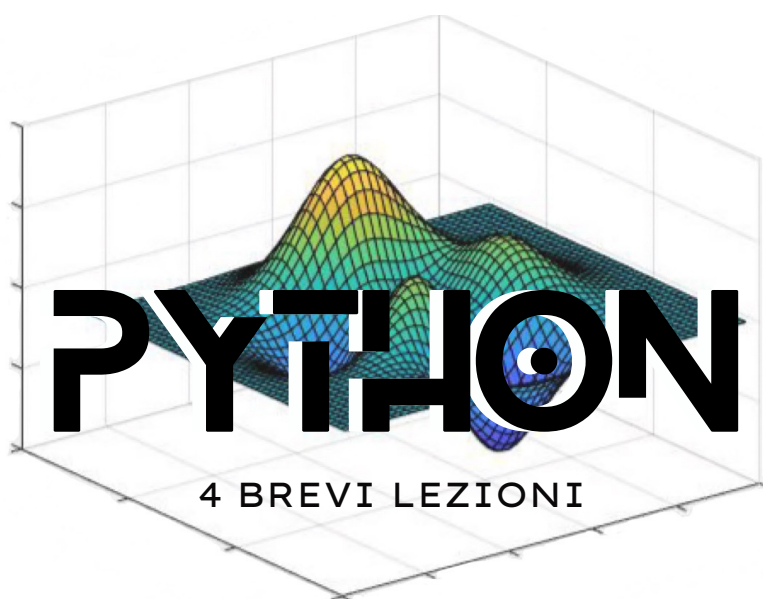


4 Brevi lezioni di Python

Francesco Zeno Costanzo

(Do you remember the) 21(st night of) September 2023



I think, it's time we blow this scene.
Get everybody and the stuff together.
Ok three, two, one, let's jam.
Seatbelts, Tank! (1999)

Indice

1	Introduzione	3
1.1	Notazioni	3
1.2	I 4 (per ora) comandamenti dell'informatica	3
2	Lezione Zero: Installazione	4
2.1	Installazione dell'ambiente: Pyzo	4
2.2	Installazione dell'interprete: Anaconda	4
2.3	Installazione dei pacchetti	4
3	The Zen of Python	5
4	Prima lezione	6
4.1	Funzione print	6
4.2	Commenti	6
4.3	Variabili	6
4.4	Operatore di Warlus	9
4.5	Librerie	10
4.6	Come leggere il Traceback	11
4.7	Un paio di trick	12

1 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 ciò causa un errore a tempo di esecuzione (e.g. il famoso segmentation fault) che 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.



1.1 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).

1.2 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.

2 Lezione Zero: Installazione

2.1 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 bit¹. 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.

2.2 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
```

2.3 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

¹Al 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>

3 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!

4 Prima lezione

4.1 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
```

4.2 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 il codice sia ad altri o anche al voi stesso del futuro, 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.

4.3 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 numeroavirgola mobile= 13.
3
4 print('Numero intero:', numerointero, 'Numero in virgola mobile:', numeroavirgola mobile)
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 "L_z" 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 fin qui 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, ne 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 $x \text{ operazione} = \text{numero}$.

4.4 Operatore di Warlus

Supponiamo di avere un frammento del codice del tipo:

```

1 measure = [2, 8, 0, 1, 1, 9, 7, 7]
2
3 info_measure = {
4     "length": len(measure),
5     "sum": sum(measure),
6     "mean": sum(measure) / len(measure),
7 }
8
9 print(info_measure)
10
11 [Output]
12 {'length': 8, 'sum': 35, 'mean': 4.375}

```

Codice abbastanza chiaro, no? Abbiamo una serie di misure e creiamo un dizionario che contenga alcune informazioni sul nostro insieme di misure. Una cosa salta subito all'occhio, le funzioni "sum" e "len" vengono chiamate due volte, cosa inutile chiaramente. Possiamo chiaramente aggiungere due righe per definire due variabili di nostro interesse in modo anche da tener traccia anche del loro valore volendo. Ma possiamo farlo in modo più compatto? Possiamo grazie all'operatore di Walrus:

```

1 measure = [2, 8, 0, 1, 1, 9, 7, 7]
2
3 info_measure = {
4     "length": (length := len(measure)),
5     "sum": (total := sum(measure)),
6     "mean": total / length,
7 }
8
9 print(info_measure)
10 print(length)
11 print(total)
12
13 [Output]
14 {'length': 8, 'sum': 35, 'mean': 4.375}
15 8
16 35

```

L'utilizzo della parentesi tonda quando si usa l'operatore di Walrus " := " è necessario, e vederemo che servirà anche poi quando tratteremo le condizioni di controllo.

4.5 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 contesto 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 dovesse essere un numero ma un array.

4.6 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
    <module>
14     run(0)
15   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 6, in
    run
16     print(err(c))
17   File "C:\Users\franc\Documents\GitHub\4BLP\Prima Lezione\codiciL1\err_trace.py", line 2, in
    err
18     b = 1/c
19 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.

4.7 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.