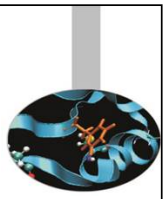
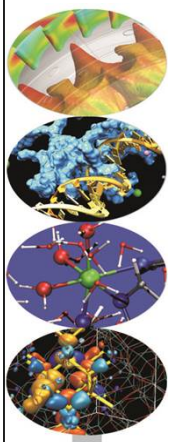
CINECA SCAI
SuperComputing Applications and Innovation



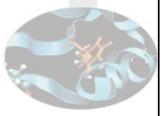
Introduzione al calcolo scientifico in Python



Mario Rosati
CINECA – Roma
m.rosati@cineca.it

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

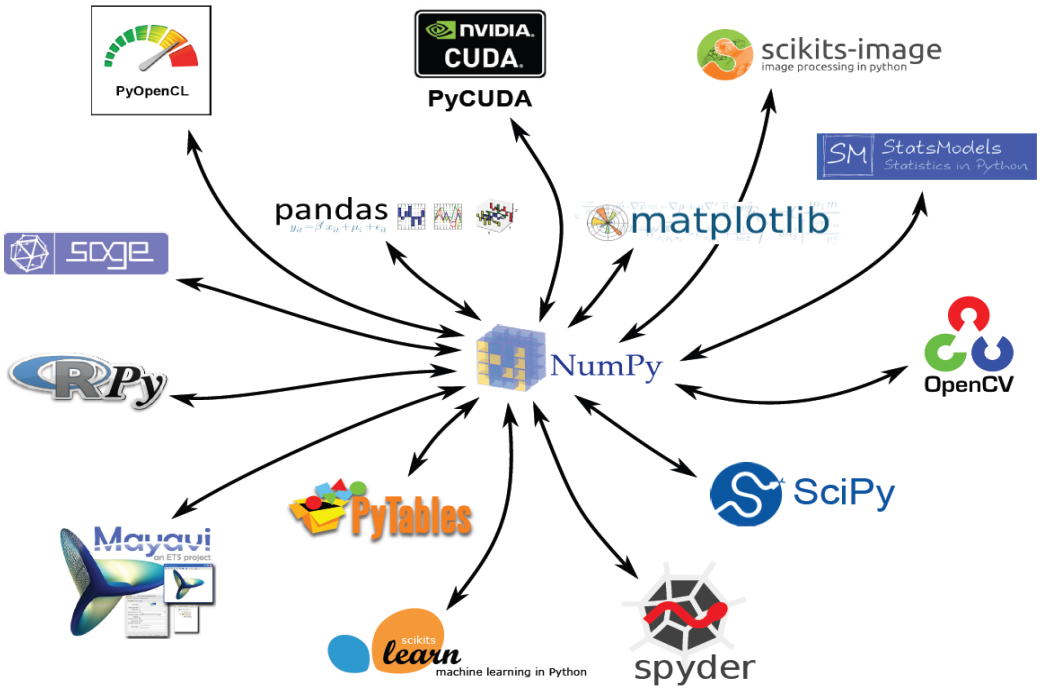


Sommario

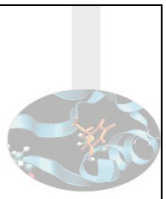
Scientific Computing in Python

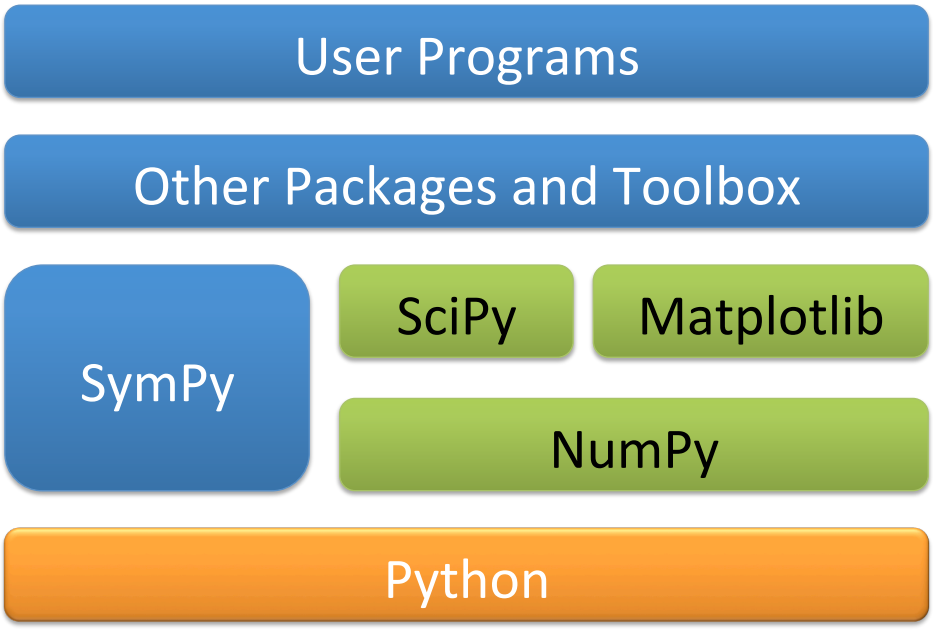
- A dispetto della sua semplicità Python è un linguaggio abbastanza potente da permettere la gestione di applicazioni complesse
- Python ha una posizione forte nell'ambito del Computing Scientifico:
  - E' open-source!
  - Ci sono significative e diffuse comunità di utenti, quindi è piuttosto facile reperire in rete aiuto e documentazione
  - Ad oggi è disponibile un esteso ecosistema di *environment*, di *package* e di librerie scientifiche per Python ed è in rapida crescita
  - Si possono scrivere codici Python che ottengono ottime performance, grazie alla forte integrazione con prodotti altamente ottimizzati scritti in C e Fortran (BLAS, Atlas, Lapack, Intel MKL®, …)
  - E' possibile sviluppare applicazioni parallele che usino la comunicazione interprocesso (MPI), il multi-threading (OpenMP) ed il GPU computing (OpenCL e CUDA)

Scientific Python Ecosystem

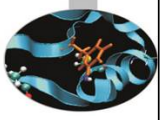


Scientific Python software stack

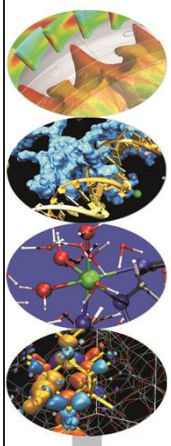




CINECA SCAI
SuperComputing Applications and Innovation



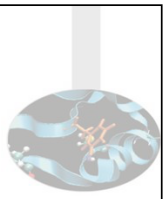
Introduzione a ***NumPy***



Mario Rosati
CINECA – Roma
m.rosati@cineca.it

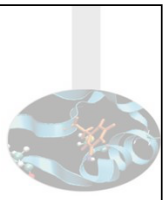
CINECA

Cos'è NumPy



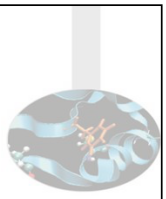
- *NumPy* (Numerical Python) è un modulo che estende Python con strutture dati e metodi utili per il calcolo tecnico scientifico.
- In Python puro abbiamo a disposizione:
  - oggetti numerici di alto livello: interi, *floating point*
  - container: liste (*insert* ed *append* a costi bassi) e dizionari (operazioni di *lookup* veloci)
- NumPy
  - Introduce un modalità naturale ed efficiente per utilizzare array multi-dimensionali
  - Aggiunge una serie di utili funzioni matematiche di base (algebra lineare, FFT, *random number*, …)

Perché usare NumPy



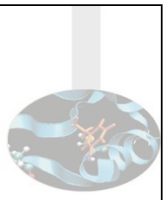
- In NumPy funzioni e metodi agiscono su interi vettori e matrici, quindi raramente si ricorre a loop espliciti, tipicamente poco efficienti
- Gli algoritimi utilizzati sono estremamente testati e disegnati con l'obiettivo di fornire alte prestazioni
- NumPy gestisce lo store degli array in memoria in modo molto più efficiente, rispetto a quanto accade in Python puro per le liste e le liste di liste
- Le operazioni di I/O di array sono significativamente più veloci
- File di grandi dimensioni possono essere *memory-mapped*, consentendo così una lettura/scrittura ottimale di grosse moli di dati
- Molte parti di NumPy sono scritte in C, cosa che rende un codice NumPy più veloce dell'analogo in Python puro
- NumPy prevede una C API, che consente di estenderne le funzionalità (questo tema non viene trattato in questa introduzione a NumPy)

Quando non usare NumPy



- Al di fuori del contesto del calcolo tecnico scientifico, l'uso di NumPy è meno utile
- Principali limitazioni:
  - NumPy non è supportato per lo sviluppo di applicazioni Google App Engine, perché molte sue parti sono scritte in C
  - Gli utenti di Jython – l'implementazione Java di Python – non posso contare sul modulo NumPy: Jython gira all'interno di una Java Virtual Machine e non può importare NumPy, perché molte sue parti sono scritte in C

Ma quanto conviene usare *NumPy*?



```
In [1]: L = range(1000)

In [2]: %timeit [i**2 for i in L]
1000 loops, best of 3: 123 us per loop

In [3]: a = np.arange(1000)

In [4]: %timeit a**2
100000 loops, best of 3: 4.6 us per loop
```

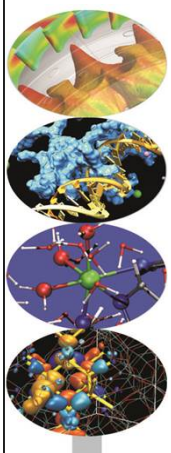
30x

NB: il metodo `arange` di NumPy è l'analogo di `range` del puro Python, ma ritorna un NumPy array

NumPy: documentazione di riferimento

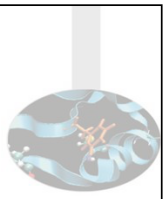
- Sito web http://docs.scipy.org
- Help interattivo in iPython

```
In [4]: import numpy as np

In [5]: np.array?
String Form:<built-in function array>
Docstring:
array(object, dtype=None, copy=True,
order=None, subok=False, ndmin=0, …

In [6]: np.con*?
np.concatenate
np.conj
np.conjugate
np.convolve
```

Introduzione a ***NumPy***: l'oggetto ndarray

Importare il modulo NumPy



- Oggetti e metodi di NumPy nel *namespace* di base

```
>>> from numpy import *
```

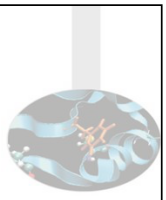
- Oggetti e metodi di NumPy in un namespace ad-hoc

```
>>> import numpy
```

o, meglio,

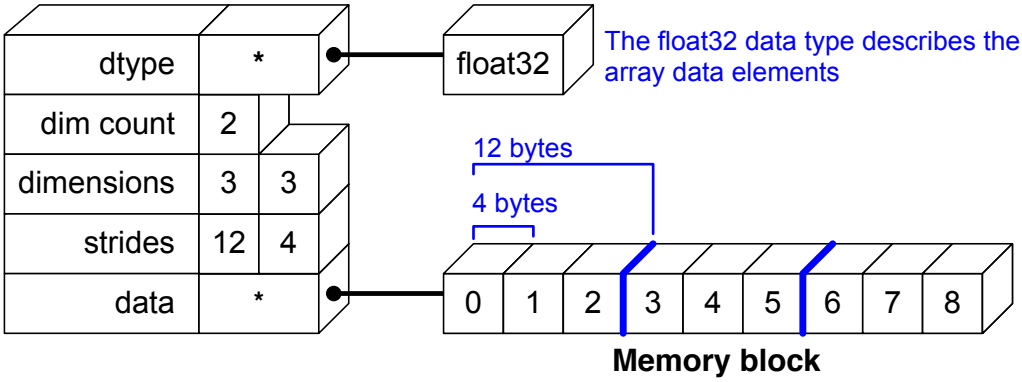
```
>>> import numpy as np   # default in molti codici e
                         nella documentazione numpy
```


NumPy array (*ndarray*)

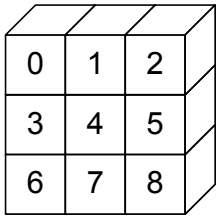


- NumPy fornisce il nuovo oggetto *ndarray*
- *ndarray* è una struttura dati omogenea (*fixed-type*) e multi-dimensionale.
- **Teminologia di base:**
  - *size* di un array: *numero di elementi presenti nell'array*
  - *rank* di un array: *numero di assi/dimensioni dell'array*
  - *shape* di un array: *dimensioni dell'array*, ovvero una tupla di interi contenente il numero di elementi per ogni dimensione

NDArray Data Structure



Python View:



Creare array

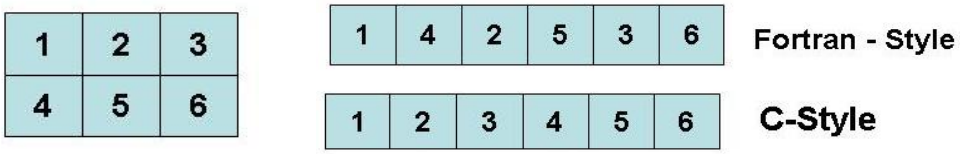
- In NumPy ci sono diverse modalità per generare un array; la più semplice consiste nell'utilizzo della funzione

  ```
  array(object, dtype=None, copy=1, order=None)
  ```

  che è utile per convertire altre strutture dati Python (liste o tuple) e ritorna un oggetto `ndarray`

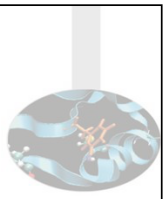
- Il primo argomento è l'oggetto da convertire, mentre gli altri (non obbligatori) sono utili rispettivamente per specificare (i) il tipo di dato, (ii) se l'operazione implica una copia in memoria dei dati dell'oggetto convertito e (iii) l'ordine in memoria (`'C'` o `'F'`) in caso di array multi-dimensionale

NB: allocazione in memoria per array multidimensionali



Creare 1D array

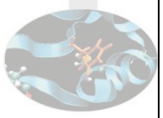
(e referenziarne alcuni attributi)



```
>>> import numpy as np

>>> a = np.array([0, 1, 2, 3])

>>> a
array([0, 1, 2, 3])

>>> a.ndim   # ritorna il numero di assi di a
1

>>> a.shape  # ritorna la tupla delle dimensioni di a
(4,)

>>> a.dtype  # ritorna il tipo degli elementi di a
dtype('int64')
```

NB: se il primo argomento è una lista di interi, su una macchina a 64 bit, la funzione `array` ritorna un array di interi a 64 bit; per ottenere un array di interi a 32 bit, deve essere utilizzato l'ulteriore argomento `dtype=int32`

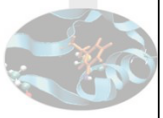
Creare 2-D, 3-D, n-D array …



```
>>> b = np.array([[0,1,2],
                  [3,4,5]])    # 2x3 array

>>> b
array([[0, 1, 2],
       [3, 4, 5]])

>>> b.ndim
2

>>> b.shape
(2,3)

>>> len(b)     # size del 1° asse
2
```

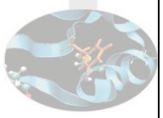
NB: argomento del metodo *array*, è una lista di liste

… Creare 2-D, 3-D, n-D array



```
>>> c = np.array([[[1], [2]], [[3], [4]]])

>>> c
array([[[1],
        [2]],

       [[3],
        [4]]])

>>> c.ndim
3


>>> b.shape
(2, 2, 1)
```

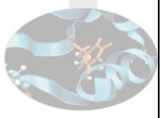

I tipi di dato: *ndtype …*



NumPy supporta molti più tipi di dato rispetto al Python puro: *21 built-in data type* che possono essere usati per creare array

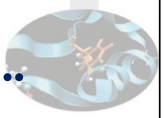
Type	Description
`bool`	*Boolean (`True or False`) stored as byte*
`int`	*Platform integer (normally either `int32` or `int64`)*
`int8`	*Byte (-128, 127)*
`int16`	*Integer (-32768, 32767)*
`int32`	*Integer (-2147483648, 2147483647)*
`int64`	*Integer (-9223372036854775808, 9223372036854775807)*

… I tipi di dato: *ndtype*



Type	Description
`uint32`	*unsigned integer (0, 4294967295)*
`uint64`	*unsigned integer (0, 18446744073709551615)*
`float`	*float64*
`float16`	*Half precision float*
`float32`	*Single precision float*
`float64`	*Double precision float*
`complex`	*complex128*

Creare array con i principali `dtype`…



- Vediamo l'uso della funzione `np.array` per creare array con i tipi di dato più utilizzati

Array di *float* e di *complex*

```
>>> import numpy as np
>>> a = np.array([1,2,3], dtype=float)    # float array
>>> a.dtype
dtype('float64')

>>> a = np.array([1+2j,3+4j])     # complex array
>>> a.dtype
dtype('complex128')
```

… Creare array con i principali `dtype`

Array di *boolean* e di stringhe

```
>>> a = np.array([True, False, False])
>>> a.dtype
dtype('bool')

>>> a = np.array(['Pippo','Pluto','Topolino'])
>>> a.dtype
dtype('S8')
```

Array di boolean

Array di stringhe

Le stringhe dell'array contengono un massimo di 8 caratteri

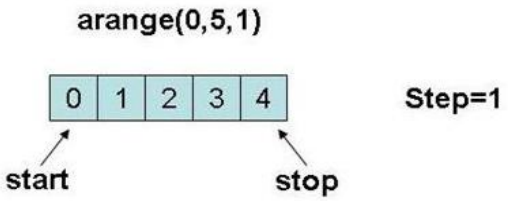
Funzioni per creare array: *arange*

- Nella pratica, raramente si crea un array elemento per elemento; all'interno di NumPy sono disponibili diverse funzionalità per inizializzare array

- La funzione ***arange*** ritorna un array di numeri equamente distribuiti (passo `step`) nell'intervallo [***start*** e ***stop)***

  `arange([start,] stop[, step], dtype=None)`

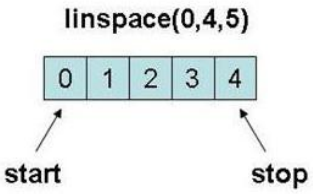
  - è l'analogo NumPy della funzione `range` del Python puro
  - se non specificato diversamente, `start` vale 0 e `step` vale 1
  - se non specificato, il tipo di dato viene ricavato dal tipo di `stop`



Funzioni per creare array: *linspace*

```
linspace(start, stop, num=50, endpoint=True,
         retstep=False)
```

- La funzione `linspace` ritorna un array di `num` numeri *floating-point* equamente distribuiti nell'intervallo `[start, stop]`



- La sequenza di numeri comprende l'estremo destro dell'intervallo, se non è specificato il parametro `endpoint=False`

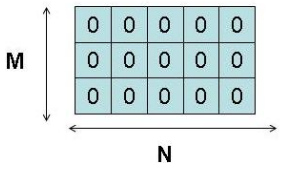
- Se `retstep=True`, oltre all'array, la funzione ritorna una tupla di 2 elementi: l'array ed un numero *floating-point* uguale alla distanza tra 2 numeri consecutivi della sequenza generata (*step*)

Altre funzioni per creare array …

- Se il contenuto di un array è a priori ignoto, è possibile utilizzare funzioni per riempire in modo sistematico l'array

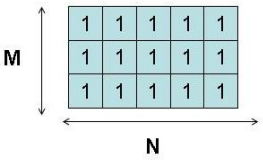
- La funzione `zeros` crea un array di dimensioni `shape` di soli zero

```
zeros(shape, dtype=float, order ='C')
```



- La funzione `ones` crea un array di dimensione `shape` di soli uno

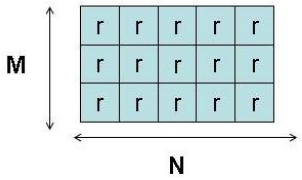
```
ones(shape, dtype=None, order ='C' )
```



…Altre funzioni per creare array

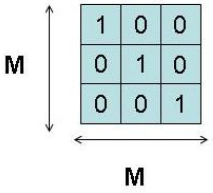
- La funzione *empty* crea un array di dimensioni *shape* senza inizializzazione.

```
empty(shape, dtype=None, order ='C')
```



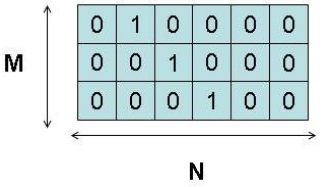
- La funzione *identity* genera la matrice identità *nxn*

```
identity(n, dtype=None)
```



- La funzione *eye* crea una matrice NxM di zero, riempendo di 1 la k-esima diagonale

```
eye(N, M=None, k=0, dtype=float)
```

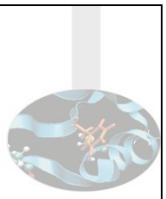


Creare array 1D di numeri casuali (Mersenne Twister)

Un assaggio del *sub-module* random di NumPy

```
>>> import numpy as np
>>> a = np.random.rand(4)    # Uniform in [0,1] distrib.
>>> a
array([0.8037485, 0.58855075, 0.84675915, 0.98103406])

>>> a = np.random.randn(4)  # Std-Normal distribution
>>> a
array([1.43748325, -0.86745428, 0.71094496, 0.90526009])

>>> np.random.seed(43587)    # Setting the random seed
```

Indexing di array …

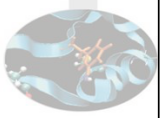


- Gli elementi di un array 1D si referenziano allo stesso modo di altre sequenze Python

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> a[0], a[2], a[-1]
(0, 2, 9)
```

- Come in C/C++ e in altre sequenze Python, gli indici cominciano da 0, diversamente dal Fortran e Matlab, in cui cominciano da 1

… Indexing di array

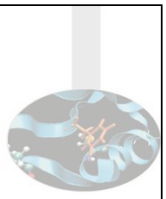


Per gli array multidimensionali, gli indici sono *tuple* di interi

```
>>> a = np.diag(np.arange(3))
>>> a
array([0, 0, 0],
      [0, 1, 0],
      [0, 0, 2])
>>> a[1,1]
1
>>> a[2,1] = 10       # 3° riga, 2° colonna
>>> a
array([0, 0, 0],
      [0, 1, 0],
      [0,10, 2])
```

- Nel caso 2D, la prima dimensione corrisponde alle righe e la seconda alle colonne
- Per array multi-dimensionali, espressioni come a [0] vengono interpretate prendendo tutti gli elementi nelle dimensioni non specificate

Slicing di array



- Come le altre sequenze Python, anche gli array possono essere ***sliced***

```
>>> a = np.arange(10))
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:9:3]          # [start:end:step]
array([2, 5, 8])
>>> a[:4]             # l'ultimo elemento non è incluso
array([0, 1, 2, 3])
```

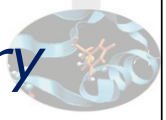
- Nessuno dei 3 componenti della *slice* è obbligatorio; per *default* `start=0`, `end` è l'ultimo elemento e *step=1*

```
>>> a[1:3]
array([1, 2])
```

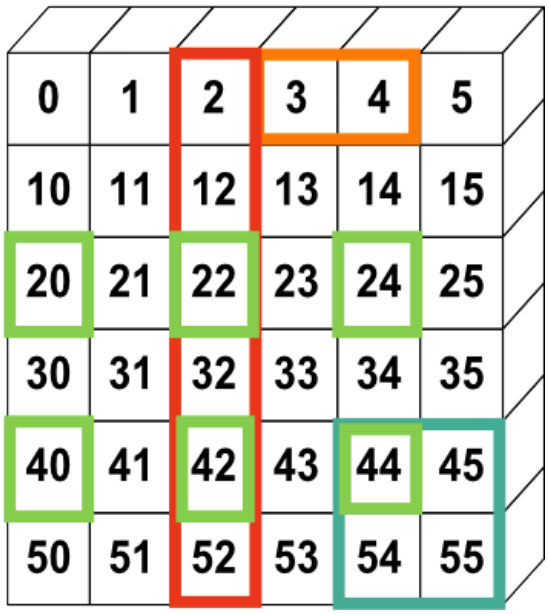
```
>>> a[::2]
array([0, 2, 4, 6, 8])
```

```
>>> a[7:]
array([7, 8, 9])
```

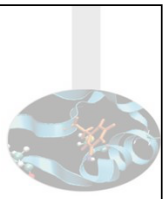

Indexing & slicing: a visual summary



```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



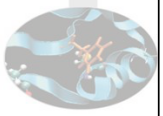
View & Copy …



- Un operazione di *slicing* crea una ***view*** dell'array originale, ovvero solo un modo differente per accedere ai dati dell'array
- Quando viene modificata la *view*, viene modificato l'array originale

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2]; b
array([0, 2, 4, 6, 8])
>>> b[0] = 10; b
array([10, 2, 4, 6, 8])
>>> a
array([10, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```


… View & Copy

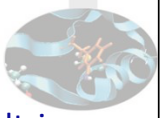


- La copia può essere "forzata", utilizzando il metodo copy dell'oggetto array, contestualmente all'operazione di *slicing*

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a[::2].copy
>>> b[0] = 10; b
array([10, 2, 4, 6, 8])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- Questo comportamento degli array NumPy può sembrare sorprendente, ma consente di risparmiare memoria e tempo d'esecuzione!

Fancy indexing ...

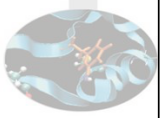


- Oltre che con lo *slicing*, in NumPy è possibile l'*indexing* di array con altri array di interi o booleani; questo metodo è noto come ***fancy indexing***
- Se un array a *fancy indexed* è assegnato ad un altro array b, gli elementi di b sono una copia degli elementi di a referenziati con il *fancy indexing*

Fancy indexing con array di boolean

```
>>> np.random.seed(3)
>>> a = np.random.random_integers(0,20,10); a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6])
>>> (a % 3 == 0)
array([False, True, False, True, False, False,
       False, True, False, True], dtype=bool)
>>> mask = (a % 3 == 0)
>>> from_a = a[mask]; from_a
array([3, 0, 9, 6])
```


... Fancy indexing ...

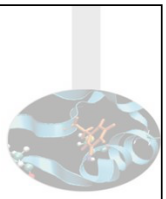


- L'*indexing* con un *mask array* può essere utilizzato per assegnare nuovi valori ad un sotto-insieme degli elementi dell'array

Modifica degli elementi di un array con fancy indexing

```
>>> np.random.seed(3)
>>> a = np.random.random_integers(0,20,10);
>>> a
array([10,  3,  8,  0, 19, 10, 11,  9, 10,  6])
>>> a[a % 3 == 0] = -2
>>> a
array([10, -2,  8, -2, 19, 10, 11, -2, 10, -2])
```

… Fancy indexing …



- L'*indexing* con un *mask array* può essere utilizzato per assegnare nuovi valori ad un sotto-insieme degli elementi dell'array

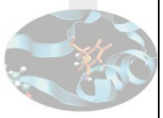
Fancy indexing con array di interi

```
>>> a = np.range(10); a
array([0,  1,  2,  3, 4, 5, 6, 7, 8, 9])
>>> a[[2, 3, 2, 4, 2]]
array([2,  3,  2,  4, 2])
>>> a[[9, 7]] = -1
>>> a
array([0,  1,  2,  3, 4, 5, 6, -1, 8, -1])
```

NB: `[2, 3, 2, 4, 2]` è una lista e alcuni indici possono essere ripetuti più volte

- Anche con un *mask* array d'interi è possibile modificare il valore di un sotto-array

… Fancy indexing



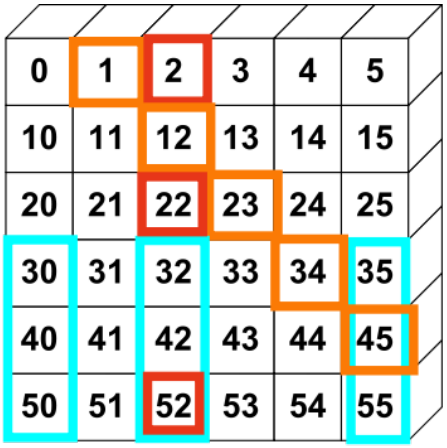
- Se un array è creato a partire dall'*indexing* di un array esistente con un *mask array di interi,* la *shape* del nuovo array è la stessa del *mask array*

Fancy indexing con array di interi

```
>>> a = np.range(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> idx = [[3, 4], [9, 7]]
>>> b = a[idx]
>>> b
array([[3, 4],
       [9, 7]])
```

Altri *fancy indexing* illustrati

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```



Iteration …

- *Se proprio necessaria*, l'iterazione sugli elementi di un array può essere effettuata con il tipico ciclo *for* lungo gli *assi* dell'array

Ciclo for sugli elementi di un array 2D (anche se da evitare!)

```
>>> a = np.empty((3,3))

>>> for i in xrange(a.shape[0]):
        for j in xrange(a.shape[1]):
            a[i,j] = i+j

>>> a
array([[ 0.,  1.,  2.],
       [ 1.,  2.,  3.],
       [ 2.,  3.,  4.]])
```

… Iteration

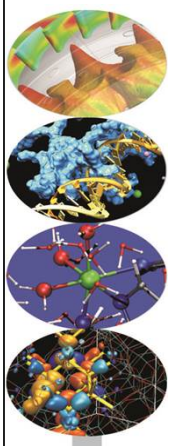
- Il ciclo *for* applicato direttamente all'array, agisce sul primo asse;
- Nell'esempio vediamo cosa succede sull'array a ottenuto nella slide precedente

```
>>> for i in a:
        print(i)

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

- Se all'array è applicato l'*iterator* `flat`, invece il ciclo for agisce su tutti gli elementi

```
>>> for i in a.flat:
        print(i)
0.0
1.0
2.0
1.0
2.0
3.0
2.0
3.0
4.0
```

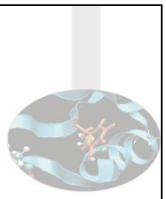
- **L'iterazione sugli array *NumPy* è estremamente inefficiente!**
- **Per le operazioni sugli array NumPy dispone di un vasto insieme di funzioni scritte in C, che operano direttamente sull'intero array**



Introduzione a ***NumPy***: laboratorio 1

CINECA SCAI
SuperComputing Applications and Innovation

NumPy: laboratorio 1



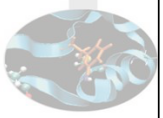
1. Creare i seguenti array, con il corretto *data-type* ed usando il minor numero di istruzioni possibile

```
[[1, 1, 1, 1],
 [1, 1, 1, 1],
 [1, 1, 1, 2],
 [1, 6, 1, 1]]

[[0., 0., 0., 0., 0.],
 [2., 0., 0., 0., 0.],
 [0., 3., 0., 0., 0.],
 [0., 0., 4., 0., 0.],
 [0., 0., 0., 5., 0.],
 [0., 0., 0., 0., 6.]]
```

Hint: consultare l'help di `np.diag`

CINECA SCAI
SuperComputing Applications and Innovation

NumPy: laboratorio 1 (2)



2. Usare la funzione `np.tile` (`np.tile?` per accedere all'help) per creare il seguente array

```
[[4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1],
 [4, 3, 4, 3, 4, 3],
 [2, 1, 2, 1, 2, 1]]
```

3. Senza scriverlo esplicitamente, generare il seguente array e, costruire un secondo array contenente solo la 2° e la 4° riga

```
[[1,  6, 11],
 [2,  7, 12],
 [3,  8, 13],
 [4,  9, 14],
 [5, 10, 15]]
```

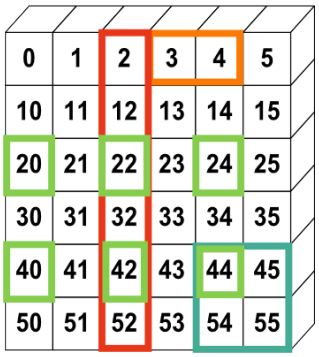
Hint: consultare l'help di `np.reshape`

NumPy: laboratorio 1 (3)

4. A partire esclusivamente dagli array `a = np.arange(10)` e `b = np.arange(5)`, costruire l'array

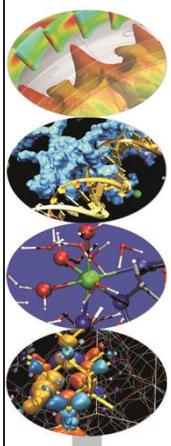
```
[0, 1, 2, 3, 4, 4, 3, 2, 1, 0]
```

5. Dato l'array in figura:
   - generare i sotto-array evidenziati in arancio, azzurro, rosso e verde
   - generare altri sotto-array a piacere, usando *indexing & slicing* discussi



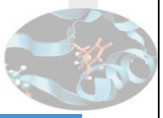
NB: l'array in figura si ottiene con l'istruzione

```
a = np.arange(6) + np.arange(0,51,10)[:,np.newaxis]
```

Introduzione a ***NumPy***:
operazioni numeriche su array

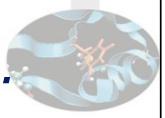
Operazioni elementari con scalari



Operazioni elementari con scalari

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1                  # somma di uno scalare
array([2, 3, 4, 5])
>>> a – 2                  # sottrazione di uno scalare
array([-1, 0, 1, 2])
>>> 3*a                    # moltiplicazione per uno scalare
array([3, 6, 9, 12])
>>> 2**a                   # potenza (base scalare)
array([2, 4, 8, 16])
>>> a = np.array([1., 2., 3., 4.])
>>> a/2                    # divisione per uno scalare
array([0.5, 1., 1.5, 2.])
```

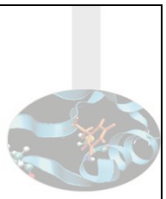

Operatori aritmetici *elementwise* ...



Gli operatori aritmetici tra array lavorano elemento per elemento

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.ones(4) + 1
>>> a – b                  # sottrazione tra array
array([-1., 0., 1., 2.])
>>> a*b                    # moltiplicazione tra array
array([2., 4., 6., 8.])
>>> a += 1; a              # autoincremento di array
array([2, 3, 4, 5])
>>> 2**(a+1) – a           # espressione con array
array([ 6, 13, 28, 59])
```

Efficienza degli operatori aritmetici elementwise



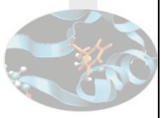
L'uso di operatori aritmetici su array NumPy produce un codice molto più efficiente rispetto al caso in cui si usi un loop del Python puro.

Prestazioni delle operazioni elementwise su array vs. loop

```
>>> a = np.arange(10000)
>>> %timeit a + 1
100000 loops, best of 3: 16.4 µs per loop
>>> l = range(10000)
>>> %timeit [i+1 for i in l]
1000 loops, best of 3: 741 µs per loop
```

45x

Occhio alla moltiplicazione

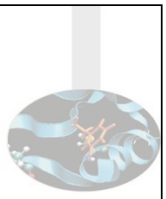


Come detto l'operatore prodotto di array opera *elementwise*, quindi, se applicato al caso 2D, il risultato non ha nulla a che fare con la moltiplicazione di matrici standard!

Moltiplicazione di array e metodo dot

```
>>> d = np.ones(3,3)

>>> d*d                 # moltiplicazione elementwise
array([1., 1., 1.],
      [1., 1., 1.],
      [1., 1., 1.])

>>> d.dot(d)            # prodotto di matrici
array([3., 3., 3.],
      [3., 3., 3.],
      [3., 3., 3.])
```

Operatori di confronto di array

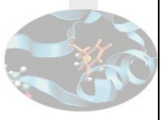


```
>>> a = np.array([1, 2, 3, 4])

>>> b = np.array([4, 2, 2, 4])

>>> a == b
array([False, True, False, True])

>>> a > b
array([True, False, True, True])

>>> c = np.array([1, 2, 3, 4])

>>> np.array_equal(a,b)
False

>>> np.array_equal(a,c)
True
```

Anche gli operatori standard di confronto lavorano *elementwise*

Per un confronto *arraywise* si usa il metodo `array_equal`

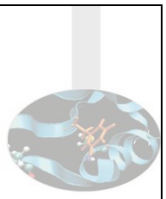
Operatori logici su array



Operazioni logiche tra array

```
>>> a = np.array([1, 1, 0, 0], dtype=bool)

>>> b = np.array([1, 0, 1, 0], dtype=bool)

>>> np.logical_or(a,b)
array([True, True, True, False], dtype=bool)

>>> np.logical_and(a,b)
array([True, False, False, False], dtype=bool)
```

Shape mismatch

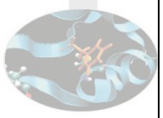


- Visto che tutte le tipologie di operatori tra array lavorano *elementwise*, gli array operandi devono avere (sempre) la stessa *shape*

```
>>> a = np.arange(4)

>>> a + np.array([1,2])
Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast
    together with shapes (4) (2)
```

- *Broadcasting*? Ne parliamo più avanti….

Universal functions

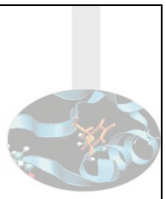


- Oltre alla definizione dell'oggetto `ndarray`, NumPy dispone anche delle cosiddette ***Funzioni Universali*** (*ufunc*)
- Si tratta di funzioni che operano sull'intero array elemento per elemento, evitando così l'uso di loop espliciti
- Le *Universal Function* sono dei *wrapper* a funzioni del core di *NumPy,* tipicamente sviluppate in C o Fortran

Una ufunc al lavoro

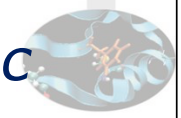
```
>>> import numpy as np

>>> a = np.arange(100)

>>> b = np.cos(a)
```

Universal functions



- Nella versione attuale di NumPy sono disponibili oltre 60 *ufunc*, che coprono una vasta tipologia di operazioni
- Alcune *ufunc* sono "nascoste" dietro gli operatori aritmetici; a titolo di esempio la funzione `np.multiply(a,b)` viene chiamata quando si effettua l'operazione `a*b` e a e/o b sono oggetti *`ndarray`*
- NumPy offre *Universal Function* per:
  - Operazioni matematiche di base
  - Operazioni trigonometriche, esponenziali e logaritmiche
  - Manipolazione di bit (es. `np.bitwise_or, np.left_shift,` …)
  - comparazione di array (es `np.maximum(a,b)`,…)
  - Operazioni *floating point* (es `np.floor, np.ceil, np.isreal,` …)
- **User guide:** http://docs.scipy.org/doc/numpy/reference/ufuncs.html

Nota sulle performance delle *ufunc*



- Le *ufunc* di NumPy sono estremamente efficienti per lavorare su array, ma possono lavorare anche sugli scalari
- Va notato però che le funzioni contenute nel modulo *math* di Python *sono più veloci* rispetto alle *ufunc* di NumPy utilizzate su variabili scalari

Confronto tra le funzioni seno di *`math`* e *NumPy* su uno scalare

```
In [1]: import math

In [2]: import numpy as np

In [3]: %timeit [math.sin(math.pi)]
1000000 loops, best of 3: 416 ns per loop

In [4]: %timeit [np.sin(np.pi)]
100000 loops, best of 3: 4.06 µs per loop
```

9x

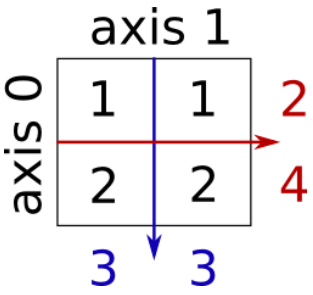
Somma degli elementi di un array

- Calcolo della somma degli elementi di un array 1D

```
>>> x = np.array([1, 2, 3, 4]); x.sum()
10
```

- Nel caso di un array multi-dimensionale

```
>>> x = np.array([[1, 1], [2, 2]])

>>> x.sum(axis=0)  # somma per colonna
array([3, 3])

>>> x.sum(axis=1)  # somma per riga
array([2, 4])

>>> x.sum()  # somma sull'intero array
6
```



Altre operazioni di riduzione

- Tutte le altre operazioni di riduzione di array lavorano come la somma (compreso l'uso dell'argomento `axis`)

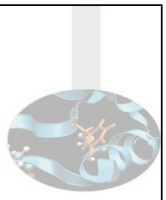
- Sono disponibili operatori di riduzione di array di diverse tipologie:
  - Statistica: `ndarray.mean()`, `ndarray.std()`, `ndarray.median()`, …
  - Calcolo di estremi: `ndarray.max()`, `ndarray.min()`, `ndarray.argmax()` – ritorna l'indice dell'elemento massimo -, `ndarray.argmin()`…
  - Logiche: `ndarray.all()` – ritorna `True` se tutti gli elementi dell'array, o lungo un asse, sono `True` – e `ndarray.any()` – ritorna `True` se almeno un elemento dell'array, o di un suo asse, è `True`

Riduzione di array con operazioni logiche: un'applicazione

Confronto di array tramite operazioni logiche di riduzione

```
>>> a = np.zeros((100,100))

>>> np.any(a != 0), np.all(a == a)
(False, True)

>>> a = np.array([1, 2, 3, 2])

>>> b = np.array([2, 2, 3, 2])

>>> c = np.array([6, 4, 4, 5])

>>> ((a <= b) & (b <= c)).all()
True
```

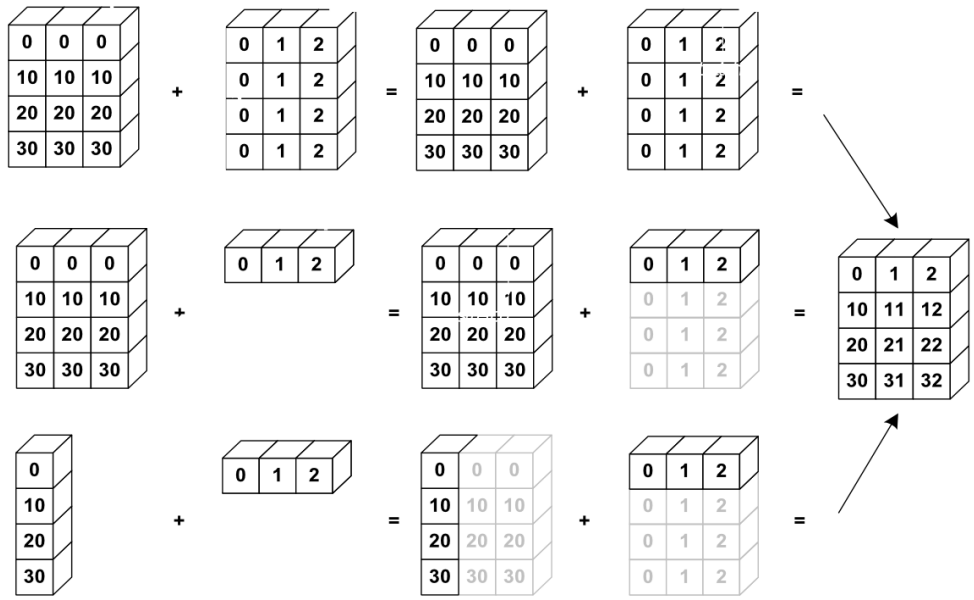

Broadcasting



- Le operazioni di base su array *NumPy* sono *elementwise, cosa concettualmente semplice e chiara* quando gli array operandi hanno la stessa *shape*
- In certe condizioni, *NumPy* permette di eseguire operazioni *elementwise*, anche su array di dimensione diversa

  trasformando gli array operanti in modo che abbiano tutti la stessa dimensione: questa conversione è chiamata ***broadcasting***
- Il *broadcasting* segue due regole*:*
  - se gli array operandi non hanno lo stesso numero di dimensioni, un 1 viene preposto alla *tupla* di `shape` dell'array più piccolo fino a che gli array non abbiano lo stesso numero di dimensioni
  - Gli array di `size` 1 lungo una particolare direzione si comportano come l'array più grande lungo quella dimensione; il valore dell'array è assunto essere lo stesso lungo tutta la direzione di *broadcast*

Un esempio di *broadcasting*

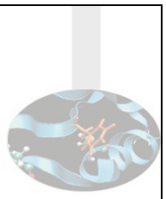


Broadcasting at work …

Esempi di broadcasting di array

```
>>> a = np.array([1,2,3])   # a.shape =(3,)

>>> b = np.array([[1,2,3],[4,5,6]])  # b.shape=(2,3)

>>> c = a + b; c.shape
(2, 3)

>>> a = np.arange(6); a = a.reshape((2,1,3)); a.shape
(2, 1, 3)

>>> b = np.arange(8); b = b.reshape((2,4,1)); b.shape
(2, 4, 1)

>>> c = a + b; c.shape
(2, 4, 3)
```

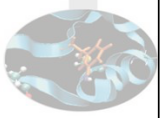
… Broadcasting at work



Il Broadcasting non sempre è possibile

```
>>> a = np.arange(15)

>>> a = a.reshape(((3,5))

>>> b = np.arange(7)

>>> a + b
ValueError          Traceback (most recent call last):
…
…
ValueError: operands could not be broadcast together
with shapes (3,5) (7)
```


Shape manipulation: flattening

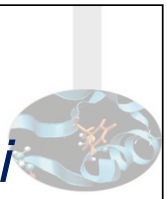


```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> a.ravel()
array([1, 2, 3, 4, 5, 6])
>>> a.T          # array trasposto
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> a.T.ravel()
array([1, 4, 2, 5, 3, 6])
```

NB: in generale, il metodo `ravel()` crea una *view* ed Il *flattening* avviene a partire dall'ultimo asse e, in sequenza, fino al primo

CINECA SCAI
SuperComputing Applications and Innovation

Shape manipulation: reshaping e aggiunta di dimensioni

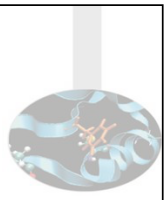


```
>>> a = np.arange(1,7); a.shape
(6,)
>>> b = a.reshape(2,3); b
array([[1, 2, 3],
       [4, 5, 6]])

>>> a = np.array([1,2,3]); a.shape
(3,)

>>> b = a[:, np.newaxis]; b.shape
(3, 1)

>>> c = a[np.newaxis, :]; c.shape
(1, 3)
```

NB: il metodo `ndarray.reshape()` può ritornare una *view* o una copia

CINECA SCAI
SuperComputing Applications and Innovation

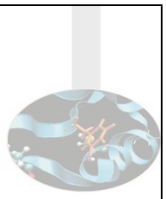
Shuffling delle dimensioni



```
>>> a = np.arange(4*3*2).reshape(4,3,2)
>>> a.shape
(4,3,2)
>>> a[0, 2, 1]
5

>>> b = a.transpose(1,2,0);

>>> b.shape
(3,2,4)
>>> b[2, 1, 0]
5
```

NB: il metodo `ndarray.transpose()` ritorna una *view*

Sorting



- Il metodo `nparray.sort()`

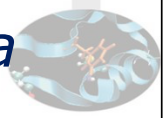
```
>>> a = np.array([[4, 3, 5], [1, 2, 1]])
>>> b = a.sort(); b
array([[3, 4, 5],
       [1, 1, 2]])
>>> a.sort(); a      #in-place sort
array([[3, 4, 5],
       [1, 1, 2]])
```

NB: ogni riga è ordinata separatamente

- *Sorting* attraverso il *fancy indexing*

```
>>> a = np.array([4, 3, 1, 2])
>>> j = np.argsort(a); j
array([2, 3, 1, 0])
>>> a[j]
array([1, 2, 3, 4])
```


Valutare una funzione 2D su griglia



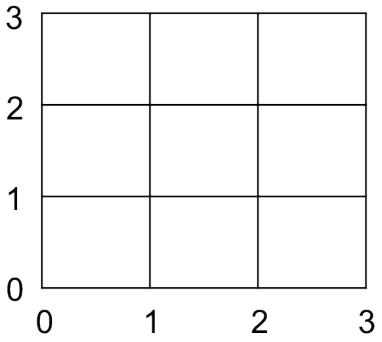
- Nelle scienze computazionali spesso capita di dover valutare una funzione su una griglia
- Data la griglia dei punti (x_i,y_i) con x_i = [0, 1, 2, 3] e y_i = [0, 1, 2, 3], vogliamo calcolare, per ciascun punto della griglia, il valore di una funzione f(x,y)
- Ingenuamente, potremmo pensare ad una soluzione NumPy come questa:

```
>>> x = np.arange(4)
>>> y = np.arange(4)
>>> def f(x,y):
...     return x**2+y
>>> f(x,y)
array([0, 2, 6, 12])
```

ma, il risultato è lungi da essere quello aspettato!

La soluzione: meshgrid

```
>>> x = np.arange(4)

>>> y = np.arange(4)

>>> xx, yy = np.meshgrid(x,y)

>>> xx
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])

>>> yy
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3]])

>>> def f(x,y):
...     return x**2+y
```




```
>>> f(xx,yy)
array([[ 0,  1,  4,  9],
       [ 1,  2,  5, 10],
       [ 2,  3,  6, 11],
       [ 3,  4,  7, 12]])
```


Type casting …

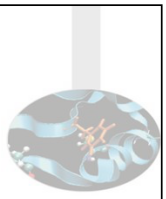
- In operazioni con dati di tipo differente, il risultato è del tipo “più grande”

```
>>> np.array([1, 2, 3]) + 1.5
array([2.5, 3.5, 4.5])
```

- Un’ assegnazione non cambia mai il tipo di dato

```
>>> a =np.array([1, 2, 3]); a.dtype
dtype('int64')

>>> a[0] = 1.9; a     # il float è troncato!
array([1, 2, 3])
```

…Type casting



- Casting esplicito

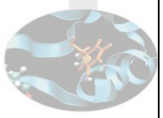
```
>>> a = np.array([1.7, 1.2, 1.9])
>>> b = a.astype(int); b  # tronca a interi
array([1, 1, 1])
```

- Arrotondamenti

```
>>> a = np.array([1.2, 1.5, 1.6, 2.5, 3.5, 4.5])

>>> b = np.around(a); b   # restano floating-point
[1., 2., 2., 2., 4., 4.]

>>> c = np.around(a).astype(int)

>>> c
array([1, 2, 2, 2, 4, 4])
```

L'arrotondamento di numeri tipo 1.5, 2.5, 3.5 e 4.5 avviene all'intero pari più vicino. Se il risultato sorprende, vedere la nota nella *docstring* `np.around()`

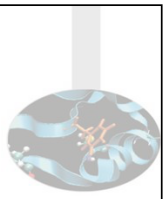
Structured data type …



- *NumPy* consente di creare e gestire array con tipi di dati strutturati, costruiti a partire dai tipi di base
- Il metodo `numpy.dtype` consente di definire il nuovo tipo di dato; supponiamo di voler creare un nuovo tipo di dato, così strutturato
  1. `'code'`: stringa di 1 carattere
  2. `'pos'`: numero *intero*
  3. `'val'`: numero *floating-point*

```
>>> dt = np.dtype([('code','S1'),('pos',int),
                   ('val',float)])

>>> a = np.array([('A',1,0.37), ('B',2,0.11)], dtype=dt)

>>> a
array([('A', 1, 0.37), ('B', 2, 0.11)],
      dtype=[('code','S1'),('pos','<i8'),('val','<f8')])
```

… Structured data type



- Utilizzando il tipo di dato ed il relativo array definiti nella slide precedente, vediamo come si referenziano gli array di dati strutturati

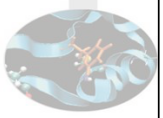
```
>>> a['code']
array(['A', 'B'], dtype='|S1')

>>> a['pos']
array([1, 2])

>>> a[['code','val']]
array([('A', 0.37), ('B', 0.11)],
      dtype=[('code', 'S1'), ('val', '<f8')])
```

- NB: sintassi alternativa di `numpy.dtype()`

```
numpy.dtype({'names': n_tuple,'formats': f_tuple})
```

in cui `n_tuple` ed `f_tuple` sono rispettivamente la tupla dei nomi dei campi del dato strutturato ed quella dei relativi formati

I/O con array *NumPy*



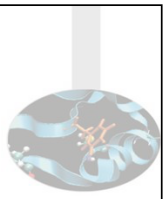
- Per la lettura di file di testo, NumPy dispone della funzione `loadtxt`

```
loadtxt(filename, dtype=<type 'float'>,
    comments='#', delimiter=None,
converters=None, skiprows=0, usecols=None,
    unpack=False, ndmin=0)
```

- La funzione `savetxt`, si occupa invece della scrittura di un array su file:

```
numpy.savetxt(filename, ndarray, fmt='%.18e',
    delimiter='', newline='\n', header='',
        footer='', comments='#')
```

np.loadtxt *at work*

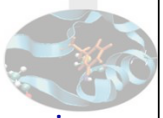


Supponiamo di dover leggere il file `tests.txt`

Student	Test1	Test2	Test3	Test4
Jane	98.3	94.2	95.3	91.3
Jon	47.2	49.1	54.2	34.7
Jim	84.2	85.3	94.1	76.4

```
>>> a = np.loadtxt('tests.txt',skiprows=2,usecols=range(1,5))

>>> a
array([[ 98.3,  94.2,  95.3,  91.3],
       [ 47.2,  49.1,  54.2,  34.7],
       [ 84.2,  85.3,  94.1,  76.4]])

>>> b = np.loadtxt('tests.txt',skiprows=2,usecols=(1,-2))

>>> b
array([[ 98.3, 95.3],
       [ 47.2, 54.2],
       [ 84.2, 94.1]])
```


I/O binario di array *NumPy …*

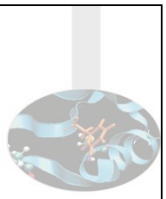


Se si deve lavorare con data set grandi, è più conveniente scrivere e leggere in formato binario. Il modo più semplice per farlo è usare il modulo della libreria standard *`cPickle`*

Scrittura su file binario degli array NumPy a1 e a2

```
>>> import cPickle

>>> file = open('tmp.dat', 'wb')

>>> file.write('This is the array a1:\n')

>>> cPickle.dump(a1, file)

>>> file.write('Here is another array a2:\n')

>>> cPickle.dump(a2, file)

>>> file.close()
```

… I/O binario di array *NumPy*

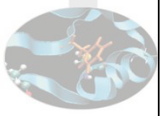


Lettura del file binario generato nella slide precedente

```
>>> file = open('tmp.dat', 'rb')
>>> file.readline()        # legge il 1° commento
>>> b1 = cPickle.load(file)
>>> file.readline()        # legge il 2° commento
>>> b2 = cPickle.load(file)
>>> file.close()
```

L'uso del modulo `cPickle` garantisce operazioni di I/O più rapide e un immagazzinamento dati a minor costo

I/O nel formato *NumPy*



NumPy prevede anche un proprio formato, non portabile, ma piuttosto efficiente nelle operazioni di I/O

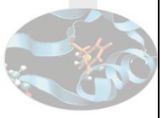
Scrittura e lettura binaria nel formato NumPy

```
>>> my_data = np.ones((3,3))
>>> np.save('myData.npy', my_data)
>>> r_data = np.load('myData.npy')
```

In Python è possibile eseguire operazioni di I/O nei formati binari più diffusi in ambito tecnico scientifico:

- Il supporto al formato HDF5 è disponibile nel modulo ad-hoc *h5py* (http://www.h5py.org/) ed in *PyTables*
- *SciPy* dispone del supporto per l'IO nei formati *NetCDF, Matlab e MatrixMarket*

Una pillola di "vettorizzazione" (1)



- I cicli *for* sono piuttosto lenti in Python.
- Uno dei vantaggi nell'utilizzo degli array NumPy sta nel fatto che molte operazioni possono essere svolte evitando *loop* espliciti: ***vettorizzazione***

Scalar version

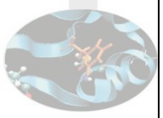
```
>>> y=zeros(len(a))
>>> for i in xrange(len(a)):
        y[i]=sin(a[i])*2
```

Vectorized Version

```
>>> a=np.arange(0,4*pi,0.1)
>>> y=sin(a)*2
```

- In alcuni casi è necessario vettorizzare esplicitamente l'algoritmo:
  - direttamente, utilizzando il metodo `vectorize()` # piuttosto lento
  - manualmente, per esempio utilizzando "opportunamente" lo *slicing*

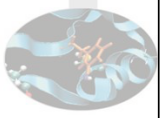
Una pillola di "vettorizzazione" (2)



Solo in alcuni casi è possibile vettorizzare un'espressione; vediamo un esempio:

```
>>> def func(x):
        if x<0: return 1
        else: return math.sin(x)

>>> func(3)
0.1411200080598672

>>> func(array([1,-2,3]))
Traceback (most recent call last):
ValueError: The truth value of an array with
more than one element is ambiguous. Use
a.any() or a.all()
```

Una pillola di "vettorizzazione" (3)

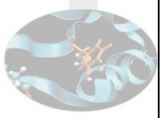


- Per poter lavorare con un array, è necessaria un'implementazione scalare della funzione

```
>>> def func_np(x):
        r = x.copy()    # result array
        for i in xrange(size(x)):
           if x[i] < 0: r[i] = 0.0
           else: r[i] = math.sin(x[i])
        return r

>>> a = np.array([1.,-2.,3.])

>>> func_np(a)
array([0.84147098, 0., 0.14112001])
```

- E' un'implementazione molto lenta e funziona solo per array 1D!

Una pillola di "vettorizzazione" (4)



- In casi come quello visto, la soluzione efficiente è nell'uso della funzione `np.where`

```
def f(x):
    if condition:
       x = <expression1>
    else:
       x = <expression2>
    return x
```

```
def f_vect(x):
    x1 = <expression1>
    x2 = <expression2>
    return np.where(condition,x1,x2)
```

Evitato l'uso del ciclo for ed ottenuto una funzione utile anche nel caso multidimensionale

Una pillola di "vettorizzazione" (5)

- Lo *slicing* di array è spesso utilizzato per la vettorizzazione di operazioni.
- Ad esempio, nel caso di applicazione di schemi d'integrazione alle differenze finite o nel processing di immagini, è comune incontrare espressioni del tipo:

$$x_k = x_{k-1} + 2x_k + x_{k+1} \quad k = 1, 2, \ldots, n-1$$

Implementazione con un loop (da evitare!)

```
for i in xrange(1,len(x)-1):
    x[i] = x[i-1] + 2*x[i] + x[i+1]
```

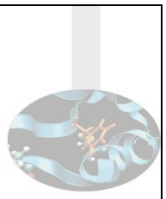
Implementazione con slicing di array: vettorizzata!

```
x[1:n-1] = x[0:n-2] + 2*x[1:n-1] + x[2:n]
```


`numpy.linalg`

- Il modulo Numpy contiene anche un sub-module dedicato all'algebra lineare; si tratta di `numpy.linalg`, presente ormai solo per ragioni storiche (era in *numarray*, da cui NumPy deriva)
- Il modulo `linalg` contiene funzioni per risolvere sistemi lineari, problemi agli autovalori, calcolare diversi tipi di fattorizzazioni di matrici, …
- Quando le performance contano, piuttosto che ricorrere al modulo `numpy.linalg` è consigliabile utilizzare le funzionalità del modulo `scipy.linalg`, che è in grado di interfacciarsi alle librerie BLAS e Lapack specifiche per la macchina in uso

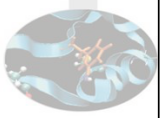
dir(numpy.linalg)



Per avere un'idea del contenuto del modulo:

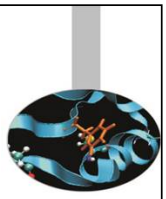
```
>>> dir(np.linalg)
['LinAlgError', 'Tester', '__builtins__',
'__doc__', '__file__', '__name__',
'__package__', '__path__', 'bench', 'cholesky',
'cond', 'det', 'eig', 'eigh', 'eigvals',
'eigvalsh', 'info', 'inv', 'lapack_lite',
'linalg', 'lstsq', 'matrix_power',
'matrix_rank', 'norm', 'pinv', 'qr', 'slogdet',
'solve', 'svd', 'tensorinv', 'tensorsolve',
'test']
```


numpy.random

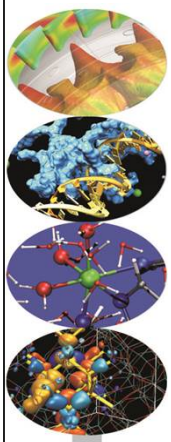


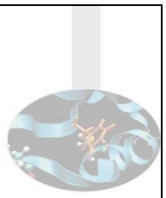
All'interno di NumPy è presente anche il modulo `random`, che contiene diverse per la generazione di numeri casuali provenienti da un vasto insieme di possibili distribuzioni. Per avere un'idea del contenuto di random

```
>>> dir(np.random)
['RandomState', 'Tester', '__RandomState_ctor', '__all__',
'__builtins__', '__doc__', '__file__', '__name__',
'__package__', '__path__', 'bench', 'beta', 'binomial',
'bytes', 'chisquare', 'dirichlet', 'exponential', 'f',
'gamma', 'geometric', 'get_state', 'gumbel', 'hypergeometric',
'info', 'laplace', 'logistic', 'lognormal', 'logseries',
'mtrand', 'multinomial', 'multivariate_normal',
'negative_binomial', 'noncentral_chisquare', 'noncentral_f',
'normal', 'np', 'pareto', 'permutation', 'poisson', 'power',
'rand', 'randint', 'randn', 'random', 'random_integers',
'random_sample', 'ranf', 'rayleigh', 'sample', 'seed',
'set_state', 'shuffle', 'standard_cauchy',
'standard_exponential', 'standard_gamma', 'standard_normal',
'standard_t', 'test', 'triangular', 'uniform', 'vonmises',
'wald', 'weibull', 'zipf']
```



Introduzione a ***NumPy***: laboratorio 2

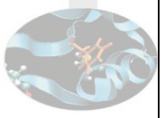




NumPy: Laboratorio 1 (4)

1. Costruire un array 2D e calcolare la media lungo le singole righe dei suoi valori che appartengono ad una colonna ad indice pari (dispari)
   hint: `np.mean?`
2. Scrivere uno script Python che stampa un array con i numeri primi compresi tra 0 e 100.
   NB: utilizzare l'algoritmo di generazione dei numeri primi noto come "Crivello di Eratostene" :
   (http://it.wikipedia.org/wiki/Crivello_di_Eratostene)

NumPy: Laboratorio 2



3. Usando lo *slicing* di array, calcolare la derivata numerica della funzione *sin(x)* tra 0 e 2π; calcolare poi la massima distanza tra la derivata numerica e quella analitica e la media della stessa distanza sull'intera griglia.

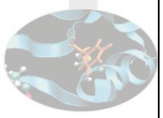
   hint: se abbiamo una griglia sufficientemente fitta in cui è valutata la funzione sin(x), la sua derivata è ben approssimata dal rapporto incrementale

4. Calcolare l'integrale Monte-Carlo unidimensionale di una funzione data (ad es *f(x)=1+2x*)

$$\int_a^b f(x)\,dx \approx \frac{b-a}{n}\sum_{i=1}^{n} f(x_i)$$

   dove i punti x_i sono numeri random uniformemente distribuiti nell'intervallo *a, b.* Costruire uno script Python che implementi tale integrazione con un *loop* esplicito e tramite vettorizzazione con NumPy. Infine, testare il cpu time delle due porzioni del programma tramite la funzione `time.clock()`

NumPy: Laboratorio 2 (2)

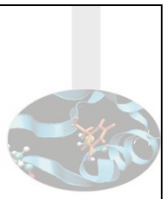


5. Dato un gioco nel quale si vince se la somma di 4 dadi è minore di 10, determinare se conviene giocarci qualora si vinca 10 volte il valore della posta
   hint: usare `np.random.radint` per creare un array bidimensionale - con shape (4, n) - di interi casuali compresi tra 1 e 6

6. leggere il file `angoli.txt` (timestep, angolo) con NumPy e costruire la distribuzione degli angoli tra 0 – 360 gradi a step di 5 gradi. Infine, disegnare il grafico *pyplot* della distribuzione
   hint: usare ad esempio

```
cont = int(angolo)/step
dist[cont] += 1
```

NumPy: Laboratorio 2 (3)



7. Leggere il file di dati `populations.txt`, contenente le popolazioni di lepri e linci (e carote) in Canada del Nord dal 1900 al 1920. Calcolare:
   - La media e la deviazione standard di ogni specie nel periodo
   - In quale anno ogni specie ha la massima popolazione
   - Quale specie ha la massima popolazione per ogni anno (*hint*: `np.argsort` e *fancy indexing*)
   - In quali anni almeno una delle popolazioni è superiore a 50.000
   - I due anni, per ogni specie, in cui la popolazione è più bassa