Programmazione Avanzata e parallela

Lezione 26

Luca Manzoni

Python ed array

Utilizzo di Numpy

- La libreria numpy
- Creazione di vettori
- Indicizzazione
- Operazioni su vettori
- Matrici
- 3D e dimensioni maggiori

Perché Numpy

Librerie per array

- Numpy è una libreria per array ad alte prestazioni
- Usata come fondamento di altre librerie (e.g., pandas)
- L’interfaccia è seguita (in parte) anche da altre librerie (e.g., PyTorch)
- Permette di passare i dati in modo efficiente ad altre librerie (e.g., Keras, Tensorflow)
- Rende “facile” il passaggio alla computazione su GPU

Numpy Array vs Liste Python

Questione di prestazioni

- Le liste Python sono più simili a tabelle hash:
  - Operazioni di append efficiente
  - Utilizzano più memoria del minimo necessario
- Gli array in numpy sono più simili a vettori in C:
  - Append può richiedere riallocazione
  - Contiguità in memoria dei dati

Numpy Array vs Liste Python

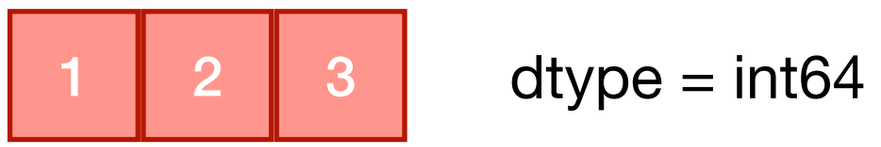
Questione di prestazioni

- Gli array in numpy sono più compatti
- Più veloci delle liste quando è possibile svolgere la stessa operazione su tutti gli elementi
- Più lenti se la loro dimensione deve essere continuamente modificata (e.g., append)
- Funzionano meglio se possono lavorare su tipi omogenei (e.g., tutti float o tutti interi) piuttosto che eterogenei

Creazione di array

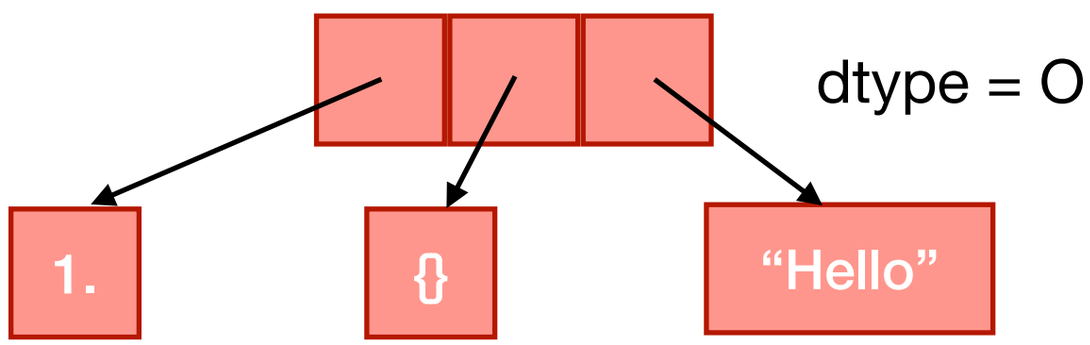
Valori di default

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




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


```
a = np.array([1.,{}, “hello”])
```



Numpy: tipi

Ed effetti sulle prestazioni

- In numpy vi è una gerarchia di tipi
- Generalmente è meglio avere tutti gli elementi dello stesso tipo per garantire buone prestazioni
- Tipi interi:
  - Con segno: int8, int16, int32, int64
  - Senza segno: uint8, uint16, uint32, uint64
- Tipi floating point: float16, float32, float64

Proprietà degli array

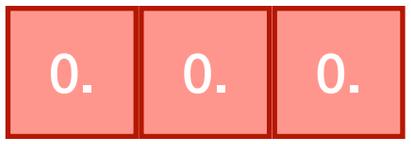
Questione di prestazioni

- Supponendo di avere un array numpy **a**:
  - **a.dtype** ritorna il tipo degli elementi dell'array
  - **a.shape** ritorna una tupla con le dimensioni dell'array (con un elemento nel caso di un vettore, due per le matrici, etc)
  - **a.size** ritorna il numero di elementi totali (e.g., una matrice 4×3 avrà 12 elementi e shape $(4,3)$)
  - Con **a.astype(nome_tipo)** possiamo convertire al tipo numpy indicato

Creazione di array

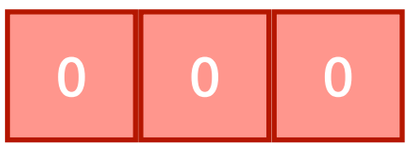
Valori di default

```
a = np.zeros(3)
```



dtype = float64

```
a = np.zeros(3, dtype=np.int16)
```



dtype = int16

```
a = np.ones(3)
```

dtype = float64

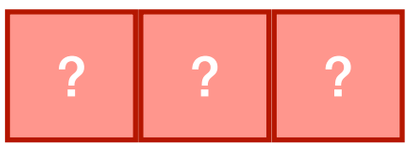
Creazione di array

Valori di default

```
a = np.full(3, 5.2)
```

dtype = float64

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



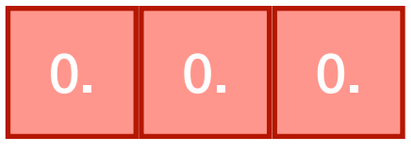
dtype = float64

La funzione non inizializza l’array!

Creazione di array

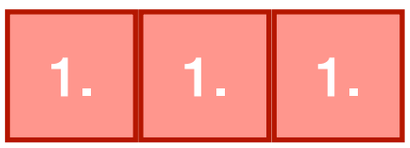
Copiare la forma

```
a = np.zeros(3)
```



dtype = float64

```
b = np.ones_like(a)
```



dtype = float64

Viene copiata la forma del vettore passato come argomento

Esistono anche **zeros_like**, **empty_like** e **full_like**

Creazione di array

Range

- Possiamo riempire un array usando i valori in un range:
  - np.arange(6) → array([0, 1, 2, 3, 4, 5])
  - np.arange(1,6) → array([1, 2, 3, 4, 5])
  - np.arange(1, 6, 2) → array([1, 3, 5])
  - np.linspace(0, 6, 4) → array([0., 2., 4., 6.])
- Attenzione che i normali problemi di precisione coi float si possono verificare quando si passano argomenti float ad **arange**

Creazione di array

Numeri casuali

- La “nuova modalità” per generare numeri in numpy prevede due passi:
  - Inizializzazione di un generatore di numeri (pseudo)casuali
  - Utilizzo del generatore per la creazione di array di valori casuali
- Perché usare la “nuova modalità”?
  - Meglio per il multi-threading
  - Più flessibile
  - Passa più test statistici per avere numeri “più casuali”

Creazione di array

Numeri casuali

- Inizializzazione del generatore:
  rng = np.random.default_rng()

- Uniforme (su interi) in $[a, b)$
  rng.integers(a, b, n)

- Se vogliamo gli estremi, quindi uniforme in $[a, b]$, possiamo aggiungere un argomento:
  rng.integers(a, b, n, endpoint=True)

Creazione di array

Numeri casuali

- Uniforme (su float) in $[a, b)$
  rng.uniform(a, b, n)

- Uniforme (su float) in $[0,1)$
  rng.random(n)

- Normale standard, ovvero $\mu = 0$ e $\sigma = 1$
  rng.standard_normal(n)

- Normale con $\mu = a$ e $\sigma = b$
  rng.normal(a, b, n)

Indicizzazione

E viste

- Possiamo usare un array numpy come una lista python per l'indicizzazione:
  - **a[i]**. Singolo elemento
  - **a[i:j]**. Range di indici tra i (incluso) e j (escluso)
  - **a[start:end:step]**. Range di indici (con step)
  - Se non presente si assume tutto il range:
    - **a[::2]** per esempio prende solo gli indici pari dell'array

Indicizzazione

E viste

- Contrariamente alle liste python non viene creata una copia, ma solo una vista (view)
- Potete pensare a una vista come a un puntatore all'array originale con un nuovo insieme di indici
- Se modifichiamo la vista modifichiamo anche l'originale!
- Solo alcuni metodi ci permettono di tornare una copia dell'array
  - e.g., col metodo **.copy()**

Indicizzazione

Fancy indexing

- Se passiamo un array di indici ci viene ritornata una *copia* dell'array originale solo per gli indici indicati:
  - **A = np.array([6, 7, 3, 4, 8])**
  - **B = A[[0, 2, 3]]** è **[A[0], A[2], A[3]]**
- Se passiamo un array di Booleani della stessa lunghezza dell'array originale stiamo dando una "maschera di selezione":
  - **A[[True, True, False, False, True]]** è **[6, 7, 8]**

Indicizzazione

Condizione Booleane

- Possiamo esprimere una condizione su un array e ci viene ritornato un vettore di Booleani:
  - **A = np.array([6, 7, 3, 4, 8])**
    B = A > 4
  - **B** è **[True, True, False, False, True]**
- Quindi possiamo usare queste condizioni per indicizzare gli array:
  - **B = A[A > 4]** per ottenere **[6, 7, 8]**

Indicizzazione

Condizione Booleane

- Possiamo combinare array di Booleani con **&** (and), **|** (or), **^** (xor) e **~** (not)

- Per ottenere un valore unico da una condizione possiamo chiederci se tutti i valori rispettano una condizione o almeno uno la rispetta:

  - **np.any(condizione)** è la forma esistenziale (vero se esiste un valore che rispetta la condizione)

  - **np.all(condizione)** è la forma universale (vero se tutti i valori la rispettano)

Clipping e where

Modificare con condizioni

- Una cosa comune è limitare il valore degli elementi in un array:
  - **np.clip(a, min, max)**
    se un elemento è inferiore a min viene messo a min, se è superiore a max viene messo a max
- Per ottenere gli indici dove una condizione è verificata possiamo usare:
  - **np.where(A > 5)**
  - Viene ritornata una tupla (nel caso 1D con un solo elemento) con un array di indici in cui la condizione è verificata

Funzioni utili

- Diverse funzioni sono già implementati in numpy:
  - **np.sqrt(A)**, $\sqrt{x}$
    np.exp(A), e^x
    np.log(A), $\ln(x)$
  - **np.sin(A)**, **np.arcsin(A)**, **np.arctan(A)**, …
  - **np.floor(A)**, **np.ceil(A)**, **np.round(A)**
  - **np.dot(A, B)**, prodotto interno. Esprimibile anche come A @ B
  - **np.sort(A)**, ritorna una copia ordinata di A

Metodi utili

- Diverse *metodi* sono già implementati in numpy:
  - **A.sum()**
  - **A.mean(), A.var(), A.std()**
  - **A.min()**
  - **A.max()**
  - **A.argmax()**
  - **A.argmin()**