Python: Librerie Scientifiche: NumPy

Rights & Credits

Introduzione alle librerie scientifiche

Come abbiamo visto precedentemente, codici basati su linguaggi di programmazione ad alto livello come Python sono generalmente più lenti di codici compilati, per esempio prodotti usando il linguaggio Fortran. Ragioni per la lentezza sono l'assegnazione del tipo di ogni oggetto in fase di esecuzione (in inglese detta *dynamic typing*) e l'accesso dei dati.
La soluzione a tale limite consiste, fondamentalmente, negli strumenti seguenti:

- *NumPy*, una libreria scientifica che offre modi molto efficienti di creare arrays (abbiamo visto il significato di *array* nella lezione introduttiva), tenerle in memoria ed effettuare computazione con esse;
- altre librerie scientifiche (*SciPy*, *Matplotlib* e *Pandas*) che fanno uso di oggetti prodotti da `NumPy` e consentono elaborazioni scientifiche efficienti ed avanzate, nonché rappresentazioni di alto livello dei risultati;

Passiamo in rapidissima rassegna le librerie menzionate sopra.

NumPy è una libreria per il linguaggio di programmazione Python che fornisce un notevole supporto per la creazione e gestione di arrays multidimensionali, unitamente a molte fuzioni matematiche di alta qualità per operare sulle arrays. De facto, NumPy è lo standard per la computazione.

Matplotlib è una libreria di grafica per il linguaggio di programmazione Python e la sua estensione numerica/matematica NumPy. Essa fornisce anche tutti gli strumenti per poter inglobare grafica in applicazioni varie, incluso il presente jupyter notebook.

SciPy è una libreria Python usata per il calcolo sicentifico e tecnico. Contiene moduli per l'ottimizzazione, l'algebra lineare, l'interpolazione, l'integrazione, la risoluzione di equazioni differenziali, le funzioni speciali, le trasformate di Fourier, il processamento di segnali e immagini e altri tipi di analisi matematica fondamentali in ambito scientifico. Noi useremo vari moduli di SciPy.

Pandas è una libreria Python per la manipolazione e l'analisi di dati. In particolare, essa offre strutture di dati e operazioni per la manipolazione numerica di tabelle e serie temporali.

NumPy

Impotare NumPy

Importando la libreria NumPy (o numpy), di solito si sceglie l'alias `np`. Questa non è affatto una scelga obbligata, ma è così diffusa che si ritrova in manuali, tutorials, ecc. Quindi, convenientemente, la faremo anche noi:

In [3]:
```
import numpy as np
```

NumPy fornisce due tipi principali di oggetti:

- `ndarray`,
- `ufunc` (= funzione universale ovvero, in inglese, *universal function*).

ndarray

Una ndarray ("nd" sta per "a n dimensioni ovvero *n-dimensional*) è una collezione di oggetti (elementi) dello stesso tipo, cioè *omogenei*. Gli elementi occupano tutti lo stesso numero di bytes e sono, di solito, di tipo numerico, anche se si possono definire arrays con elementi di tipo diverso. Una ndaray fatta da un solo elemento rappresenta uno scalare; una ndarray unidimensionale rappresenta un vettore; una ndarray bidimensionale rappresenta una matrice; una ndarray con più di due dimensioni rappresenta un tensore in generale. Inoltre, dal punto di vista della programmazione, tali oggetti sono definiti con attributi e metodi che consentono di gestirli in una miriade di modi, sottoponendoli a tantissime operazioni e funzioni matematiche.
Una ndarray può essere creata mediante la funzione ndarray() di `np`, cioè `np.ndarray()`, che ha la seguente sintassi:

```
np.ndarray(shape, dtype=float, ...)
```

Con tale istruzione si crea una array con certe proprietà (rinvenibili con `help`), che è un oggetto di tipo `ndarray`. Il parametro `shape` ("forma" in inglese) è una tupla o lista di interi non-negativi che specificano le dimensioni della array. Per esempio, se vogliamo creare una matrice `2 x 3`, dobbiamo fornire l'argomento `(2,3)` oppure `[2,3]`. L'altro parametro, `dtype`, specifica il tipo di formato degli elementi e il suo valore di default è `float`. La linea di comando di sopra crea una struttura di tipo ndarray come indicato, e la si può salvare assegnandola ad una variabile, ma non permette di scegliere i valori degli elementi, che sono molto spesso praticamente zero. L'accento è sulla struttura in sé, non su una specifica, desiderata array con certi elementi:

In [2]:
```
a = np.ndarray([2,3])
print(a)
```

```
[[3.33772792e-307 4.22786102e-307 2.78145267e-307]
 [4.00537061e-307 2.23419104e-317 0.00000000e+000]]
```

In [3]:
```
type(a)
```

Out[3]:
```
numpy.ndarray
```

reshape()

Gli stessi elementi possono essere riarrangiati in una array di forma diversa mediante la funzione, o *routine*, reshape() di `np`, che richiede almeno due parametri in ingresso: il nome della array di cui cambiare la forma e la nuova forma desiderata:

In [4]:
```
b = np.reshape(a, [1,6])
print(b)
```

```
[[3.33772792e-307 4.22786102e-307 2.78145267e-307 4.00537061e-307
  2.23419104e-317 0.00000000e+000]]
```

In [ ]:
```
Alternativamente, si può procedere come segue:
```

In [12]:
```
c = a.reshape([1,6])
print(c)
```

```
[[8.88517653e-312 8.88517656e-312 8.88517656e-312 8.88517656e-312
  8.88517656e-312 8.88517652e-312]]
```

o ancora più semplicemente così:

In [14]:
```
c = a.reshape(1,6)
print(c)
```

```
[[8.88517653e-312 8.88517656e-312 8.88517656e-312 8.88517656e-312
  8.88517656e-312 8.88517652e-312]]
```

np.array()

Se vogliamo creare una array con dati valori degli elementi (piuttosto che avere una struttura di array poi da riempire opportunamente), in generale usiamo la funzione np.array() di NumPy, che ha la seguente sintassi:

`np.array(object, dtype=None, ...)`

Sopra, `object` rappresenta proprio la array da creare con i valori forniti da noi, che avrà una struttura del tipo `ndarray` e quindi sarà utilizzabile da tutte le funzioni di NumPy e dalle altre librerie scientifiche che si appoggiano su NumPy. Siccome il tipo di elementi è dedotto da Python direttamente dai valori forniti in ingresso, a `dtype` viene semplicemente

assegnato il valore di default `None`.
Costruiamo una array `A` usando `np.array`:

In [6]:
```
A = np.array([[1.0,3,4],[0,2,7]])
print(A)
```

```
[[1. 3. 4.]
 [0. 2. 7.]]
```

Come abbiamo detto, la array creata è di tipo `ndarray`, come quando si usa `np.ndarray()`, e infatti

In [26]:
```
type(A), type(A) is type(a)
```

Out[26]:
```
(numpy.ndarray, True)
```

ma adesso abbiamo riempito la array con gli elementi desiderati. Inoltre, si deve ricordare che non potremmo creare un oggetto dello stesso tipo semplicemente con un'assegnazione del tipo `A = [[1.0,3,4],[0,2,7]]`, in quanto verrebbe fuori un oggetto diverso, non con le proprietà di una `ndarray`. Invece, con l'istruzione di sopra l'oggetto creato non solo conterrà gli elementi da noi desiderati, ma sarà pure dotato di tutti gli attributi e metodi di una `ndarray` che potete vedere con `help(A)`. La funzione `array` di Numpy (cioè `np.array`) rappresenta il metodo standard per creare gli oggetti `ndarray` di NumPy. Siccome gli elementi di una `ndarray` devono essere tutti dello stesso tipo, e siccome il tipo degli oggetti viene deciso da Python in fase di esecuzione, se eseguissimo per esempio `B = np.array(["a", 5, 6])`, Python dovrebbe trasformare la stringa in un numero o, viceversa, i due numeri in stringhe. La prima cosa non è possibile, per cui Python farà la seconda (in caso di tipi misti, il tipo `str` ha priorità sugli altri):

In [24]:
```
B = np.array(["a", 5, 6])
print(B)
```

```
['a' '5' '6']
```

Dal momento che `A` è una `ndarray` possiamo applicare ad essa funzioni come `reshape`:

In [8]:
```
C = np.reshape(A, [3,2])
C
```

Out[8]:
```
array([[1., 3.],
       [4., 0.],
       [2., 7.]])
```

np.zeros_like()

La funzione np.zeros_like() prende come argomento necessario una data array e restituisce un'array con la stessa forma e tutti gli elementi nulli:

In [12]:
```
D = np.zeros_like(A)
print(D)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

np.zeros()

La funzione np.zeros(), con sintassi

```
np.zeros(shape, dtype=float, ...)
```

crea direttamente una array con la forma richiesta e gli elementi tutti zero e del tipo richiesto (`float` di default).

In [14]:
```
E = np.zeros([3, 3])
E
```

Out[14]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [15]:
```
F = np.zeros([3, 3], dtype = int)
F
```

Out[15]:
```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

np.full()

La funzione np.full(), con sintassi

```
np.full(shape, fill_value, dtype=None, ...)
```

crea una array con la forma richiesta in input e tutti gli elementi uguali al valore richiesto `fill_value` (valore con cui riempire la array, cioè valore di riempimento ovvero, in inglese, *fill value*):

In [18]:
```
G = np.full((3,4),7)
print(G)
```

```
[[7 7 7 7]
 [7 7 7 7]
 [7 7 7 7]]
```

np.ones()

La funzione np.ones(), con sintassi

`np.ones(shape, dtype=Nones, ...)`

crea una array con tutti gli elementi uguali ad 1:

In [40]:
```
H = np.ones((3,4),int)
print(H)
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

np.shape()

np.shape() è una funzione che restituisce la forma (*shape*) di una data ndarray:

In [41]:
```
np.shape(H)
```

Out[41]:
```
(3, 4)
```

Vi è anche un attributo degli oggetti di tipo ndarray chiamato shape, che fornisce la forma della ndarray come una tupla:

In [42]:
```
H.shape
```

Out[42]:
```
(3, 4)
```

e gli si può assegnare un valore diverso, così cambiando la forma (*reshaping*) di una array:

In [43]:
```
H.shape = 6, 2
print(H)
```

```
[[1 1]
 [1 1]
 [1 1]
 [1 1]
 [1 1]
 [1 1]]
```

Chiaramente la riassegnazione poteva essere equivalentemente fatta scrivendo `(6,2)` oppure `[6,2]` sul lato destro.

dtype

L'attributo dtype (il nome deriva da una contrazione dell'inglese *data type*) descrive il tipo degli elementi di una ndarray:

In [47]:
```
A.dtype, B.dtype, H.dtype
```

Out[47]:
```
(dtype('float64'), dtype('<U11'), dtype('int32'))
```

o, per leggerli in modo più chiaro,

In [48]:
```
print(A.dtype, B.dtype, H.dtype)
```

```
float64 <U11 int32
```

`float64` significa un float che viene scritto usando 64 bits, quindi 8 bytes. `<U11` e un tipo di stringa `Unicode` che si usa fintantoché il numero di caratteri è minore o uguale a 11. Dopo tale lunghezza, ogni stringa ha un tipo `<Un` , dove `n` è il numero di caratteri nella stringa:

In [55]:
```
N = np.array(["abcdefghi", 5, 6])
P = np.array(["abcdefghijk", 5, 6])
Q = np.array(["abcdefghijklmno", 5, 6])
print(N.dtype, P.dtype, Q.dtype)
```

```
<U11 <U11 <U15
```

`int32` denota un intero a 32 bits con segno. Per cambiare

Si noti che i tipi `float64` , `<U11` , `int32` , ecc. corrispondono a classi `np.float64` , `np.<U11` , `np,int32` , ecc. di Numpy. Inoltre, esiste anche una funzione np.dtype() per assegnare un desiderato, specifico `dtype` agli elementi di una ndarray. Per esempio,

In [70]:
```
dt = np.dtype(np.int64)
R = np.array([[1,0],[0,1]],dtype = dt)
R
```

Out[70]:
```
array([[1, 0],
       [0, 1]], dtype=int64)
```

Visto che abbiamo usato un `dtype` non di default (il default per gli interi è `int62`), la stampa di `R` restituisce non solo l'informazione che si tratta di una array, ma anche il tipo di dati. Ovviamente, nessuna delle due informaioni appare se si usa `print` per stampare la array:

In [68]:
```
print(R)
```

```
[[1 0]
 [0 1]]
```

La seguente tabella mostra i tipi di dati disponibili in NumPy. Sono presenti tipi aggiuntivi rispetto a quelli di Python, ma sono tutti compatibili con l'uso di Python.

Data Type	np.dtype	char. code	Description
bool	np.bool	'b'	Boolean (True or False) stored as a byte
string	'S'	'S', 'a'	String datatype
int	np.int	'i4' or 'i8'	Platform integer (normally either int32 or int64)
int8	np.int8	'i1'	Byte (-128 to 127)
int16	np.int16	'i2'	Integer (-32768 to 32767)
int32	np.int32	'i4'	Integer (-2147483648 to 2147483647)
int64	np.int64	'i8'	Integer (-9223372036854775808 to 9223372036854775807)
uint32	np.uint32	'u4'	Unsigned integer (0 to 4294967295)
uint64	np.uint64	'u8'	Unsigned integer (0 to 18446744073709551615)
float	np.float	'f8'	Shorthand for float64
float32	np.float32	'f4'	Single precision float, sign bit, 8 bits exponent, 23 bits mantissa
float64	np.float64	'f8'	Double precision float, sign bit, 11 bits exponent, 52 bits mantissa
complex64	np.complex64	'c8'	Complex number represented by two 32-bit floats
complex128	np.complex128	'c16'	Complex number represented by two 64-bit floats

Dato, per esempio,

In [69]:
```
z = 1.3
type(z)
```

Out[69]:
```
float
```

si può usare il tipo del dato `z` per denire il tipo (`dtype`) di elementi di una array:

In [74]:
```
S = np.array([[1,0],[0,1]],dtype = type(z))
S
```

Out[74]:
```
array([[1., 0.],
       [0., 1.]])
```

In [78]:
```
S.dtype
```

Out[78]:
```
dtype('float64')
```

Il tipo `float` di Python corrisponde al default `float64` di NumPy, per cui, sopra, esso non è stato indicato quando abbiamo rihiesto di vedere `S` .

astype()

Il metodo astype() consente di creare una copia della ndarray da cui lo si richiama cambiandone il tipo di elementi a quello fornito come argomento del metodo:

In [93]:
```
T = S.astype(np.int64)
U = S.astype('S')
T, U
```

Out[93]:
```
(array([[1, 0],
        [0, 1]], dtype=int64),
 array([[b'1.0', b'0.0'],
        [b'0.0', b'1.0']], dtype='|S32'))
```

Altri attributi di una ndarray

In agginuta a shape e dtype, menzioniamo ndim e size, che forniscono, rispettivamente, il numero di dimensioni e il numero totale di elementi di una ndarray:

In [111…
```
print(T.ndim, T.size)
```

```
2 4
```

np.arange()

Abbiamo già incontrato la funzione intrinseca `arange` di NumPy, np.arange(), con la sintassi

`np.arange(start,stop,step)`

che consente di creare valori numerici tra `start` e `stop` con il passo `step`. Se i valori sono interi, essi sono come quelli prodotti da `range`, ma sono generati come elementi di una `ndarray`:

In [101…
```
ar = np.arange(1,3,0.5)
ar
```

Out[101…
```
array([1. , 1.5, 2. , 2.5])
```

In [102…
```
print(ar)
```

```
[1.  1.5 2.  2.5]
```

np.linspace()

Una funzione di NumPy molto spesso usata per creare arrays di numeri equispaziati su intervalli specificati è np.linspace(), che ha la sintassi

`np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)`

Come si vede sopra, invece di assegnare il passo, si può scegliere il numero di punti `num` da usare, che di default è 50. Se endpoint è True (cioè è lasciato al suo valore di default), il

valore fornito come `stop` viene incluso. Il passo, dato da `(stop - start)/num` viene riportato se si assegna `True` al parametro `retstep`.

In [109…
```
d = np.linspace(0,10,30,retstep=True)
d
```

Out[109…
```
(array([ 0.        ,  0.34482759,  0.68965517,  1.03448276,  1.37931034,
         1.72413793,  2.06896552,  2.4137931 ,  2.75862069,  3.10344828,
         3.44827586,  3.79310345,  4.13793103,  4.48275862,  4.82758621,
         5.17241379,  5.51724138,  5.86206897,  6.20689655,  6.55172414,
         6.89655172,  7.24137931,  7.5862069 ,  7.93103448,  8.27586207,
         8.62068966,  8.96551724,  9.31034483,  9.65517241, 10.        ]),
 0.3448275862068966)
```

Lettura di ndarray da file

Per leggere una ndarray da un file usiamo la *routine* (una funzione) np.loadtxt() (il nome deriva dall'inglese `load the text` (carica il testo), che ha la sintassi

`np.loadtxt(nome_file, dtype=float, ...)`

Per usarla sul file `AT_TA.xyz`, dobbiamo definire un opportuno tipo di dato strutturato (structured data type) o composito. Due esempi di come ciò possa essere fatto sono mostrati sotto.

In [201…
```
xyzdt = np.dtype("S3,3f8")
print(np.loadtxt("AT_AT.xyz",dtype = xyzdt))
```

```
[(b'C', [ 2.40000000e-02,  4.89700000e+00,  0.00000000e+00])
 (b'N', [ 8.77000000e-01,  3.90200000e+00,  0.00000000e+00])
 (b'C', [ 7.10000000e-02,  2.77100000e+00,  0.00000000e+00])
 (b'C', [ 3.69000000e-01,  1.39800000e+00,  0.00000000e+00])
 (b'N', [ 1.61100000e+00,  9.09000000e-01,  0.00000000e+00])
 (b'N', [-6.68000000e-01,  5.32000000e-01,  0.00000000e+00])
 (b'C', [-1.91200000e+00,  1.02300000e+00,  0.00000000e+00])
 (b'N', [-2.32000000e+00,  2.29000000e+00,  0.00000000e+00])
 (b'C', [-1.26700000e+00,  3.12400000e+00,  0.00000000e+00])
 (b'C', [-1.46200000e+00, -3.13500000e+00,  0.00000000e+00])
 (b'O', [-2.56200000e+00, -2.60800000e+00,  0.00000000e+00])
 (b'N', [-2.98000000e-01, -2.40700000e+00,  0.00000000e+00])
 (b'C', [ 9.94000000e-01, -2.89700000e+00,  0.00000000e+00])
 (b'O', [ 1.94400000e+00, -2.11900000e+00,  0.00000000e+00])
 (b'C', [ 1.10600000e+00, -4.33800000e+00,  0.00000000e+00])
 (b'C', [ 2.46600000e+00, -4.96100000e+00,  0.00000000e+00])
 (b'C', [-2.40000000e-02, -5.05700000e+00,  0.00000000e+00])
 (b'H', [ 2.39075432e+00,  1.54352175e+00, -1.28500000e-05])
 (b'H', [-2.69149972e+00,  2.64677530e-01,  6.90000000e-07])
 (b'H', [-2.11571099e+00,  5.07405583e+00, -1.53000000e-06])
 (b'H', [ 2.98947390e-01,  5.94300395e+00, -8.40000000e-07])
 (b'H', [-2.11973726e+00, -5.06096733e+00,  9.07950000e-04])
 (b'H', [ 2.65044292e+00, -5.54070171e+00, -9.10560030e-01])
 (b'H', [ 2.61075523e+00, -5.62663325e+00,  8.56981870e-01])
 (b'H', [ 3.21930700e+00, -4.17294246e+00,  5.26498900e-02])
 (b'H', [-6.21196000e-03, -6.14218557e+00,  9.90760000e-04])
 (b'N', [-1.29100000e+00,  4.49800000e+00,  0.00000000e+00])
 (b'N', [-1.28400000e+00, -4.50000000e+00,  0.00000000e+00])
 (b'H', [ 1.76827351e+00, -1.08568960e-01,  3.22147900e-02])
 (b'H', [-4.23936200e-01, -1.38870836e+00,  1.53915700e-02])
 (b'H', [ 1.48456902e+00,  9.44344440e-01,  3.36349928e+00])
 (b'H', [ 4.68976250e-01, -1.37197318e+00,  3.36734069e+00])
 (b'N', [-3.68800000e+00,  2.88000000e+00,  3.38000000e+00])
 (b'N', [ 1.60600000e+00, -4.39500000e+00,  3.38000000e+00])
 (b'C', [-2.85900000e+00,  3.97600000e+00,  3.38000000e+00])
 (b'N', [-1.58400000e+00,  3.67200000e+00,  3.38000000e+00])
 (b'C', [-1.57100000e+00,  2.28400000e+00,  3.38000000e+00])
 (b'C', [-5.23000000e-01,  1.34800000e+00,  3.38000000e+00])
 (b'N', [ 7.69000000e-01,  1.68200000e+00,  3.38000000e+00])
 (b'N', [-8.53000000e-01,  3.80000000e-02,  3.38000000e+00])
 (b'C', [-2.14800000e+00, -2.96000000e-01,  3.38000000e+00])
 (b'N', [-3.22300000e+00,  4.89000000e-01,  3.38000000e+00])
 (b'C', [-2.86100000e+00,  1.78300000e+00,  3.38000000e+00])
 (b'C', [ 6.60000000e-01, -3.39600000e+00,  3.38000000e+00])
 (b'O', [-5.40000000e-01, -3.61600000e+00,  3.38000000e+00])
 (b'N', [ 1.17400000e+00, -2.12200000e+00,  3.38000000e+00])
 (b'C', [ 2.50700000e+00, -1.75900000e+00,  3.38000000e+00])
 (b'O', [ 2.81800000e+00, -5.72000000e-01,  3.38000000e+00])
 (b'C', [ 3.44500000e+00, -2.85900000e+00,  3.38000000e+00])
 (b'C', [ 4.91100000e+00, -2.56400000e+00,  3.38000000e+00])
 (b'C', [ 2.95300000e+00, -4.10500000e+00,  3.38000000e+00])
 (b'H', [-4.69881725e+00,  2.86668739e+00,  3.38000115e+00])
 (b'H', [-3.24632742e+00,  4.98695124e+00,  3.38000084e+00])
 (b'H', [ 1.04198710e+00,  2.65971991e+00,  3.38002708e+00])
 (b'H', [-2.32834527e+00, -1.36898148e+00,  3.37999169e+00])
 (b'H', [ 1.26115232e+00, -5.34193929e+00,  3.37764522e+00])
```

```
(b'H', [ 5.07058350e+00, -1.49691412e+00,  3.21723827e+00])
(b'H', [ 5.37806105e+00, -2.82614822e+00,  4.33504715e+00])
(b'H', [ 5.43365622e+00, -3.11037666e+00,  2.58951659e+00])
(b'H', [ 3.60546736e+00, -4.97177252e+00,  3.37713880e+00])]
```

In [202…

```
xyzdt = np.dtype([('A','S3'),('x',np.float64),('y',np.float64),('z',np.float64)])
print(np.loadtxt("AT_AT.xyz",dtype = xyzdt))
```

```
[(b'C',  0.024     ,  4.897     ,  0.00000000e+00)
 (b'N',  0.877     ,  3.902     ,  0.00000000e+00)
 (b'C',  0.071     ,  2.771     ,  0.00000000e+00)
 (b'C',  0.369     ,  1.398     ,  0.00000000e+00)
 (b'N',  1.611     ,  0.909     ,  0.00000000e+00)
 (b'N', -0.668     ,  0.532     ,  0.00000000e+00)
 (b'C', -1.912     ,  1.023     ,  0.00000000e+00)
 (b'N', -2.32      ,  2.29      ,  0.00000000e+00)
 (b'C', -1.267     ,  3.124     ,  0.00000000e+00)
 (b'C', -1.462     , -3.135     ,  0.00000000e+00)
 (b'O', -2.562     , -2.608     ,  0.00000000e+00)
 (b'N', -0.298     , -2.407     ,  0.00000000e+00)
 (b'C',  0.994     , -2.897     ,  0.00000000e+00)
 (b'O',  1.944     , -2.119     ,  0.00000000e+00)
 (b'C',  1.106     , -4.338     ,  0.00000000e+00)
 (b'C',  2.466     , -4.961     ,  0.00000000e+00)
 (b'C', -0.024     , -5.057     ,  0.00000000e+00)
 (b'H',  2.39075432,  1.54352175, -1.28500000e-05)
 (b'H', -2.69149972,  0.26467753,  6.90000000e-07)
 (b'H', -2.11571099,  5.07405583, -1.53000000e-06)
 (b'H',  0.29894739,  5.94300395, -8.40000000e-07)
 (b'H', -2.11973726, -5.06096733,  9.07950000e-04)
 (b'H',  2.65044292, -5.54070171, -9.10560030e-01)
 (b'H',  2.61075523, -5.62663325,  8.56981870e-01)
 (b'H',  3.219307  , -4.17294246,  5.26498900e-02)
 (b'H', -0.00621196, -6.14218557,  9.90760000e-04)
 (b'N', -1.291     ,  4.498     ,  0.00000000e+00)
 (b'N', -1.284     , -4.5       ,  0.00000000e+00)
 (b'H',  1.76827351, -0.10856896,  3.22147900e-02)
 (b'H', -0.4239362 , -1.38870836,  1.53915700e-02)
 (b'H',  1.48456902,  0.94434444,  3.36349928e+00)
 (b'H',  0.46897625, -1.37197318,  3.36734069e+00)
 (b'N', -3.688     ,  2.88      ,  3.38000000e+00)
 (b'N',  1.606     , -4.395     ,  3.38000000e+00)
 (b'C', -2.859     ,  3.976     ,  3.38000000e+00)
 (b'N', -1.584     ,  3.672     ,  3.38000000e+00)
 (b'C', -1.571     ,  2.284     ,  3.38000000e+00)
 (b'C', -0.523     ,  1.348     ,  3.38000000e+00)
 (b'N',  0.769     ,  1.682     ,  3.38000000e+00)
 (b'N', -0.853     ,  0.038     ,  3.38000000e+00)
 (b'C', -2.148     , -0.296     ,  3.38000000e+00)
 (b'N', -3.223     ,  0.489     ,  3.38000000e+00)
 (b'C', -2.861     ,  1.783     ,  3.38000000e+00)
 (b'C',  0.66      , -3.396     ,  3.38000000e+00)
 (b'O', -0.54      , -3.616     ,  3.38000000e+00)
 (b'N',  1.174     , -2.122     ,  3.38000000e+00)
 (b'C',  2.507     , -1.759     ,  3.38000000e+00)
 (b'O',  2.818     , -0.572     ,  3.38000000e+00)
 (b'C',  3.445     , -2.859     ,  3.38000000e+00)
 (b'C',  4.911     , -2.564     ,  3.38000000e+00)
 (b'C',  2.953     , -4.105     ,  3.38000000e+00)
 (b'H', -4.69881725,  2.86668739,  3.38000115e+00)
 (b'H', -3.24632742,  4.98695124,  3.38000084e+00)
 (b'H',  1.0419871 ,  2.65971991,  3.38002708e+00)
 (b'H', -2.32834527, -1.36898148,  3.37999169e+00)
 (b'H',  1.26115232, -5.34193929,  3.37764522e+00)
```

```
(b'H',  5.0705835 , -1.49691412,  3.21723827e+00)
(b'H',  5.37806105, -2.82614822,  4.33504715e+00)
(b'H',  5.43365622, -3.11037666,  2.58951659e+00)
(b'H',  3.60546736, -4.97177252,  3.37713880e+00)]
```

Il caratter `b` che compare all'inizio di ogni riga della array deriva dal fatto che `np.loadtxt()` lavora su bytes (*byte mode*) mentre Python usa `Unicode` e contrassegna le stringhe di bytes on `b`.

Alcuni metodi che operano sulle ndarrays

Ecco una lista di alcuni metodi di ndarray (sotto usiamo `arr` per denotare la ndarray):

metodo	cosa produce
arr.copy()	copia della array
arr.reshape()	array con una nuova forma
arr.teanspose o arr.T	array trasposta
arr.H	array trasposta coniugata (come T se reale)
arr.sum()	somma degli elementi della array
arr.prod()	prodotto degli elementi della array
arr.min/max()	valore minimo/massimo
arr.argmin/argmax()	valore dell'indice del minimo/massimo
arr.var/mean/std	varianza, media, deviazione standard

Esempi

In [197…
```
v = np.array([[5.0, 5],[1, 4]])
print(v)
```

```
[[5. 3.]
 [1. 4.]]
```

In [198…
```
w = v.copy()
print(w)
```

```
[[5. 3.]
 [1. 4.]]
```

In [199…
```
print(w is v, w == v,sep='\n')
```

```
False
[[ True  True]
 [ True  True]]
```

In [205…
```
v.min(), v.argmin(), v.max(), v.argmax()
```

Out[205… (1.0, 2, 5.0, 0)

In [208…
```
v.T
```

Out[208…
```
array([[5., 1.],
       [3., 4.]])
```

In [214…
```
v.sum(), v.prod()
```

Out[214… (13.0, 60.0)

Funzioni universali e routines di NumPy

Una funzione universale (detta ufunc usando una nomenclatura abbreviata derivante dall'inglese *universal function*) opera su ndarrays elemento per elemento. In questa sezione esaminiamo alcune delle molte *ufuncs* e altre funzioni (*routines*) disponibili in NumPy. Consideriamo le due arrays seguenti (che, in termini matematici, descrivono due matrici 2 x 2):

In [64]:
```
M = np.array([[1,3],[2,4]])
N = np.array([[4,3],[0,3]])
print(M, N, sep='\n')
```

```
[[1 3]
 [2 4]]
[[4 3]
 [0 3]]
```

La loro differenza è

In [65]:
```
M - N
```

Out[65]:
```
array([[-3,  0],
       [ 2,  1]])
```

Il segno meno è una notazione abbreviata a cui siamo abituati che, in realtà, richiama l'uso della *ufunc* np.subtract() (la documentazione di `help` recita: "The `-` operator can be used as a shorthand for `np.subtract` on ndarray"):.

In [66]:
```
np.subtract(M,N)
```

Out[66]:
```
array([[-3,  0],
       [ 2,  1]])
```

Analogamente, c'è una *ufunc* np.add() che somma due matrici elemento per elemento.
La *ufunc* np.sqrt() fa la radice quadrata degli elementi di una array se essi sono tutti non-negativi:

In [67]:
```
np.sqrt(M)
```

Out[67]:
```
array([[1.        , 1.73205081],
       [1.41421356, 2.        ]])
```

Abbiamo già parlato della *ufunc* np.power(). Mostriamo qualche esempio:

In [83]:
```
print(np.power(M,N), np.power(M,2), np.power([1, 2, 3], 3), sep='\n\n')
```

```
[[ 1 27]
 [ 1 64]]

[[ 1  9]
 [ 4 16]]

[ 1  8 27]
```

La *ufunc* np.multiply() effettua la *moltiplicazione di Hadamard* delle due arrays, cioè la moltiplicazione elemento per elemento:

In [68]:
```
np.multiply(M,N)
```

Out[68]:
```
array([[ 4,  9],
       [ 0, 12]])
```

La notazione breve (in inglese, *shorthand notation*) per tale operazione è `*`:

In [69]:
```
M * N
```

Out[69]:
```
array([[ 4,  9],
       [ 0, 12]])
```

La *ufunc* np.matmul() effettua la moltiplicazione standard, "riga per colonna", delle due arrys:

In [70]:
```
np.matmul(M,N)
```

Out[70]:
```
array([[ 4, 12],
       [ 8, 18]])
```

La notazione breve per tale operazione è `@`:

In [71]:
```
M @ N
```

Out[71]:
```
array([[ 4, 12],
       [ 8, 18]])
```

NumPy contiene anche un set di speciali funzioni chiamate routines. La *routine* np.dot() effettua il prodotto scalare di due arrays `A` e `B`.

- Se `A` e `B` sono due arrays `1D`, quindi rappresentanti due vettori, `dot` ne fa il prodotto scalre propriamente detto o prodotto interno. Se i loro elementi sono complessi, vengono moltiplicati per come sono, senza fare prima i complessi coniugati degli elementi di `A` (o `B`). Se si vuole fare quest'ultima cosa, si usa invece la *routine* np.vdot().

- Se `A` e `B` sono due arrays di forma `2D`, `dot` è una moltiplicazione matriciale che usa `matmul`.
- Se `A` o `B` è una array di forma `0D`, scioè uno scalare, `dot` funziona come `multiply`
- ...

Esempi

In [72]:
```
np.dot(M,N)
```

Out[72]:
```
array([[ 4, 12],
       [ 8, 18]])
```

In [73]:
```
k = 1.5
print(np.dot(k,M))
```

```
[[1.5 4.5]
 [3.  6. ]]
```

In [74]:
```
V = np.array([1,-1,8]); W = np.array([[2],[0],[-0.25]])
print(V, W, sep='\n\n')
```

```
[ 1 -1  8]

[[ 2.  ]
 [ 0.  ]
 [-0.25]]
```

In [75]:
```
print(np.dot(V,W))
```

```
[0.]
```

In [76]:
```
cv = np.array([1-1j,2+3j]); cw = np.array([[1j],[1]])
print(np.dot(cv,cw), np.vdot(cv,cw), sep='\n')
```

```
[3.+4.j]
(1-2j)
```

Generazione di numeri random con NumPy

Il modulo np.random implementa vari generatori di numeri pseudo-random che possono anche essere sfruttati per generare valori per una varietà di distribuzioni di probabilità. Generatori di numeri random (RNG = Random Number Generator) e generatori di numeri pseudo-random (PRNG = PseudoRandom Number Generator) sono spesso usati in algoritmi di Machine Learning (= ML) e intelligenza artificiale (in inglese, Artificial Intelligence = AI).

Come unico esempio, consideriamo la funzione np.random.rand() del modulo di cui sopra. Essa è una *funzione di convenienza* per la generazione di numeri random che è utile quando si importa codice da MATLAB. La sua sintassi è

```
np.random.rand(d0, d1, ..., dn)
```

dove `d0`, `d1`, ..., `dn` sono le dimensioni della array di numeri random generata. Per esempio:

In [84]:
```
np.random.rand()
```

Out[84]:
```
0.5313847001386298
```

In [85]:
```
np.random.rand(3)
```

Out[85]:
```
array([0.779705  , 0.55488726, 0.47112732])
```

In [86]:
```
np.random.rand(3,2)
```

Out[86]:
```
array([[0.57546877, 0.02824374],
       [0.48149286, 0.51115758],
       [0.1866403 , 0.05583081]])
```

Si noti che `np.random.rand(3)` e lo stesso di `np.random.rand(1,3)`:

In [88]:
```
np.random.rand(1,3)
```

Out[88]:
```
array([[0.91450701, 0.64596476, 0.74940154]])
```

Funzioni *ad hoc* con NumPy

Si possono anche definire facilmente funzioni *ad hoc* che lavorano sulle ndarrays. A tal fine, usiamo la *routine* fromfunction(), che ha la seguente sintassi:

```
np.fromfunction(funzione, shape, *, dtype=<class 'float'>, ...)
```

dove la `funzione` è definita *ad hoc* per le nostre esigenze. Supponiamo, per esempio, di essere interessati a costruire una ndarray bidimensionale, che quindi rappresenta una matrice. Tale *routine* crea una ndarray con la forma `shape` e a ciascuna posizione `i,j` scrive il valore di `function(i,j)`. Facciamo un esempio:

In [26]:
```
def fhyp(i, j):
    return j**2 - i**2
np.fromfunction(fhyp, (4,4))
```

Out[26]:
```
array([[ 0.,  1.,  4.,  9.],
       [-1.,  0.,  3.,  8.],
       [-4., -3.,  0.,  5.],
       [-9., -8., -5.,  0.]])
```

Esercizio

Costruire una funzione che legga due arrays bidimensionali con elementi di tipo `float` da file diversi, le sommi e stampi la array risultante.