INDICE

- Operazioni tra array elemento per elemento
  - Comparazione
    - fra array e costanti
    - fra due array (con lo stesso numero di elementi e la stessa shape)
  - Operazioni logiche su array
  - Broadcasting
    - Somma in 2D con array con stesso numero di righe ma diverso numero di colonne
    - Somma in 2D con array con stesso numero di colonne ma diverso numero di righe
    - Somma in 2D tra un vettore riga ed un vettore colonna
  - Regole e condizioni per il Broadcasting
  - Aggiunta di dimensioni ad un array (senza aggiungere elementi)
    - con resize /reshape
    - con slices e np.newaxis
  - Applicazione passo passo delle regole del broadcasting ad un esempio
    - Esempio di uso del broadcasting
      - provare in aula
      - soluzione:
      - soluzione alternativa
- ufunc e vettorizzazione
  - cicli vs ufunc
  - Vettorizzazione di funzioni
- Fancy Indexing
  - Indicizzazione con array di bool: le maschere
    - Indicizzazione con maschere ed assegnamento
  - Indicizzazione con array di bool e slices (dimensione per dimensione)
    - assegnamento
  - Indicizzazione con array di interi
    - Indicizzazione di array monodimensionali
      - con array di indici monodimensionali
      - con array di indici di shape generica
    - Indicizzazione di array con più dimensioni
      - Con un singolo array di indici
        - singolo array di indici monodimensionale
        - singolo array degli indici con più dimensioni
      - Con array di indici passati per ogni dimensione dell'array da indicizzare
    - Indicizzazione con array di interi ed assegnamento
  - Fancy indexing -> copia
    - Fancy indexing + elaborazione + riassegnamento
- Esercizio: integral approximation

ripartiamo da dove avevamo lasciato.

Operazioni tra array elemento per elemento

Ci sono una sessantina di funzioni che permettono di operare con gli array elemento per elemento.

http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs

Queste funzioni sono usate anche quando si usano gli operatori matematici, di comparazione etc. in modo tradizionale.

ad esempio quando uso l'operatore + per sommare due array, l'interprete chiama la `ufunc` (universal function) numpy.add

Passiamo in rassegna gli aspetti meno banali che interessano gli operatori element-wise

Comparazione

la comparazione tra array da luogo a degli array di bool ...

fra array e costanti

In [8]:
```
a=np.arange(5*4).reshape(5,4)
a
```

Out[8]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

In [9]:
```
a>7
```

Out[9]:
```
array([[False, False, False, False],
       [False, False, False, False],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
```

fra due array (con lo stesso numero di elementi e la stessa shape)

numpy.random.randint

In [10]:
```
#genero un array con shape (5,4) e interi random tra 0 e 20 (escluso, come nei range)
b=np.random.randint(0,20,(5,4))
b
```

Out[10]:
```
array([[ 5,  7, 10, 12],
       [ 2, 15,  5, 18],
       [ 1,  4,  8, 11],
       [ 8,  2,  5,  4],
       [ 5,  4, 17, 17]])
```

In [11]:
```
a>b
```

Out[11]:
```
array([[False, False, False, False],
       [ True, False,  True, False],
       [ True,  True,  True, False],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
```

Operazioni logiche su array

Le operazioni logiche avvengono elemento per elemento solo se si usano gli operatori bitwise (& , |, ~ , ^) o le corrispondenti ufunc (comparison-functions)
Gli operatori logici and, or , not si riferiscono all'array nella sua interezza.

In [15]:
```
a
```

Out[15]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

In [14]:
```
(a > 5) & (a < 14)
```

Out[14]:
```
array([[False, False, False, False],
       [False, False,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True, False, False],
       [False, False, False, False]], dtype=bool)
```

In [16]:
```
(a < 5) | (a > 14)
```

Out[16]:
```
array([[ True,  True,  True,  True],
       [ True, False, False, False],
       [False, False, False, False],
       [False, False, False,  True],
```

```
[ True,  True,  True,  True]], dtype=bool)
```

OCCHIO ALLA precedenza degli operatori logici su quelli di comparazione:

In [18]:
```
a<5 | a>14
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-18-47b8fe18ff4a> in <module>()
----> 1 a<5 & a>14

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

l'operazione che fa scaturire l'errore è 5|a che viene eseguita a causa della precedenza di | sugli operatori di comparazione

Broadcasting

Ma cosa succede quando voglio operare con array di dimensioni diverse?

Il broadcasting è l'insieme di regole che permette alla ufunc di operare su array che non hanno esattamente le stesse dimensioni.

Ciò non è sempre possibile ma il broadcasting amplia di molto le possibilità di operare.

Prima di cercare di riassumere le regole e le condizioni per l'applicazione del broadcasting, vediamo degli esempi:

Somma in 2D con array con stesso numero di righe ma diverso numero di colonne

In [26]:
```
a=np.arange(3*4).reshape(3,4)
print('a=\n',a)#(3,4)

b=np.arange(10,40,10).reshape(3,1)
print('b=\n',b)
```

```
a=
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
b=
 [[10]
 [20]
 [30]]
```

In [27]:
```
a+b
```

Out[27]:
```
array([[10, 11, 12, 13],
       [24, 25, 26, 27],
       [38, 39, 40, 41]], dtype=int32)
```

l'operazione sopra può essere pensata come eseguita in due fasi:

prima trasformo b in modo da avere le stesse dimensioni di a replicandolo 4 volte (regola 2, vedi sotto)

In [28]:
```
newb=np.concatenate((b,b,b,b),1)
newb
```

Out[28]:
```
array([[10, 10, 10, 10],
       [20, 20, 20, 20],
       [30, 30, 30, 30]])
```

NOTA BENE: questa trasformazione NON avviene effettivamente, semplicemente la ufunc add fa riferimento ai valori contenuti nella prima ed unica colonna di b per effettuare la somma

poi sommo a con newb

In [29]:
```
a+newb
```

Out[29]:
```
array([[10, 11, 12, 13],
       [24, 25, 26, 27],
```

```
       [38, 39, 40, 41]], dtype=int32)
```

Somma in 2D con array con stesso numero di colonne ma diverso numero di righe

In [30]:
```
a=np.arange(3*4).reshape(4,3)
print('a=\n',a)#(3,4)

b=np.arange(10,40,10).reshape(1,3)
print('b=\n',b)
```

```
a=
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
b=
 [[10 20 30]]
```

In [31]:
```
a+b
```

Out[31]:
```
array([[10, 21, 32],
       [13, 24, 35],
       [16, 27, 38],
       [19, 30, 41]], dtype=int32)
```

al solito è come se prima avessi trasformato b replicandolo 4 volte (regola 2)

In [32]:
```
newb = np.concatenate((b,b,b,b),0)
newb
```

Out[32]:
```
array([[10, 20, 30],
       [10, 20, 30],
       [10, 20, 30],
       [10, 20, 30]])
```

e poi:

In [33]:
```
a+newb
```

Out[33]:
```
array([[10, 21, 32],
       [13, 24, 35],
       [16, 27, 38],
       [19, 30, 41]], dtype=int32)
```

Somma in 2D tra un vettore riga ed un vettore colonna

In [82]:
```
a=np.arange(3).reshape(1,3) # una riga, tre colonne
b=np.arange(10,50,10).reshape(4,1) # (4,1) quattro righe, 1 colonna
print('a=\n',a)
print('b=\n',b)
```

```
a=
 [[0 1 2]]
b=
 [[10]
 [20]
 [30]
 [40]]
```

In [83]:
```
a+b
```

Out[83]:
```
array([[10, 11, 12],
       [20, 21, 22],
       [30, 31, 32],
       [40, 41, 42]], dtype=int32)
```

In questo caso, per effettuare il broadcasting devo trasformare entrambi i vettori

trasformo a in modo che abbia lo stesso numero di righe di b replicandolo 4 volte (regola 2)

In [84]:
```
newa = np.concatenate((a,a,a,a),0)
newa
```

Out[84]:
```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

trasformo b in modo che abbia lo stesso numero di colonne di a replicandolo 3 volte (regola 2)

In [85]:
```
newb = np.concatenate((b,b,b),1)
newb
```

Out[85]:
```
array([[10, 10, 10],
       [20, 20, 20],
       [30, 30, 30],
       [40, 40, 40]])
```

e quindi sommo

In [86]:
```
newa+newb
```

Out[86]:
```
array([[10, 11, 12],
       [20, 21, 22],
       [30, 31, 32],
       [40, 41, 42]], dtype=int32)
```

Regole e condizioni per il Broadcasting

Ci sono due fondamentali regole alla base del broadcasting:

- Regola 1: se gli array in ingresso non hanno le stesse dimensioni (ndim diversi) prependo un 1 alla shape dell'array più piccolo fino a che ottengo ndim uguali.
- Regola 2:
  - un array con profondità 1 secondo un particolare asse viene (idealmente) trasformato in un array che ha una profondità lungo lo stesso asse pari a quella dell'array più grande.
  - Il valore dell'array originale viene copiato in tutte le celle estese dell'array trasformato

Ne consegue che due o più array sono compatibili per il broadcasting se si verifica una delle tre seguenti condizioni:

- condizione a) hanno esattamente la stessa shape
- condizione b) hanno lo stesso numero di dimensioni e la 'lunghezza' di ciascuna dimensione o è una lunghezza comune o è uno
- condizione c) hanno un numero diverso di dimensioni ma quelli che ne hanno meno soddisfano la condizione b dopo aver aggiunto un numero opportuno di 1 a sinistra della shape di partenza

Negli esempi che abbiamo visto è stata applicata solo la regola 2 e quindi abbiamo esaminato casi in cui si verificavano le condizioni a e b. Vediamo un caso nel quale si manifesta la condizione c e si applica la regola 1.

Aggiunta di dimensioni ad un array (senza aggiungere elementi)

Per capire il broadcasting è utile avere chiaro come fare a modificare la shape di un array aggiungendogli dimensioni ma non elementi (vedi regola 1).

a è un array a 1 dimensione con 5 elementi:

In [11]:
```
a=arange(5)
print(a.shape)
print(a)
```

```
(5,)
```

```
[0 1 2 3 4]
```

con resize /reshape

trasformo a in un array di due dimensioni con 1 riga e 5 colonne

In [12]:
```
a.resize(1,5)
print('a.shape=',a.shape)
print('a=',a)
```

```
a.shape= (1, 5)
a= [[0 1 2 3 4]]
```

trasformo a in un array con 5 righe ed una colonna

In [13]:
```
a.resize(5,1)
print('a.shape=',a.shape)
print('a=',a)
```

```
a.shape= (5, 1)
a= [[0]
 [1]
 [2]
 [3]
 [4]]
```

trasformo a in un array con 5 piani, 1 riga ed una colonna

In [14]:
```
a.resize(5,1,1)
print('a.shape=',a.shape)
print('a=',a)
```

```
a.shape= (5, 1, 1)
a= [[[0]]

 [[1]]

 [[2]]

 [[3]]

 [[4]]]
```

etc etc

con slices e np.newaxis

In [15]:
```
a=arange(5)
print(a.shape)
print(a)
```

```
(5,)
[0 1 2 3 4]
```

restituisce una vista di a con 1 riga e 5 colonne

In [16]:
```
a[np.newaxis , :]
```

Out[16]:
```
array([[0, 1, 2, 3, 4]])
```

restituisce una vista di a con 5 righe e 1 colonna

In [17]:
```
a[: , None] #NB np.newaxis è equivalente a None
```

Out[17]:
```
array([[0],
       [1],
       [2],
       [3],
```

```
       [4]])
```

restituisce una vista di a con 5 piani, 1 riga e 1 colonna

In [18]:
```
a[:,None,None]
```

Out[18]:
```
array([[[0]],

       [[1]],

       [[2]],

       [[3]],

       [[4]]])
```

Applicazione passo passo delle regole del broadcasting ad un esempio

Considero la somma in 2D tra un vettore 1D con 3 elementi ed un vettore colonna

Controllate bene le differenze rispetto al caso precedente!!!

In [90]:
```
a=np.arange(3)# 3 elementi
b=np.arange(10,50,10).reshape(4,1) # (4,1) quattro righe, 1 colonna
print('a=\n',a)
print('b=\n',b)
```

```
a=
 [0 1 2]
b=
 [[10]
 [20]
 [30]
 [40]]
```

NOTA BENE: c'è differenza tra i due array v1 e v2:

In [89]:
```
v1 = np.arange(3)
v2 = np.arange(3).reshape(1,3)
print('v1 = ',v1)
print('v2 = ',v2)
print('v1.shape = ',v1.shape)
print('v2.shape = ',v2.shape)
```

```
v1 =  [0 1 2]
v2 =  [[0 1 2]]
v1.shape =  (3,)
v2.shape =  (1, 3)
```

v1 è un array a 1 dimensione con 3 elementi
v2 è un array con 2 dimensioni e 3 elementi su una riga

il broadcasting in questo caso può essere scomposto in 3 passaggi

Prima applico la regola 1 e aggiungo una dimensione ad a senza però aggiungere alcun elemento, usando slice e newaxis:

In [91]:
```
newa1 = a[np.newaxis ,  :] # avrei potuto usare anche newa1=a.reshape(1,3) ...
newa1
```

Out[91]:
```
array([[0, 1, 2]])
```

Poi applico la regola 2 e trasformo newa1 in modo da avere lo stesso numero di righe di b replicandolo 4 volte

In [93]:
```
newa2 = np.concatenate((newa1,newa1,newa1,newa1),0)
newa2
```

Out[93]:
```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
```

```
       [0, 1, 2]])
```

Quindi applico nuovamente la regola 2 e trasformo b in modo da avere lo stesso numero di colonne di newa2 replicandolo 3 volte lungo l'asse 1

In [94]:
```
newb = np.concatenate((b,b,b),1)
newb
```

Out[94]:
```
array([[10, 10, 10],
       [20, 20, 20],
       [30, 30, 30],
       [40, 40, 40]])
```

ed infine sommo

In [95]:
```
newa2+newb
```

Out[95]:
```
array([[10, 11, 12],
       [20, 21, 22],
       [30, 31, 32],
       [40, 41, 42]], dtype=int32)
```

vi ricordo che io avevo ottenuto lo stesso risultato con a+b !!!!!!!!

In [96]:
```
a+b
```

Out[96]:
```
array([[10, 11, 12],
       [20, 21, 22],
       [30, 31, 32],
       [40, 41, 42]], dtype=int32)
```

Esempio di uso del broadcasting

generare il seguente array con 10x10x10 elementi:

array([[[0, 10, ..., 80, 90], [100, 110, ..., 180, 190], ..., [800, 810, ..., 880, 890], [900, 910, ..., 980, 990]], [[1000, 1010, ..., 1080, 1090], [1100, 1110, ..., 1180, 1190], ..., [1800, 1810, ..., 1880, 1890], [1900, 1910, ..., 1980, 1990]], ..., [[8000, 8010, ..., 8080, 8090], [8100, 8110, ..., 8180, 8190], ..., [8800, 8810, ..., 8880, 8890], [8900, 8910, ..., 8980, 8990]], [[9000, 9010, ..., 9080, 9090], [9100, 9110, ..., 9180, 9190], ..., [9800, 9810, ..., 9880, 9890], [9900, 9910, ..., 9980, 9990]]], dtype=int32)

provare in aula

soluzione:

In [6]:
```
#creo un array che rappresenta l'incremento relativo ai diversi piani (1000)
#l'array è strutturato per piani (di un solo elemento)
a=np.arange(0,10000,1000)[:,None,None]
print('a.shape=',a.shape)
print('a=',a)
```

```
a.shape= (10, 1, 1)
a= [[[   0]]

 [[1000]]

 [[2000]]

 [[3000]]

 [[4000]]

 [[5000]]

 [[6000]]

 [[7000]]

 [[8000]]
```

```
[[9000]]]
```

In [8]:
```
#creo un array che rappresenta l'incremento relativo alle righe (100)
b=np.arange(0,1000,100)[None,:,None]
print('b.shape=',b.shape)
print('b=',b)
```

```
b.shape= (1, 10, 1)
b= [[[  0]
  [100]
  [200]
  [300]
  [400]
  [500]
  [600]
  [700]
  [800]
  [900]]]
```

In [9]:
```
#creo un array che rappresenta l'incremento relativo alle colonne (10)
c=np.arange(0,100,10)[None,None,:]
print('c.shape=',c.shape)
print('c=',c)
```

```
c.shape= (1, 1, 10)
c= [[[ 0 10 20 30 40 50 60 70 80 90]]]
```

In [ ]:
```
np.set_printoptions(threshold=10)
np.set_printoptions(edgeitems=2)
print(a+b+c)
```

soluzione alternativa

In [20]:
```
a=np.arange(0,10000,1000)[:,None,None]#uguale al precedente caso
b2=np.arange(0,1000,100)[:,None]  #una dimensione meno del precedente caso
c2=np.arange(0,100,10)   #due dimensioni in meno
print('a.shape=',a.shape)
print('-'*20)
print('b.shape=',b.shape)
print('b2.shape=',b2.shape)
print('-'*20)
print('c.shape=',c.shape)
print('c2.shape=',c2.shape)
a+b2+c2
```

```
a.shape= (10, 1, 1)
--------------------
b.shape= (1, 10, 1)
b2.shape= (10, 1)
--------------------
c.shape= (1, 1, 10)
c2.shape= (10,)
```

Out[20]:
```
array([[[   0,   10, ...,   80,   90],
        [ 100,  110, ...,  180,  190],
        ...,
        [ 800,  810, ...,  880,  890],
        [ 900,  910, ...,  980,  990]],

       [[1000, 1010, ..., 1080, 1090],
        [1100, 1110, ..., 1180, 1190],
        ...,
        [1800, 1810, ..., 1880, 1890],
        [1900, 1910, ..., 1980, 1990]],

       ...,
       [[8000, 8010, ..., 8080, 8090],
        [8100, 8110, ..., 8180, 8190],
        ...,
        [8800, 8810, ..., 8880, 8890],
```

```
       [8900, 8910, ..., 8980, 8990]],

      [[9000, 9010, ..., 9080, 9090],
       [9100, 9110, ..., 9180, 9190],
       ...,
       [9800, 9810, ..., 9880, 9890],
       [9900, 9910, ..., 9980, 9990]]], dtype=int32)
```

ufunc e vettorizzazione

cicli vs ufunc

se vogliamo sommare due array monodimensionali di N elementi, va e vb, possiamo effettuare un ciclo
nella seguente funzione misuro il tempo necessario ad eseguire l'operazione di somma escuso il tempo necessario a creare l'array dei risultati

In [34]:
```
import time

def sommaciclo(va,vb):
    #determino il numero di elementi
    N=min(len(va),len(vb))
    #creo ed inizializzo l'array dei risultati
    vres = np.zeros(N)

    t1 = time.clock()

    #somma con un ciclo sugli elementi
    #dei vettori
    for i in range(N):
        vres[i] = va[i] + vb[i]

    t2 = time.clock()

    retval=t2-t1
    print('sommaciclo {1} elementi, t={0:.6f}'.format(retval,N))
    return retval
```

Oppure posso usare la somma tra vettori che richiama implicitamente la ufunc numpy.add
nella seguente funzione misuro il tempo necessario ad eseguire l'operazione di somma compreso il tempo necessario a creare l'array dei risultati che non è scorporabile

In [35]:
```
def sommavect(va,vb):
    t1 = time.clock()

    vr = va + vb

    t2 = time.clock()

    retval=t2-t1
    print('sommavect  {1} elementi, t={0:.6f}'.format(retval,va.size))
    return retval
```

Adesso confronto il tempo di esecuzione delle due somme al crescere del numero di elementi contenuti

In [43]:
```
for N in [100,1000,10000,100000,1000000]:
    va=np.arange(N)
    vb=np.arange(200,200+N)
    print('-'*30)
    tc=sommaciclo(va,vb)
    tv=sommavect(va,vb)
    print('ratio = {0}'.format(tc/tv))
```

```
------------------------------
```

```
sommaciclo 100 elementi, t=0.000385
sommavect  100 elementi, t=0.000041
ratio = 9.310810791934884
-----------------------------
sommaciclo 1000 elementi, t=0.003660
sommavect  1000 elementi, t=0.000034
ratio = 106.50406541529249
-----------------------------
sommaciclo 10000 elementi, t=0.044367
sommavect  10000 elementi, t=0.000162
ratio = 274.2901550729541
-----------------------------
sommaciclo 100000 elementi, t=0.163127
sommavect  100000 elementi, t=0.000460
ratio = 354.96717332723364
-----------------------------
sommaciclo 1000000 elementi, t=1.130004
sommavect  1000000 elementi, t=0.004410
ratio = 256.23337133975
```

Come si può vedere al crescere di N la versione 'vettorizzata' è più conveniente
Tale convenienza dipende principalmente dal fatto che il ciclo sugli elementi è eseguito nella routine compilata e non mediante l'interprete!!!!

Vettorizzazione di funzioni

data una generica funzione di python non è detto che questa sia compatibile con array passati come argomento.
Ad esempio consideriamo la seguente myfunc

In [86]:
```
def myfunc(a, b):
        """Return a+b if a>b, otherwise return a-b"""
        if a > b:
                return a + b
        else:
                return a - b
```

definiamo a e b

In [87]:
```
a=np.arange(5)
b=np.random.randint(0,10,(5))
a,b
```

Out[87]:
```
(array([0, 1, 2, 3, 4]), array([4, 6, 0, 5, 1]))
```

se applichiamo myfunc ad a e b l'interprete segnala un errore

In [88]:
```
myfunc(a,b)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-88-1e77756ca839> in <module>()
----> 1 myfunc(a,b)

<ipython-input-86-84a12f4517f5> in myfunc(a, b)
      1 def myfunc(a, b):
      2         """Return a+b if a>b, otherwise return a-b"""
----> 3         if a > b:
      4                 return a + b
      5         else:

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

(a>b) è un array di bool che è valutato in un costrutto if che richiede una variabile riconducibile a vero o falso ....

Se vogliamo che myfunc lavori come una ufunc ovvero elemento per elemento sull'array posso crearne una nuova versione studiandola ad-hoc usando le maschere (che vedremo tra poco):

In [89]:
```
def myfunc2(a,b):
       a.shape == b.shape :
```

```
    if
        retval = np.zeros_like(a)
        i = a>b
        # print(i)
        retval[i] = a[i] + b[i]
        i = ~i
        #print(i)
        retval[i] = a[i] - b[i]
        return retval
    else:
        return np.array([])
```

In [90]:
```
print(a)
print(b)
myfunc2(a,b)
```

```
[0 1 2 3 4]
[4 6 0 5 1]
```

Out[90]:
```
array([-4, -5,  2, -2,  5])
```

Questa funzione adesso lavora con array che però devono avere la stessa forma.

Nel caso a e b siano di forma diversa ma compatibili con il broadcasting la funzione restituisce un array vuoto

In [91]:
```
a=np.arange(5)
b=np.random.randint(0,10,(5)).reshape(5,1)
print('a=',a)
print('b=',b)
```

```
a= [0 1 2 3 4]
b= [[8]
 [5]
 [5]
 [0]
 [8]]
```

In [92]:
```
myfunc2(a,b)
```

Out[92]:
```
array([], dtype=float64)
```

esiste un modo semplice per trasformare una funzione di python ordinaria in una ufunc che lavora su array elemento per elemento e supporta il broadcasting e cioè utilizzare la funzione numpy.vectorize:

In [93]:
```
myfunc_v = np.vectorize(myfunc)
```

in questo caso la nuova funzione supporterà pienamente anche il broadcasting:

In [94]:
```
myfunc_v(a,b)
```

Out[94]:
```
array([[-8, -7, -6, -5, -4],
       [-5, -4, -3, -2, -1],
       [-5, -4, -3, -2, -1],
       [ 0,  1,  2,  3,  4],
       [-8, -7, -6, -5, -4]])
```

la funzione 'vettorizzata' lavora anche con numeri ma restituisce comunque un array

In [95]:
```
myfunc_v(3,2)
```

Out[95]:
```
array(5)
```

Fancy Indexing

Indicizzazione con array di bool: le maschere

In [30]:
```
a=np.arange(2*3*4).reshape(2,3,4)
a
```

Out[30]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

Abbiamo già visto che gli operatori di comparazione danno come risultato degli array di bool con la stessa shape degli array di partenza

In [31]:
```
mask = a>8
mask
```

Out[31]:
```
array([[[False, False, False, False],
        [False, False, False, False],
        [False,  True,  True,  True]],

       [[ True,  True,  True,  True],
        [ True,  True,  True,  True],
        [ True,  True,  True,  True]]], dtype=bool)
```

se uso una matrice di bool (che chiamo maschera) per indicizzare un array dove la shape della maschera corrisponde a quella dell'array di interesse
ottengo un array monodimensionale che contiene tutti gli elementi dell'array corrispondenti ad un elemento True della maschera

In [32]:
```
a[mask]
```

Out[32]:
```
array([ 9, 10, ..., 22, 23])
```

Indicizzazione con maschere ed assegnamento

Questa espressione può essere usata per assegnare nuovi valori ad un array

In [37]:
```
a[mask]=1000
a
```

Out[37]:
```
array([[[   0,    1,    2,    3],
        [   4,    5,    6,    7],
        [   8, 1000, 1000, 1000]],

       [[1000, 1000, 1000, 1000],
        [1000, 1000, 1000, 1000],
        [1000, 1000, 1000, 1000]]])
```

NB: se assegno a[mask] ad un nuovo array questo NON si comporta come una vista ma ottengo una copia!!!

In [55]:
```
b=a[mask]
b[0]=50000
print(a)
print(b.flags.owndata)
```

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

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
True
```

Indicizzazione con array di bool e slices (dimensione per dimensione)

L'indicizzazione con bool può essere effettuata anche secondo un'altra modalità.
Al solito illustriamola con degli esempi.

In [52]:
```
a=np.arange(2*3*4).reshape(2,3,4)
a
```

Out[52]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

seleziono uno dei due piani (quello che corrisponde a True nell'array di bool)

In [53]:
```
secondopianodidue = np.array([False,True])
b=a[secondopianodidue,:,:]
print(b)
```

```
[[[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

seleziono una delle righe (per ogni piano)

In [7]:
```
secondarigaditre  = np.array([False , True , False])
print(a[: , secondarigaditre , :])
```

```
[[[ 4  5  6  7]]

 [[16 17 18 19]]]
```

In [27]:
```
secondaeterzacolonnadiquattro = np.array( [False , True , True , False])
print(a[: , : , secondaeterzacolonnadiquattro])
```

```
[[[ 1  2]
  [ 5  6]
  [ 9 10]]

 [[13 14]
  [17 18]
  [21 22]]]
```

In [9]:
```
print(a[secondopianodidue , secondarigaditre , secondaeterzacolonnadiquattro])
```

```
[17 18]
```

assegnamento

anche in questo caso posso assegnare un nuovo valore agli elementi dell'array originario

In [57]:
```
a[secondopianodidue,:,:]=200000
a
```

Out[57]:
```
array([[[     0,      1,      2,      3],
        [     4,      5,      6,      7],
        [     8,      9,     10,     11]],

       [[200000, 200000, 200000, 200000],
        [200000, 200000, 200000, 200000],
        [200000, 200000, 200000, 200000]]])
```

ma con se assegno un nome al sub-array indicizzato ottengo una copia!!!

In [58]:
```
b=a[secondopianodidue,:,:]
b[0]=-10000
print(a)
print(b.flags.owndata)
```

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

 [[200000 200000 200000 200000]
  [200000 200000 200000 200000]
  [200000 200000 200000 200000]]]
True
```

Indicizzazione con array di interi

Indicizzazione di array monodimensionali

per indicizzare un array 1D come a

In [10]:
```
a=arange(15)**2
print(a)
```

```
[  0   1   4   9  16  25  36  49  64  81 100 121 144 169 196]
```

con array di indici monodimensionali

posso usare un array 1D ed ottengo un array 1D come risultato

In [11]:
```
i= np.array([2,2,5,7,14])#NB elementi ripetuti
print(a[i])
```

```
[  4   4  25  49 196]
```

con array di indici di shape generica

più in generale il risultato dell'indicizzazione ha la shape dell'array di indici utilizzato

In [12]:
```
i=np.array([[1,3],[1,6],[14,14]])
print(a[i])
```

```
[[  1   9]
 [  1  36]
 [196 196]]
```

Indicizzazione di array con più dimensioni

consideriamo come array da indicizzare il seguente

In [6]:
```
a=np.arange(5*4).reshape(5,4)
a
```

Out[6]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

Con un singolo array di indici

singolo array di indici monodimensionale

se indicizzo a con un array monodimensionale questo viene interpretato come un array di indici che si riferiscono alla prima dimensione di a, in questo caso le righe

indicizzo due volte la seconda riga:

In [8]:
```
a[np.array([1,1])]
```

Out[8]:
```
array([[4, 5, 6, 7],
       [4, 5, 6, 7]])
```

singolo array degli indici con più dimensioni

indicizzo tre volte la seconda riga e tre volte la terza e le metto in un array bidimensionale di due colonne

In [12]:
```
b = a[np.array([  [1,1,1], [2,2,2] ])]
b
```

Out[12]:
```
array([[[ 4,  5,  6,  7],
        [ 4,  5,  6,  7],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [ 8,  9, 10, 11],
        [ 8,  9, 10, 11]]])
```

ricapitolando, passo un indice che ha shape(2,3):

In [18]:
```
c = np.array([  [1,1,1], [2,2,2] ])
c.shape
```

Out[18]:
```
(2, 3)
```

ciascun indice intero si riferisce alla prima dimensione di un array che ha shape

In [19]:
```
a.shape
```

Out[19]:
```
(5, 4)
```

ciascuna riga (prima dimensione) di a ha quindi 4 elementi (colonne) e quindi il risultato dell'indicizzazione avrà come shape

In [20]:
```
b.shape
```

Out[20]:
```
(2, 3, 4)
```

Con array di indici passati per ogni dimensione dell'array da indicizzare

riferiamoci sempre ad un array bidimensionale di 5x4 elementi

In [21]:
```
a
```

Out[21]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

estraggo il terzo ed il quinto elemento della seconda colonna (NB indicizzo con array distinti)

In [23]:
```
a[np.array([2,4]),np.array([1])]
```

Out[23]:
```
array([ 9, 17])
```

gli array per le diverse dimensioni devono essere array distinti NON integrati in un altro array che verrebbe interpretato come relativo alla sola prima dimensione (vedi paragrafi precedenti):

In [34]:
```
a[np.array([np.array([2,4]),np.array([1,1])])]
```

Out[34]:
```
array([[[ 8,  9, 10, 11],
        [16, 17, 18, 19]],

       [[ 4,  5,  6,  7],
```

```
       [ 4,  5,  6,  7]]])
```

sopra ho estratto la terza e la quinta riga e due volte la seconda riga ....

Indicizzazione con array di interi ed assegnamento

Anche questo tipo di indicizzazione può essere usata per l'assegnamento di nuovi valori agli elementi dell'array, ma attenzione agli indici ripetuti:

In [63]:
```
a = np.arange(10)
```

In [64]:
```
listaindici = [2  ,2   ,3  ,3   ,4  , 4]
listavalori = [20 ,200 ,30 ,300 ,40 , 400]
a[listaindici] = listavalori
a
```

Out[64]:
```
array([  0,   1, 200, 300, 400,   5,   6,   7,   8,   9])
```

... i valori della lista dei valori sono assegnati all'array indicizzato da sinistra a destra ...

Fancy indexing -> copia

a differenza che gli slice, il Fancy indexing restituisce delle copie:

sia nel caso di indici interi

In [65]:
```
b=a[listaindici]
b.flags.owndata
```

Out[65]:
```
True
```

sia nel caso di indicizzazione con maschere

In [66]:
```
b=a[a>100]
b.flags.owndata
```

Out[66]:
```
True
```

In [67]:
```
a
```

Out[67]:
```
array([  0,   1, 200, 300, 400,   5,   6,   7,   8,   9])
```

Fancy indexing + elaborazione + riassegnamento

Le modifiche effettuate sugli elementi di b non hanno effetto sugli elementi di a.

Se voglio manipolare e riassegnare i valori indicizzati devo quindi procedere in più passaggi:

In [72]:
```
np.set_printoptions(threshold=50)
a=np.random.randint(0,200,(7,7))
print(a)
```

```
[[124  13 137 190 172 175 179]
 [ 79   2  20  47  93  99 166]
 [165  62   6  86 199  20 161]
 [  7 140  42 128  19 145 194]
 [  7  91  95  51 105  39  63]
 [146 178  62  83 134 116 122]
 [  3 140  91  42  25 103   9]]
```

In [76]:
```
#indicizzo gli elementi di interesse e creo un array che li contiene
i=a>100
print('i=',i)
b=a[i]
```

```
print('-'*20)
print('b=',b)
```

```
i= [[ True False  True  True  True  True  True]
 [False False False False False False  True]
 [ True False False False  True False  True]
 [False  True False  True False  True  True]
 [False False False False  True False False]
 [ True  True False False  True  True  True]
 [False  True False False False  True False]]
--------------------
b= [124 137 190 172 175 179 166 165 199 161 140 128 145 194 105 146 178 134
 116 122 140 103]
```

In [77]:
```
#elaboro
b = -b**2
print('b=',b)
```

```
b= [-15376 -18769 -36100 -29584 -30625 -32041 -27556 -27225 -39601 -25921
 -19600 -16384 -21025 -37636 -11025 -21316 -31684 -17956 -13456 -14884
 -19600 -10609]
```

In [78]:
```
#riassegno
a[i]=b
print('a=',a)
```

```
a= [[-15376     13 -18769 -36100 -29584 -30625 -32041]
 [    79      2     20     47     93     99 -27556]
 [-27225     62      6     86 -39601     20 -25921]
 [     7 -19600     42 -16384     19 -21025 -37636]
 [     7     91     95     51 -11025     39     63]
 [-21316 -31684     62     83 -17956 -13456 -14884]
 [     3 -19600     91     42     25 -10609      9]]
```

Esercizio: integral approximation

- Scrivere una funzione $f(a, b, c)$ che restituisce $a^b - c$.
- Scrivere una (o più) funzione di integrazione tale che:
  - accetta come parametro una funzione f come la precedente.
  - accetta come parametro tre numeri interi $n0, n1, n2$
  - genera (o comunque si riconduce a, vedi note) un array con shape $(n0, n1, n2)$ che contiene i valori $f(a_i, b_j, c_k)$ ottenuti calcolando la funzione f su i punti (a_i, b_j, c_k) che formano una griglia regolare nel cubo unitario [0,1] x [0,1] x [0,1]
  - determina l'integrale $\int_0^1 \int_0^1 \int_0^1 (a^b - c)\, da\, db\, dc$ approssimandolo come media dei valori dell'array di cui al punto precedente.

NOTE:

- Il risultato esatto è $log(2) - \frac{1}{2}$ valutare l'entità dell'approssimazione al crescere del numero dei punti, secondo lo schema fornito nello script fornito come 'traccia'
- provare a generare una griglia completa di punti (a_i, b_j, c_k) (manualmente o con numpy.mgrid
- provare a sfruttare il broadcasting (manualmente o con numpy.ogrid)
- valutare anche numpy.ogrid + broadcast_arrays
- Nel caso si implementino più versioni verificarne e confrontarne i tempi di esecuzione (vedi script)