

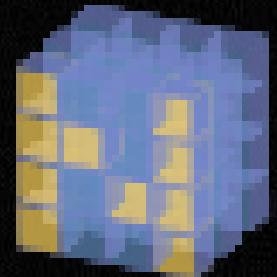
NumPy

17634 | VISIONE ARTIFICIALE



NumPy

- ▶ NumPy (Numerical Python): package fondamentale per il **calcolo scientifico** in Python.
 - Consente di lavorare in modo efficiente su grandi quantità di dati memorizzati come **vettori** e **matrici**.
- ▶ Python fornisce:
 - Oggetti numerici di alto livello (int, float, ...)
 - Comode strutture dati: tuple, liste, dizionari
- ▶ Tuttavia gestire grossi moli di dati direttamente in Python risulterebbe poco efficiente
- ▶ NumPy arricchisce Python con:
 - Un **array multi-dimensionale** efficiente e con potenti funzionalità
 - Sofisticate tecniche per operare su tale struttura dati
 - Utili funzioni matematiche di base (algebra lineare, FFT, numeri random, ...)

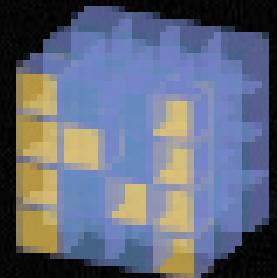


NumPy



Vantaggi di NumPy

- ▶ Funzioni e operatori agiscono su interi vettori: raramente si ricorre a loop esplicativi, tipicamente poco efficienti
- ▶ L'accesso agli elementi degli array è simile alle liste in Python (parentesi quadre, slicing)
- ▶ Codice scritto in C e altamente ottimizzato
- ▶ Gli algoritmi utilizzati sono ben testati e progettati per fornire buone prestazioni
- ▶ Gli array contengono elementi tutti dello stesso tipo (es. tutti interi a 32 bit) memorizzati in modo contiguo: un approccio meno flessibile rispetto alle strutture dati Python ma molto più efficiente
- ▶ Le operazioni di I/O di array sono significativamente più veloci



NumPy

NumPy e OpenCV

- ▶ OpenCV-Python, il modulo che consente di utilizzare OpenCV da Python **utilizza NumPy** per memorizzare e operare sulla maggior parte dei dati.
- ▶ In OpenCV le immagini con più canali sono semplicemente rappresentate come array tridimensionali. Utilizzando NumPy per memorizzarle, l'accesso ai pixel può essere effettuato in modo comodo ed efficiente (indexing, slicing, ...)
- ▶ Altre strutture dati utilizzate in OpenCV (es. filtri, matrici di trasformazione, vettori di caratteristiche) sono array NumPy e questo semplifica lo sviluppo del codice e l'interazione con i tanti altri package software basati su NumPy.



Utilizzare NumPy

► Installazione:

- Il modulo NumPy è già incluso nelle principali distribuzioni di software scientifico Python (es. Anaconda)
- Altre informazioni si possono trovare sul sito ufficiale: <https://numpy.org>

► Importazione:

- È sufficiente importare il modulo `numpy`
- È consuetudine importarlo con il nome locale `np`

```
import numpy as np  
  
print(np.__version__)
```



La classe ndarray

- ▶ La classe principale in NumPy: tutti gli array NumPy sono oggetti di questo tipo
- ▶ Implementa un **array multidimensionale omogeneo** (tutti gli elementi hanno lo stesso tipo, di solito numerico)
- ▶ Alcuni importanti **attributi** di ogni oggetto ndarray:
 - **ndim** - Il numero di dimensioni, chiamati anche assi (axes)
 - **shape** - Una tupla di interi che indica il numero di elementi lungo ciascuna dimensione
 - **size** - Il numero totale di elementi dell'array
 - **dtype** - Il tipo degli elementi
 - **itemsize** - La dimensione (in byte) di ogni elemento
 - **data** - Il buffer contenente gli elementi (normalmente non serve: si accede agli elementi con le parentesi quadre)



Esempio di array

```
a = np.array([[ 0,  1,  1,  2,  3],  
             [ 5,  8, 13, 21, 34],  
             [55, 89, 144, 233, 377]])  
  
print('type:', type(a))  
  
print('ndim:', a.ndim)  
  
print('shape:', a.shape)  
  
print('size:', a.size)  
  
print('dtype:', a.dtype)  
  
print('itemsize:', a.itemsize)
```

type: <class 'numpy.ndarray'>



ndim: 2

shape: (3, 5)

size: 15

dtype: int32

itemsize: 4

Creare un array: la funzione array()

```
# La funzione array consente di creare un array partendo da una lista o tupla Python
# Il tipo dell'array viene dedotto dagli elementi, oppure può essere specificato
a = np.array([2,3,5])
print(a.ndim, a.shape, a.dtype, a.itemsize)

a = np.array([2,3,5], np.uint8)
print(a.ndim, a.shape, a.dtype, a.itemsize)
```

```
1 (3,) int32 4
```



```
1 (3,) uint8 1
```

```
# Sequenze di sequenze sono trasformate in array bidimensionali,
# sequenze di sequenze di sequenze in array tridimensionali e così via
```

```
a = np.array([[ 0,  1,  1,  2,  3],
              [ 5,  8, 13, 21, 34],
              [ 55, 89, 144, 233, 377]])
print(a.ndim, a.shape, a.dtype, a.itemsize)
```

```
a = np.array([ [ [1,2] ], [ [3,4] ] ])
print(a.ndim, a.shape, a.dtype, a.itemsize)
```

```
2 (3, 5) int32 4
```



```
3 (2, 1, 2) int32 4
```



Altri modi per creare un array

```
# empty() crea un array lasciando i valori non inizializzati  
a = np.empty((2,7), np.int16) # 2 righe, 7 colonne  
print(a)
```

```
# zeros() e ones(): valori a zero/uno
```

```
print(np.zeros(5))  
print(np.ones(3, np.int))  
print(np.zeros((2,3)))
```

```
# arange() è simile a range() di Python
```

```
a = np.arange(100, 110, 2)  
print(a)
```

```
# identity() crea una matrice identità
```

```
a = np.identity(3)  
print(a)
```

```
[[ -4352 26016 546 0 4 0 1]  
 [ 29184 -16960 25809 546 0 -16992 25809]]
```



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

```
[1 1 1]
```

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

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

```
[100 102 104 106 108]
```

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

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

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



Tipi di dati numerici

► I tipi numerici di Python e NumPy:

- NumPy supporta una varietà di tipi numerici primitivi maggiore (interi a 8/16/32/64 bit, float a 32/64 bit, ...): corrispondono sostanzialmente a quelli disponibili in C
- Non esiste un tipo primitivo NumPy che corrisponde al tipo int di Python (che non ha limiti di lunghezza)

► La tabella a fianco riassume i principali tipi numerici primitivi di NumPy e i corrispondenti tipi in C

Tipo NumPy	Equivalente C	Note
np.byte	signed char	Dipendente dalla piattaforma
np.ubyte	unsigned char	Dipendente dalla piattaforma
np.short	signed short	Dipendente dalla piattaforma
np ushort	unsigned short	Dipendente dalla piattaforma
np.intc	signed int	Dipendente dalla piattaforma
np.uintc	unsigned int	Dipendente dalla piattaforma
np.int_	signed long	Dipendente dalla piattaforma
np.uint	unsigned long	Dipendente dalla piattaforma
np.longlong	long long	Dipendente dalla piattaforma
np.ulonglong	unsigned long long	Dipendente dalla piattaforma
np.half	-	Float a 16 bit (mezza precisione)
np.single	float	Singola precisione
np.double	double	Doppia precisione
np.longdouble	long double	Precisione estesa
np.csingle	single complex	Num. complesso prec. singola
np.cdouble	double complex	Num. complesso prec. doppia
np.clongdouble	long double complex	Num. complesso prec. estesa



Tipi di dati numerici (alias a dimensione fissa)

- ▶ Molti dei tipi numerici primitivi NumPy dipendono, come i corrispondenti C, dalla piattaforma:
 - Ad esempio in Windows np.int_ (come il corrispondente tipo C long) è a 32 bit, mentre su altre piattaforme è a 64 bit
- ▶ Per evitare potenziali problemi, sono disponibili **alias** per i tipi NumPy che hanno un **numero di bit prefissato** e indipendente dalla piattaforma
- ▶ È anche possibile utilizzare tipi Python per indicare tipi NumPy:
 - int (Python) → np.int_
 - float (Python) → np.float_
 - complex (Python) → np.complex_

Tipo NumPy	Equivalente C	Note
np.int8	int8_t	1 byte [-128, 127]
np.uint8	uint8_t	1 byte [0, 255]
np.int16	int16_t	2 byte [-32768, 32767]
np.uint16	uint16_t	2 byte [0, 65535]
np.int32	int32_t	4 byte [-2147483648, 2147483647]
np.uint32	uint32_t	4 byte [0, 4294967295]
np.int64	int64_t	8 byte [-9223372036854775808, 9223372036854775807]
np.uint64	uint64_t	8 byte [0, 18446744073709551615]
np.float32	float	4 byte
np.float64	double	8 byte (corrisponde a float in Python)
np.float_	double	8 byte (corrisponde a float in Python)
np.complex64	float complex	8 byte
np.complex128	double complex	16 byte (corrisp. a complex in Python)
np.complex_	double complex	16 byte (corrisp. a complex in Python)



Operazioni di base

```
# Gli operatori aritmetici si applicano agli array elemento-per-elemento:  
# il risultato viene tipicamente memorizzato in un nuovo array  
  
a = np.array( [25,36,49,64] )  
b = np.arange( 4 )  
print(b)  
  
c = a-b  
print(c)  
  
print(b**2)  
  
print(np.sqrt(a))  
  
# I confronti seguenti producono array di  
# valori booleani con le stesse dimensioni di a  
print(a < 37)  
  
b = np.array( [25,37,49,63] )  
print(a == b)
```

```
[0 1 2 3]
```



```
[25 35 47 61]
```

```
[0 1 4 9]
```

```
[5. 6. 7. 8.]
```

```
[ True  True False False]
```

```
[ True False  True False]
```

Prodotto

13

```
# Anche l'operatore prodotto '*' opera elemento-per-elemento;
# Dalla v3.5 di Python è disponibile l'operatore '@' per i prodotti fra matrici
A = np.array( [[1,1],
               [0,1]] )
B = np.array( [[2,0],
               [3,4]] )

print(A * B) # Prodotto elemento-per-elemento

print(A @ B) # Prodotto tra matrici

# Un vettore "colonna": shape = (3, 1)
c = np.array([[1], [2], [3]])
# Un vettore "riga": shape = (1, 3)
r = np.array([[3,0,2]])

print(c @ r) # (3,1)x(1,3) = (3,3)
print(r @ c) # (1,3)x(3,1) = (1,1)
```

[[2 0]
[0 4]]



[[5 4]
[3 4]]

[[3 0 2]
[6 0 4]
[9 0 6]]

[[9]]

Operatori di assegnamento

```
# Gli operatori di assegnamento come += o *= modificano  
# l'array esistente senza doverne creare uno nuovo  
  
a = np.ones((2,3))  
b = np.array([[1,0,2], [0,3,0]])  
  
id_a = id(a)  
  
a *= b  
  
print(a)  
  
print(id_a, id(a), id_a==id(a))  
  
a = a * b  
  
print(a)  
  
print(id_a, id(a), id_a==id(a))
```

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

```
2346790588544 2346790588544 True
```

```
[[1. 0. 4.]  
 [0. 9. 0.]]
```

```
2346790588544 2346791113632 False
```



Type cast

```
a = np.ones(3, np.int32)
b = np.array([1.4, 1.5, 1.6])
print(a.dtype, b.dtype)

# In una operazione fra array di tipo diverso,
# il tipo del risultato corrisponde a quello più
# preciso (o più generale) dei due
c = a + b

print(c)

print(c.dtype)

# Per creare un nuovo array cambiando il tipo di un
# array esistente, si può usare il metodo astype()
a = np.arange(10, dtype = np.uint8)
print(a.dtype)

a = a.astype(np.uint64)
print(a.dtype)
```

int32 float64



[2.4 2.5 2.6]

float64

uint8

uint64

Alcune operazioni su un array

```
# Crea una matrice contenente numeri casuali nell'intervallo [0,1)
a = np.random.random((2,3))
print(a)
```

```
# Calcola il valore minimo, massimo e la somma
print(f'Min: {a.min()}')
```

```
print(f'Max: {a.max()}')
```

```
print(f'Sum: {a.sum()}')
```

```
# Il calcolo, invece che su tutti gli elementi,
# può essere lungo uno specifico asse
print(f'Somma di ogni colonna: {a.sum(axis=0)}')
```

```
print(f'Somma di ogni riga: {a.sum(axis=1)}')
```

```
[[0.80024488 0.95481118 0.29314655]
 [0.20377526 0.98809386 0.8159563 ]]
```

```
Min: 0.2037752645645331
```

```
Max: 0.9880938639265971
```

```
Sum: 4.05602804986859
```

```
Somma di ogni colonna: [1.00402014 1.94290504 1.10910285]
```

```
Somma di ogni riga: [2.04820261 2.00782542]
```



Funzioni universali (ufunc)

- ▶ Funzioni NumPy che operano su ndarray **elemento-per-elemento**
 - Alcune ufunc sono automaticamente chiamate quando si usano i corrispondenti operatori Python (ad esempio np.add() è chiamata quando si usa l'operatore '+' fra due ndarray)
- ▶ Alcuni esempi di ufunc:
 - **Operazioni matematiche** - add, subtract, multiply, divide, power, exp, log, sqrt
 - **Trigonometria** - sin, cos, tan, arcsin, arccos, arctan, arctan2, sinh, cosh, tanh, deg2rad, rad2deg
 - **Operazioni sui bit** - bitwise_and, bitwise_or, bitwise_xor, invert, left_shift, right_shift
 - **Confronto** - greater, greater_equal, less, less_equal, not_equal, equal, logical_and, logical_or, logical_xor, logical_not, maximum, minimum, fmax, fmin
 - **Floating point** - isfinite, isinf, isnan, fabs, floor, ceil, trunc

Indicizzazione e slicing su array monodimensionali

18

```
# Negli array NumPy monodimensionali, l'accesso agli elementi è analogo a
# quanto avviene con liste e altre sequenze in Python
a = np.arange(10)**3
print(a)

print(a[2]) # Accesso a un elemento

print(a[2:5]) # Slicing: dall'elemento 2 al 4

a[:6:2] = 42 # Modifica elementi di posto 0, 2, 4
print(a)

print(a[::-1]) # Step negativo: ordine inverso

a[::2] += a[1::2] # a[i] += a[i+1], i=0,2,4,6,8
print(a)

a[:] = -1 # Modifica tutti gli elementi
print(a)
```



```
[ 0  1   8  27  64 125 216 343 512 729]
8
[ 8 27 64]

[ 42  1 42 27 42 125 216 343 512 729]
[729 512 343 216 125 42 27 42  1 42]

[ 43  1 69  27 167 125 559 343 1241 729]
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

Indicizzazione e slicing su array multidimensionali

```
a = np.fromfunction(lambda i,j: i*10+j, (3,5), dtype=int)
print(a)

# Riga 2, Colonna 3
print(a[2, 3])

# Righe 0 e 1, colonna 2
print(a[:2, 2])

# La colonna 1
print(a[:, 1])

# La riga 1
print(a[1, :])

# La riga 1: eventuali indici mancanti
# sono sostituiti con ':'
print(a[1])
```

0	1	2	3	4
10	11	12	13	14
20	21	22	23	24



Slicing su array multidimensionali: altri esempi

```
a = np.fromfunction(lambda i,j: i*10+j, (3,5), dtype=int)
print(a)
```

```
# Rigue 0 e 2, colonne 0 e 3
print(a[::-2, ::3])
```

```
# Rigue 1 e 2
print(a[1:3])
```

```
# Rigue 0 e 1, colonne 0, 1 e 2
print(a[:2, :3])
```

```
# Ultime due righe
print(a[-2:])
```

0	1	2	3	4
10	11	12	13	14
20	21	22	23	24

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]]
```



```
[[ 0  3]
 [20 23]]
```

```
[[10 11 12 13 14]
 [20 21 22 23 24]]
```

```
[[ 0  1  2]
 [10 11 12]]
```

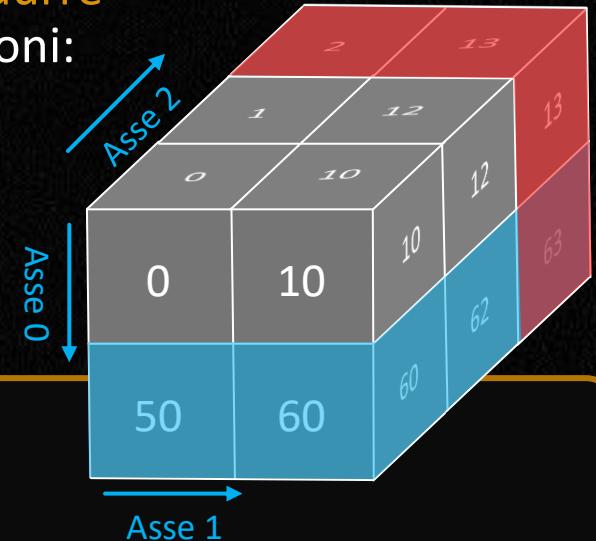
```
[[10 11 12 13 14]
 [20 21 22 23 24]]
```

Slicing con ellissi (...)

- ▶ Ellipsis: oggetto Python predefinito che può essere indicato semplicemente con ...
- ▶ NumPy lo utilizza nello slicing per rappresentare tutti i ':' che mancano per produrre una tupla di indicizzazione completa. Ad esempio se v è un array con 5 dimensioni:
 - $v[1, 2, \dots]$ equivale a $v[1, 2, :, :, :]$
 - $v[\dots, 3]$ equivale a $v[:, :, :, :, 3]$
 - $v[4, \dots, 5, :]$ equivale a $v[4, :, :, 5, :]$
- ▶ Ovviamente questo è utile solo in array con tre o più dimensioni

```
# Inizializza un array con 3 dimensioni
c = np.array( [[[ 0,  1,  2],
                 [ 10, 12, 13]],
                [[ 50, 51, 52],
                 [ 60, 62, 63]]])
print(c.shape)

print(c[1,...]) # Equivale a c[1] e a c[1,:,:]
print(c[...,:,2]) # Equivale a c[:, :, 2]
```



(2, 2, 3)

```
[[50 51 52]
 [60 62 63]]
```

```
[[ 2 13]
 [52 63]]
```



Iterare su un array

```
a = np.arange(7)
print(a)

# Si può iterare sugli elementi un array monodimensionale
for x in a:
    print(x, end='; ')
print()

a = np.fromfunction(lambda i,j: i*10+j, (3,5), dtype=int)
print(a)

# Iterando su una matrice si ottengono le righe...
for r in a:
    for x in r: # ... su cui si può ulteriormente iterare
        print(x, end='; ')
    print()

# L'attributo .flat è un iteratore su tutti gli elementi
for x in a.flat:
    print(x, end=', ')
```

[0 1 2 3 4 5 6]



0; 1; 2; 3; 4; 5; 6;

[[0 1 2 3 4]
 [10 11 12 13 14]
 [20 21 22 23 24]]

0; 1; 2; 3; 4;
10; 11; 12; 13; 14;
20; 21; 22; 23; 24;

0, 1, 2, 3, 4, 10, 11, 12, 13, 14,
20, 21, 22, 23, 24,

Modifica della forma

```
a = np.arange(12)
print(a.shape)
print(a)

# reshape() restituisce gli stessi con forma diversa
b = a.reshape(2, 6)
print(b.shape)
print(b)

# Se una dimensione è -1, viene calcolata automaticamente
c = a.reshape(2, 2, -1)
print(c.shape)
print(c)

a.resize(2,6) # resize() modifica l'array stesso
print(a)

# ravel() restituisce i dati come array monodimensionale
d = a.ravel()
print(d)
```

(12,)
[0 1 2 3 4 5 6 7 8 9 10 11]

(2, 6)
[[0 1 2 3 4 5]
 [6 7 8 9 10 11]]

(2, 2, 3)
[[[0 1 2]
 [3 4 5]]
 [[6 7 8]
 [9 10 11]]]

[[0 1 2 3 4 5]
 [6 7 8 9 10 11]]

[0 1 2 3 4 5 6 7 8 9 10 11]



Altri modi per modificare la forma

```
a = np.arange(12).reshape(2,6)
print(a)

# Aggiunta di nuove dimensioni con np.newaxis
b = a[np.newaxis, ...]
print(b.shape)
print(b)
c = a[np.newaxis, ..., np.newaxis, np.newaxis]
print(c.shape)
# np.newaxis non è altro che un riferimento a None
print(np.newaxis is None)

# Scambiare le dimensioni con transpose() o .T
d = a.T
e = a.transpose()
print(d.shape, e.shape)

# squeeze() elimina eventuali dimensioni a uno
f = c.squeeze()
print(f.shape)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
```



```
(1, 2, 6)
[[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]]
```

```
(1, 2, 6, 1, 1)
```

```
True
```

```
(6, 2) (6, 2)
```

```
(2, 6)
```

Concatenare array

```
a = np.floor(10*np.random.random((2,2)))
print(a)

b = np.floor(10*np.random.random((2,2)))
print(b)

print( np.vstack((a,b)) )

print( np.hstack((a,b)) )

# column_stack() affianca array 1D come colonne
# di un array 2D
x = np.array([4.,2.])
y = np.array([3.,8.])
print( np.column_stack((x,y)) )

# notare la differenza se invece di usa hstack()
print( np.hstack((x,y)) )
```

```
[[5. 6.]
 [2. 0.]]
```



```
[[4. 8.]
 [1. 7.]]
```

```
[[5. 6.]
 [2. 0.]
 [4. 8.]
 [1. 7.]]
```

```
[[5. 6. 4. 8.]
 [2. 0. 1. 7.]]
```

```
[[4. 3.]
 [2. 8.]]
```

```
[4. 2. 3. 8.]
```

Copie e viste di array

```
a = np.arange(7)
print(a)
# Come per qualsiasi variabile Python, l'assegnamento
# non crea un nuovo oggetto, solo un nuovo riferimento
b = a
print(b is a)

# Lo slicing (come altre operazioni) crea una vista: un
# nuovo oggetto che condivide gli stessi dati. a.base
# è un riferimento all'oggetto su cui è costruita la vista
c = a[3::2]
print(c is a, c.base is a)
print(c)
c[0] = -1
print(a)

d = a.copy() # Il metodo copy() crea una copia dell'array
print(d is a, d.base is a, d.base)
d[0] = -1
print(d, a)
```

[0 1 2 3 4 5 6]



True

False True
[3 5]

[0 1 2 -1 4 5 6]

False False None

[-1 1 2 -1 4 5 6] [0 1 2 -1 4 5 6]



Broadcasting

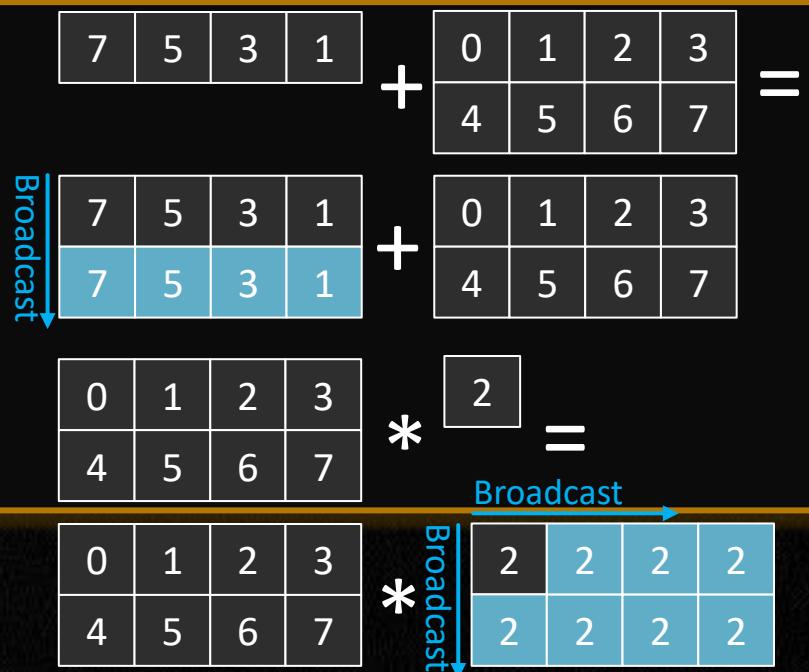
- Comodo e potente strumento con cui NumPy consente alle ufunc di agire su input con forme diverse.
- Prima regola: se gli array di input non hanno lo stesso numero di dimensioni, un "1" viene anteposto alla shape dell'array più piccolo finché il numero di dimensioni non coincide.
- Seconda regola: le dimensioni a 1 sono trattate come se il numero di elementi fosse pari a quello dell'array con più elementi lungo le corrispondenti dimensioni: si assume che tutti gli elementi lungo tale dimensione siano uguali.
- Se dopo l'applicazione di queste due regole gli array hanno la stessa forma, l'operazione viene eseguita.

```
a = np.array([7, 5, 3, 1])
print(a)
b = np.arange(8).reshape(2,-1)
print(b)
# Somma con broadcasting
c = a + b
print(c)

# Prodotto con broadcasting
print(b * 2)
```



```
[7 5 3 1]
[[0 1 2 3]
 [4 5 6 7]]
[[ 7  6  5  4]
 [11 10  9  8]]
[[ 0  2  4  6]
 [ 8 10 12 14]]
```



Indicizzare con un array di indici

28

```
a = np.arange(12)**2
print(a)

# Oltre a interi e slicing, si possono
# utilizzare array di interi per
# indicizzare altri array
idx1 = np.array( [ 1,1,3,8,5 ] )
print(a[idx1])

# L'array di interi può anche
# essere multidimensionale: il
# risultato ha la forma dell'array
# usato come indice
idx2 = np.array( [ [3, 4], [9, 7] ] )
print(a[idx2])
```

```
[ 0  1  4  9 16 25 36 49 64 81 100 121]
```



```
[ 1  1  9 64 25]
```

```
[[ 9 16]
 [81 49]]
```

Indicizzare un array multidimensionale con array di indici

```
# Se l'array a è multidimensionale si può passare un array  
# di indici per ciascuna dimensione, purché abbiamo la stessa  
# forma (o si possa fare broadcast fra loro).  
# Come per lo slicing, si può utilizzare ':' o '...'  
# per non dover specificare tutte le dimensioni.  
a = (np.arange(12)**2).reshape(3,4)  
print(a)  
  
idx2 = np.array([1,1,2])  
print(a[idx2,:]) # oppure print(a[idx2])  
  
print(a[:,idx2])  
  
idx_r = np.array( [ [0,0,0], [1,1,1] ] )  
idx_c = np.array( [ [2,3,2], [0,0,0] ] )  
print(a[idx_r,idx_c])
```



```
[[ 0  1  4  9]  
 [16 25 36 49]  
 [64 81 100 121]]  
  
[[ 16 25 36 49]  
 [ 16 25 36 49]  
 [ 64 81 100 121]]  
  
[[ 1  1  4]  
 [ 25 25 36]  
 [ 81 81 100]]  
  
[[ 4  9  4]  
 [16 16 16]]
```

Indicizzare con array di booleani

```
# Nell'indicizzazione con array di interi si specificano
# gli indici da considerare, invece in questo caso si scelgono
# esplicitamente quali elementi si vogliono e quali no.
# Il modo più semplice è utilizzare array booleani con la
# stessa forma dell'array originale.
a = np.arange(12).reshape(3,4)
print(a)

b = a > 4
print(b) # array booleano con la stessa forma di a

print(a[b]) # un array 1D con gli elementi considerati

# La selezione con elementi booleani è molto utile per
# modificare solo gli elementi che soddisfano un certo criterio.
criterio = a%3 == 0
print(criterio)

print(a[criterio])
```



```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[[False False False False]
 [False True True True]
 [ True True True True]]

[ 5  6  7  8  9 10 11]

[[ True False False True]
 [False False True False]
 [False True False False]]

[0 3 6 9]
```

Riepilogo

- ▶ La conoscenza di NumPy è fondamentale per lavorare in Python con OpenCV e tante altre librerie software
 - Il tipo di dato fondamentale introdotto da NumPy è **ndarray**, un **array multidimensionale omogeneo**
 - I tipi di dati numerici utilizzati in NumPy sono diversi da Python e corrispondono a quelli del C
- ▶ I concetti fondamentali da apprendere e su cui esercitarsi sono:
 - Creazione di array con **np.array()**, **np.zeros()**, **np.ones()**, **np.arange()**, etc.
 - Operatori di base, principali **ufunc** e **broadcasting**
 - Type cast fra array, **copie** e **viste**
 - Indicizzazione semplice e con **slicing**
 - Indicizzazione con **array di interi** e **array di booleani**
 - Modificare la forma di array con **reshape()**, utilizzo di **np.newaxis**, **np.hstack()**, etc.



Per approfondire

- ▶ I siti da cui iniziare:
 - Sito ufficiale NumPy: <https://numpy.org>
 - Sito con la documentazione: <https://numpy.org/devdocs>
 - Manuale con i dettagli di tutte le funzionalità: <https://numpy.org/devdocs/reference>
- ▶ Libro completo (open source) di Nicolas P. Rougier (Inria):
 - <http://www.labri.fr/perso/nrougier/from-python-to-numpy>

