UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

AImage Lab

Laurea in Ingegneria Informatica - Sede di Mantova
a.a. 2020-2021

Introduzione a Numpy

Lorenzo Baraldi

Numpy

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Numpy è la libreria di riferimento per il calcolo scientifico in Python
- Mette a disposizione un oggetto array multi-dimensionale ad elevate prestazioni, e le funzioni per operare con tali oggetti.

Arrays

- Un array numpy è una griglia di valori n-dimensionale (**tensore**), tutti dello stesso tipo, indicizzata da una tupla di numeri interi non negativi. Il numero di dimensioni è detto **rango** dell'array; mentre la sua **shape** è una tupla di interi che indica la dimensione del tensore lungo ogni dimensione.
- Possiamo inizializzare gli array numpy usando liste Python nidificate e accedere agli elementi utilizzando le parentesi quadre:

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

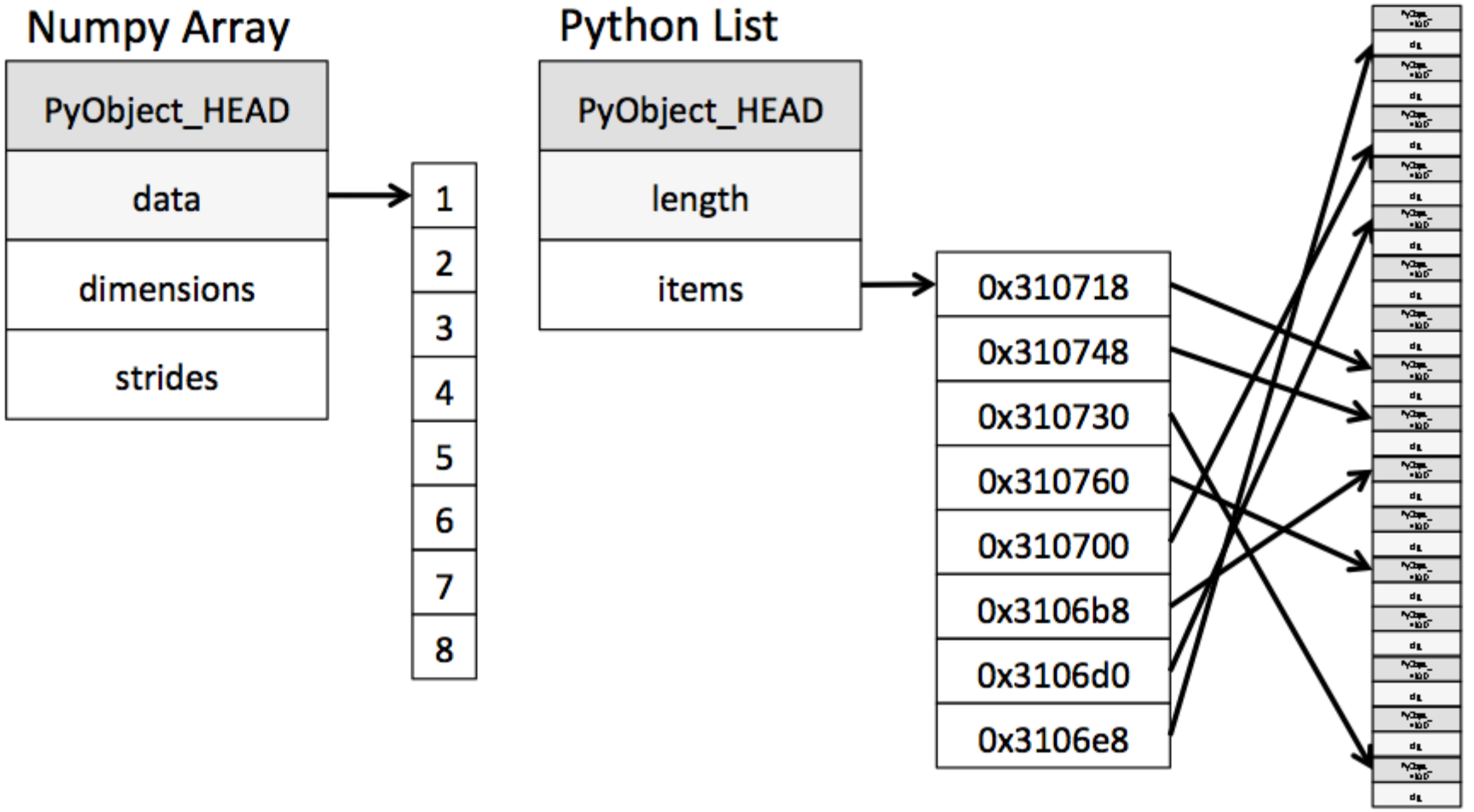
ARRAYS

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Numpy fornisce anche molte funzioni per creare gli array:

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

Python list vs Numpy ndarray



Quindi, np.ndarray è *più efficiente*!

Array indexing

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- **Slicing:** Come le liste Python, gli array numpy possono essere indicizzati (indexing). Poiché gli array numpy possono essere multidimensionali, è necessario specificare una sezione per ogni dimensione della matrice:

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

Array indexing

- È inoltre possibile combinare l'indicizzazione intera con l'indicizzazione a sezioni. Tuttavia, in questo modo si produrrà una matrice di rango inferiore rispetto all'array originale.

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

ARRAY INDEXING

- Integer array indexing

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

ARRAY INDEXING

- **Boolean array indexing:** Il boolean indexing consente di selezionare elementi arbitrari di un array. Spesso questo tipo di indicizzazione viene utilizzata per selezionare gli elementi di una matrice che soddisfano alcune condizioni. Ad esempio:

```
import numpy as np
a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2)    # Find the elements of a that are bigger than 2;
                      # this returns a numpy array of Booleans of the same
                      # shape as a, where each slot of bool_idx tells
                      # whether that element of a is > 2.
print(bool_idx)       # Prints "[[False False]
                      #          [ True  True]
                      #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

DATATYPES

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Ogni numpy array è una griglia di elementi dello stesso tipo. Numpy fornisce un ampio set di tipi di dati numerici che è possibile utilizzare per costruire gli array. Numpy tenta di indovinare un tipo di dati quando si crea un array, ma le funzioni che costruiscono gli array in genere includono anche un argomento facoltativo per specificare in modo esplicito il tipo di dati.

```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

ARRAY MATH

- Le funzioni matematiche di base operano in modo element-wise sugli array e sono disponibili sia tramite operator overloading che come funzioni nel modulo numpy:

```
import numpy as np
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

```
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

ARRAY MATH

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- L'operatore * è la moltiplicazione element-wise, non la moltiplicazione tra matrici. Usiamo invece la funzione dot per calcolare l'inner product tra vettori, per moltiplicare un vettore e una matrice e per moltiplicare le matrici.

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

ARRAY MATH

- Numpy fornisce molte funzioni utili per l'esecuzione di calcoli su array; uno dei più utili è somma:

```
import numpy as np
x = np.array([[1,2],[3,4]])
print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"

x = np.array([[1,2,0],[3,4,0]])
```

ARRAY RESHAPING

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Oltre a calcolare le funzioni matematiche utilizzando array, spesso abbiamo bisogno di cambiare la shape di un array. L'esempio più semplice di questo tipo di operazione è la matrice trasposta; per trasporre una matrice, è sufficiente utilizzare l'attributo T di un oggetto array:

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)     # Prints "[[1 2]
             #          [3 4]]"
print(x.T)   # Prints "[[1 3]
             #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)     # Prints "[1 2 3]"
print(v.T)   # Prints "[1 2 3]"
```

BROADCASTING

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Il Broadcasting è un potente meccanismo che consente a numpy di lavorare con matrici di forme diverse quando si eseguono operazioni aritmetiche. Spesso abbiamo un array più piccolo e un array più grande, e vogliamo usare l'array più piccolo più volte per eseguire alcune operazioni sull'array più grande.
- Si supponga, ad esempio, di voler aggiungere un vettore costante a ogni riga di una matrice. Potremmo farlo così:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

BROADCASTING

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Questo funziona; tuttavia, quando la matrice è molto grande, il calcolo di un ciclo in Python potrebbe essere lento. Si noti che l'aggiunta del vettore a ogni riga della matrice equivale a formare una matrice che «impila» più copie di del vettore verticalmente, e poi eseguire la somma element-wise delle due matrici. Potremmo implementare questo approccio in questo modo:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

BROADCASTING

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Il broadcasting ci permette di eseguire questo calcolo senza creare effettivamente più copie di v. Si consideri questa versione, che utilizza il broadcasting:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

- L'istruzione y = x + v funziona anche se x ha shape (4, 3) e v ha shape (3,) grazie al broadcasting; questa istruzione funziona come se v avesse effettivamente la shape (4, 3), dove ogni riga è una copia di v, e la somma fosse stata eseguita element-wise.

BROADCASTING

- Il broadcasting tra due array segue queste regole:

1. Se i due array non hanno lo stesso rango, Numpy antepone alla shape dell'array di rango inferiore degli 1 fino a quando entrambe le shape hanno la stessa lunghezza.
2. I due array così ottenuti sono detti *compatibili lungo una dimensione* se hanno lo stesso numero di elementi su quell'asse o se uno dei due array ha lunghezza 1 su tale asse.
3. Si può applicare il broadcasting tra due array solo se sono compatibili lungo tutte le loro dimensioni.
4. Dopo il broadcasting, ogni array si comporta come se avesse shape uguale al massimo element-wsie delle shape dei due array di input.
5. In qualsiasi dimensione in cui uno dei due array aveva lunghezza 1 e l'altro array aveva lunghezza maggiore di 1, il primo array si comporta come se fosse stata copiato lungo tale dimensione.

a (3, 4) ← rango 2

b (1, 4,) ← rango 1

(3, 4)

c c ➔ COMPATIBILE, si può fare il broadcasting

➔ a+b non mi tira errore

a (3, 4, 10, 32, 64) ← rango 5

b (1, 1, 1, 1, 4,) ← rango 1

c c c c nc ➔ NON COMPATIBILE, NON si può fare il broadcasting

a+b ➔ mi tira errore

Lavorare con le immagini

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Le immagini sono di solito memorizzate in formati compressi (JPG, PNG, BMP, ...). La compressione può essere lossy (con perdita di informazione) o senza perdita (come nel caso di PNG).
- Per leggere un'immagine, abbiamo bisogno di un decodificatore in grado di leggere il formato di input e restituire i valori dei pixel.
- Per il resto del corso, supporremo sempre che le immagini in scala di grigi siano rappresentate come tensori con shape (H, W) e che le immagini a colori siano rappresentate come tensori (3, H, W), dove i canali di colore (primo asse) saranno in ordine BGR o RGB.
- I pixel sono in genere rappresentati con interi senza segno a 8 bit (aka np.uint8 in Numpy), anche se possono essere convertiti in altri tipi di dati per l'elaborazione.

Aritmetica intera e saturata

- È necessario prestare molta attenzione quando si utilizzano gli interi a 8 bit e in particolare ai loro limiti numerici (0-255) e al tipo di dati restituito di ogni operazione.
- Inoltre, dobbiamo sempre assicurarci che il risultato dell'elaborazione rientri nei limiti numerici: se una funzione tenta di portare un pixel a 256, il suo valore di ritorno deve essere bloccato a 255; se una funzione restituisce un valore negativo per un pixel, il valore viene bloccato a 0.

Verifica delle soluzioni

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

- Gli esercizi in genere richiederanno di scrivere una funzione o una classe che implementa un operatore di image processing.
- Per controllare la correttezza delle soluzioni, useremo un valutatore on-line che controlla in automatico il codice che inviate.