# Linguaggi di programmazione nella fusione

Edmondo Giovannozzi

Introduzione a Python.

edmondo.giovannozzi@enea.it

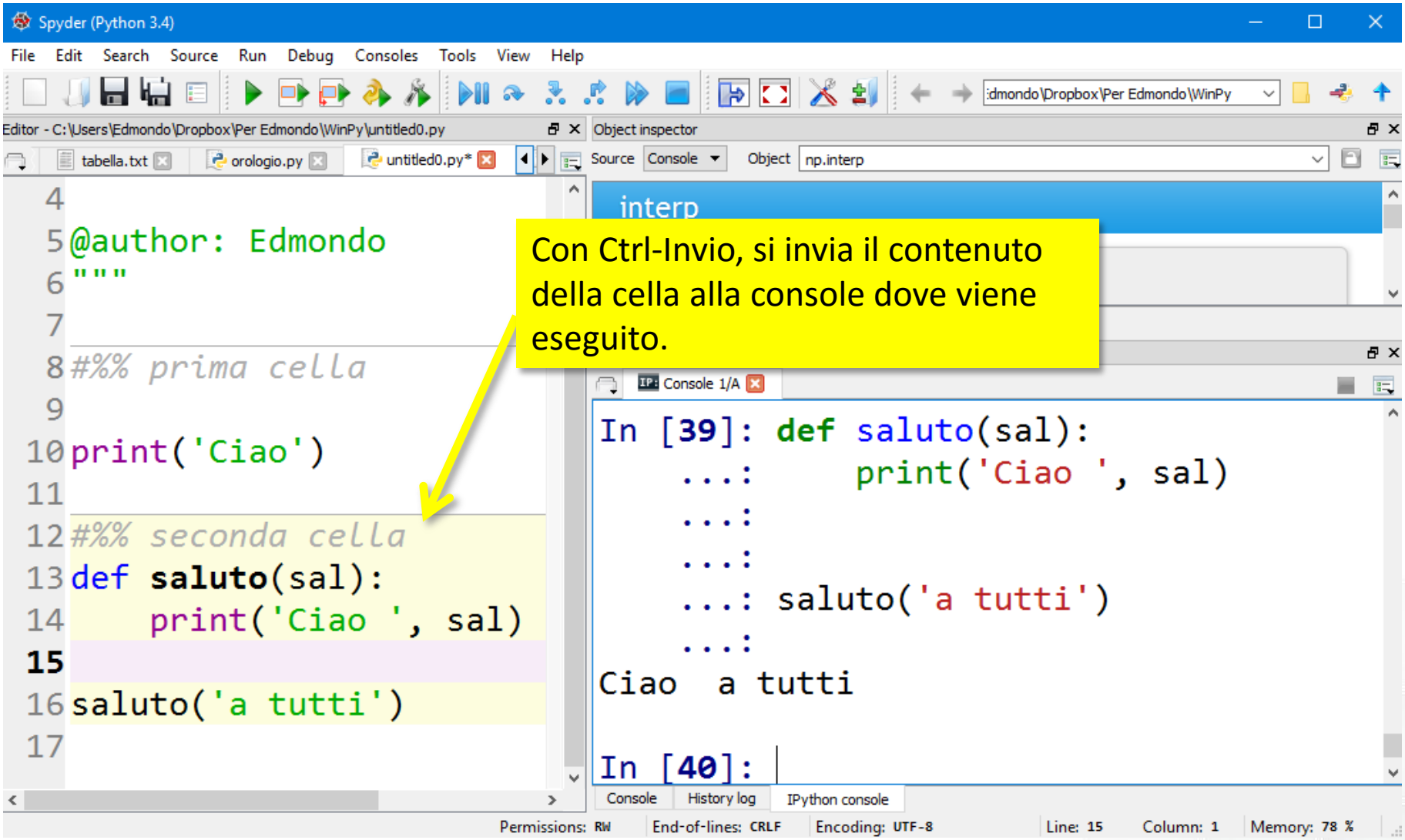
# Introduzione

- 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 (in particolare l'istruzione print è diventata una funzione).
- Help su:
  - https://www.python.org/
  - http://www.scipy.org/
- Distribuzioni che useremo:
  - Winpython (http://winpython.github.io/)
  - Anaconda (https://www.continuum.io/)
  - Altre distribuzioni (enthought, activestate).
- Diverse implementazioni di Python: CPython (quella di default), Jithon (java). IronPython (.NET), etc.

# Come eseguire Python

- Spyder (IDE)
  - Console, editor dei programmi, varie viste (variabili, outline, etc.)
- Ipython (Jupiter) notebook
  - All'interno di un browser, celle esegibili e di testo (supporta LaTex per le formule)
- IPython + editor esterni (geany, etc.)
- Su linux la prima riga di un file se contiene "#!/*path*/python" permette di eseguire il file con python.
  - gli argomenti della linea di comando sono disponibili in **sys.argv**, ma è meglio usare il modulo **argparse**.

# spyder


Spyder (Python 3.4)
File Edit Search Source Run Debug Consoles Tools View Help
Editor - C:\Users\Edmondo\Dropbox\Per Edmondo\WinPy\untitled0.py
tabella.txt
orologio.py
untitled0.py*
4
5 @author: Edmondo
6 """
7
8 #%% prima cella
9
10 print('Ciao')
11
12 #%% seconda cella
13 def saluto(sal):
14 print('Ciao ', sal)
15
16 saluto('a tutti')
17
Object inspector
Source Console Object np.interp
interp
Con Ctrl-Invio, si invia il contenuto della cella alla console dove viene eseguito.
Console 1/A
In [39]: def saluto(sal):
...: print('Ciao ', sal)
...:
...:
...: saluto('a tutti')
...:
Ciao a tutti
In [40]:
Console History log IPython console
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 15 Column: 1 Memory: 78 %

# ipython notebook

jupyter testpercorso

File Edit View Insert Cell Kernel Help | Python 3

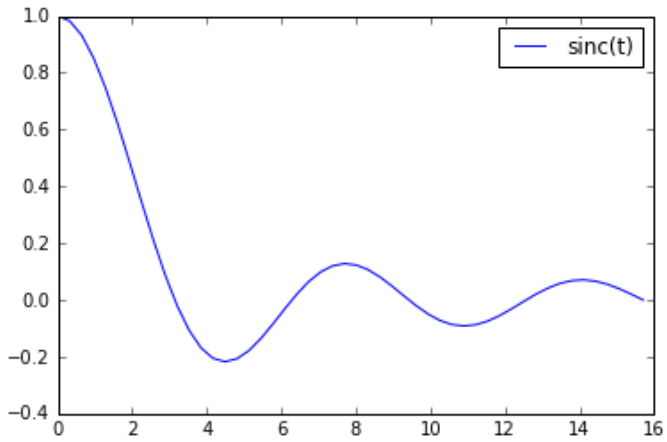
Code | Cell Toolbar: None

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

In [6]: t = np.linspace(0.001,5*np.pi)
        y = np.sin(t)/t
```

Plot della funzione:

$$\text{sinc} = \frac{\sin(t)}{t}$$

```
In [10]: plt.plot(t,y,label="sinc(t)");
         plt.legend();
```




All'interno di un browser.

Si possono inserire celle di testo anche con delle formule scritte in LaTex

Le figure sono inserite nel testo.

I file in formato .ipynb possono essere scambiati.

# Python 2 vs Python 3

Principali differenze (utente normale)

- print è una funzione in Py3:
  - Py2: >>> print 'Ciao: ', 34
  - Py3: >>> print('Ciao: ', 34)
- Divisione tra interi:
  - Py2: >>> 3/2 → 1
  - Py3: >>> 3/2 → 1.5
- In Py3 range, map, zip, etc. vengono espansi quando servono:
  - Py2: >>> **range**(5) → [0, 1, 2, 3, 4]
  - Py3: >>> **range**(5) → range(0, 5)
  - Py3: >>> **range**(0,50,10)[3] → 30
- In Py3 le stringhe sono formate da caratteri unicode .

# 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** (IPython) 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
```

# Moduli e package

```
>>> import numpy
```
Importa il package **numpy**

```
>>> import numpy as np
```
Importa il package **numpy** cambiandogli il nome

```
>>> from numpy import linalg
```
Importa il modulo **linalg** del package **numpy**

```
>>> from numpy.linalg import lstsq
```
Importa la funzione **lstsq** del modulo **numpy.linalg**

Python è organizzato con moduli e package, ogni funzionalità aggiuntiva si ottiene importando il relativo modulo:

```
>>> import re
>>> import os
>>> import sys
>>> import argparse
Etc.
```

Espressioni regolari

Accesso al sistema operativo

Informazioni sul sistema

Parsing degli argomenti

# 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):
␣␣␣␣completo = 'Ciao '+cosa
␣␣␣␣print(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 viene così scritto automaticamente in maniera ordinata.

# Assegnazione

Il segno di = associa ad un nome un riferimento ad un oggetto che diventa raggiungibile.

```
>> sonounastringa = 'Ciao come stai'
```

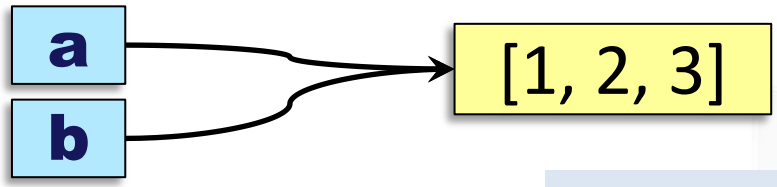
Nome — Oggetto

```
>> quelladiprima = sonounastringa
```

Altro nome — Oggetto puntato dal nome

```
>> a = [1,2]
>> b = a
>> a.append(3)
>> b
   [1, 2, 3]
```

Per le variabili come le stringhe ed i numeri, che sono immutabili, nessuna differenza rispetto alla interpretazione consueta.
Diverso è il comportamento delle variabili mutabili come le liste.



# Principali tipi predefiniti

## Immutabili:

| | | |
|---|---|---|
| numerici | interi | >> 2 |
| | reali | >> 5.6 |
| | complessi | >> 1 + 2j |
| | booleani | >> true<br>>> false |
| stringhe | >> "Ciao come stai"<br>>> 'Bene e tu' | |
| tuple | >> (1,)<br>>> (1,2,'Ciao',4)<br>>> **tuple**(…) | |
| range | >> range(8)<br>>> range(0,4,2) | |
| frozenset | >> frozenset((1,'Ciao',4.5)) | |
| NoneType | >> None | |

## Mutabili:

| | |
|---|---|
| liste | >> []<br>>> [1,2,'Ciao']<br>>> **list**((1, 2, 'Ciao')) |
| dizionari | >> {1:'primo', 'due':4.5}<br>>> **dict**([(1, 'primo'), ('due',4.5)]) |
| set | >> {'ciao', 4.5, 7}<br>>> **set**([4.5, 7, 'ciao']) |

Da numpy

| | |
|---|---|
| ndarray | >> np.array([1,2,3,4]) |

Sono mutabili anche gli oggetti istanze di una classe definita dall'utente.

| |
|---|
| sequenze |
| set |
| mapping |

# numeri

```
>> 3 + 5
    8
>> 3.5 * 6.7
    23.4499999
>> 7 / 3
    2.33
>> 7.0/3
    2.33
>> 7.0 // 3.0
    2.0
>> 3**2
    9
>> 8 % 3
    2
>> 1 + 5j
   (1+5j)
>> True and False
   False
>> True or False
   True
```

| Operatori | |
|---|---|
| +, - | Somma, sottrazione |
| *, / | Moltiplicazione, divisione |
| //, % | Divisione intera, resto (modulo) |
| ** | Elevazione a potenza |
| @ | Moltiplicazione matriciale (almeno Python 3.5, numpy 1.10) |

In Python 2 la divisione "/" è una divisione intera se gli operandi sono interi.

# stringhe

```
>> "Ciao come stai"
   'Ciao come stai'

>> 'Bene e tu?'
    'Bene e tu'

>> print(' Bene! \n Bene!')
 Bene!
 Bene!

>> print(r' Bene! \n Bene!')
 Bene! \n Bene!

>> a = """Ciao,
come va?
Bene!"""

>> print(a)
Ciao,
come va?
Bene!

>> len(s)
   6
```

Singole o doppie quote equivalenti

Come in C "\n" viene interpretato come un carattere di line-feed (a capo su unix).

La r disabilita l'interpretazione della barra inversa.

Le tre doppie virgolette iniziano e terminano delle stringhe che si sviluppano su più linee (mantenendo gli a capo correttamente)

Con len la lunghezza

# Stringhe (2)

```
>> 3 * "Ciao, " + "Roma!"
   'Ciao, Ciao, Ciao, Roma!'
```

Si ripetono con * e si concatenano con +

```
>> s = "abcdef"
>> s[3]
   'd'
>> s[3:5]
   'de'
```

Si estrae un carattere od una sottostringa (a partire da zero)

Sono immutabili, non possono essere cambiate

```
>> s[3] = 'K'
TypeError: 'str' object does not support item assignment
```

```
>> "ma che dici".split()
 ['ma', 'che', 'dici']
```

In Python3 le stringhe sono formate da caratteri unicode e non bytes. I metodi *obj*.**encode()**, *obj*.**decode()** le trasformano in bytes e viceversa.

| Operatori e metodi | |
|---|---|
| **+** | concatenazione |
| ***** | ripetizione |
| *obj*.**upper()** | Maiuscolo |
| *obj*.**lower()** | minuscolo |
| *obj*.**split()** | spezzetta |
| *obj*.**startswith(***pref***)** | True se inizia con *pref* |

# Conversioni

```
>> a = 3.2
>> type(a)
   float
>> b = str(a)
>> b
  '3.2'
>> float(b)
  3.2
>> int(b)
ValueError: invalid literal for int() with base 10: '3.2'
>> int('54')
  54
>> " %d  %5.2f" % (5, 6.7)
 ' 5   6.70'
>> " {1:02d} {0} {pluto}".format(5.3, 2, pluto=6.7)
 ' 02 5.3 6.7'
```

| Funzioni di conversione | |
|---|---|
| **str** | trasforma in stringa |
| **int** | trasforma in intero |
| **float** | trasforma in float |
| *%* | formattazione C like |
| *obj*.**format()** | formattazione avanzata |

non è trasformabile in un intero

# 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[1] = 'Mondo'
 [1, 'Mondo', 'Ciao', 3.4]

>> b[1] = 'Mondo'
TypeError: 'tuple' object does not support item assignment
```

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

Le tuple sono immutabili, non possono essere modificate.

# Liste altre operazioni

```
>> a = [1,2,3,4]
>> a[2:2] = [10,20,30]
   [1, 2, 10, 20, 30, 3, 4]
>> del a[2:3]
   [1, 2, 20, 30, 3, 4]
>> a.append(5)
 [1, 2, 20, 30, 3, 4, 5]
>> [1,2] + [5,6]
  [1, 2, 5, 6]
>> 3 * [3,4]
  [3, 4, 3, 4, 3, 4]
>> del a
>> a
  NameError: name 'a' is not defined
```

Inserire elementi in una lista.

cancellare elementi da una lista.

appendere elementi ad una lista

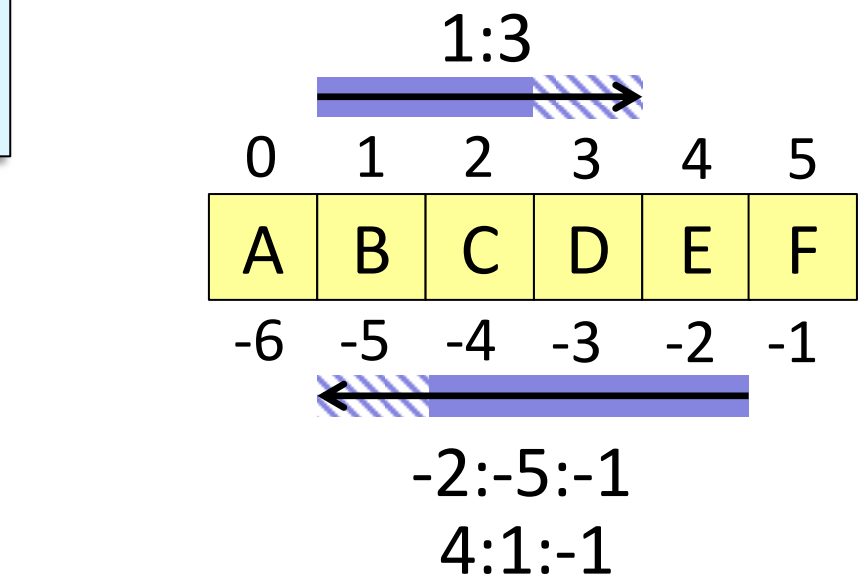
Concatenazione e ripetizione di una lista

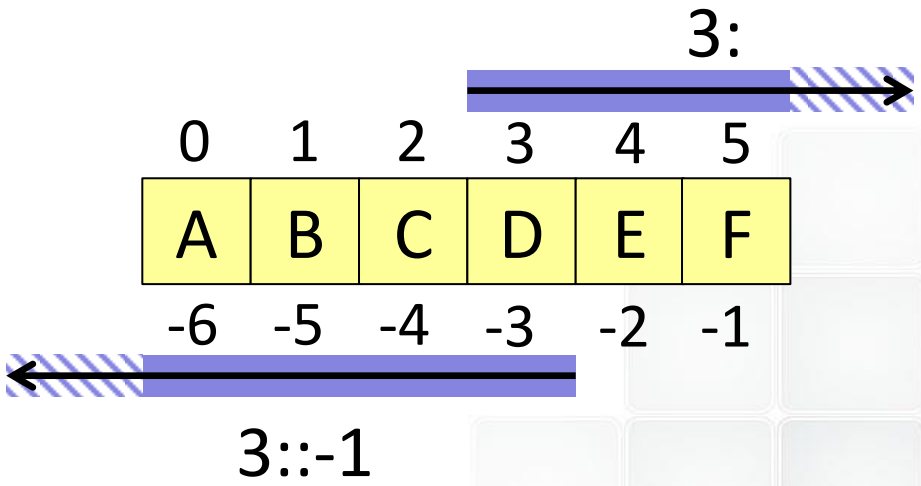
L'istruzione **del** si può usare per cancellare una variabile.

# Sezioni (slice)

Si applicano alle sequenze: liste, tuple, strighe, range e ndarray.

```
>> s = "ABCDEF"
>> s[:3]        → 'ABC'
>> s[1:3]       → 'BC'
>> s[-2:-5:-1]  → 'EDC'
>> s[4:1:-1]    → 'EDC'
>> s[3:]        → 'DEF'
>> s[3::-1]     → 'DCBA'
>> s[::-1]      → 'FEDCBA'
>> len(s)       → 6
```



# spacchettamento sequenze

```
>> (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

# set

```
>> a = {'Ciao', 'b', 1, 2}
>> b = {'Bene', 2, 'b', 3}
>> a | b
 {1,2,'b',3,'Ciao','Bene'}
>> a & b
 {2, 'b'}
>> a - b
 {1, 'Ciao'}
```

| Operatori | |
|---|---|
| **\|** | Unione |
| **&** | Intersezione |
| **-** | differenza fra set |
| **^** | differenza simmetrica |
| **<, >, <=, >=** | test se sottoinsieme. |
| ***obj*.add(*elem*)** | aggiungi un elemento al set |
| **in** | Test di appartenennza |

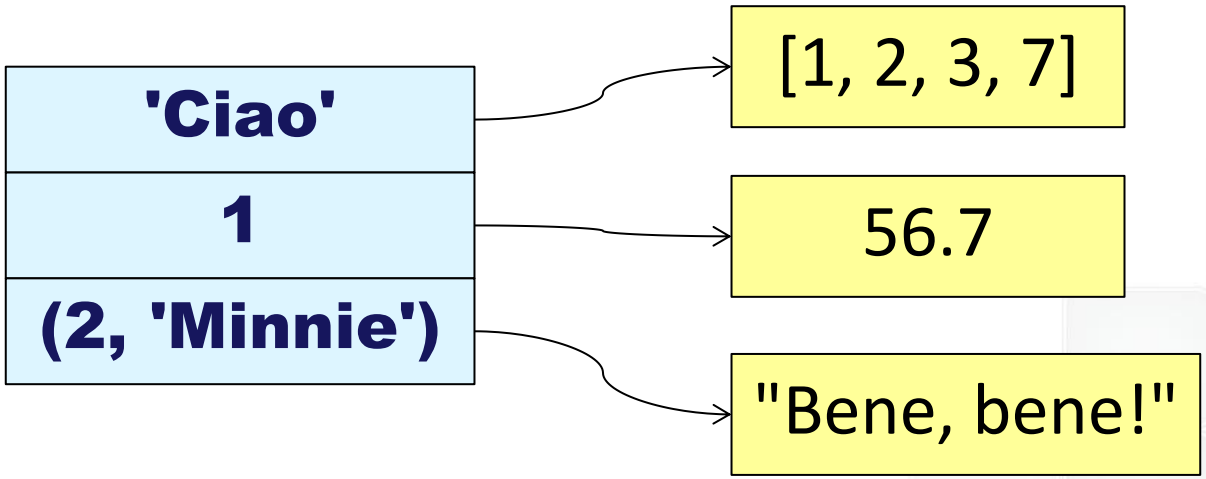
```
>> {1,2} < a
  True
>> 'Bene' in b
  True
```

I **set** sono mutabili, i **frozenset** immutabili.
Gli elementi di un set devono essere **hashable**, ovvero:

- Immutabili, e se contenitori contenere solo elementi **hashable**.
- Oggetti definiti dall'utente (salvo alcuni casi), ogni oggetto è diverso da chiunque altro (anche se della stessa classe)

# Dizionari

- Associa a delle variabili chiave (a parità di valore) altre variabili (riferimenti ovviamente).
- Usati in molte parti di Python per il funzionamento interno.
- Le chiavi devono essere **hashable**.



# Dizionari

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

Mutabili. Gli elementi si accedono tramite una chiave, che deve essere **hashable**.

Elenco delle chiavi e dei valori.

In Python 2.x sono delle liste

Esistenza di una chiave

# comprehension et al.

```
>> a = [1, 2, 3, 4, 4]
>> alist = [i**2 for i in a if i>2]
   [9, 16, 16]
>> aiter = (i**2 for i in a if i>2)
   <generator object <genexpr> at …>
>> list(aiter)
   [9, 16, 16]
>> aset = {i**2 for i in a if i>2}
    {9, 16}
>> adict = {i:i**2 for i in a if i>2}
    {3: 9, 4: 16}
>> list(zip([1,2],[3,4],[10,20]))
   [(1, 3, 10), (2, 4, 20)]
```

Comprehension può creare liste, generatori, set e dizionari.

Per accoppiare elementi di più liste: zip
c = **zip**(lista1, lista2, lista3)
Per disaccoppiarli:
l1, l2, l3 = zip(*c)

# ndarray

```
>> 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

Le dimensioni sono passate con una tuple od una lista.

a partire da una lista

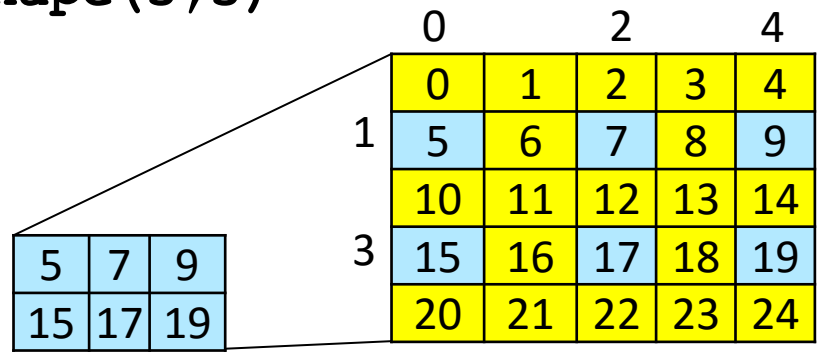
il tipo numerico sottostante

# Sezioni (slice) di un array

```
>>> a = np.arange(25).reshape(5,5)
>>> b = a[1:4:2,0:5:2]
```

1, 3

0, 2, 4

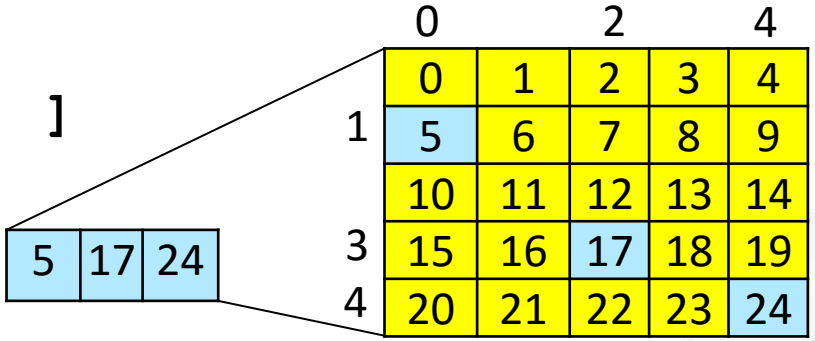



E' una vista, se si modifica **b** anche **a** viene modificata

## Indexing Avanzato:

```
>>> c = a[ [1,3,4], [0,2,4] ]
```

I dati vengono copiati. Modificare **c** non cambia **a**.



# operazioni tra array

```
>> r = np.random.random((2,3))
array([[ 0.90,  0.27,  0.08],
       [ 0.64,  0.83,  0.32]])
>> a = np.array([1, 2, 3])
>> r * a
array([[ 0.90,  0.54,  0.24],
       [ 0.64,  1.66,  0.96]])
>> b = np.array([10,100])
>> r * b[:,np.newaxis]
array([[  9.0,   2.7,   0.8],
       [ 64.0,  83.0,  32.0]])
```

Estensione automatico delle dimensioni unitarie degli array (broadcasting).

Si assume che ogni array sia preceduto da tutte le dimensioni unitarie che servono.

L'oggetto numpy **newaxis** serve ad inserire delle dimensioni unitarie al posto giusto.

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.

# 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'
```

| Operatori | |
|---|---|
| **and, or, not** | **and**, **or** e **not** logico |
| **in** | appartenenza |
| **not in** | non appartenenza |
| **>, <, >=, <=, ==, !=** | uguaglianze e disuguaglianze |
| **is** | identità |
| **is not** | non identità |

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)
   array([ True, False,  True,  True], dtype=bool)
>> ~ (b == a)
   array([ True,  True, False,  True], dtype=bool)
>> (b >= a) & (b == a)
   array([False, False,  True, False], dtype=bool)
>> (b > a) ^ (b < a)
   array([ True,  True, False,  True], dtype=bool)
```

| OR

~ NOT

& AND

^ XOR

Gli operatori bitwise possono essere usati per 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)*10
   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])
>> c = np.c_[3:6, 10:40:10]
   array([[ 3, 10],
          [ 4, 20],
          [ 5, 30]])
>> c.ravel()
 array([ 3, 10, 4, 20, 5, 30])
>> a = np.array([3,1,2])
>> a.sort()
   [1, 2, 3]
```

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

| metodi e funzioni | |
|---|---|
| *np*.**arange** | genera un vettore in sequenza |
| *np*.**r_**[...] | concatena lungo le righe |
| *np*.**c_**[...] | concatena per colonne |
| *np*.**flatnonzero** | indici dei valori diversi da 0 |
| *obj*.**ravel**() | come vettore 1D (per righe) |
| *obj*.**sort**() | ordina il vettore sul posto |
| *obj*.**T** | trasposta |
| *obj*.**copy**() | crea una copia |
| *obj*.**reshape**() | cambia lo shape |
| *obj*.**shape** | shape dell'array |
| *obj*.**size** | numero di elementi |

# argmin, argmax, etc.

```
>> a = np.array([[10, 20, 3, 4],
                 [2,   7, 5, 6]])
>> a.amin(axis=0)
   array([2, 7, 3, 4])
>> id0 = a.argmin(axis=0)
>> id1 = np.r_[0:4];
   id0 = [1, 1, 0, 0]
   id1 = [0, 1, 2, 3]

>> a[id0, id1]
 array([2, 7, 3, 4])
```

| | |
|---|---|
| *np*.**meshgrid**() | genera una griglia (come in Matlab). |
| *np*.**indices**() | ritorna un array di indici in forma di griglia. |
| *obj*.**min**(), *obj*.**max**() | valere minimo, massimo dell'array (lungo un asse se specificato) |
| *obj*.**argmin**(), *obj*.**argmax**() | Indici dei valori minimi |
| *obj*.**sum**(), *obj*.**prod**() | somma, prodotto di un array |
| *obj*.**cumsum**(), *obj*.**cumprod**() | somma e prodotto cumulativi |
| *np*.**maximum**(), *np*.**minimum**() | Massimo, minimo elemento per elemento tra diversi vettori. |
| *np*.**diff**() | differenza fra elementi di un array |
| *np*.**gradient**() | gradiente numerico |
| *np*.**trapz**() | Integrazione trapezoidale |

# 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
```

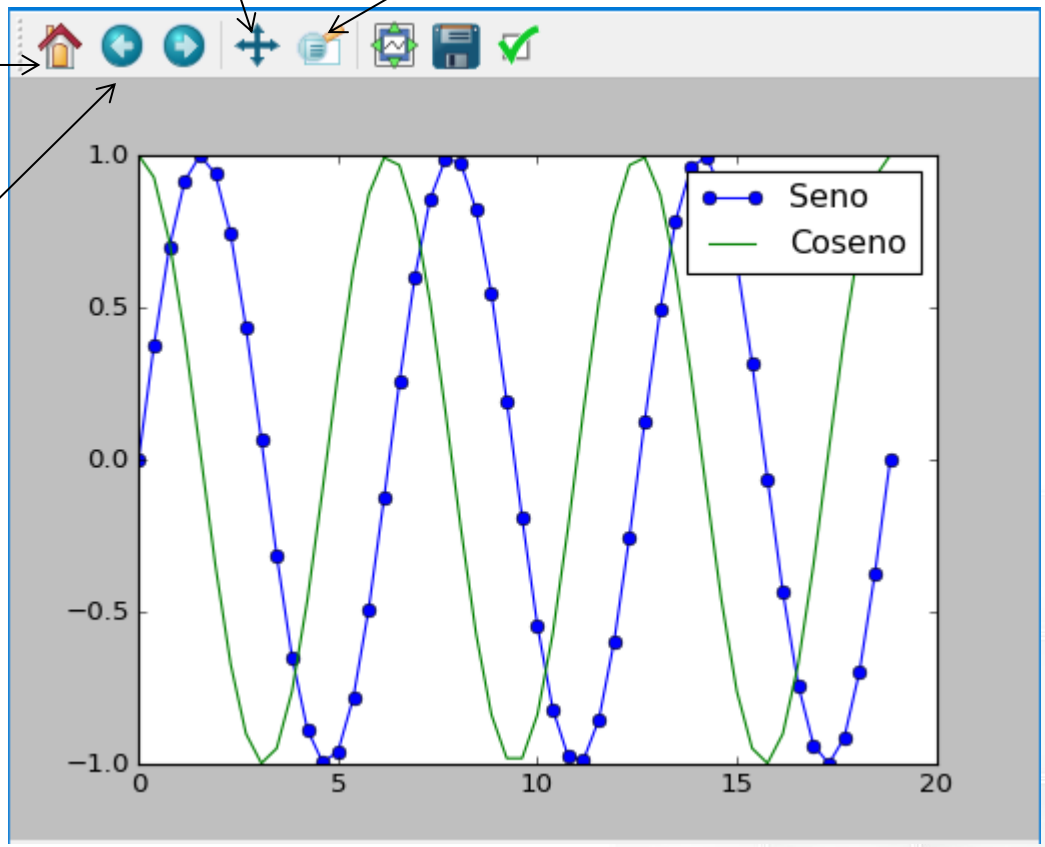
# Figure

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

zoom rettangolare

Riporta la figura alle dimensioni originarie

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



# 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
```

# 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
```

Ciclo while, map e lambda

```
>> ipl = 0
>> while ipl <5:
…     ipl += 1
…     print(ipl)

>> a = map(lambda x: x**2 + 2,[1,2,3,4] )
>> list(a)
```

# Principali strutture (for)

**esempiocicli2.py**

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

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

la funzione:

```
range(start, stop, step)
```

ritorna un iterabile che genera una lista di interi:

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

In Py2 ritorna direttamente una lista di interi.

# 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]
>> a = fibonacci(3)
>> next(a) → 2
>> next(a) → 3
>> next(a) → 5
>> next(a) → Exception: StopIteration
```

**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, end='')
```

```
>>> %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, end='')

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

Aperto di default in sola lettura e come file di testo.

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['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 |

```
>> import pandas as pd
>> a = pd.read_csv('tabella.txt',delim_whitespace=True)
     t  te   ne
0  1.0  10  100
1  2.3  35  118
2  3.1  21  250
```

# Lettura di un file Matlab

```
>> from scipy.io import loadmat

>> a = loadmat('test.mat')
>> a.keys()
 dict_keys(['t', 'co', '__globals__', '__header__',
            '__version__', 'info', 'cella', 'si'])
```

Ritorna un dizionario

In blu le variabili presenti nel file.

```
>>> print(a['t'].shape)
    (1, 100)
```

Tutte le variabili in Matlab sono almeno matrici con 2 dimensioni

Provate a fare un plot di **si** vs **t** e **co** vs **t**

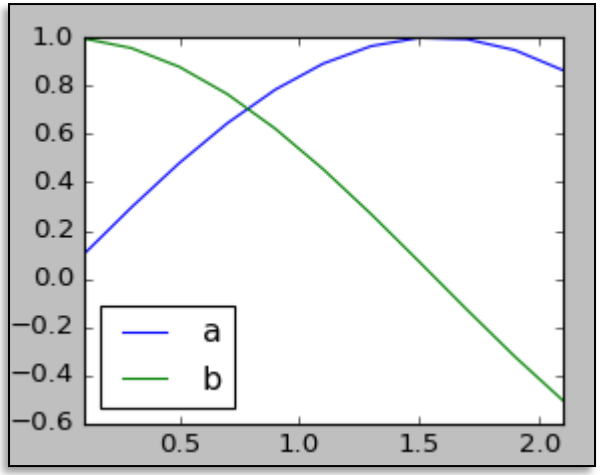
# Lettura file altri formati

Si possono leggere files di altri formati:

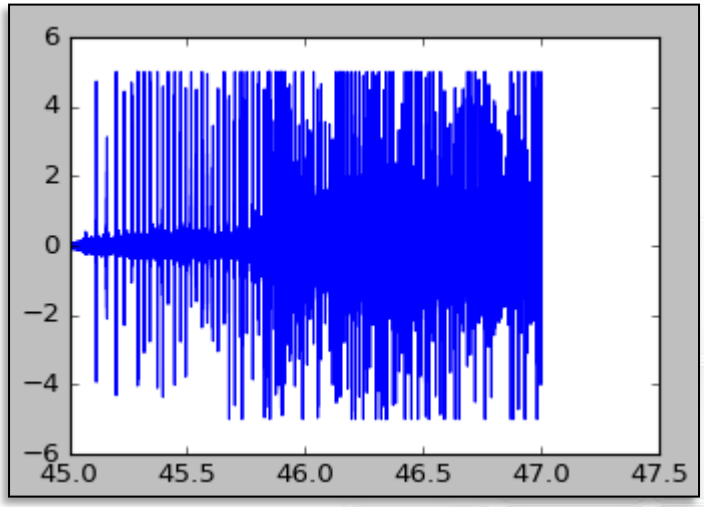
- Excel
  - http://www.python-excel.org/
  - XlsxWriter, xlrd,xlwt
- Kaleidagraph (con una routine a parte)
- HDF5
  - http://www.h5py.org/: **import h5py**
  - http://www.pytables.org/ : **import tables**
- Netcdf
  - **scipy.io.netcdf.netcdf_file**
- Mdsplus
  - Per accedere ai dati della fusione

# Excel, HDF5

```
>> import pandas as pd
>> a = pd.read_excel("OctaveTest.xlsx")
     tempo          a          b
0      0.1   0.099833   0.995004
1      0.3   0.295520   0.955336
…
>> a.set_index('tempo', inplace=True)
>> a.plot()
```




```
>> import h5py
>> f = h5py.File('sh74826coils.h5','r')
>> list(f.keys())
['channels', 'kc1data', 'phi', 'time']
>> plt.plot(f['time'],f['kc1data'][0])
>> f.close()
```



# 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):
    """ 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)
```

# 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 in Py2, non necessario in Py3).
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**.

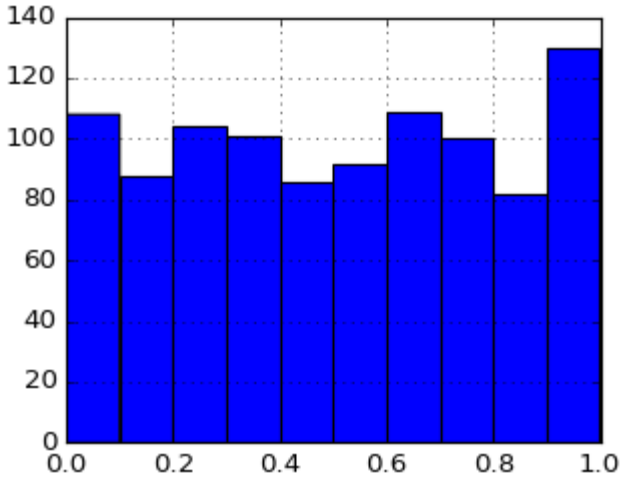
# Librerie Scientifiche

Sito web di riferimento: www.scipy.org

- numpy: package di base per array multidimensionali
- scipy: libreria fondamentale per il calcolo scientifico
- sympy: analisi simbolica (non potente come Mathematica)
- pandas: statistica di base
- matplotlib: Grafica 2D

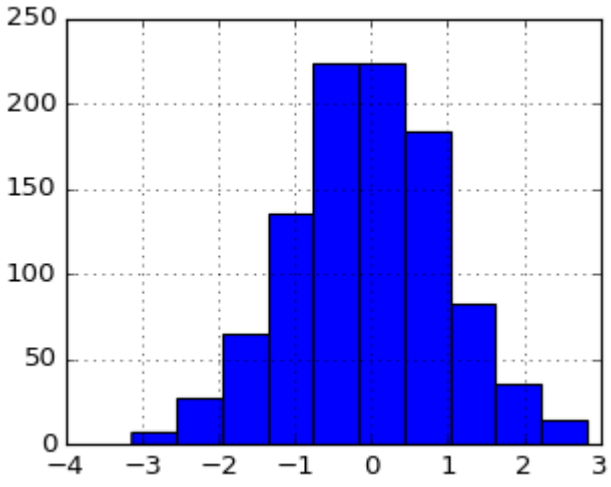
# numeri random

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




```
>> a = np.random.rand(1000)
>> plt.figure(figsize=(4.3,3.35),facecolor='w')
>> plt.hist(a)
>> plt.grid('on')
```

```
>> a = np.random.randn(1000)
>> plt.hist(a)
```



# Algebra lineare

```
>> from numpy.linalg import solve, lstsq, svd
>> a = np.array([[3,1], [1,2]])
>> b = np.array([9,8])
>> x = solve(a, b)
   array([ 2.,  3.])

>> a @ x
   array([ 9.,  8.])

>> a = np.array([[3,1],[1,2],[1,1]])
>> b = np.array([9,8,11])
>> x,res,rank,s = lstsq(a, b)
   [ 2.,  4.], 30., 2, [3.8729, 1.4142]



>> U, s, V = svd(a, full_matrices=False)
```

| | | | | | |
|---|---|---|---|---|---|
| 3 | 1 | × | 2 | = | 9 |
| 1 | 2 | | 3 | | 8 |

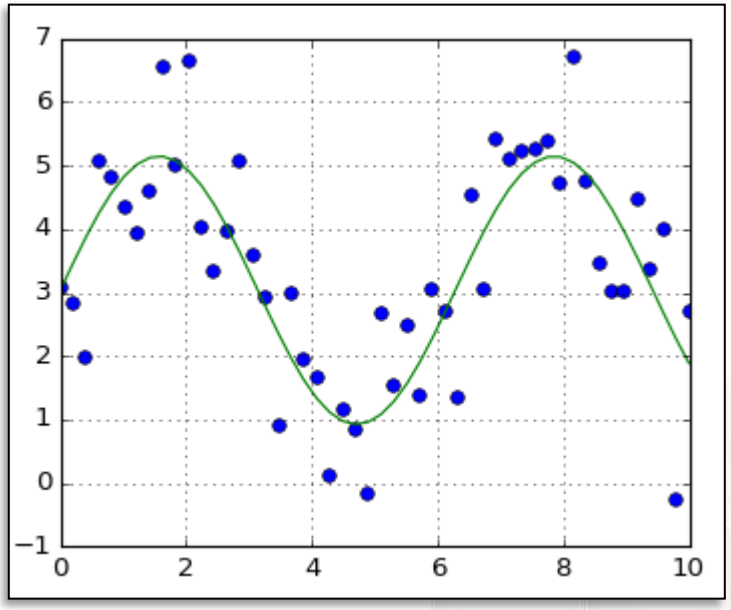
| | | | | |
|---|---|---|---|---|
| 3 | 1 | × | 2 | ~ | 9 |
| 1 | 2 | | 4 | | 8 |
| 1 | 1 | | | | 11 |

Ricordiamoci:

@ è la moltiplicazione matriciale (Py 3.5 numpy 1.10). Usare np.**dot**(a, x) se non disponibile.

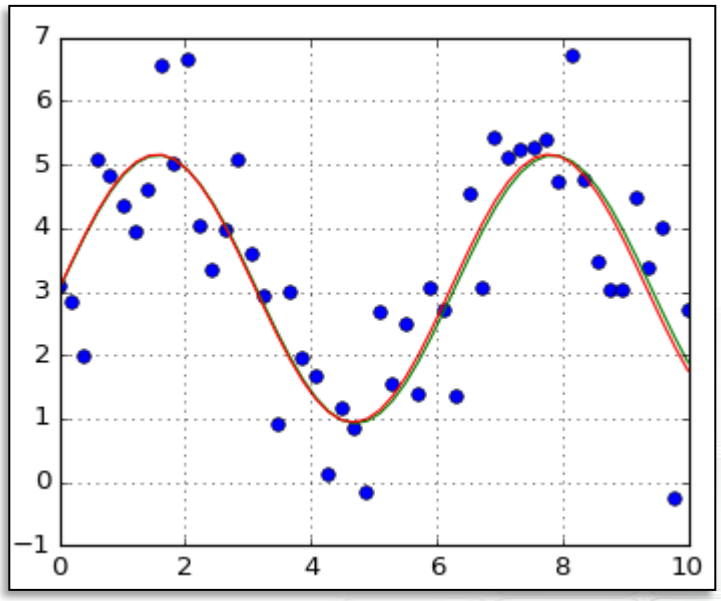
# Fit lineare

```
>> n = 50
>> t = np.linspace(0,10,n)
>> y =   3.0   +  2.0 * np.sin(t) + np.random.randn(n)

>> a = np.c_[np.ones(n),np.sin(t)]
>> x,res,rank,s = lstsq(a, y)

>> plt.plot(t, y, 'o')
>> plt.plot(t, a @ x)

>> x
  array([ 3.03484206,  2.1113669 ])
```



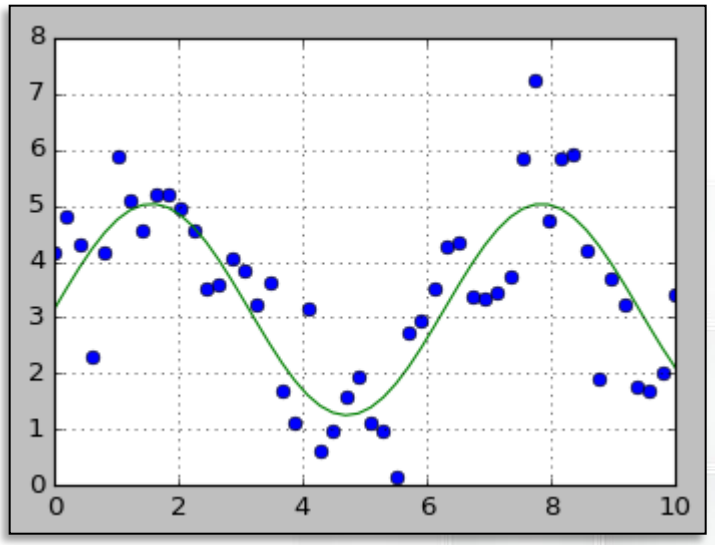
# Fit non lineare

```
>> from scipy.optimize import curve_fit

>> def funfit(t, a, b, omega):
...     return a + b*np.sin(omega*t)

>> popt, pcov = curve_fit(funfit, t, y)
   array([ 3.05010557,  2.10975173,  1.00864033])

>> plt.plot(t, funfit(t, *popt))
```



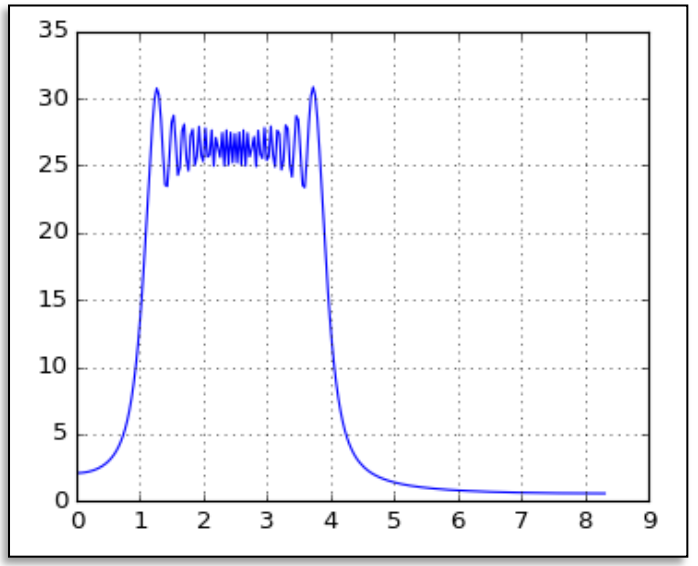
# Pandas e statsmodel

```
>> import statsmodels.formula.api as smf
>> import pandas as pd
>> n = 50
>> t = np.linspace(0,10,n)
>> y =   3.0   +  2.0 * np.sin(t) + np.random.randn(n)

>> df = pd.DataFrame({'t':t, 'y':y})

>> mod = smf.ols(formula='y ~ np.sin(t)',data=df)

>> res = mod.fit()

>> plt.plot(t,res.fittedvalues)
>> res.summary()
```



# Fourier

```
>>> n=500
>>> t = np.linspace(0,30,n)
>>> dt = t[1] - t[0]
>>> x = np.sin(2*np.pi*t*(1+t/20.))

>>> xfourier = np.fft.rfft(x)
>>> xfreq = np.fft.rfftfreq(n, dt)

>>> plt.plot(xfreq, np.abs(xfourier))
>>> xfourier
array([ 19.879 +0.000e+00j,  19.909 -5.013e-02j,
        19.995 -1.011e-01j, ...,  -0.047 +9.495e-05j,
        -0.047 +4.747e-05j,  -0.047 +0.000e+00j])

>>> xori = np.fft.irfft(xfourier)
>>> np.angle(xfourier)

>>> xfourier.real
>>> xfourier.imag
```



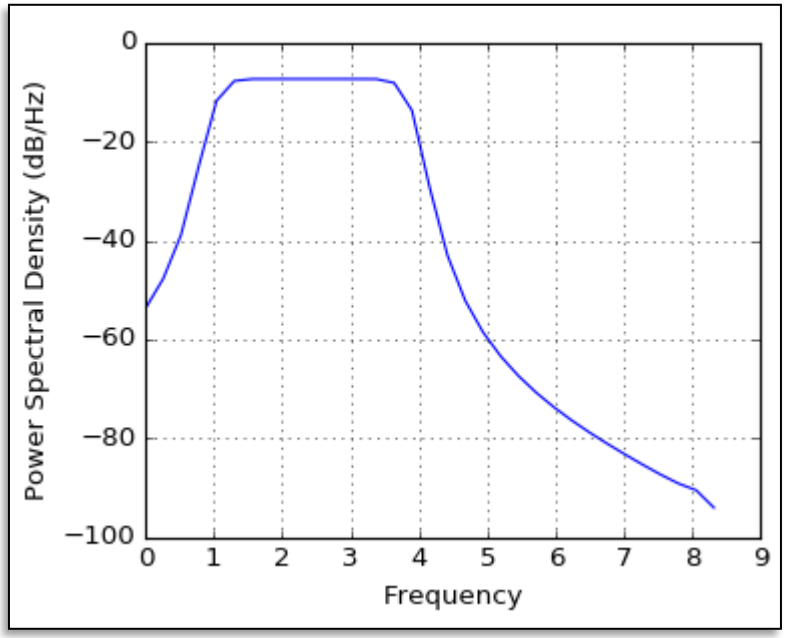

l'inversa

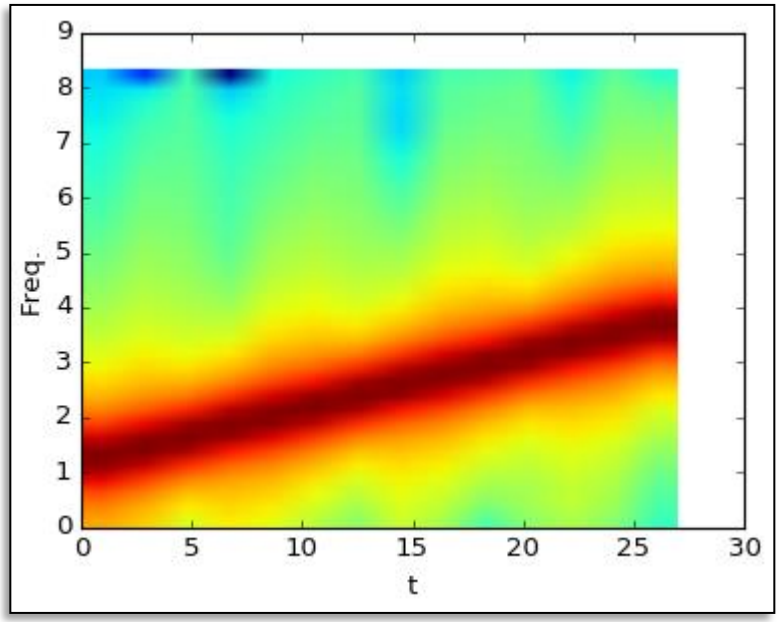
la fase

la parte reale e quella immaginaria

# Power Spectrum

```
>>> plt.psd(x, Fs=1.0/dt,NFFT=64,noverlap=32)
```

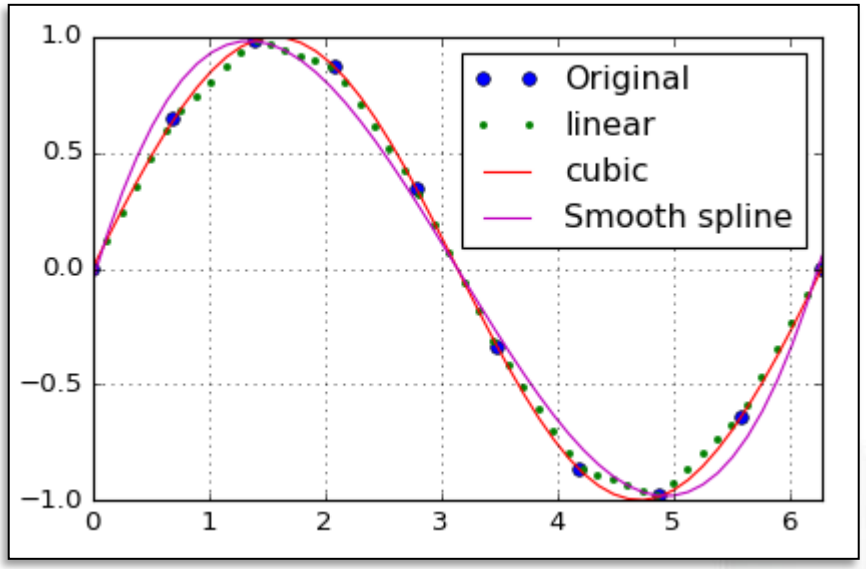



```
>>> plt.specgram(x, Fs=1.0/dt,NFFT=64,noverlap=32)
plt.xlabel('t')
plt.ylabel('Freq.')
```

# Interpolazione e spline

```
>> from scipy.interpolate import interp1d
>> from scipy.interpolate import UnivariateSpline

>> x = np.linspace(0,2*np.pi)
>> xp = np.linspace(0,2*np.pi,10)
>> yp = np.sin(xp)
```



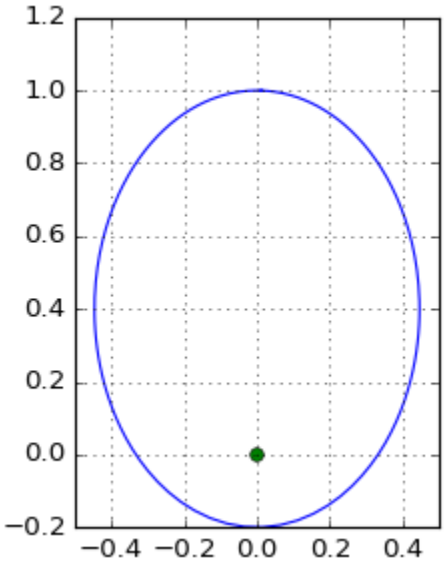

Le spline tipicamente usano le librerie DIERCKX scritte in Fortran.
Interpolazioni anche in due dimensioni.

```
>> y = np.interp(x, xp, yp)
>> f = interp1d(xp,yp,kind='cubic')
>> fu = UnivariateSpline(xp,yp,s=0.1)

>> plt.plot(xp,yp,'o', x,y,'.', x, f(x), x, fu(x))
```

# Soluzione di ODE

```
>>> from scipy.integrate import ode

>>> def forzavel(t, y, massa):
      vel = y[0:2]; pos = y[2:4]
      r = np.hypot(pos[0], pos[1])
      acc = - massa/r**3 * pos
      return np.r_[acc, vel]

>>> r = ode(forzavel).set_integrator('vode', method='adams')
>>> r.set_f_params(3.0)
>>> r.set_initial_value([1.0,0.0,0.0,1.0])
>>> t1 = 1.7; dt = 0.01

>>> sols = []
>>> while r.successful() and r.t < t1:
       sols.append((r.t, r.integrate(r.t+dt)))

>>> t,yy = zip(*sols)
>>> y=np.c_[yy]
```

Le librerie usate sono la **voda.f** , e simili scritte in Fortran, purtroppo usano dei common internamente e questo implica che solo una integrazione alla volta è possibile



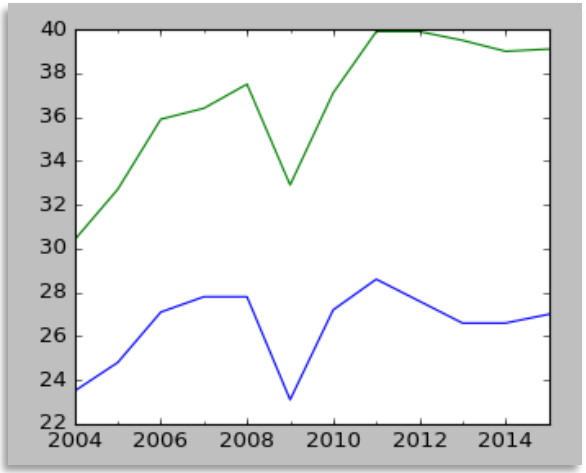
# Database EUROSTAT

http://ec.europa.eu/eurostat/web/national-accounts/data/main-tables


questi sono in nomi che ci interessano

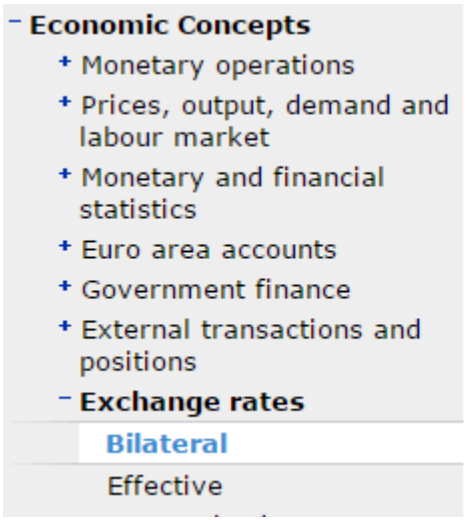
```
>> from pandasdmx import Request
>> estat = Request('ESTAT')
>> resp = estat.get(resource_type='data',  resource_id='tet00004')
>> tet4 = resp.write()
>> tet4.columns = tet4.columns.get_level_values(2)
>> tet4.IT
2004    23.5
2005    24.8
...
>> tet4.IT.plot()
>> tet4.DE.plot()
```

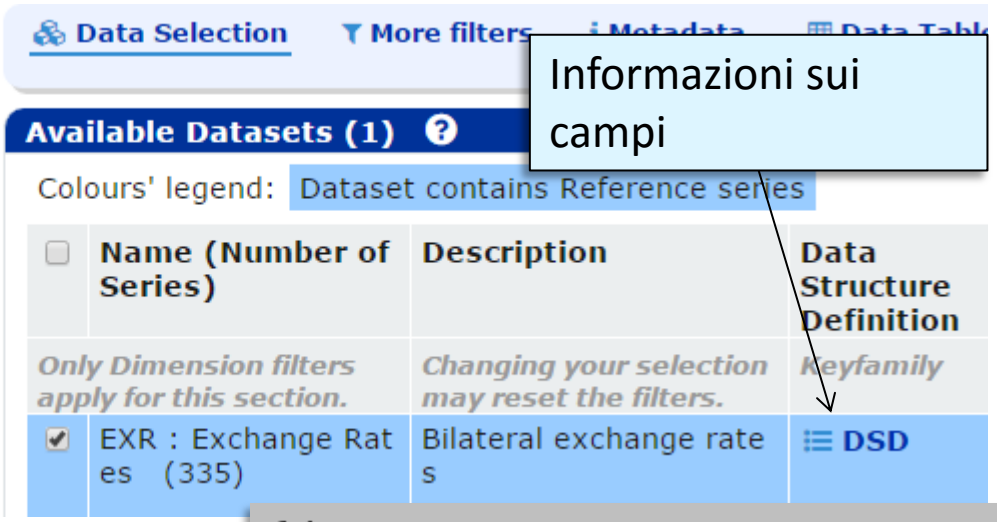
Le colonne sono un multi indice, prendiamo la parte legata ai nomi delle nazioni, e rinominiamo le colonne.



# Database ECB

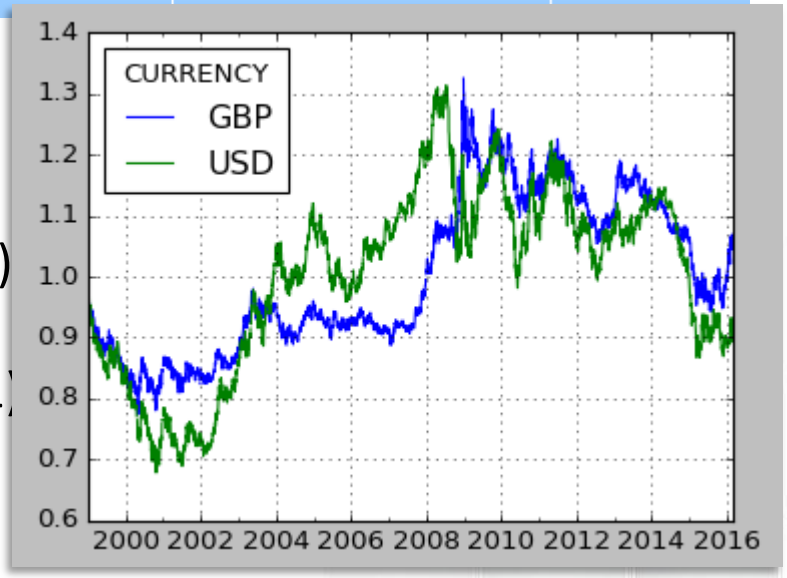
https://sdw.ecb.europa.eu/




```
>> ecb = Request('ECB')
>> resp = ecb.get(resource_type='data',
          resource_id='EXR',
          key={'FREQ':'D','CURRENCY':'USD+GBP' })
>> exr = resp.write()
>> exr.columns = exr.columns.get_level_values(1)
>> (exr / exr.mean()).plot()
>> exr.mean()
```

| | |
|---|---|
| GBP | 0.738295 |
| USD | 1.217151 |



# Esercizi per casa

- Soluzione di una ODE per un pendolo reale (forza proporzionale al sin(theta)):
  - Calcolare periodo del pendolo in funzione dell'ampiezza.
  - Mappa di Poincarè (plot della velocità vs la posizione) per diverse orbite ad energia crescente.
- Dato un segnale a + b*sin(omega*t) + noise
  - Eseguire un fit non lineare su a, b e omega.
  - Calcolare l'errore su a, b e omega in funzione dell'ampiezza del noise e confrontarlo con una serie di test numerici con ampiezza del noise crescente.
- Per gli economisti:
  - Relazione tra inflazione e disoccupazione (o qualsiasi cosa i vostri professori preferiscono) negli ultimi dieci anni in Italia e Germania.