $A$ . Vediamo ora di testare i risultati del codice fittando qualcosa di un po' più brutto.

```

1  """
2  Test
3  """
4  import numpy as np
5  import matplotlib.pyplot as plt
6  from scipy.optimize import curve_fit
7  from Lev_MaQ import lm_fit
8
9
10 def f(t, A, o1, o2, f1, f2, v, tau):
11     """fit function
12     """
13     return A*np.cos(t*o1 + f1)*np.cos(t*o2 + f2)*np.exp(-t/tau) + v
14
15 ##data
16 x = np.linspace(0, 20, 1000)
17 y = f(x, 200, 10.5, 0.5, np.pi/2, np.pi/4, 42, 25)
18 rng = np.random.default_rng(seed=69420)
19 y_noise = 1 * rng.normal(size=x.size)
20 y = y + y_noise
21 dy = np.array(y.size*[1])
22
23 ##confronto
24
25 init = np.array([101, 10.5, 0.475, 1.5, 0.6, 35, 20])
26
27 pars1, covm1, iter = lm_fit(f, x, y, init, sigma=dy, tol=1e-8)
28 dpar1 = np.sqrt(covm1.diagonal())
29 pars2, covm2 = curve_fit(f, x, y, init, sigma=dy)
30 dpar2 = np.sqrt(covm2.diagonal())
31 print("-----codice-----|-----scipy-----")
32 for i, p1, dp1, p2, dp2 in zip(range(len(init)), pars1, dpar1, pars2, dpar2):
33     print(f"pars{i} = {p1:.5f} +- {dp1:.5f} ; {p2:.5f} +- {dp2:.5f}")
34
35 print(f"numero di iterazioni = {iter}")
36
37 chisq1 = sum(((y - f(x, *pars1))/dy)**2.)

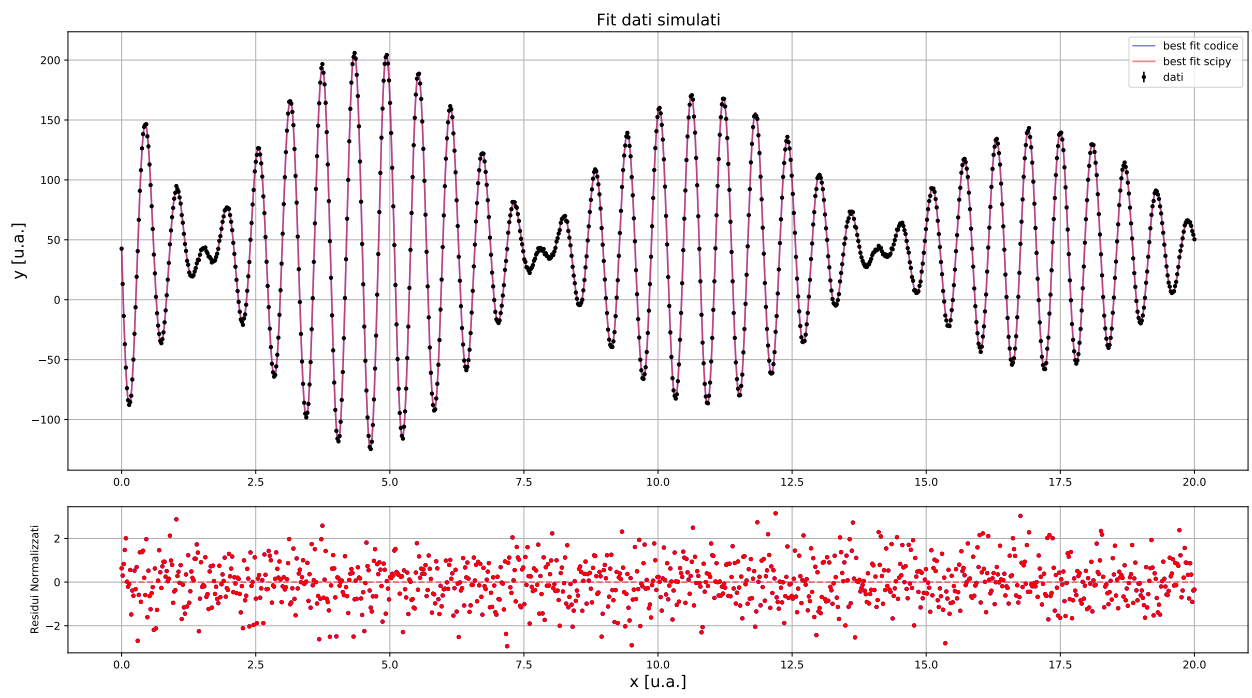
```

```

38 chisq2 = sum(((y - f(x, *pars2))/dy)**2.)
39 ndof = len(y) - len(pars1)
40 print(f'chi quadro codice = {chisq1:.3f} ({ndof:d} dof)')
41 print(f'chi quadro numpy = {chisq2:.3f} ({ndof:d} dof)')
42
43
44 c1 = np.zeros((len(pars1),len(pars1)))
45 c2 = np.zeros((len(pars1),len(pars1)))
46 #Calcoliamo le correlazioni e le inseriamo nella matrice:
47 for i in range(0, len(pars1)):
48     for j in range(0, len(pars1)):
49         c1[i][j] = (covm1[i][j])/(np.sqrt(covm1.diagonal()[i])*np.sqrt(covm1.diagonal()[j]))
50         c2[i][j] = (covm2[i][j])/(np.sqrt(covm2.diagonal()[i])*np.sqrt(covm2.diagonal()[j]))
51 #print(c1) #matrice di correlazione
52 #print(c2)
53
54 ##Plot
55 #Grafichiamo il risultato
56 fig1 = plt.figure(1)
57 #Parte superiore contenente il fit:
58 frame1=fig1.add_axes((.1,.35,.8,.6))
59 #frame1=fig1.add_axes((trasla lateralmente, trasla verticalmente, larghezza, altezza))
60 frame1.set_title('Fit dati simulati',fontsize=15)
61 plt.ylabel('y [u.a.]',fontsize=15)
62 plt.grid()
63
64
65 plt.errorbar(x, y, dy, fmt='.', color='black', label='dati') #grafico i punti
66 t = np.linspace(np.min(x), np.max(x), 10000)
67 plt.plot(t, f(t, *pars1), color='blue', alpha=0.5, label='best fit codice') #grafico del best
68     fit
69 plt.plot(t, f(t, *pars2), color='red', alpha=0.5, label='best fit scipy') #grafico del best
70     fit scipy
71 plt.legend(loc='best')#inserisce la legenda nel posto migliore
72
73 #Parte inferiore contenente i residui
74 frame2=fig1.add_axes((.1,.1,.8,.2))
75
76 #Calcolo i residui normalizzati
77 ff1 = (y - f(x, *pars1))/dy
78 ff2 = (y - f(x, *pars2))/dy
79 frame2.set_ylabel('Residui Normalizzati')
80 plt.xlabel('x [u.a.]',fontsize=15)
81
82 plt.plot(t, 0*t, color='red', linestyle='--', alpha=0.5) #grafico la retta costantemente zero
83 plt.plot(x, ff1, '.', color='blue') #grafico i residui normalizzati
84 plt.plot(x, ff2, '.', color='red') #grafico i residui normalizzati scipy
85 plt.grid()
86 plt.show()
87
88 [Output]
89
90 -----codice-----|-----scipy-----
91 pars0 = 199.85504 +- 0.17712 ; 199.85504 +- 0.17712
92 pars1 = 10.50005 +- 0.00009 ; 10.50005 +- 0.00009
93 pars2 = 0.49990 +- 0.00008 ; 0.49990 +- 0.00008
94 pars3 = 1.57040 +- 0.00087 ; 1.57040 +- 0.00087
95 pars4 = 0.78579 +- 0.00067 ; 0.78579 +- 0.00067
96 pars5 = 41.92350 +- 0.03125 ; 41.92350 +- 0.03125
97 pars6 = 24.99194 +- 0.05652 ; 24.99194 +- 0.05652
98 numero di iterazioni = 6
99 chi quadro codice = 969.017 (993 dof)
100 chi quadro numpy = 969.017 (993 dof)

```

Non abbiamo stampato la matrice di covarianza per avere un po' più di ordine. Vediamo che tra i due non ci sono differenze, siamo felici. Vediamo anche il grafico:





## 9 Lezione Bonus: Version control

Ok che le lezioni dovrebbero essere 4, ma come avete potuto vedere dall'indice ci sono una serie svariata di appendici e quindi tanto vale aggiungere questa lezione bonus che potrebbe essere molto utile. Tra l'altro potete capire da quanto appena scritto che questa sezione è stata aggiunta molto dopo. Non sappiamo nemmeno se sarà effettivamente una lezione che verrà tenuta. Però nella convinzione che possa essere utile io comunque provo a spiegarvelo. Come sapete queste note e i vari codici sono disponibili su un sito internet chiamato GitHub. GitHub è un sito che serve proprio per queste tipo di cose e per interagirci si può usare git o più semplicemente l'applicazione github desktop. Spieghiamo ora brevemente cos'è il controllo versione. Fondamentalmente immaginate di dover lavorare con più persone ad un progetto, che sia un codice software o una relazione di laboratorio. Quindi è necessario che tutti abbiano le stesse informazioni sui più recenti sviluppi fatti. Per quanto riguarda lo scrivere le relazione e basta magari uno può dire che esistono modi per scrivere e condividere il sorgente, ad esempio overleaf; ma non c'è solo la relazione, c'è anche il codice di analisi dati che sarebbe giusto tutti abbiano. Quindi quello che facciamo è creare un luogo remoto, ad esempio una repository su GitHub, in cui sarà caricato il codice e tutte le altre informazioni. Ogni membro potrà quindi farne una copia, tramite Git o github desktop, sul suo pc, quindi una copia locale della repository, che potrà modificare all'occorrenza. Tale copia si fa con il comando "**git clone ...**" vedremo poi il perchè dei tre puntini. Una volta fatta le varie modifiche voi potete, allo stesso modo con cui avete scaricato la versione originale, potete caricarla dalla vostra repository locale sulla repository in remoto sul server. Vediamo un semplice schemino per rendere chiare le idee.

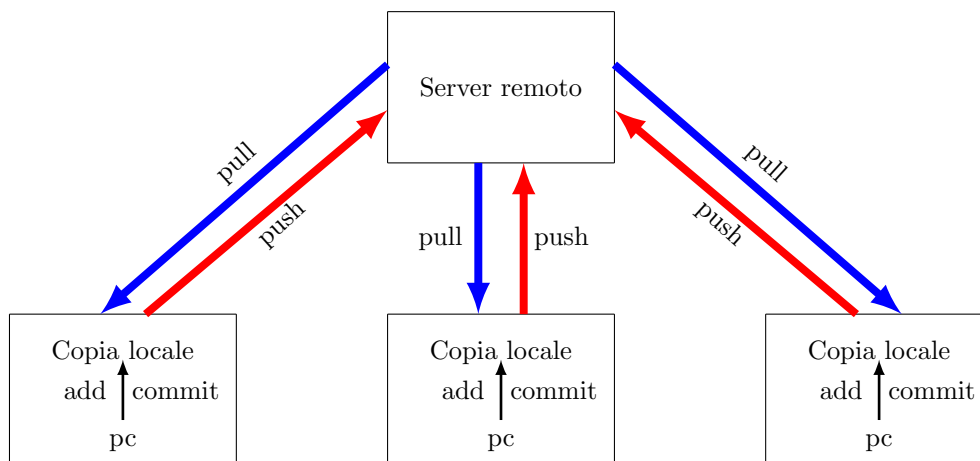


Figura 3: Schema controllo versione distribuito.

I nomi che vedete accanto alle frecce sono i nomi delle operazioni, banalmente i comandi da dare a git. Spieghiamo un attimo: avete un certo file e lo modificate; dovete quindi aggiornare il vostro repository locale facendo "**git add nomefile.estensione**" e "**git commit nomefile.estensione**" (su github desktop vi basterà premere il tasto commit). Fatto ciò per spedire la copia in remoto vi basterà fare "**git push**". Se invece la copia locale è indietro rispetto alla copia remota, per aggiornare vi basterà fare "**git pull**". Ovviamente tutti questi comandi vanno scritti su shell, su github desktop vi basta premere i bottoni in cui ci sta scritto push, pull o che altro sia; abbastanza semplice. Buona norma è aggiungere un commento quando si fa il commit; commento che sarà visibile sulla repository locale. da shell basta fare "**git commit nomefile.estensione -m 'commento da inserire'**". Ciò è importante per far capire cosa avete fatto; infatti questo sistema di collaborazione tiene traccia di tutte le versioni di ciò su cui state lavorando, come una vera e propria cronologia, in modo che sia semplice risalire a vecchie versioni in caso di errori. Ora in linea di principio anche dentro il server remoto possono succedere cose, tra **fork**, **pull request**, **issue** e **merge**. spieghiamo brevemente di che si tratta: Sia A la repository sull'account GitHub di persona A (ho molta fantasia). Voi potete premere dove ci sta scritto **fork** e creare così una copia della repository sul vostro account e fare tutto come detto sopra, creando un ramo secondario. Una volta fatto tutto potete fare una **pull request** cioè chiedere che la vostra modifica entri nel ramo principale dalla quale avete eseguito il **fork**; se chi gestisce la repository (persona A) è d'accordo allora lui farà il **merge**. Tutto ciò può anche essere fatto su una vostra repository ovviamente. Infine un **issue** è una cosa che voi su GitHub aprite dicendo al mondo: sta cosa non funge. Tutto ciò nella speranza che qualcuno veda il vostro problema e attraverso tutti i meccanismi di sopra, o semplicemente con un commento vi aiuti a risolverlo o vi dia consigli sul da farsi. Veniamo all'ultima questione lasciata in sospeso: "**git clone ...**" cosa ci va al posto dei tre puntini? Potete fare due cose: potete banalmente mettere l'HTTPS, prendendo l'url, ma lo trovate anche nella repository su GitHub premendo dove c'è un bottone verde con scritto "Code". Lì trovate anche l'SSH, altra cosa che potete utilizzare, secondo me più comodo perchè

una volta che configurate l'SSH dopo aver creato il vostro account GitHub potrete fare tutto quello che volete senza che git vi chieda varie ed eventuali password. Per configurare la chiave SSH potete seguire i due link nell'ordine: prima lui <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent> e poi lui <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account>. Con github desktop non avrete questi problemi nel clone e nel resto.