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:

```
def area(a, b):
    """
    restituisce l'area del rettangolo
    di lati a e b
    """
    A = a*b #calcolo dell'area
    return A

#chiamiamo la funzione e stampiamo subito il risultato
print(area(3, 4))
print(area(2, 5))

"""
Se la funzione non restituisce nulla
ma esegue solo un pezzo di codice,
si parla propriamente di procedura
e il valore restituito e' None.
"""
def procedura(a):
    a = a+1

print(procedura(2))

"""
Volendo si possono creare anche funzioni
che non hanno valori in ingresso:
"""
def pigreco():
    return 3.14
print(pigreco())

[Output]
12
10
None
3.14
```

Portiamo all'attenzione due fatti importati:

- 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 logicalmente 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 un 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:

```
def assoluto(x):
    """
    restituisce il valore assoluto di un numero
    """
    # se vero restituisci x
    if x >= 0:
        return x
    #altrimenri restituisci -x
    else:
        return -x

print(assoluto(3))
print(assoluto(-3))

[Output]
3
3
```

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

```
def segno(x):
    """
    funzione per capire il segno di un numero
    """
    #se vero ....
    if x > 0:
        return 'Positivo'
    #se invece ....
    elif x == 0:
        return 'Nullo'
    #altrimenti ....
    else:
        return 'Negativo'

print(segno(5))
print(segno(0))
print(segno(-4))

[Output]
Positivo
Nullo
Negativo
```

7.2.2 Cicli: while, for

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

```
def fattoriale(n):
    """
    Restituisce il fattoriale di un numero
    """
    R = 1
    #finche' e' vero fai ...
    while n > 1:
        R *= n
        n -= 1
    return R

print(fattoriale(5))

[Output]
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:

```
def fattoriale(n):
    """
    restituisce il fattoriale di un numero
    """
    R = 1
    #finche' i non arriva ad n fai ...
    for i in range(1, n+1):
        R = R*i
    return R

print(fattoriale(5))

[Output]
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 range(1, n+1) = [1,2,..., 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:

```
import numpy as np

def trova_pari(array):
    """
    restituisce un array contenente solo
    i numeri pari dell'array di partenza
    """
    R = np.array([]) #array da riempire
    #per ogni elemento in arrary fai ...
    for elem in array:
        if elem%2 == 0:
            R = np.append(R,elem)
    return R

a = np.array([i for i in range(0, 11)])
"""
il precedente e' un modo piu' conciso di scrivere:
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
"""
print(a)
print(trova_pari(a))

[Output]
[ 0  1  2  3  4  5  6  7  8  9 10]
[ 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:

```
import numpy as np

#creiamo un array
array = np.linspace(0, 1, 5)

"""
in questo modo posso iterare contemporaneamente
sia sugli indici sia sugli elementi dell'array
"""
for index, elem in enumerate(array):
    print(index, elem)

[Output]
0 0.0
1 0.25
2 0.5
3 0.75
4 1.0
```

zip:

```
import numpy as np

```

```
#creiamo tre un array
array1 = np.linspace(0, 1, 5)
array2 = np.linspace(1, 2, 5)
array3 = np.linspace(2, 3, 5)
"""
in questo modo posso iterare contemporaneamente
sugli elementi di tutti gli array
"""
for a1, a2, a3 in zip(array1, array2, array3):
    print(a1, a2, a3)

[Output]
0.0 1.0 2.0
0.25 1.25 2.25
0.5 1.5 2.5
0.75 1.75 2.75
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):

```
def molt(*numeri):
    """
    restituisce il prodotto di n numeri
    """
    R = 1
    for numero in numeri:
        R *= numero
    return R

print(molt(2, 7, 10, 11, 42))
print(molt(5, 5))
print(molt(10, 10, 2))

[Output]
64680
25
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 ninete 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 cosi: curve_fit(lambda x, m, q : x*m + q, x, y, ...); ma ciò non vieta comunque di dargli un nome.

```
f = lambda x : 3*x
print(f(2))

h = lambda x, y, z : x*y + z
print(h(2, 3, 4))

[Output]
6
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.

```
def g(x):
    '''restituisce potenza x-esima di y
    '''
    return lambda y : y**x

G = g(3) # potenza cubica
print(G(4))

#============================================
#============================================

def m(f, y, x):
    '''passo una funzione ad una funzione
    '''
    return f(y) - x

print(m(lambda x : x**2, 5, 1))

[Output]
64
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:

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

Per leggeri:

```
import numpy as np

#Leggiamo da un file di testo classico
path = 'dati.txt'
dati1, dati2 = np.loadtxt(path, unpack=True)
"""
unpack=True serve proprio a dire che vogliamo che
dati1 contenga la prima colonna e dati2 la seconda
La prima riga avendo il cancelletto verra' saltata
"""

#se vogliamo invece che venga letto tutto come una matrice scriviamo:
path = 'dati.txt'
dati = np.loadtxt(path) # sarebbe unpack = False
#dati sara' nella fattispecie una matrice con due colonne e 6 righe


#leggere da file.csv
path = 'dati.csv'
dati1, dati2 = np.loadtxt(path,usecols=[0,1], skiprows=1, delimiter=',',unpack=True)
"""
si capisce senza troppa fatica che usecols indica le colonne che vogliamo leggere
skiprows il numero di righe da saltare e delimeter indica il carattere che separa le colonne
"""
```

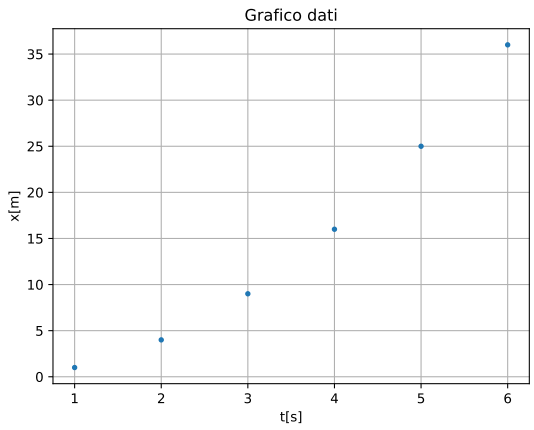
Un interessante tipo di file sono i file con estensione "npy" comodi se i dati soono di dimensioni elevate:

```
import time
import numpy as np

x = np.linspace(0, 1, int(2e6)) # dati

#=====================================================
# file txt
#=====================================================

# salviamo su file
start = time.time()
```

```
path = r'c:\Users\franc\Desktop\dati.txt'
f = open(path, 'w')
for i in x:
    f.write(f'{i} \n')
f.close()
end = time.time() - start

print(f"tempo di scrittura txt: {end} s")

#legggiamo da file
start = time.time()
X = np.loadtxt(path, unpack=True)
end = time.time() - start

print(f"tempo di lettura   txt: {end} s")

#=======================================================
# file npy
#=======================================================

# salviamo su file
start = time.time()
path = r'c:\Users\franc\Desktop\dati.npy'
np.save(path, x)
end = time.time() - start

print(f"tempo di scrittura npy: {end} s")

#legggiamo da file
start = time.time()
X = np.load(path, allow_pickle='TRUE')
end = time.time() - start

print(f"tempo di lettura   npy: {end} s")

[Output]
tempo di scrittura txt: 5.610752582550049 s
tempo di lettura   txt: 9.489601373672485 s
tempo di scrittura npy: 0.016022205352783203 s
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:

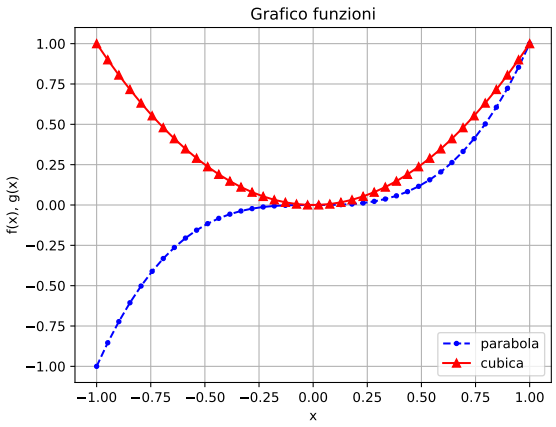
```
import numpy as np
import matplotlib.pyplot as plt

#Leggiamo da un file di testo classico
path = 'dati.txt'
dati1, dati2 = np.loadtxt(path, unpack=True)

plt.figure(1) #creiamo la figura

#titolo
plt.title('Grafico dati')
#nomi degli assi
plt.xlabel('t[s]')
plt.ylabel('x[m]')
#plot dei dati
plt.plot(dati1,dati2, marker='.',linestyle='')
#aggiungiamo una griglia
plt.grid()
#comando per mostrare a schermo il grafico
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:

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

def f(x):
    """
    restituisce il cubo di un numero
    """
    return x**3

def g(x):
    """
    restituisce il quadrato di un numero
    """
    return x**2

#array di numeri equispaziati nel range [-1,1] usiamo:
x = np.linspace(-1, 1, 40)

plt.figure(1) #creiamo la figura

#titolo
plt.title('Grafico funzioni')
#nomi degli assi
plt.xlabel('x')
plt.ylabel('f(x), g(x)')
#plot dei dati
plt.plot(x, f(x), marker='.', linestyle='--', color='blue', label='parabola')
plt.plot(x, g(x), marker='^', linestyle='-', color='red', label='cubica')
#aggiungiamo una leggenda
plt.legend(loc='best')
#aggiungiamo una griglia
plt.grid()
#comando per mostrare a schermo il grafico
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 conosce tutti gli altri comandi possibili per migliorare/abbellire il grafico da adre 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 logaritimica o altro sull'asse x (analogo sarà sulle y mutatis mutandis).

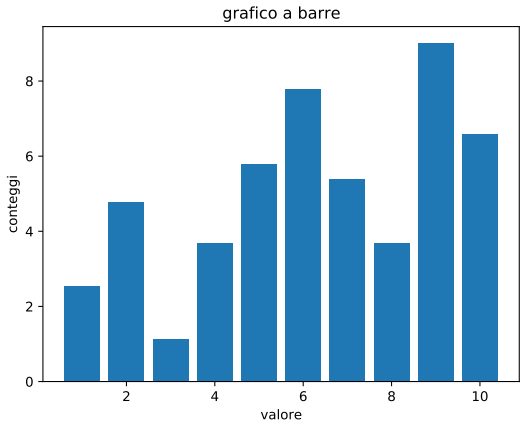


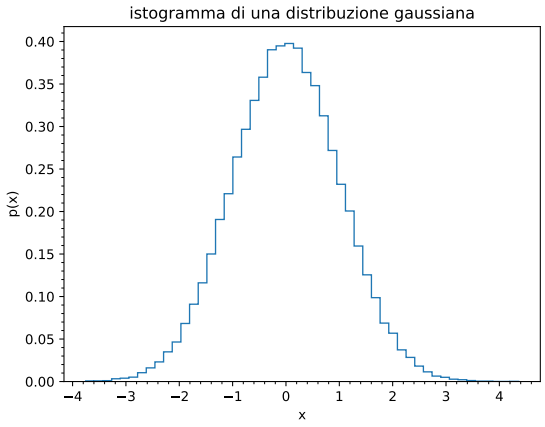
Ultima menzione da fare sono gli istogrammi:

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

plt.figure(1)
plt.title('grafico a barre')
plt.xlabel('valore')
plt.ylabel('conteggi')
# Sull'asse x utilizziamo un array di 10 punti equispaziati.
x = np.linspace(1,10,10)
# Sull'asse y abbiamo, ad esempio, il seguente set di dati:
y = np.array([2.54, 4.78, 1.13, 3.68, 5.79, 7.80, 5.4, 3.7, 9.0, 6.6])

# Il comando per la creazione dell'istogramma corrispondente e':
plt.bar(x, y, align='center')

plt.figure(2)
plt.title('istogramma di una distribuzione gaussiana')
plt.xlabel('x')
plt.ylabel('p(x)')
```

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

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:

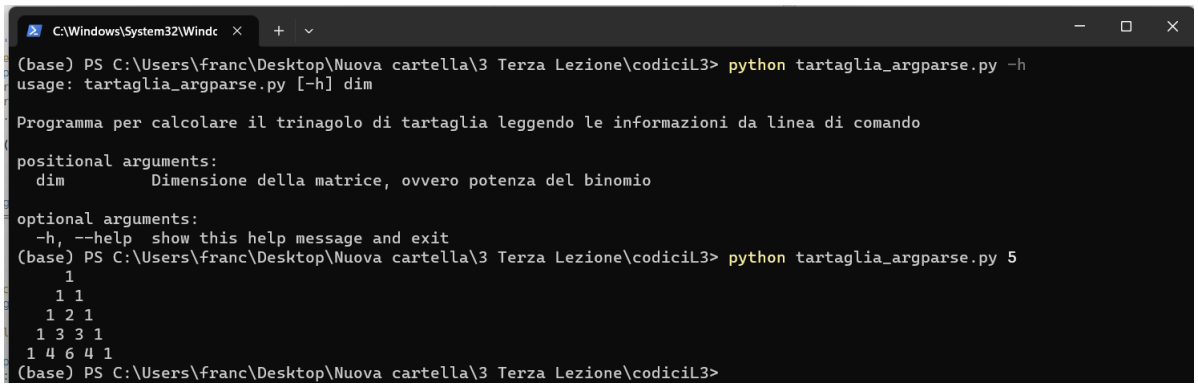
```
"""
Programma per calcolare il trinagolo di tartaglia
"""
import numpy as np

# leggo da input un valore e lo rendo intero
# la stringa verra stampata su shell
n = int(input("Ordine del triangolo: "))
a = np.zeros((n, n), dtype=int) # matrice per i coefficienti

# calocolo i coefficienti del trinagolo
a[0,0] = 1
for i in range(1, n):
    a[i, 0] = 1
    for j in range(1, i):
        a[i, j] = a[i-1, j-1] + a[i-1, j]
    a[i,i] = 1

# stampo a schermo
for i in range(n):
    for j in range(i+1):
        # solo per fare la forma a piramide
        if j == 0 : # non funziona con numeri a due cifre
            print(*[""]*(n-i),a[i, j], end='')
        else:
            print("",a[i, j], end='')
     print()

[Output]
Ordine del triangolo: 5
     1
    1 1
```

```
   1 2 1
  1 3 3 1
 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:

```
"""
Programma per calcolare il trinagolo di tartaglia
"""
import argparse
import numpy as np

description='Programma per calcolare il trinagolo di tartaglia leggendo le informazioni da
    linea di comando'
# descrizione accessibile con -h su shell
parser = argparse.ArgumentParser(description=description)
parser.add_argument('dim',  help='Dimensione della matrice, ovvero potenza del binomio')
args = parser.parse_args()
n = int(args.dim) # accedo alla variabile tramite il nome messo a linea 10

a = np.zeros((n, n), dtype=int) # matrice per i coefficienti

# calocolo i coefficienti del trinagolo
a[0,0] = 1
for i in range(1, n):
    a[i, 0] = 1
    for j in range(1, i):
        a[i, j] = a[i-1, j-1] + a[i-1, j]
    a[i,i] = 1

# stampo a schermo
for i in range(n):
    for j in range(i+1):
        # solo per fare la forma a piramide
        if j == 0 : # non funziona con numeri a due cifre
            print(*[""]*(n-i),a[i, j], end='')
        else:
            print("",a[i, j], end='')
    print()
```

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



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:

```
import time
import numpy as np

#inizio a misurare il tempo
start = time.time()


a1 = 0       #variabile che conterra' il risultato
N = int(5e6) # numero di iterazioni da fare = 5 x 10**6

#faccio il conto a 'mano'
for i in range(N):
```

```
    a1 += np.sqrt(i)

#finisco di misurare il tempo
end = time.time()-start

print(end)

#inizio a misurare il tempo
start = time.time()

#stesso conto ma fatto tramite le librerie di python
a2 = sum(np.sqrt(np.arange(N)))

#finisco di misurare il tempo
end = time.time()-start

#sperabilmente sara' minore del tempo impiegato prima
print(end)

[Output]
11.588378429412842
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 steso 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 magri qualcosa è stato implementato male. Esiste un costrutto che ci permette di gestire gli errori in maniera tranquilla diciamo. Facciamo un semplice esempio:

```
a = 0
b = 1/a
print(b)

[Output]
Traceback (most recent call last):
  File "<tmp 1>", line 5, in <module>
    b = 1/a
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 da 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 da 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.

```
a = 0
try :
    b = 1/a
except ZeroDivisionError:
    b = 1

print(b)

[Output]
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.

```
"""
leggo un valore da shell
uso del comando try per evitare che venga letto
qualcosa che non sia un numero: e.g. una stringa
"""
try:
    b = int(input('scegliere un valore:'))
except ValueError:
    print('hai digitato qualcosa diverso da un numero, per favore ridigitare')
    b = int(input('scegliere un valore:'))

#se si sbaglia a digitare di nuovo il codice si arresta per ValueError

#controllo se e' possibile proseguire
if b > 7 :
    #se vero si blocca il codice sollevando l'eccezione
    messaggio_di_errore = 'il valore scelto risulta insensato in quanto nulla supera 7, misura
     massima di ogni cosa'
    raise Exception(messaggio_di_errore)

[Output]
8
Traceback (most recent call last):
  File "<tmp 1>", line 18, in <module>
    raise Exception(messaggio_di_errore)
Exception: il valore scelto risulta insensato in quanto nulla supera 7, misura massima di ogni
     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 cilo).
3. Stessa cosa di sopra ma ora ogni curva deve avere un colore e uno linstyle 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. Sovrappore i plot di un istogramma e della funzione di ditribuzione 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:

```
def Area(l, n):
    '''
    funzione per calcolare l'area di un poligono
    regolare di lato l e numero lati n
    '''
    a = l/2 * 1/np.tan(np.pi/n) # apotema
    p = n*l                     # perimetro
    A = p*a/2                   # area
    return A

l = [3, 4, 5]         # dimesioni dei lati, (terna pitagorica)
n = np.arange(4, 12) # numero di lati dei poligoni

A3, A4, A5 = [], [], []
# liste in cui ci saranno le aree dei poligoni di lato rispettivamente 3, 4, 5

```

```
for n_i in n:      # loop sul numero di lati
    for l_i in l: # lup sulla dimesione

        A0 = Area(l_i, n_i) # calcolo dell'area

        if l_i == 3:  # conservo a seconda dei lati
            A3.append(A0)
        elif l_i == 4:
            A4.append(A0)
        elif l_i == 5:
            A5.append(A0)


for i in range(len(n)):
    print(f'A3 + A4 = {A3[i]+A4[i]:.3f}')
    print(f'   A5   = {A5[i]:.3f}')
```

Secondo:

```
power = [0.5, 1, 2]          # potenze
x = np.linspace(0, 1, 1000) # range sulle x

plt.figure(1)
for p in power:
    plt.plot(x, x**p) # un plot alla volta sulla stessa figura

# bellurie
plt.grid()
plt.title("Esercizio 2")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.show()
```

Terzo:

```
power = [0.5, 1, 2]                          # potenze
color = ['k', 'r', 'b']                      # colore di ogni curva
lnsty = ['-', '--', '-.']                    # stile di ogni curva
label = [r'$\sqrt{x}$', r'$x$', r'$x^2$']  # nome della curva
x = np.linspace(0, 1, 1000)

plt.figure(1)
for p, c, ls, lb in zip(power, color, lnsty, label):
    # un plot alla volta sulla stessa figura
    plt.plot(x, x**p, c=c, linestyle=ls, label=lb)

#bellurie
plt.grid()
plt.title("Esercizio 3")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend(loc='best')
plt.show()
```

Quarto:

```
def read():
    '''
    funzione che legge da input un numero con la condizione che esso
    sia maggiore di zero e che dia la possibilita' di inserirlo
    nuovamente finche' la condizione non e' verificata.
    Volendo si puo' generalizzare il codice passando la condizione come input
    '''

    while True:  # Il codice deve runnare finche' non inserisco un numero buono

        try: # provo a leggere il numero e a renderlo intero
            x = int(input("Iserisci un numero: "))

        except ValueError: # se non riesco sollevo l'eccezione
            print(f"Fra ti ho chiesto di mettere un numero") # messaggio di errore
            continue # questo comando fa ripartire il ciclo da capo

        if x > 0: # se la lettura e' andata a buon fine verifico la condizione
            return x # se e' verificata ritorno il numero
        else :
            # alrimenti stampo un messaggio di errore
            print("In numero inserito e' minore di zero, sceglierne un altro.")
```

```
            continue # e faccio ripartire il ciclo da capo


x = read()
print(f"Il numero letto e': {x}")
```

Quinto:

```
Area = lambda l, n : (l/2 * 1/np.tan(np.pi/n))*(n*l)/2

l = [3, 4, 5]         # dimesioni dei lati, (terna pitagorica)
n = np.arange(4, 12) # numero di lati dei poligoni

A3, A4, A5 = [], [], []
# liste in cui ci saranno le aree dei poligoni di lato rispettivamente 3, 4, 5

for n_i in n:      # loop sul numero di lati
    for l_i in l: # lup sulla dimesione

        A0 = Area(l_i, n_i) # calcolo dell'area

        if l_i == 3:  # conservo a seconda dei lati
            A3.append(A0)
        elif l_i == 4:
            A4.append(A0)
        elif l_i == 5:
            A5.append(A0)


for i in range(len(n)):
    print(f'A3 + A4 = {A3[i]+A4[i]:.3f}')
    print(f'    A5    = {A5[i]:.3f}')
```

Sesto:

```
# Gaussiana
m = 0
s = 1
z = [np.random.normal(m, s) for _ in range(int(1e5))]

# Plot dati
plt.figure(1)
plt.hist(z, bins=50, density=True, histtype='step', label='dati')
plt.grid()
plt.xlabel("x")
plt.ylabel("P(x)")
plt.title("Distribuzione gaussiana")

# Plot curva
x = np.linspace(-5*s + m, 5*s + m, 1000)
plt.plot(x, np.exp(-(x-m)**2 / (2*s**2))/np.sqrt(2*np.pi*s**2), 'b', label=f"N({m}, {s})")
plt.legend(loc='best')
plt.show()
```

Settimo:

```
def obs(x):
    '''
    funzione che dato un set di dati x in input
    restituisce media e deviazione standard
    '''
    N = len(x)

    media = sum(x)/N
    varianza = sum( (x - media)**2)/(N*(N-1))
    dev_std = np.sqrt(varianza)

    return media, dev_std

dati = np.array([0, 3, 5, 1, 5, 667, 2, 4, 9, 3, 33])

m, dm = obs(dati)
print(f"media = {m:.3f} +- {dm:.3f}")
```