Pacchetto Numpy

Python e Calcolo Scientifico

Un elaboratore può essere utilizzato per risolvere problemi scientifici

Tipicamente, si tratta di applicazioni caratterizzate da:

- Grandi quantità di dati
- Elaborazioni complesse ed onerose
- Composizione di diversi sottoproblemi

Del linguaggio Python abbiamo detto che:

- Permette di sviluppare molto velocemente
- ...Ma in termini di esecuzione è piuttoto lento

Rispetto al calcolo scientifico:

- La prima caratteristica rappresenta un grosso vantaggio
- ...Ma la seconda è apparentemente una barriere invalicabile

Pacchetto `numpy`

Il problema delle prestazioni viene risolto attraverso pacchetti esterni

Paccheti dedicati possono offrire:

- Strutture dati adatte a gestire grandi quantità di informazioni
- Algoritmi efficienti per problemi di occorrenza frequente

Entrambi possono essere implementati in linguaggi efficienti come C, C++, o Fortran

Il principale di questi pacchetti si chiama `numpy` ed offre:

- Una struttura dati per gestire dati in forma tensoriale
- Algoritmi per diversi problemi di calcolo numerico molto comuni

Permette così di fare in Python quello che tradizionalmnete si faceva in Matlab

Classe `numpy.array`

La struttura dati principale offerta da `numpy` si chiama `array`

Da un punto di vista matematico rappresenta un tensore

- Un tensore è una collezione n-dimensionale di elementi contigui
- Intuitivamente, è la generalizzazione di una matrice ad n dimensioni
- 1 dimensione = vettore, 2 dimensioni = matrice, > 3 dimensioni = tensore

Dal punto di vista implementativo

- I dati di un `array` sono memorizzati come sequenza mono-dimensionale
- L'`array` ha una forma che indica il numero di elementi per ogni dimensione
- La forma viene utilizzata per determinare come accedere agli elementi


Classe `numpy.array`

Vediamo come esempio una matrice $2x3$

La matrice vera è propria è:

$$\begin{pmatrix} x_{0,0} & x_{0,1} & x_{0,2} \\ x_{1,0} & x_{1,1} & x_{1,2} \end{pmatrix}$$

...Ma viene memorizzata (e.g.) per righe, come:

$$\begin{pmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{1,0} & x_{1,1} & x_{1,2} \end{pmatrix}$$

- La forma è in questo caso $(2, 3)$
- L'indice bidimensionale (i, j) corrisponde all'indice lineare $3i + j$

Utilizzo di `numpy`

`numpy` non fa parte dell'installazione minima di Python

- È pre-installato in alcune distribuzioni (e.g. Anaconda)
- ...E si può installare in ogni caso usando un package manager
  - E.g. `conda install numpy` o `pip install numpy`

`numpy` si può importare nel solito modo

...Canonicamnte lo si abbrevia come `np`

```
In [1]: import numpy as np
```

- La documentazione è reperibile online
- ...Ed accessibile con `help(numpy)` o ahche `help('numpy')`


Creazione di Array

Si può convertire una collezione sequenziale in un array:

In [2]:
```
x = [1, 2, 3]
a = np.array(x)
print('Collezione originale:', x)
print('Array:', a)
```

```
Collezione originale: [1, 2, 3]
Array: [1 2 3]
```

- La forma di un array è disponibile nell'attributo `shape`

In [3]:
```
a.shape
```

Out[3]: (3,)

- `shape` è sempre una tupla (in questo caso con un solo elemento)

Creazione di Array

Usando collezioni innestate is ottengono array multi-dimensionali

E.g. una lista di liste diventa un array bidimensionale

In [4]:
```
x = [[1, 2, 3],
     [4, 5, 6]]
a = np.array(x)
print('Collezione originale:', x)
print('Array:')
print(a)
print('Forma:', a.shape)
```

```
Collezione originale: [[1, 2, 3], [4, 5, 6]]
Array:
[[1 2 3]
 [4 5 6]]
Forma: (2, 3)
```

- In questo caso la tupla in `shape` ha due elementi
- I.e. numero di righe e numero di colonne

Creazione di Array

`numpy` fornisce funzioni per costruire particolari array

Per un array nullo si usa `zeros`

In [5]:
```
shape = (2, 3) # numero di righe e colonne
print(np.zeros(shape))
```

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

Per un array unitario si usa `ones`

In [6]:
```
print(np.ones(shape))
```

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


Creazione di Array

`numpy` fornisce funzioni per costruire particolari array

Per un array riempito con un valore a scelta si usa `full`

In [7]:
```
shape = (2, 3) # numero di righe e colonne
val = np.NaN
print(np.full(shape, np.NaN))
```

```
[[nan nan nan]
 [nan nan nan]]
```

- `NaN` sta per Not a Number
- È l'equivalente di un valore mancate in calcolo numerico

Per la matrice di identità si usa `eye`:

In [8]:
```
n = 3
print(np.eye(n))
```

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

Creazione di Array

`numpy` fornisce funzioni per costruire particolari array

Per un array di interi consecutivi si usa `arange`

In [9]:
```
x = np.arange(1, 10)
print(x)
```

```
[1 2 3 4 5 6 7 8 9]
```

Per un array di valori equispaziati si usa `linspace`:

In [10]:
```
start, stop, num = 0, 8, 10
x = np.linspace(start, stop, num)
print(x)
```

```
[0.         0.88888889 1.77777778 2.66666667 3.55555556 4.44444444
 5.33333333 6.22222222 7.11111111 8.        ]
```

- Il valore di default per `num` è 50


Tipo di un Array

Tutte gli elementi di un array devono essere dello stesso tipo

- Il tipo degli elementi è accessibile attraverso l'attribyto `dtype`

In [11]:
```
x = np.zeros(3)
x.dtype
```

Out[11]:
```
dtype('float64')
```

- Se si converte in array una lista con elementi eterogenei
- ... `numpy` cerca di tradurre gli elementi in un unico dipo

In [12]:
```
x = np.array([1, 2.3, True])
print(x, x.dtype)
```

```
[1.  2.3 1. ] float64
```


Operazioni su Array

Gli operatori di Python sono ridefiniti per gli array

In particolare, funzionano elemento per elemento

- Qualche esempio con gli operatori aritmetici

In [13]:
```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
print('x + y:', x + y)
print('x * y:', x * y)
print('x - y:', x - y)
print('x / y:', x / y)
print('y % 2:', y % 2)
```

```
x + y: [5 7 9]
x * y: [ 4 10 18]
x - y: [-3 -3 -3]
x / y: [0.25 0.4  0.5 ]
y % 2: [0 1 0]
```


Operazioni su Array

Gli operatori di Python sono ridefiniti per gli array

In particolare, funzionano elemento per elemento

- Qualche esempio con gli operatori di confronto

In [14]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
print('x <= y:', x <= y)
print('x == y:', x == y)
```

```
x <= y: [ True  True False]
x == y: [False  True False]
```

- Il risultato sono degli array di valori logici


Operazioni su Array

Gli operatori di Python sono ridefiniti per gli array

In particolare, funzionano elemento per elemento

- Gli operatori `&`, `|` e `~` non lavorano bit per bit
- ...Ma elemento per elemento

In [15]:
```
print('~(x <= y):', ~(x <= y))
print('(x <= y) | (x >= y):', (x <= y) | (x >= y))
print('(x <= y) & (x >= y):', (x <= y) & (x >= y))
```

```
~(x <= y): [False False  True]
(x <= y) | (x >= y): [ True  True  True]
(x <= y) & (x >= y): [False  True False]
```

- Occorre fare un po' di attenzione alle priorità
- E.g. `&` e `|` hanno una priorità più alta degli operatori di confronto
- Soluzione: usare le parentesi


Accesso ad Array

Per accedere ad un array si usa l'operatore di indicizzazione, i.e. `[]`

Per accedere ad un singolo elemento si usa una tupla come indice

In [16]:
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x[0, 2]) # riga 0, colonna 2
```

```
3
```

È anche possibile l'accesso mediante slice (come con le liste)

In [17]:
```
print(x[0, :]) # l'intera riga 0
print(x[:, 1]) # l'intera colonna 1
print(x[:2, :2]) # prime due righe e due colonne
```

```
[1 2 3]
[2 5]
[[1 2]
 [4 5]]
```

Si può usare uno slice per ciascuna dimensione

Accesso ad Array

Si può accedere con una collezione di indici

Si utilizza di solito nel caso di array mono-dimensionali

In [18]:
```
x = np.array([2, 4, 6, 8, 10, 12])
idx = [0, 2, 4]
print(x[idx]) # accesso agli indici 0, 2 e 4
```

```
[ 2  6 10]
```

- Prima si ottiene una collezione con gli indici desiderati
- ...Quindi la si passa come argomento all'operatore di indicizzazione
- I.e. tra le parentesi quadre `[]`

Il risultato è un array con gli elementi agli indici specificati


Accesso ad Array

Si può accedere utilizzando una "maschera" logica

- La maschera è un secondo array, con la stessa dimensione
- ...E contenente valori logici

In [19]:
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x)
mask = np.array([[True, True, False], [False, False, True]])
print(mask)
```

```
[[1 2 3]
 [4 5 6]]
[[ True  True False]
 [False False  True]]
```

- Usando la maschera come indice si ottiene un array monodimensionale
- ...Con gli elementi agli indici aventi `True` nella maschera

In [20]:
```
print(x[mask])
```

```
[1 2 6]
```

Accesso ad Array

Si può accedere utilizzando una "maschera" logica

Si usa di solito per recuperare gli elementi che soddisfano una data condizione

```
In [21]: x = np.array([[1, 2, 3], [4, 5, 6]])
         x[x % 2 == 0]

Out[21]: array([2, 4, 6])
```

In questo esempio:

- L'espressione `x % 2 == 0` restituisce una maschera logica
- ...Che viene usata per accedere all'array

Il risultato sono gli elementi con valore pari

Assegnamento con Array

Si possono assegnare elementi individuali in un array

...Esattamente come per le liste:

In [22]:
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x)
x[1, 1] = -1
print(x)
```

```
[[1 2 3]
 [4 5 6]]
[[ 1  2  3]
 [ 4 -1  6]]
```

Assegnamento con Array

Si possono assegnare intere sottoparti di un array

E.g. si può assegnare una colonna:

In [23]:
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x)
x[:, 1] = [-1, -1]
print(x)
```

```
[[1 2 3]
 [4 5 6]]
[[ 1 -1  3]
 [ 4 -1  6]]
```

...O una riga:

In [24]:
```
x[0, :] = [-1, -1, -1]
print(x)
```

```
[[-1 -1 -1]
 [ 4 -1  6]]
```

Assegnamento con Array

Si possono assegnare intere sottoparti di un array

- Se le dimensioni della porzione di array selezionata
- ...Sono diverse dalle dimensioni dell'oggetto assegnato
- `numpy` tenta di adattare il secondo al primo

Il caso più tipico è l'assegnamento di uno scalare:

In [25]:
```
x = np.array([[1, 2, 3], [4, 5, 6]])
print(x)
x[:2, :2] = -1
print(x)
```

```
[[1 2 3]
 [4 5 6]]
[[-1 -1  3]
 [-1 -1  6]]
```

- In questo caso tutti gli elementi selezionati
- ...vengono sostituiti con lo scalare

Funzioni e Metodi in `numpy`

`numpy` fornisce diverse funzioni per lavorare con array

Vediamo un po' di funzioni aritmetiche:

In [26]:
```
x = np.array([1, 2, 3, 4])
print(np.square(x)) # quadrato elemento per elemento
print(np.sqrt(x)) # radice quadrata elemento per elemento
print(np.exp(x)) # esponenziale elemento per elemento
print(np.log(x)) # logaritmo elemento per elemento
print(np.sin(x)) # seno elemento per elemento
print(np.cos(x)) # coseno elemento per elemento
```

```
[ 1  4  9 16]
[1.         1.41421356 1.73205081 2.        ]
[ 2.71828183  7.3890561  20.08553692 54.59815003]
[0.         0.69314718 1.09861229 1.38629436]
[ 0.84147098  0.90929743  0.14112001 -0.7568025 ]
[ 0.54030231 -0.41614684 -0.9899925  -0.65364362]
```

Funzioni e Metodi in `numpy`

`numpy` fornisce diverse funzioni per lavorare con array

Vediamo qualche di funzioni di aggregazione:

In [27]:
```
x = np.array([1, 2, 3, 4])
print(np.prod(x)) # prodotto degli elementi
print(np.sum(x)) # somma degli elementi
print(np.mean(x)) # media
print(np.std(x)) # deviazione standard
```

```
24
10
2.5
1.118033988749895
```

Funzioni e Metodi in `numpy`

`numpy` fornisce diverse funzioni per lavorare con array

Vediamo qualche di funzioni per lavorare con numeri pseudo casuali:

In [28]:
```
np.random.seed(42) # scelta del "seed"
shape = (4,)
print(np.random.random(shape)) # generazione di numeri casuali in [0,1)
print(np.random.randint(low=0, high=4, size=shape)) # generazione di numeri casuali interi
vals = [2, 4, 6, 8]
print(np.random.choice(vals, size=shape)) # elementi casuali da una co
```

```
[0.37454012 0.95071431 0.73199394 0.59865848]
[2 1 2 2]
[6 6 8 2]
```

- Le funzioni in questa categoria sono nel modulo `np.random`
- Vi sono altri moduli utili (al solito: vedere la documentazione!)

Funzioni e Metodi in `numpy`

Alcune funzioni sono disponibili anche come metodi

Vantaggi di `numpy`

`numpy` ci permette di ottenere codice più leggibile ed efficiente

E.g. supponiamo di dover sommare due sequenze di numeri

- Prima risolviamo il problema usando Python "nativo"

In [30]:
```
%%time
n = 20000000
a = [i for i in range(n)]
b = [i for i in range(n)]
c = [v1 + v2 for v1, v2 in zip(a, b)]
```

```
CPU times: user 1.58 s, sys: 476 ms, total: 2.05 s
Wall time: 2.05 s
```

- Il comando `%%time` stampa il tempo impiegato ad eseguire la cella


Vantaggi di `numpy`

`numpy` ci permette di ottenere codice più leggibile ed efficiente

E.g. supponiamo di dover sommare due sequenze di numeri

- Ora risolviamo il problema con `numpy`

In [31]:
```
%%time
n = 20000000
a = np.arange(n)
b = np.arange(n)
c = a + b
```

```
CPU times: user 346 ms, sys: 99 ms, total: 445 ms
Wall time: 444 ms
```

- La versione fatta con `numpy` è più leggibile
- ...E quasi 5 volte più veloce!