Linguaggi di programmazione nella fusione

Edmondo Giovannozzi

Introduzione a Python.

Introduzione

- Interpretato
- Orientato agli Oggetti
- Vasta Libreria
- Molto usato nella analisi dei dati
- Specifiche del linguaggio:
  - 2.x compatibile con il passato.
  - 3.x ha alcune incompatibilità con la 2.x (inparticolare l'istruzione print è diventata una funzione).
- https://www.python.org/
- http://www.scipy.org/

Per iniziare

- Installare Versione(2.x) > 2.6:

  ipython, matplotlib, numpy, scipy
- Su linux:

  ipython --pylab

  matplotlib, numpy, scipy importati automaticamente
  - l'opzione --pylab è equivalente ad eseguire:
    `from matplotlib.pylab import *`
- Su windows lanciare pythonxy o equivalente:
  - https://code.google.com/p/pythonxy/
  - http://continuum.io/
  - https://www.enthought.com/
  - http://www.activestate.com/activepython

come calcolatore

```
>>> 3+5
    8
>>> 3.5 * 6.7
    23.4499999
>>> 7/3
     2
>>> 7.0/3
     2.33
>>> 7.0 // 3.0
     2.0
>>> 3**2
     9
```

Divisione intera se argomenti interi (Python 2.x)

Divisione intera

Assegnazione

Il segno di = assegna un nome ad una variabile che diventa raggiungibile.

```
>>>  sonounastringa = ‘Ciao come stai’
```

Nome

Variabile

```
>>> quelladiprima = sonounastringa
```

Altro nome

Variabile puntata dal nome

Per le variabili come le stringhe ed i numeri, che sono immutabili, nessuna differenza rispetto alla interpretazione consueta.

Liste

Le liste sono un primo esempio di variabili mutabili.

```
>>> lista_a = [1, 2, 3]
>>> lista_b = lista_a
```

i due nomi **lista_a**, e **lista_b** puntano alla stessa lista

```
>>> lista_a.append(7)
>>> lista_b
    [1, 2, 3, 7]
```

Diversi tipi di variabili sono mutabili: liste, dizionari, set, oggetti (tra cui i numpy array).
Tra i tipi immutabili abbiamo: numeri, stringhe e tuple (ed i frozen_set)

Liste, e tuple

```
>>> a = [1, 2, ‘Ciao’, 3.4]
>>> a_singola = [3.4]

>>> b = (1,2, ‘ciao’)
>>> b_singola = (3.4, )

>>> a[0]
   1
>>> b[-1]
  ‘ciao’
>>> a[0:2]
  [1,  2]
```

Lista

Lista con un solo elemento

Tupla

Tupla con un solo elemento
notare la virgola finale

Gli indici partono da 0

Gli indici negativi partono dalla fine

In una sezione l’ultimo indice viene escluso

spacchettamento tuple

```
>>> (a, b) = (1, 2)
>>> a
    1
>>> b
    2
```

spacchettamento della tupla

```
>>> aa, bb = 3, 4
>>> aa
    3
>>> bb
    4
```

Le parentesi non sono necessarie

```
>>> aa, bb = bb, aa
>>> aa
    4
>>> bb
    3
```

Usato per scambiare i valori di due variabili e per spacchettare i valori tornati dalle funzioni

Sezioni

```
>>> a = [1,2,3,4,5]
>>> a[:3]
    [1, 2, 3]
>>> a[2:]
    [3, 4, 5]
>>> a[2:-1]
    [3, 4]
>>> a[::-1]
    [5, 4, 3, 2, 1]
>>> a[:0:-1]
    [5, 4, 3, 2]
>>> len(a)
     5
```

I primi 3 elementi

a partire dal terzo fino alla fine

a partire dal terzo escludendo l'ultimo

in ordine inverso

in ordine inverso escludendo il primo

numero di elementi

comprehension et al.

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [ i**2 for i in a if i>2]
    [9, 16, 25]

>>> aa = [1, 2, 3]
>>> bb = [10, 20, 30]
>>> cc = zip(aa, bb)
     [(1, 10), (2, 20), (3, 30)]

>>> dd, ee = zip(*cc)
>>> dd
   (1, 2, 3)
>>> ee
   (10, 20, 30)
```

List coprehension, per generare una lista a partire da un'altra.

zip, per accoppiare elementi di più liste.

Può anche essere usato per l'operazione inversa.

array

```
>>> import numpy as np
>>> a = np.linspace(0, 2, 5)
>>> a
  array([ 0., 0.5,  1., 1.5, 2.])
>>> a.size
   5
>>> b = np.zeros((2,4))
>>> b
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> b.shape
    (2, 4)
>>> c = np.array([1.0, 2.0, 3.0])
>>> c.dtype
   dtype('float64')
```

spaziati linearmente

il numero totale di elementi dell'array

Notare che le dimensioni sono passate con una tuple

a partire da una lista

il tipo numerico sottostante

operazioni tra array

```
>>> r = np.random.random((2,3))
array([[ 0.90895715,  0.27791328,  0.08318425],
       [ 0.643648  ,  0.83921606,  0.32521858]])
>>> a = np.array([1, 2, 3])
>>> r*a
array([[ 0.90895715,  0.55582656,  0.24955276],
       [ 0.643648  ,  1.67843213,  0.97565575]])
>>> b = np.array([10,100])
>>> r*b[:,np.newaxis]
array([[  9.08957148,   2.77913282,   0.83184253],
       [ 64.36480046,  83.92160635,  32.52185839]])
```

r

0.90	0.27	0.08
0.64	0.83	0.32

* a

1	2	3
1	2	3

↓ Esteso automaticamente sulle dimensioni iniziali

r

0.90	0.27	0.08
0.64	0.83	0.32

* b

10	10	10
100	100	100

→ Esteso tramite **newaxis.** Aggiunge un asse di dimensioni unitarie, che poi si estende automaticamente

Dizionari

```
>>> aa = {}
>>> aa['ciao'] = 56
>>> aa['test'] =  3.46
>>> aa
   {'ciao': 56, 'test': 3.46}
>>> aa['test']
   3.46
>>>  aa.keys()
   ['ciao', 'test']
>>> aa.values()
   [56, 3.46]
>>> aa.items()
   [('ciao', 56), ('test', 3.46)]
>>> len(aa)
   2
```

Mutabili. Gli elementi si accedono tramite una chiave. Non necessariamente una stringa.

Elenco delle chiavi e dei valori.

Grafica

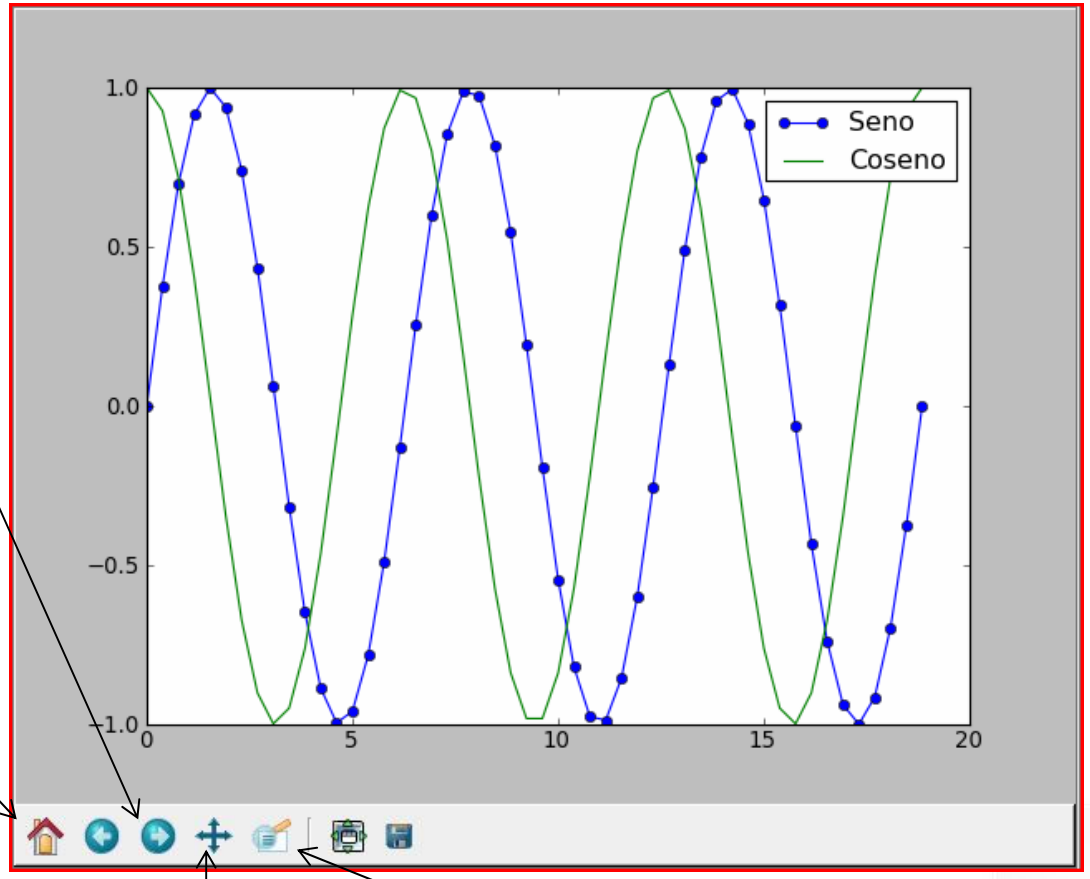
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> t = np.linspace(0, 6*np.pi)
>>> y = np.sin(t)
>>> plt.plot(t, y, '-o', label='Seno')
>>> plt.plot(t, np.cos(t), label='Coseno')
>>> plt.legend()
>>> plt.show()
```

Con l'opzione **--pylab** gli import non sono necessari come non è necessario qualificare le funzioni con **np** o **plt**. Il comando **show**() finale è anche superfluo, non però all'interno di script.

```
In[1]: t = linspace(0, 6*pi)
In[2]: plot(t, sin(t), label='Seno')
In[3]: plot(t, cos(t), label='Coseno')
In[4]: legend()
In[5]: clf()  # Cancella la figura
```

Figure

Ci si muove avanti ed indietro lungo la storia degli zoom o spostamenti effettuati.



Riporta la figura alle dimensioni originarie

Zoom spostamento dinamico. Spostamento se si muove il mouse con il tasto sinistro abbassato. Zoom se si spinge il tasto destro

zoom rettangolare

Esecuzione di un programma

- All'interno di IPython

  In [1]: %run nomeprogramma

- all'interno di qualsiasi interprete Python

  >>> import nomeprogramma

- Dalla shell del sistema operativo

  python nomeprogramma.py

Nel seguito lanceremo i programmi con %run all'interno di IPython

Primo programma

saluto.py

```
def ciao(cosa):
    print ‘Ciao ’ + cosa
```

Definizione di una funzione.
cosa è un argomento

```
>>> %run saluto
>>> ciao('a tutti')
Ciao a tutti
```

Con **%run** le definizioni sono direttamente accessibili, con **import** sono qualificate dal nome del **modulo** importato

```
>>> import saluto
>>> saluto.ciao(‘anche a te’)
Ciao anche a te
```

Primo programma

saluto.py

```
def ciao(cosa):
    completo = ‘Ciao ’+cosa
    print completo
```

I blocchi iniziano con «:» nella istruzione che precede il blocco.

saluto.py

```
def ciao(cosa):
ÞÞÞcompleto = ‘Ciao ’+cosa
ÞÞÞprint completo
def …
```

Il blocco di istruzioni termina quando l’indentazione ritorna al livello precedente.

Non ci sono rispetto agli altri linguaggi dei terminatori del blocco (come } in C/C++).

Un programma così è automaticamente scritto ordinatamente.

Principali strutture (if)

esempio.py

```
def controlla(val):
    if val>0:
        print ‘val > 0’
    elif val == 0:
        cosa = val + 5
        print cosa
    else:
        print ‘val < 0’
```

If

```
if <condizione1> :
    <istruzioni1>
elif <condizione2> :
    <istruzioni2>
else:
    <istruzioni3>
```

```
>>> %run esempio
>>> controlla(7)
val > 0
>>> controlla(0)
5
```

espressioni logiche

```
>>> (3 > 2 or 5 < 3) and not 7 != 7
    True

>>> 2 in [1, 2, 3]
    True
>>> 2 not in [1, 2, 3]
    False

>>> a = [1, 2, 3]
>>> b = a
>>> b is a
    True
>>> b = a[:]
>>> b is a, b == a
    (False, True)

>>> 1 < 2 < 3
   True
>>> 1 < 4 < 3
   False

>>> 'Grande' if 5 > 3 else 'piccolo'
   'Grande'
>>> 'Grande' if 5 > 8 else 'piccolo'
   'piccolo'
```

I classici operatori

Controlla se un elemento appartiene ad una sequenza

Controlla se due nomi si riferiscono allo stesso oggetto. Vedere la differenza con == che controlla se due oggetti hanno lo stesso valore

Operatori di comparazione possono essere messi uno dopo l'altro, il significato è ovvio.

espressione con if in linea

bitwise «or» «and» «not»

```
>>> a = np.array([1,2,3,4])
>>> b = np.array([8,0,3,7])
>>> (b == a) | (b > a)          | OR
  array([ True, False,  True,  True], dtype=bool)
>>> ~ (b == a)                  ~ NOT
  array([ True,  True, False,  True], dtype=bool)
>>> (b >= a) & (b == a)         & AND
  array([False, False,  True, False], dtype=bool)
>>> (b > a) ^ (b < a)           ^ XOR
  array([ True,  True, False,  True], dtype=bool)
```

Gli operatori bitwise possono essere usati creare dei vettori logici. Hanno una precedenza superiore agli operatori di comparazione, perciò devono essere protetti con delle parentesi.

vettori logici et al.

```
>>> a = np.arange(6)
  array([ 0, 10, 20, 30, 40, 50])
>>> a[a<20] = -2
  array([-2, -2, 20, 30, 40, 50])
>>> np.flatnonzero(a>20)
  array([3, 4, 5])


>>> b = np.r_[3:7, 21]
  array([ 3,  4,  5,  6, 21])


>>> a = np.arange(0,30,10)
>>> b = np.arange(3)
>>> np.hstack((a,b))
 array([ 0, 10, 20,  0,  1,  2])
>>> c = np.vstack((a,b))
 array([[ 0, 10, 20],
        [ 0,  1,  2]])
>>> c.ravel()
 array([ 0, 10, 20,  0,  1,  2])
```

Un vettore logico può essere usato al posto degli indici per selezionare degli elementi.

La funzione flatnonzero ritorna gli indici stessi (del vettore considerato 1D).

hstack, **vstack** per concatenare i vettori (Attenzione in ingresso una tupla di vettori).

ravel per renderlo monodimensionale (C like)

argmin, argmax, etc.

```
>>> a = np.random.random((5,6))
>>> b = np.random.random((5,6))
```

Genero due matrici di numeri random.

```
>>> idm = a.argmax(axis=0)
>>> i2 = np.indices(idm.shape)
```

Cerco, con **argmax()**, la posizione del massimo in ogni colonna della matrice **a**. Non posso usare direttamente l'uscita di **argmax** per trovare il massimo, devo usare **indices** che mi restituisce il valore degli altri indici.

```
>>> print a.max(axis=0)
>>> print a[idm, i2]
```

Il metodo **max(axis=0)** mi dà direttamente il massimo lungo le colonne.

```
>>> print b[idm, i2]
```

Ma l'uso di **indices** mi permette di trovare i valori in **b** corrispondenti ai massimi di **a**.

Principali strutture (for)

esempiocicli.py

```
def controlla(vals):
    for v in vals:
        print 'v: {0}'.format(v)
        if v<0: break
    else:
        print 'tutti > 0'
```

for

```
for <var> in <iterable>:
    <istruzioni>
    … break
    … continue
else:
    <istruzioni else>
```

```
>>> %run esempiocicli
>>> controlla([1,2])
v: 1
v: 2
tutti > 0
>>> controlla([1,-2,3])
v: 1
v: -2
```

Principali strutture (for)

esempiocicli2.py

```
def finoa(n):
    for i in range(n):
        print 'i: {}'.format(i)
```

```
>>> %run esempiocicli2
>>> controlla(3)
i: 0
i: 1
i: 2
>>> range(3)
 [0, 1, 2]
>>> range(2, 8, 3)
 [2, 5]
```

la funzione:

`range(start, stop, step)`

ritorna una lista di interi:

`start` :0 se non specificato
`stop` : escluso
`step` :1 se non specificato

Iteratori e Generatori

Le liste, le tuple gli array possono essere trasformate in un iteratore ed usate nei cicli for.

Un particolare tipo di iteratori si ottengono tramite i generatori, funzioni che contengono l'istruzione **yield**.

```
>>> %run generatore
2
3
5
8
13

>>> list(fibonacci(7))
 [2, 3, 5, 8, 13, 21, 34]
```

generatore.py

```
def fibonacci(n):
    a, b = 1, 1
    for i in range(n):
        a, b = b, a+b
        yield b

for i in fibonacci(5):
    print i
```

Eccezioni (try..except)

esempiotry.py

```
def testtry(n):
    a = [1,2,3]
    try:
        print ‘a = ‘, a
        print ‘a[n] = ‘, a[n]
    except IndexError:
        print ‘Non ci siamo’
```

try

```
try:
    <istruzioni1>
except <eccezioni>:
    <istruzioni2>
```

```
>>> %run esempiotry
>>> testtry(2)
a =  [1, 2, 3]
a[n] =  3
>>> testtry(7)
a =  [1, 2, 3]
a[n] =  Non ci siamo!
```

Ci sono altre possibilità, tipo la clausola **finally:**, **else:**, etc.

Che non mostriamo per semplicità.

Lettura file e **with:**

letturafile.py

```
def leggimi():
    with open('letturafile.py') as f:
        for i, line in enumerate(f):
            print i, line,
```

```
>>> %run letturafile.py
>>> leggimi()
0 def leggimi():
1     with open('letturafile.py') as f:
2         for i, line in enumerate(f):
3             print i, line,

>>> list(enumerate(['a','b','c']))
    [(0, 'a'), (1, 'b'), (2, 'c')]
```

Si può usare **enumerate** per avere anche l'indice di un iterabile.

Lettura matrici da file

Il modulo numpy provvede funzioni per la lettura di matrici o tabelle di numeri. Esistono anche funzioni per la scrittura o lettura di file matlab

```
>>> import numpy as np
>>> a = np.genfromtxt('tabella.txt',names=True)
>>> a['t']
   array([ 1. ,  2.3,  3.1])
>>> a['te']
   array([ 10.,  35.,  21.])
>>> a['ne']
    array([ 100.,  118.,  250.])
```

tabella.txt

t	te	ne
1	10	100
2.3	35	118
3.1	21	250

Funzioni

```
>>> %run funzioni
>>> pluto(2)
a =  2
b =  3.0
c =  [3.0]
  (2, 3.0, [3.0])
>>> pluto(1,c=[5, 6])
a =  1
b =  3.0
c =  [5, 6, 3.0]
   (1, 3.0, [5, 6, 3.0])
>>> aa, bb, cc = pluto(2)
>>> aa
   2
>>> bb
   3.0
>>> cc
  [3.0]
```

funzioni.py

```
def pluto(a, b=3.0, c=None):

   print 'a = ', a
   print 'b = ', b

   if c is None:
      c = []
   c.append(b)

   print 'c =', c

   return a, b, c
```

Docstring

Una Docstring è una stringa che documenta il codice. Deve essere la prima istruzione dopo la definizione di una funzione, classe, etc.

Tipicamente si usano le stringhe multiline che iniziano e terminano con: """

funzioni.py

```
def pluto(a, b=3.0, c=None):
    """ Test optional argument
    b and c are optional
    """
    print 'a = ', a
    print 'b = ', b

    if c is None:
        c = []
    c.append(b)

    print 'c =', c

    return a, b, c
```

None

- Usato per indicare la mancanza di qualche cosa
- E' un tipo a se stante **NoneType** che ha una sola variabile di quel tipo ovvero **None**.
- Una funzione che non ritorna nulla (ovvero che non ha un istruzione **return**) in realtà ritorna **None**.
- Si controlla l'uguaglianza di un oggetto con None tramite: *nomeoggetto* **is None**.
- Si usa tipicamente se abbiamo un argomento di default di tipo mutabile:

```
def pluto(arg=None)
    if arg is None:
        arg = []
```

- Attenzione diverso dal float NAN:

```
>>> a = np.sqrt(np.array([1.0, 2.0, -2.0, 3.0]))
>>> np.isnan(a)
 array([False, False,  True, False], dtype=bool)
```

Lista di argomenti variabile

```
>>> positionalarg(4, 5, 'Ciao')
(4, 5, 'Ciao')
>>> a = [4, 5, 'Ciao']
>>> positionalarg(*a)
(4, 5, 'Ciao')
```

argumentlist.py

```
def positionalarg(*arg):
    print arg

def keywordarg(**kwarg):
    print kwarg
```

Gli argomenti sono inseriti in una tupla. Una lista od una tupla con lo * davanti vengono spacchettate negli argomenti.

```
>>> keywordarg(pippo=5,pluto=6,topo='Ciao')
{'topo': 'Ciao', 'pippo': 5, 'pluto': 6}
>>> b = {'primo':3.4, 'sec':7.8}
>>> keywordarg(**b)
{'primo': 3.4, 'sec': 7.8}
```

Gli argomenti passati tramite keyword diventano dizionari (e viceversa).

Classi

```
>>> %run myclass
>>> a = MyClass('Pluto')
>>> a
 Nome: Pluto
>>> a.add(' e Clara')
>>> a
 Nome: Pluto e Clara
>>> dir(a)
['__class__',
 ....
 '__init__',
 ....
 'add',
 'nome']
```

myclass.py

```
class MyClass(object):
   def __init__(self, nome):
      self.nome = nome

   def __repr__(self):
      rstr = [‘Nome: ‘ + str(self.nome)]
      return “\n”.join(rstr)

   def add(self, suff):
      self.nome += suff
```

MyClass discende da object (sempre consigliabile).
Ha tre metodi:
__init__ il costruttore.
__repr__ viene invocato quando Python vuole rappresentarlo. ritorna una stringa.
add che aggiunge un suffisso al nome.
Il primo argomento di ogni metodo è **self** corrispondente all’oggetto stesso.

Classe Ereditarietà

```
>>> %run myclass2

>>> a = Point(2,3)
>>> a
 x=2, y=3

>>> b = Square(2,3,5)
>>> b
 x=2, y=3 w=5

>>> isinstance(b, Square)
   True
>>> isinstance(b, Point)
   True
```

myclass2.py

```
class Point(object):
   def __init__(self, x, y):
      self.x, self.y = x, y

   def __repr__(self):
      return 'x={0}, y={1}'.format(self.x,self.y)

class Square(Point):
   def __init__(self, x, y, width=1.0):
      super(Square,self).__init__(x,y)
      self.width = width

   def __repr__(self):
      rst = super(Square,self).__repr__()
      return rst + ', w='+str(self.width)
```

Square discende da Point che discende da object.
I metodi della classe genitore si accedono tramite **super**.
Per controllare se un oggetto appartiene ad una clase che discende da quella data si usa **isinstance**. Anche se si preferisce il duck typing, provare ad accedere ai metodi, al massimo darà un errore che si intercetta con **try…except**.

Convenzioni e metodi standard

- In Python nulla è realmente protetto, anche se il linguaggio definisce delle convenzioni che forzano una sorta di protezione.
- In genere metodi o routine che iniziano con un underscore sono considerati privati (ma non c'è nulla che li protegga realmente).
- Metodi che iniziano con due underscore sono privati e parzialmente protetti quando la classe viene estesa.
- I metodi speciali iniziano e terminano con due underscore:
  - __init__ costruttore
  - __repr__ rappresentazione
  - __getattr__ per emulare l'accesso ad una attributo:
    - a.x --> a.__getattr('x')
  - __getitem__ per emulare un tipo contenitore (tipo le liste):
    - a[x] --> a.__getitem__('x')
  - __add__, __mul__, etc. per emulare somme, moltiplicazioni etc.

Conclusioni

- Python è un linguaggio molto esteso. Questa è stata solo una introduzione.
- Scipy è una libreria che permette, FFTs, integrazioni numeriche, ODE, etc.
- Moltissime librerie sono disponibili su internet.
- Grazie per il vostro tempo!