

Autori: Domenico Lembo, Giuseppe Santucci and Marco Schaerf

Dipartimento di Ingegneria informatica, automatica e gestionale



This notebook is distributed with license Creative Commons CC BY-NC-SA

La libreria NumPy

1. NumPy
2. Il tipo array di NumPy
3. Creazione di un array in NumPy
4. Attributi della classe ndarray (array)
5. Indicizzazione, slicing e iterazione
6. Algebra lineare in NumPy
7. Esercizi su matrici come NumPy array
8. Progetto finale

Numpy

Questa lezione è basata sul tutorial [Quickstart](#) del sito ufficiale di [NumPy](#) (in inglese). Per eseguire questo notebook dovete installare sul vostro computer il modulo numpy, istruzioni dettagliate sono disponibili [qui](#), comunque per la maggior parte delle installazioni dovete solo aprire una shell di comandi e dare il comando: `pip3 install numpy` ; per installarlo nell'ambiente notebook dovete dare il comando `%pip3 install numpy` .

Perchè? Ottimizzazione!

NumPy fa parte di un pacchetto di moduli Python per il calcolo scientifico, noi abbiamo già visto, brevemente, anche [Matplotlib](#).

```
In [ ]: %pip3 install numpy
```

Il tipo array di NumPy

L'oggetto principale di NumPy è l'array multidimensionale **omogeneo**. È una tabella di elementi (solitamente numeri), tutti dello stesso tipo, indicizzati da una tupla di numeri interi non negativi. In NumPy le dimensioni sono chiamate assi.

Ad esempio, le coordinate di un punto nello spazio [1, 2, 1] sono rappresentate su un asse. L'asse contiene 3 elementi, quindi diciamo che ha una lunghezza di 3.

Nell'esempio mostrato di seguito, l'array ha 2 assi. Il primo asse ha una lunghezza di 2, il secondo asse ha una lunghezza di 3. Corrisponda una shape di pandas (2,3), ovvero 2 righe per tre colonne.

```
[[1., 0., 0.], [0., 1., 2.]]
```

N.B. per evitare qualunque confusione useremo i due termini:

- asse/i: coordinata/e dell'array
- lunghezza asse (o dimensione asse): numero di elementi presenti in un asse

Un array 3D (tridimensionale) ha tre assi :-). La dimensione degli assi ha una lunghezza che dipende dal numero di elementi inseriti.

La classe di array di NumPy si chiama `ndarray`. È anche conosciuto con l'alias `array`. Notate che `numpy.array` non è uguale alla classe Standard Python Library `array.array`, che gestisce solo array 1D e offre molte meno funzionalità.

- `ndarray.ndim`: numero degli assi
- `ndarray.shape`: analoga a quanto visto per pandas, `shape` è una tupla di `ndim` interi che contiene la lunghezza di ogni asse
- `ndarray.size`: numero complessivo di elementi dell'array

Un esempio:

```
In [ ]: import numpy as np

#Creiamo, ad esempio, un array a due assi 2x3 (cioè con 2 righe e 3 colonne)
a = np.array([[1., 0., 0.], [0., 1., 2.]])

print('a=',a)

print('numero assi=',a.ndim)
print('shape=',a.shape)
print('numero elementi=',a.size)
```

Per accedere a un elemento di indici `i` e `j` di un array bidimensionale `m` si può usare la notazione delle liste di liste (cioè `m[i][j]`) oppure la notazione semplificata `m[i,j]`. Ovviamente, questo è valido anche per array di dimensione superiore a 2.

Un esempio:

```
In [ ]: import numpy as np

#Creiamo, ad esempio, un array 2x3 (cioè con 2 righe e 3 colonne)
a = np.array([[1., 0., 0.], [0., 1., 2.]])
print('a=',a)
print()
#Stampiamo uno specifico elemento, ad esempio,
#quello sulla riga 0 colonna 1
print('elemento riga 0 colonna 1',a[0][1]) #notazione standard per liste di
print('elemento riga 0 colonna 1',a[0,1]) #notazione semplificata di NumPy
# stampa tramite doppio ciclo

print()
for i in range(a.shape[0]):      # righe
    for j in range(a.shape[1]):  # colonne
```

```
print(a[i,j],end='\t')
print()
```

Creazione di un array in NumPy

Ci sono molti modi per creare un array in NumPy, si può direttamente creare un array fornendo tutti i dati, come visto sopra, si può trasformare una lista in array oppure si possono usare le numerose funzioni di inizializzazione presenti in NumPy. **Notate che le dimensioni degli assi dell'array vanno fornite come tuple**, cioè scritte tra parentesi.

La funzione `print()` applicata ad un oggetto di tipo array lo stampa automaticamente per righe, se è tridimensionale stampa un piano 2D per volta. Vediamo degli esempi:

```
In [ ]: # Crea un array 2D di uno con 3 righe e 4 colonne
print(np.ones((3,4)))
```

Si noti esplicitamente che nell'esempio sopra, la tupla (3,4) è la shape dell'array e denota contemporaneamente:

- il numero degli assi (lunghezza della tupla)
- la lunghezza di ciascun asse (valori della tupla)

La stampa di un array con 3 o più assi richiede stampe ripetute:

```
In [ ]: # Crea un array 3D di zeri con shape 2x3x4 ed ogni elemento di tipo intero (
print(np.zeros((2,3,4),dtype=np.int16)) #notate la stampa fatta di 2 matrici
```

```
In [ ]: # Crea un array 2D di valori casuali (random)
print(np.random.random((2,2)))
```

```
In [ ]: # Crea un array 2D vuoto
print(np.empty((3,2)))
```

```
In [ ]: # Crea un array 2D pieno con il valore 7 (tutti gli elementi valgono 7)
print(np.full((2,2),7))
```

```
In [ ]: # Crea un array 1D con i valori da 10 a 50 (escluso) con passo 5
print(np.arange(10,50,5))
```

```
In [ ]: # Crea un array 1D con 9 valori uniformemente spazati tra 0 e 2 (inclusi)
print(np.linspace(0,2,9))
```

Rappresentazione interna degli array ed i metodi `reshape()` e `flatten()`

NumPy rappresenta in memoria gli array usando zone contigue di memoria. Di fatto, l'array viene rappresentato sempre **come un array monodimensionale**, dove le righe vengono scritte una dopo l'altra. Un array di dimensioni (3,4) e un array di dimensioni (2,3,2) sono rappresentati in memoria nella stessa maniera: un array monodimensionale di 12 elementi (12). Per questo motivo, è possibile cambiare facilmente **la shape** lasciando invariato il numero degli elementi, ovvero la **size**, usando il metodo

`reshape()` . Se vogliamo trasformare un array nella sua versione monodimensionale possiamo usare il metodo `flatten()` . Vediamo degli esempi:

```
In [ ]: a = np.random.random((3,4)) #12 elementi, shape (3,4)
        print(a)
```

```
In [ ]: b = a.reshape((2,3,2)) #12 elementi, shape (2,3,2)
        print(b)
```

```
In [ ]: #c = a.flatten() #oppure c = a.reshape((12))
        print(c)
```

Attributi, operatori e funzioni della classe ndarray (array)

Oltre agli attributi **ndim**, **shape**, e **size**, già introdotti in precedenza, la classe ndarray offre:

- **dtype**: il tipo di elementi nella matrice. Si può creare o specificare i tipi usando i tipi standard di Python. Inoltre NumPy fornisce tipi propri: `numpy.int32`, `numpy.int16` e `numpy.float64` sono alcuni esempi.
- **itemsize**: la dimensione in byte di ciascun elemento dell'array. Ad esempio, un array di elementi di tipo `float64` ha `itemsize 8` (= 64/8), mentre uno di tipo `complex32` ha `itemsize 4` (= 32/8). È equivalente a `ndarray.dtype.itemsize`.
- `+, -, *, **, <, =, >` : operatori che operano su **tutta** la matrice

```
In [ ]: a = np.array ([2,3,4])
        print('dtype=',a.dtype)
        b = np.array ([1.2, 3.5, 5.1])
        print('dtype=',b.dtype)
        c = np.array([[1,2], [3,4]], dtype = complex)
        print(c)
        print('dtype=',c.dtype)
```

Operazioni di base

Gli operatori aritmetici sugli array si applicano a tutti gli elementi. Di regola, un nuovo array viene creato e riempito con il risultato.

```
In [ ]: a = np.array ([20,30,40,50])
        b = np.array ([4,4,4,4])
        c = a-b
        print(c)
        print()
        print('a*7=',a*7)
        print('a*b=',a*b)
```

```
In [ ]: print(b**2)
        print(np.sin(a))
```

```
In [ ]: print(a)
        print(a < 35)
```

N.B. A differenza di molti linguaggi a matrice, l'operatore del prodotto `*` opera elemento per elemento negli array NumPy, cioè moltiplica gli elementi nella stessa posizione dei 2 arrays, che devono avere le stesse dimensioni. Il prodotto tra matrici può essere eseguito utilizzando l'operatore `@` (in python ≥ 3.5) o il metodo `dot()`

```
In [ ]: A = np.array([[1,1],
                    [0,1]])
        B = np.array([[2,0],
                    [3,4]])
        print(A * B) # è il prodotto elemento per elemento
        print(A @ B) # è il prodotto tra matrici
        print(A.dot(B)) # un altro prodotto tra matrici

# A @ B =      [1,1]   [2,0]      [1*2+1*3, 1*0+1*4]      [5,4]
#              [0,1]   [3,4]      [0*2+1*3, 0*0+1*4]      [3,4]
```

Alcune operazioni, come `+=` e `*=`, agiscono per modificare un array esistente anziché crearne uno nuovo.

```
In [ ]: a = np.ones((2,3), dtype = int)
        print(a)
        a *= 3
        print(a)
```

Molte operazioni unarie, come calcolare la somma di tutti gli elementi dell'array, sono implementate come metodi della classe ndarray.

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

        print('sum=', a.sum())
        print('min=', a.min())
        print('max=', a.max())
```

```
In [ ]: a = np.random.random((2,3))
        print('a=', a)
        b = np.random.random(10)
        print('b=', b)

        # per trovare l'indice in cui si trova il massimo (od il minimo) si può usare
        # la funzione (NON METODO) argmax (argmin). Se l'array è monodimensionale restituisce
        # l'indice, altrimenti restituisce l'indice dell'array flat (appiattito) in cui si trova
        # le righe sono messe di seguito. In caso di più elementi pari al massimo (o minimo)
        # argmax restituisce l'indice più piccolo fra tutti quelli

        print('argmax_b', np.argmax(b))
        print('argmin_b', np.argmin(b))

        print('argmax_a', np.argmax(a)) #di [0.96961376 0.9839157 0.36553127 0.29110
        print('argmin_b', np.argmin(b)) #di [0.96961376 0.9839157 0.36553127 0.29110
```

Per impostazione predefinita, queste operazioni si applicano all'array come se fosse un elenco di numeri, indipendentemente dalla sua forma. Tuttavia, specificando il parametro `axis` è possibile applicare un'operazione lungo l'asse specificato di un array:

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

# somma di ogni colonna, restituisce un array con una dimensione in meno
print('somma delle righe=', a.sum(axis = 0))
print()
print('somma delle colonne=', a.sum(axis = 1))

In [ ]: # min di ogni riga, restituisce un array con una dimensione in meno
print(b.min(axis = 1))

In [ ]: # somma cumulativa lungo ogni riga
print(b.cumsum(axis = 1))
```

Indicizzazione, slicing e iterazione

Le matrici multidimensionali possono essere indicizzate, suddivise e ripetute, in modo simile alle liste e ad altre sequenze di Python, ma anche **contemporaneamente** su più assi usando la notazione semplificata. Attraverso lo slicing si può, ad esempio, estrarre una colonna della matrice od anche le colonne dispari. Vediamo alcuni esempi.

```
In [ ]: b = np.arange(9) #crea array 1D di 12 elementi
print(b)

In [ ]: print(b[2])      # stampa l'elemento di indice 2
print(b[1:8:2]) # da indice 1 a indice 8 (escluso) passo 2

In [ ]: b = np.arange(9) #crea array 1D di 12 elementi
d=b[::-1]

print(d.shape)
print(d)

In [ ]: # Consideriamo ora un caso a 2D

c = np.arange(20).reshape(4,5) #crea array 4x5 da un array 1D di 20 elementi
print(c)

In [ ]: print('l'elemento in posizione 2,3 è',c[2,3]) # stampa l'elemento di indice
print(c[1:4:2,:]) #seleziona le righe 1 e 3 e tutte le colonne

In [ ]: print(c)
print()
print(c[0:2,0:3]) #seleziona le righe 0 e 1 e le colonne 0, 1, e 2

In [ ]: print(c)
print()
print(c[1:3,1:4]) #seleziona le righe 1 e 2 e le colonne 1, 2 e 3

In [ ]: print(c)
print()
print(c[:,::2]) #seleziona le colonne pari
```

```
In [ ]: print(c)
        print()
        d=c[:,2:3] #seleziona la colonna 2
        print('la dimensione dell\'array è', d.shape)
        print('il suo tipo è',d.dtype)
        print(d)
```

```
In [ ]: # Vediamo ora degli esempi di iterazioni con la print()
        b = np.arange(4) #crea array 1D di 12 elementi
        print('b=',b)
        print()
        for i in b:
            print(i ** 2) # calcola il quadrato degli elementi di b e stampa ciascun
```

```
In [ ]: c = np.arange(6).reshape(2,3) #crea array 2x3 da un array 1D di 6 elementi
        print(c)
        print()
        print('c[0]=' ,c[0])
        print()
        for i in c:
            print(i ** 2) # calcola, riga per riga, il quadrato degli elementi
```

```
In [ ]: for i in range(c.shape[0]):    # stampa ciascun valore di c su una riga diversa
        for j in range(c.shape[1]):
            print(c[i,j])
```

```
In [ ]: for riga in c:    # versione alternativa che itera sugli elementi e non sugli indici
        for elem in riga:
            print(elem)
```

Esercizi su matrici come NumPy array

1. Esercizio: Ricerca dell'elemento massimo: Funzione che prende in input una matrice e restituisce il valore massimo presente; assume che la matrice non sia vuota.
2. Esercizio: Ricerca della riga a somma massima: Funzione che prende in input una matrice e restituisce l'indice della riga di somma massima presente; assume che la matrice non sia vuota e che nel caso ci siano più righe di somma massima la funzione restituisca la riga con indice minore.
3. Esercizio: Somma di matrici (stessa dimensione).
4. Esercizio: Prodotto di matrici (dimensioni compatibili).

```
In [ ]: # Creiamo 2 matrici 4x3 a e b ed una matrice c 3x4, tutte con numeri random
        a = np.random.random((4,3))
        b = np.random.random((4,3))
        c = np.random.random((3,4))
        print(a)
        print()
        print(b)
        print()
        print(c)
        print()
```

```
In [ ]: # Esercizio 1: basta usare la funzione np.max()
        print(np.max(a))
```

```
In [ ]: # Esercizio 2: Scansiona le righe e trova il massimo
sommari_ghe = np.sum(a,axis=1) #calcolo il vettore somma delle righe
print(sommari_ghe)
print(np.argmax(sommari_ghe)) #restituisco l'indice dell'elemento massimo
```

```
In [ ]: # Esercizio 3: basta usare l'operatore +
print(a+b) # Il risultato ha le stesse dimensioni di a e b
```

```
In [ ]: # Esercizio 4: basta usare l'operatore @
print(a@c) # Poiché a è 4x3 e c 3x4 il risultato è un array 4x4
```

Esercizio: calcolo del più grande punto di sella

Scrivere una funzione che prende in input un array bidimensionale **a** e restituisce (se esiste) la tupla (i,j) della posizione del più grande punto di sella presente nell'array. Se non esiste deve restituire la tupla (-1,-1). Si definisce punto di sella (i,j) una posizione tale che $a[i,j]$ è il minimo valore della riga i e il massimo valore della colonna j o viceversa. Assumete, per semplicità, che ogni riga e ogni colonna abbia uno e un solo massimo e minimo.

```
In [ ]: import numpy as np

def maxPuntoSella(a):
    ris = (-1,-1) # per ora non ho trovato punti di sella
    minrighe = np.argmin(a,axis=1) # Trova gli indici dei minimi per riga e
    maxrighe = np.argmax(a,axis=1) # Trova gli indici dei massimi per riga
    mincol = np.argmin(a,axis=0) # Trova gli indici dei minimi per colonna
    maxcol = np.argmax(a,axis=0) # Trova gli indici dei massimi per colonna
    # un punto di sella (i,j) minimo sulle righe e massimo sulle colonne ha
    # j = minrighe[i] e i = maxcol[j], la colonna j è la minima sulla riga i
    # è la massima nella colonna j. per l'altro tipo di punto di sella vale
    for i in range(a.shape[0]):
        j = minrighe[i] # cerco punti di sella minriga-maxcol
        if i == maxcol[j]: # Trovato un punto di sella minriga-maxcol
            if ris == (-1,-1): # primo punto di sella trovato
                ris = (i,j)
            elif a[i,j] > a[ris]: # Trovato un nuovo punto di sella, più grande
                ris = (i,j)
        j = maxrighe[i] # cerco punti di sella maxriga-mincol
        if i == mincol[j]: # Trovato un punto di sella maxriga-mincol
            if ris == (-1,-1): # primo punto di sella trovato
                ris = (i,j)
            elif a[i,j] > a[ris]: # Trovato un nuovo punto di sella, più grande
                ris = (i,j)
    return ris
```

```
In [ ]: ps=(-1,-1)
while ps==(-1,-1):
    a=np.random.random((5,5))
    ps=maxPuntoSella(a)
print(a,maxPuntoSella(a))
```

Altre funzioni e librerie

Python ha un numero molto elevato di altri moduli e librerie predefinite che sono a disposizione dei programmatori, quali ad esempio [SciPy](#) per il calcolo scientifico, [Pandas](#)

per l'analisi dei dati, [Keras](#), [TensorFlow](#) e [PyTorch](#) per il cosiddetto *deep learning* e molte altre. Qui ora mostriamo solo alcune funzionalità della libreria *IPython.display* che saranno utili per lo sviluppo del progetto finale che descriveremo a breve.

Caricare, visualizzare e cancellare immagini

La libreria *IPython.display* ci mette a disposizione 3 funzioni per le immagini che ci saranno utili, cioè le funzioni *Image*, *display* e *clear_output*. Il loro comportamento è il seguente:

- La funzione `Image(url)` carica un'immagine da un indirizzo web (url) o e restituisce un oggetto di tipo *image*
- La funzione `display(image)` visualizza sullo schermo un oggetto di tipo *image*
- la funzione `clear_output()` cancella quanto visualizzato dalla cella.

Vediamo un semplice esempio:

```
In [ ]: from IPython.display import display, Image, clear_output

url1 = "https://upload.wikimedia.org/wikipedia/commons/thumb/c/c9/Interno_de
url2 = "https://upload.wikimedia.org/wikipedia/commons/thumb/d/d8/Colosseum_

image1 = Image(url1) # carica l'immagine alla url1 nell'oggetto image1
image2 = Image(url2) # carica l'immagine alla url2 nell'oggetto image2
image3=Image(open('Eiffel.png',"rb").read())
display(image1) # visualizza image1
input() # il programma aspetta un invio ('enter') prima di andare avanti
clear_output() # cancella output
display(image2) # visualizza image1
input() # il programma aspetta un invio ('enter') prima di andare avanti
clear_output() # cancella output
display(image3) # visualizza image2
input() # il programma aspetta un invio ('enter') prima di andare avanti
clear_output() # cancella output e termina
```

Progetto finale

Sviluppiamo insieme un progetto finale che sfrutta quello che abbiamo imparato nel corso. Il progetto consiste nello sviluppare un piccolo gioco che propone immagini e chiede di indovinare la città e l'attrazione rappresentata nella foto. Il gioco lo sviluppiamo nel prossimo (e ultimo) notebook di questo corso.