Introduzione agli script Python per l'analisi dati

Sommario

L'analisi dati richiede spesso di eseguire ripetutamente operazioni semplici, ad esempio nel caso di calcolo di medie o deviazioni standard, oppure nel caso di fit numerici. È in questi casi che il calcolatore ci può aiutare a velocizzare le operazioni. Un semplice *script*, basato su un linguaggio ad alto livello, è quello che serve nella quasi totalità dei casi.

Nell'esperienza di oggi vedremo alcuni esempi di base che possono essere utili nelle future esperienze di laboratorio. Utilizzeremo il linguaggio Python, noto per essere semplice da usare, dotato di un ampia libreria standard e di librerie aggiuntive (scipy, numpy, matplotlib, etc.) che lo rendono molto pratico per l'analisi dati.

L'ambiente di sviluppo

Python può essere usato sia in maniera interattiva, aprendo il suo interprete e inserendo le istruzioni una alla volta, che per eseguire script salvati su file di testo. Il primo caso si presta ad operazioni immediate da fare una volta sola. Il secondo, su cui ci soffermeremo, è più adatto quando lo script è più complesso di qualche semplice riga e quando si vuole avere la libertà di fare piccole modifiche allo script ed eseguirlo più volte senza troppi sforzi.

Potete usare un qualunque editor di testo (emacs, gedit, vi, etc.) per scrivere il vostro codice, ed eseguirlo nel terminale a riga di comando, con comandi come "python myscript.py". Esistono anche ambienti di sviluppo integrati come Pyzo (disponibile nei PC in laboratorio) che forniscono sia dei tab per scrivere e salvare i vostri script, gli strumenti per eseguire il codice, un insieme di librerie pre-installate ed una shell python interattiva.

Il terminale

È consigliato crearsi *directory*, (ovvero una cartella) dedicata per ciascun gruppo ed esperienza in cui lavorare e mettere i propri file (script, dati, grafici, etc.). Partendo dai comandi base di terminale del sistema operativo Linux:

- `pwd` restituisce sullo schermo il nome della directory corrente;
- `ls` elenca i file contenuti in una certa directory;
- `cd` cambia directory (es. "`cd plasduino`");
- `mkdir` crea directory (es. "`mkdir esperienza_dadi`");
- `cp` copia file (es. "`cp plasduino/data/dati.txt mydir`");
- `mv` sposta o rinomina un file (es. "`mv dati.txt dati_1.txt`")
- `rm` cancella file (es. "`rm dati.txt`")

Apriamo un terminale, controlliamo in che cartella siamo (con *pwd*), creiamo la nostra cartella di lavoro (con *mkdir*) e entriamoci (con *cd*). A questo punto facciamo partire Pyzo e creiamo un nuovo scrip nella nostra cartella di lavoro.

Primo programma

È tradizione, iniziare ad apprendere i fondamenti della sintassi di un linguaggio di programmazione, con un semplice programma che restituisca una stringa di testo. Nel caso di Python, si scrive un output su terminale semplicemente con la funzione *print*[1]:

```
# primo programmino
print('Hello world!')
```

Giusto un po' più complesso

Notare che *print* restituisce sempre una stringa di testo e quindi prova a convertire in questo formato anche variabili che testo non sono. Ecco un esempio con definizione e manipolazione di variabili, in cui si vede anche come importare la libreria delle funzioni matematiche:

```
# definisco x e y come interi
x = 2
y = 3
# stampo la somma
print (x+y)
#importo una libreria con funzioni matematiche
# ne vedremo dopo le caratteristiche
import numpy
# calcolo la radice quadrata della somma
z = numpy.sqrt(x+y)
# stampo la radice quadrata della somma
print (z)
# forzando solo 3 cifre dopo la virgola
print ('%.3f' % z)
# forzando ad intero
print (int(z))
```

Cosa succede se c'è un errore nel codice? Ad esempio se proviamo a fare la radice quadrata di una lettera:

```
import numpy
x = numpy.sqrt('a')
```

Otteniamo un messaggio di errore con indicata la linea dove è stato trovato l'errore ed il suo tipo. In questo caso "TypeError: Not implemented for this type", ovvero la funzione *sqrt* non funziona con le stringe di testo, ma si aspetta numeri. È bene leggere con cura tali messaggi (*Traceback*) di solito sono abbastanza chiari da indicare bene quale errore abbiamo fatto.

1. In tutte le lingue del mondo: http://helloworldcollection.de/

Leggere e usare un file di dati

Può capitare che i dati da analizzare siano scritti in un file di testo, vediamo quindi come leggerlo. Iniziamo dal crearne uno con un po' di dati, ad esempio due colonne separate da una tabulazione (tasto "tab")

```
#x      y
1.0     2.08
2.0     4.44
3.0     6.09
4.0     6.88
```

Proviamo a leggerlo e a riempire due liste con numeri *float* (ovvero in virgola mobile). È fondamentale l'indentazione, ovvero che righe di comandi in uno stesso gruppo siano allineate verticalmente.

```
# apriamo il file
fin = open('dati.txt')
# cosa succede se faccio print?
print(fin)
# andiamo attraverso ogni linea e
# riempiamo una lista di numeri:
ListaX = []
ListaY = []
for line in fin:
  if not line.startswith('#'):
    [x,y] = line.split('\t') # separiamo sui tab
    print (x,y) # diamo un occhiata al contenuto
    ListaX.append(float(x)) # forziamo a float
    ListaY.append(float(y))

# vediamo il risultato:
print(ListaX)
print(ListaY)
```

Media aritmetica

Vogliamo definire una funzione per il calcolo della media di una lista di dati.

$$m = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$

```
# definiamo la funzione
def media(x):
  n = len(x) # lunghezza della lista
  media = 0. # inizializziamo la media
  for i in x:
    media += i
  media /=n
  # restituiamo la media
  return media

# inventiamoci un set di valori
dati = [1,2,3,4,5,6]
# calcoliamone la media
print(media(dati))
```

Notare che la media è inizializzata con un float, cosa succede se invece uso un numero intero (*int*?

Deviazione standard

Proviamo ora ad aggiungere alla media, il calcolo della deviazione standard:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - m)^2} \tag{2}$$

```
# ci serve la radice quadrata
from numpy import sqrt

# definiamo la funzione
def getstats(x):
  # media come prima
  n = len(x) # lunghezza della lista
  media = 0. # inizializziamo la media
  for i in x:
    media += i
  media /=n
  # ora somma dei quadrati:
  ss = 0.
  for i in x:
    ss += (i-media)**2
  # normalizzazione:
  stdev = sqrt(ss/(n-1))
  # in output i 2 valori
  return media, stdev

# inventiamoci un set di valori
dati = [1,2,3,4,5,6]
# e usiamo la funzione
print(getstats(dati))
```

Notiamo che nella funzione precedente abbiamo fatto 2 *loop*, ovvero due cicli sullo stesso set di dati. C'è un modo per rendere la funzione più veloce evitando il secondo loop? Possiamo provare a manipolare la formula della deviazione standard:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i^2 + m^2 - 2x_i m)} \tag{3}$$

$$= \sqrt{\frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 + nm^2 - 2m\sum_{i=1}^{n} x_i\right)} \tag{4}$$

$$= \sqrt{\frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n * m^2\right)} \tag{5}$$

e notiamo che basta la somma dei valori e dei suoi quadrati per costruire media e deviazione standard. Scriviamo quindi una nuova funzione:

```
# ci serve la radice quadrata
from numpy import sqrt

# definiamo la funzione
def getstats2(x):
  # media come prima
  n  = len(x) # lunghezza della lista
  s  = 0. # inizializziamo somma
  ss = 0. # inizializziamo somma dei quadrati
```

```
  for i in x:
    s += i
    ss += i*i
  # calcoliamo media:
  media =s/n
  # calcoliamo stdev:
  stdev = sqrt((ss - n*media**2)/(n-1))
  # in output i 2 valori
  return media, stdev

# inventiamoci un set di valori
dati = [1,2,3,4,5,6]
# e usiamo la funzione
print(getstats2(dati))
```

Notate differenze tra il risultato delle due funzioni? Sapendo che la seconda è più veloce, potete immaginare casi in cui è preferibile la prima?

NumPy

Una libreria molto usata per il calcolo numerico e scientifico in Python è NumPy (http://www.numpy.org/). Il suo scopo è principalmente la manipolazione di *array* multidimensionali. Un array (unidimensionale) è una sequenza ordinata di numeri, quindi abbastanza simile ad una lista, ma con differenze fondamentali. Vediamo, nell'esempio seguente, alcune caratteristiche degli array di numpy:

```
# cominciamo importando numpy
import numpy as np
# creiamo quindi il nostro primo array:
a = np.array([2,3,4])
# e proviamo a stamparlo
print(a)
# Prima cosa importante: il tipo di variabile
# e' fissato per tutti per tutti gli elementi
print (a.dtype) # in questo caso un int
# tipi diversi si possono specificare:
b = np.array([2,3,4], 'd')
print (b.dtype)

# il metodo append() esiste solo per le liste:
l = [2,3,4]
l.append(5)
print (l)
a.append(5) # Riuscite a capire il Traceback?
# per aggiungere elementi la sintassi e':
a = np.insert(a, len(a), 5.2)
print(a) # Perche' compare 5 e non 5.2?

# le operazioni tra array sono molto immediate:
print (a+a)
print (a**2)
# ma attenzione a usare il metodo giusto:
A = np.array( [[1,1],
               [0,1]] )
B = np.array( [[2,0],
               [3,4]] )
print (A*B)           # elementwise product
print (A.dot(B))      # matrix product
```

Media e deviazione standard con NumPy

Come prima, ma stavolta usando array[2]:

```
import numpy as np

def getstatsNP(sample):
    mean = sample.sum()
    variance = (sample**2).sum()
    n = len(sample)
    mean /= n
    variance = (variance - n*mean**2.0)/(n - 1)
    return mean, np.sqrt(variance)

# Vediamo i risultati:
dati = np.array([1,2,3,4,5,6]) # con int
print(getstatsNP(dati))
dati = np.array([1,2,3,4,5,6],'d') # con float
print(getstatsNP(dati))
```

Grafici con matplotlib

Esistono diverse librerie grafiche che consentono la creazione di grafici di vario tipo. In questa esperienza vedremo brevemente alcuni esempi con la libreria matplotlib. Cominciamo con un semplice grafico cartesiano:

```
# cominciamo ad importare la libreria
import numpy as np
import matplotlib.pyplot as plt
# inventiamoci un set di dati
x = np.arange(1,9,1)
y = np.array([2.08, 4.44, 6.09, 6.88, 8.61,
                  9.79, 11.9, 12.6])
xe = np.array(8*[0.1])
ye = np.array(8*[0.3])
# creiamo uno scatter plot
plt.errorbar(x, y, ye)
# magari mettiamo anche gli errori in x
plt.errorbar(x, y, ye, xe, 'o')
# e mostriamo il grafico finale
plt.show()
```

È bene imparare a formattare i grafici in modo che siano meglio comprensibili. Proviamo a trarre ispirazione dalle codice seguente, potete intuire a cosa servono le varie righe?

```
plt.rc('font', size = 18)
plt.title('Law of motion', y = 1.02)
plt.xlabel('t [s]')
plt.ylabel('s [m]', labelpad = 25)
plt.xlim(0, 10)
plt.ylim(0, 18)
plt.grid(color = 'gray')
```

Infine sarebbe tutto inutile se non foste in grado di salvare il grafico finale:

```
plt.savefig('mioplot.pdf') # o .png etc.
```

Oltre agli *scatterplot* ci sono altri tipi di grafici disponibili. Un altro esempio è il grafico a barre:

2. In realtà NumPy offre anche funzioni predefinite per per il calcolo delle statistiche campione

```
import numpy as np
import matplotlib.pyplot as plt
# inventiamo i x e y (valori e occorrenze)
valori     = np.array([1,2,3,4,5])
occorrenze = np.array([.2,.5,2,.6,.7])

plt.title('Law of motion', y = 1.02)
plt.xlabel('Valori')
plt.ylabel('Occorrenze')
plt.xlim(0, 6)
plt.ylim(0, 3)
plt.grid(color = 'gray')
plt.bar(valori, occorrenze, width = 1)
plt.savefig('barplot.png')
```

Notare come sono "centrate" le barre sull'asse x