Algebra lineare in Python con NumPy

Programmazione di Applicazioni Data Intensive

Laurea in Ingegneria e Scienze Informatiche
DISI – Università di Bologna, Cesena

Proff. Gianluca Moro, Roberto Pasolini
nome.*cognome*@unibo.it

Outline

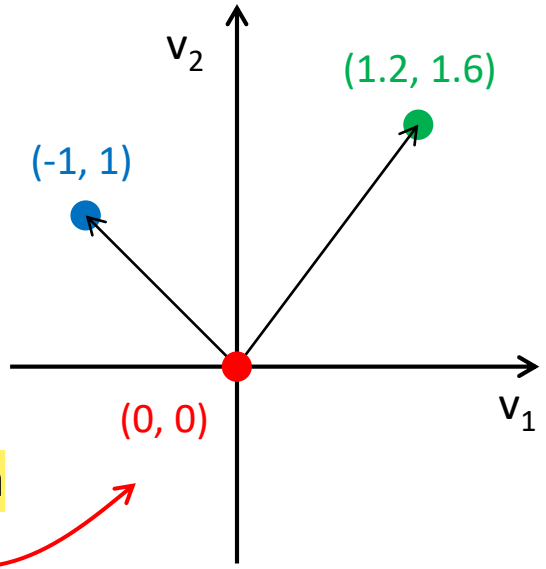
- Richiami di algebra lineare: vettori, matrici, operazioni di base
- Perché l'algebra lineare in data science, machine learning e deep learning?
- Introduzione a NumPy ed agli array
- Creazione di array
- Selezione di elementi dagli array
- Operazioni sugli array
- Funzionalità di algebra lineare

Richiami di Algebra Lineare: Vettori

- L'***algebra lineare*** studia vettori, matrici, spazi vettoriali, sistemi di equazioni lineari e trasformazioni lineari
- Un ***vettore*** è una tupla di *n* numeri (*componenti*) reali

  $\mathbf{v} = (v_1, \ldots, v_n)$ es. (12, 4.5, 2.1)

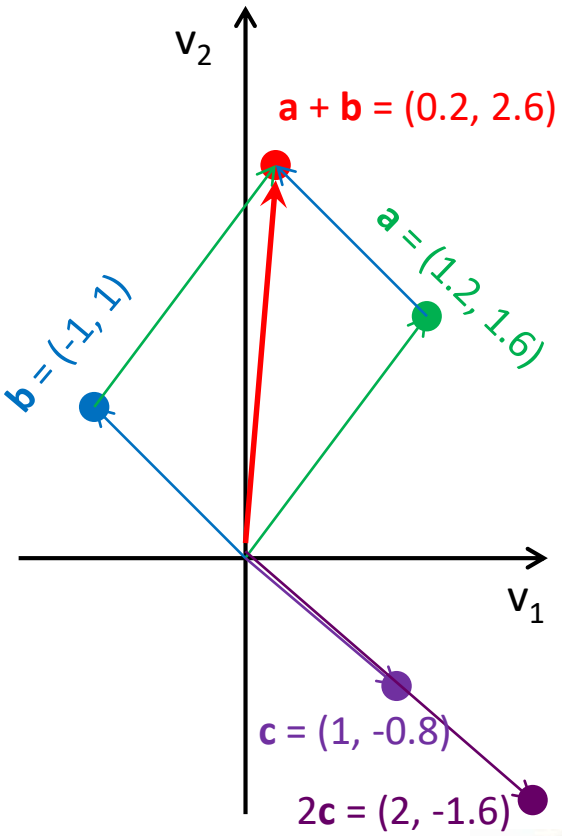
- Geometricamente le componenti di un vettore sono le coordinate di un punto in uno spazio *N*-dimensionale
  - nella figura sono visualizzati esempi di vettori a 2 valori in un piano 2D con origine in 0,0
  - direzione di **v** è data dalla retta passante per gli estremi, il verso dall'orientamento della freccia
  - Il vettore nullo ha tutte le componenti a 0
  - il modulo di **v** è la lunghezza, il vettore unitario ha modulo 1



Richiami di Algebra Lineare: Operazioni di Base sui Vettori

- La somma di due vettori di pari dimensioni è data dal vettore con le componenti sommate una ad una
  - es. (1.2, 1.6) + (-1, 1)
    = (1.2-1, 1.6+1)
    = (0.2, 2.6)

- Il prodotto tra uno *scalare* x (un numero singolo) ed un vettore è dato dal vettore con le componenti moltiplicate per x
  - es. $2 \cdot (1, -0.8) = (2 \cdot 1, -2 \cdot 0.8) = (2, -1.6)$
    $-1 \cdot (1, -0.8) = (-1, 0.8)$

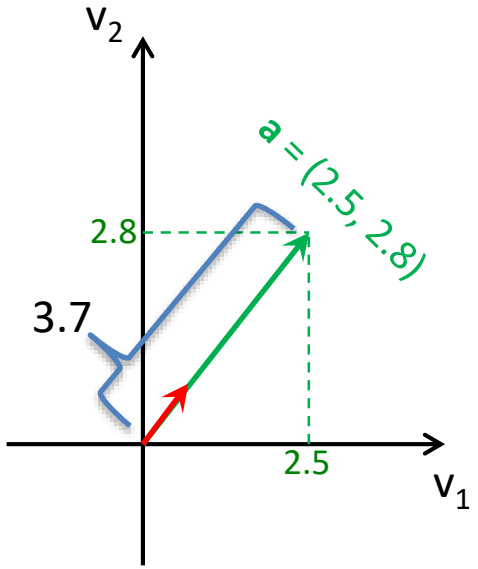


Norma Euclidea di un Vettore

- La norma euclidea di un vettore **a** (detta anche norma 2) è la radice quadrata della somma dei quadrati delle sue componenti

$$\|\mathbf{a}\| = \sqrt[2]{\sum_{i=1}^{n} a_i^2}$$

Sia la norma 2 (o norma euclidea) che il modulo di un vettore misurano la lunghezza del vettore



- e.g. *sia* $\mathbf{a} = (2.5,\ 2.8)$

$$\|\mathbf{a}\| = \sqrt[2]{2.5^2 + 2.8^2} = 3.7$$

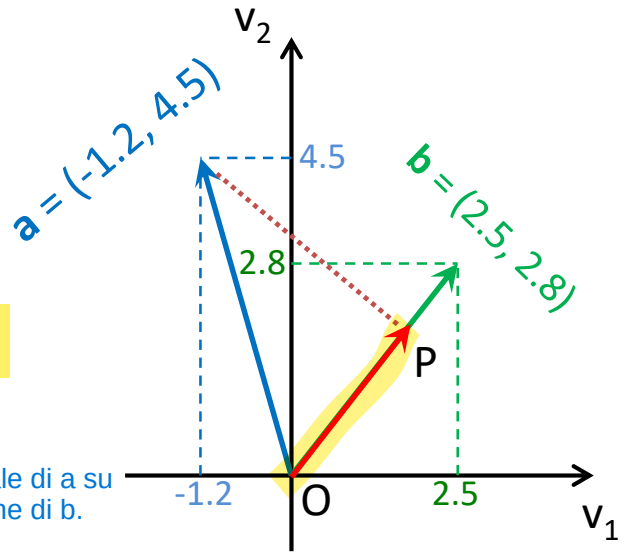
- la norma 2 è la lunghezza di **a** ed equivale ad applicare il teorema di Pitagora alle componenti
- con $\frac{\mathbf{a}}{\|\mathbf{a}\|}$ si ottiene il vettore unitario di **a** che ha norma 1 (0.67, 0.75)

Richiami di Algebra Lineare: Prodotto Scalare tra due Vettori (i)

- Il *prodotto scalare* (*dot product*) tra due vettori a, b di dimensioni *n* è la somma dei prodotti delle rispettive coppie di componenti

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i \cdot b_i$$

- e.g. **a** = (-1.2, 4.5), **b** = (2.5, 2.8),
  a·b =-1.2 x 2.5 + 4.5 x 2.8 = 9.6
- **a·b** geometricamente è indicatore della lunghezza della proiezione ortogonale OP di **a** su **b**



Questa quantità è la lunghezza della proiezione ortogonale di a su b, chiamata anche componente scalare di a nella direzione di b.

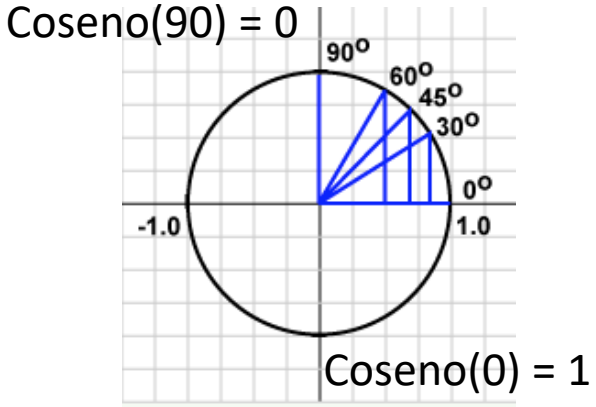
  - minore è l'angolo tra loro, maggiore è la lunghezza della proiezione ortogonale

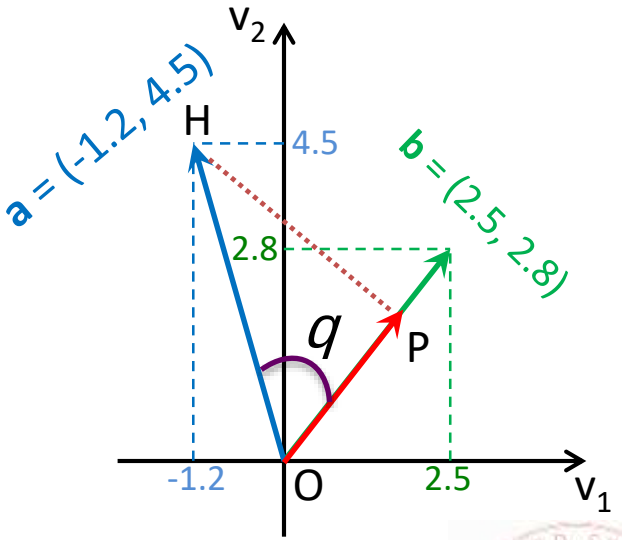
Richiami di Algebra Lineare: Prodotto Scalare tra due Vettori (ii)

- **a·b** è indicatore della lunghezza della proiezione ortogonale OP di **a** su **b**, perché ?
- sia q l'angolo tra due vettori che formano un triangolo, dalla geometria sappiamo che $OP = OH \times \text{coseno}(q)$
- cioè $\overline{OP} = \|\mathbf{a}\| \times \text{coseno}(q)$
- ma il prodotto scalare tra i vettori a, b è

$$\mathbf{a} \times \mathbf{b} = \|\mathbf{a}\| \times \|\mathbf{b}\| \times \text{coseno}(q) = \|\mathbf{b}\| \times OP$$

- perciò **a·b** è la proiezione ortogonale di **a** su **b**, scalata rispetto alla lunghezza di **b**





- maggiore è il prodotto scalare, più due vettori sono simili

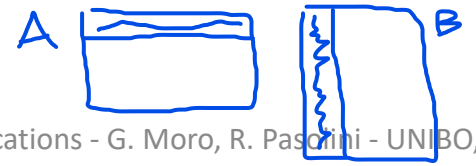
il prodotto scalare è massimo quando i due vettori puntano nella stessa direzione, e si riduce (fino a diventare negativo) man mano che si allontanano da quella direzione.

Richiami di Algebra Lineare: Matrici

- Una ***matrice*** *m*×*n* è una tabella di numeri reali con *m* righe e *n* colonne, e.g. 2x3

$$A_{2,3} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \quad \text{es.:} \begin{bmatrix} 2.1 & -4 & 2.4 \\ -3.2 & 1.1 & 5 \end{bmatrix}$$

  - Ogni riga ed ogni colonna è rappresentabile come un vettore
  - Una matrice con una sola riga (1×*n*) o una sola colonna (*m*×1) è detta rispettivamente *matrice riga* o *matrice colonna*
  - Una matrice è *nulla* se tutti gli elementi sono 0
  - La somma tra matrici di pari dimensioni ed il prodotto tra scalari e matrici avviene per singoli valori, come per i vettori operazioni note anche come *element wise operation*
- Il prodotto tra due matrici $A_{m,p}$ x $B_{p,n}$ è una matrice *m*x*n*
  - dove la cella *i,j* è data dal prodotto scalare tra il vettore riga *i* di A con il vettore colonna *j* di B



Perché Vettori e Matrici ?

I dati reali sono modellati con Vettori e Matrici

- Utenti x Prodotti
  - ogni riga è un vettore utente e le componenti > 0 sono i prodotti che ha acquistato (o recensito)

	Prod1	Prod2	Prod3	Prod4	Prod5
Carl	0	1	0	4	0
Mike	0	1	2	0	3
Jake	0	3	0	3	5
Tom	3	3	1	5	4

- Auto: ogni vettore è un'auto con le proprie caratteristiche
  - Costi di manutenzione dell'anno al tempo t
  - Costi di manutenzione dell'anno al tempo t + 1 VARIABILE DA PREDIRE
  - Costi di assicurazione, Costi di proprietà (bollo), Costi totali di rifornimento di carburante, etc. etc.
- Abitazioni: ogni vettore è un'abitazione con proprie variabili
  - mq, vetustà, quartiere, stanze, bagni, piano
  - prezzo di vendita VARIABILE DA PREDIRE

NumPy

- ***NumPy*** è una libreria Python di uso comune per la gestione di array ad *N* dimensioni e l'algebra lineare
- Molte operazioni sugli array usano la normale sintassi Python
  - indicizzazione con `[…]`, operatori `+`, `−`, `*`, ...
- Le operazioni con oggetti NumPy sono spesso 10+ volte più veloci rispetto all'uso di oggetti standard Python (es. liste)
- Array e funzioni di NumPy sono usati da molte altre librerie
- Tutte le funzionalità sono contenute nel package `numpy`, convenzionalmente importato col nome "`np`"

```
>>> import numpy as np
```

N-Dimensional Array

ndarray

- `ndarray` è un array a *N* dimensioni di valori dello stesso tipo
- Ogni `ndarray` ha come attributi fondamentali:
  - il tipo di valori che contiene
  - il numero di dimensioni e il numero di elementi lungo ogni dimensione
- I valori in un `ndarray` sono mutabili, ma **non** è possibile cambiare il tipo e il numero totale di valori
- NumPy supporta determinati tipi di valori, di cui sono comuni:
  - numeri reali a 8 byte (`np.float64`)
  - numeri interi a 8 byte (`np.int64`)
- Vettori e matrici sono rappresentati da array con valori numerici, rispettivamente a 1 e 2 dimensioni
  - si possono creare array a più di 2 dimensioni (noti anche come *tensori*)

Attributi dei `ndarray`

Gli attributi di un oggetto `ndarray` includono

- `ndim`: numero di dimensioni (dette anche *assi*)
  - ad es. 1 per i vettori e 2 per le matrici
- `shape`: *forma* dell'array, una tupla che indica il numero di valori lungo ciascuna dimensione
  - ad es. `(4, )` per un vettore con 4 elementi
    `(2, 3)` per una matrice 2×3 (2 righe e 3 colonne)
  - per qualsiasi array `x`: `len(x.shape) == x.ndim`
- `size`: numero totale di valori
  - pari al prodotto tra gli elementi di `shape`
- `dtype`: tipo dei valori contenuti

Creare un ndarray

- La funzione `array` crea un `ndarray` da una sequenza (es. lista)
- Il tipo dell'array è di default determinato dai valori contenuti
  - con soli numeri interi, il tipo è `int64` o `int32`
  - se almeno un numero è decimale, il tipo è `float64`
  - è possibile specificare un tipo con l'argomento opzionale `dtype`
  - il metodo `astype` restituisce una copia dell'array cambiando il dtype

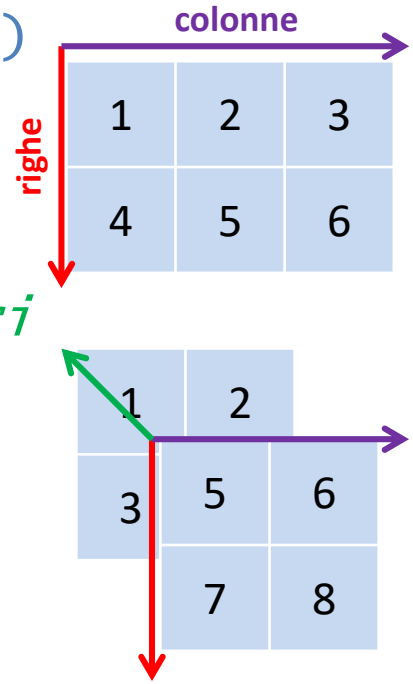
```
>>> x = np.array([1, 2.5, 5])
>>> x          # stampa i contenuti dell'array
array([ 1. ,  2.5,  5. ])
>>> x.dtype    # tipo di dati nell'array
dtype('float64')
>>> x.ndim     # numero di dimensioni
1
>>> x.shape    # numero di valori per dimensione
(3,)
```

1	2.5	5

Creare un ndarray Multidimensionale

- Con liste innestate, si creano matrici e array a più dimensioni

```
>>> x = np.array([ [1, 2, 3], [4, 5, 6] ])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.dtype   # tutti numeri int -> interi
dtype('int64')
>>> x.ndim, x.size
2, 6
>>> x.shape   # matrice 2x3
(2, 3)
>>> y = np.array([[[1,2], [3,4]], [[5,6], [7,8]]])
>>> y.shape
(2, 2, 2)
```



Array con Valori Costanti

- La funzione `full` crea un array di forma indicata con tutti i valori inizializzati a quello dato
  - è possibile specificare un `dtype`, se non indicato è dedotto dal valore

```
>>> np.full((3, ), 7)  # vettore di 3 valori tutti 7
array([7, 7, 7])
```

- Per valori 0 o 1 si possono usare le funzioni `zeros` e `ones`

```
>>> np.zeros((2, 3))   # matrice 2x3 di zeri
array([[0., 0., 0.],
       [0., 0., 0.]])
```

- La funzione `empty` crea un array di forma indicata senza inizializzarlo, lasciando dati "dirty" presenti in memoria
  - utilizzabile se **tutti** i valori saranno poi impostati manualmente

```
>>> data = np.empty((2, 3))
array([[ -1.72723371e-077,  -4.33230621e-311,   9.64734874e-315],
       [  4.89584003e-085,   4.89209922e-085,   4.80427054e-309]])
```

Vettori con Intervalli di Valori

- La funzione `arange` crea un vettore di numeri a intervalli regolari in modo analogo alla funzione `range` di Python

```
>>> np.arange(5)   # da 0 (implicito) a 5 escluso
array([0, 1, 2, 3, 4])
>>> np.arange(10, 20, 2.5) # da 10 a 20 escluso con passo 2,5
array([10. , 12.5, 15. , 17.5])
```

- La funzione `linspace` è simile, ma fa specificare la lunghezza desiderata del vettore piuttosto che l'incremento (cioè quanti elementi voglio di quel intervallo)

```
>>> np.linspace(0, 1, 4)   # 4 valori tra 0 e 1
array([0., 0.3333333, 0.6666667, 1.])
```

Crea una sequenza di num valori equidistanti in scala logaritmica tra start e stop.

- `geomspace` è simile a `linspace`, ma fa una progressione esponenziale invece che lineare (c'è anche `logspace`)

```
>>> np.geomspace(1, 1000, 4)
array([1., 10., 100., 1000.])
```

Qui ogni valore è 10 volte il precedente, allora progressione esponenziale.

Array con Valori Casuali

- `np.random` offre funzioni per generare array di valori casuali
- La funzione `seed` imposta il seed per le chiamate successive

```
>>> np.random.seed(42)
```

- `random` restituisce valori distribuiti uniformemente in [0, 1)

```
>>> np.random.random((2, 3))   # matrice casuale 2x3
array([[0.31637396, 0.5995927 , 0.19594632],
       [0.04312813, 0.84134461, 0.5603397 ]])
```

- `randint` restituisce valori interi tra a (incluso) e b (escluso)

```
>>> np.random.randint(1, 7, 3) # min 1, max 6, 3 valori
array([6, 1, 4])
```

- `binomial, normal, geometric` ...valori da varie distribuzioni
  - in queste funzioni è possibile specificare parametri (es. media e dev. standard per `normal`)

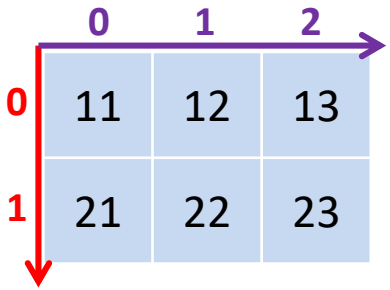
Accesso agli Elementi di un Array

- L'accesso ad elementi di un vettore funziona come per le liste
  - si può anche qui usare -*n* per accedere all'*n*-ultimo elemento

```
>>> x = np.array([10, 20, 30, 40, 50])
>>> x[1], x[-2]   # secondo e penultimo elemento
20, 40
```

- Per array a più dimensioni, va indicato un indice per ciascuna

```
>>> y = np.array([ [11, 12, 13], [21, 22, 23] ])
>>> y[0, 2]   # prima riga, terza colonna
13
```



- Si può assegnare un valore ad un elemento

```
>>> y[0, 2] = 100
>>> y
array([[ 11,  12, 100],
       [ 21,  22,  23]])
```

Accesso a Intervalli di Vettori (*slicing*)

- Come per le liste, si possono estrarre intervalli di vettori

```
>>> x = np.array([10, 20, 30, 40, 50])
>>> y = x[2:4]
>>> y
array([30, 40])
```

- Si possono usare tutte le varianti di intervallo ammesse per le liste: indici negativi, inizio/fine impliciti e selezionare ogni N

```
>>> x[:3]      # primi 3 elementi
array([10, 20, 30])
>>> x[-3:]     # ultimi 3 elementi
array([30, 40, 50])
>>> x[1::2] # dal 2° ogni 2 elementi (indice dispari)
array([20, 40])
```

Viste degli Array

- Un array ottenuto selezionando un intervallo di *x* è una ***vista*** dell'array *x*, che condivide con esso la memoria usata
  - per questo qualsiasi modifica su uno ha effetto anche sull'altro
  - da una vista si può risalire all'array originale con l'attributo `base`
  - si può ottenere una copia indipendente dalla base col metodo `copy`

```
>>> x = np.array([10, 20, 30, 40, 50])
>>> y = x[2:4]
>>> y[1] = 100
>>> x                    → array([10, 20, 30, 100, 50])
>>> y.base is x          → True
```

- Altre operazioni (es. conversioni di tipo) per cui non è possibile creare una vista restituiscono già una copia
  - se si tratta di una copia, l'attributo `base` è `None`

Intervalli di Array Multidimensionali

- Gli intervalli si usano allo stesso modo in array a N dimensioni

```
>>> x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> x[:2, 1:3]   # prime due righe, 2a e 3a colonna
array([[2, 3], [5, 6]])
```

1	2	3
4	5	6
7	8	9

- "**:**" indica tutti gli indici lungo una dimensione

```
>>> x[:, :2]     # tutte le righe, prime 2 colonne
array([[1, 2], [4, 5], [7, 8]])
```

1	2	3
4	5	6
7	8	9

- Gli elementi "**:**" più a destra si possono omettere

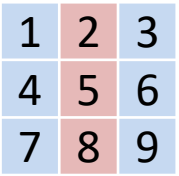
```
>>> x[-2:, :]    # ultime 2 righe (tutte le colonne)
array([[4, 5, 6], [7, 8, 9]])
```

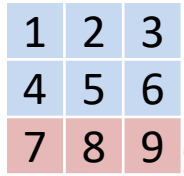
1	2	3
4	5	6
7	8	9

Vettori Riga e Colonna

- Se invece di un intervallo si seleziona un singolo indice di una dimensione, questa viene rimossa dall'array risultante
- Ad esempio, selezionando una singola riga o una singola colonna di una matrice (2D), otteniamo un vettore (1D)

```
>>> x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> x.shape
(3, 3)
>>> x[:, 1]   # tutte le righe, seconda colonna
array([2, 5, 8])
>>> x[:, 1].shape   # è un vettore, non una matrice!
(3, )
>>> x[-1]   # ultima riga (tutte le colonne)
array([7, 8, 9])
```





Selezione di Elementi con Lista di Indici

- Al posto di un indice singolo o un intervallo è possibile passare una lista o un vettore di interi come indici

```
>>> x = np.array([ [1, 2, 3, 4], [5, 6, 7, 8] ])
>>> x[:, [0, 2]]   # tutte le righe, colonne 0 e 2
array([[1, 3],
       [5, 7]])
```

1	2	3	4
5	6	7	8

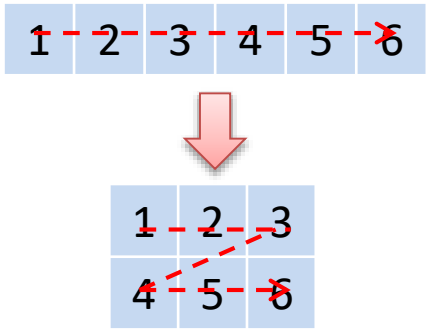
- Passando molteplici sequenze di *N* interi come indici otteniamo un vettore con *N* elementi selezionati
  - al contrario degli intervalli, questo metodo di selezione restituisce sempre array con copie dei dati

```
>>> x[[0, 1], [2, 3]]   # vettore [ x[0,2], x[1,3] ]
array([3, 8])
```

1	2	3	4
5	6	7	8

Cambiare la Forma di un Array (1)

- La forma di un array può essere cambiata in una che preveda lo stesso numero complessivo di elementi
- L'ordine dei valori rimane invariato iterando le dimensioni dalla prima (esterna) all'ultima (interna)
  - ad es. in matrici i valori sono letti riga per riga, colonna per colonna
- Il metodo `reshape` crea una matrice di forma diversa ma con gli stessi dati
  - `reshape` può restituire una vista o una copia in base alla contiguità o meno nella memoria fisica




```
>>> x = np.array([1, 2, 3, 4, 5, 6])
>>> x.reshape((2, 3))
array([[1, 2, 3],
       [4, 5, 6]])
```

Stesso numero di elementi, ma in 2 righe e 3 colonne

Cambiare la Forma di un Array (2)

- Nella tupla passata a `reshape` è possibile inserire un `-1` (non più d'uno) per determinare automaticamente la lunghezza di un asse in base al numero totale di elementi

```
>>> x = np.array([[1, 2, 3],
                  [4, 5, 6]])
>>> x.reshape((-1, 2))  →  array([[1, 2],
                                  [3, 4],
                                  [5, 6]])
```

x·2 = 2·3 = 6 → x=3

- Il metodo `ravel` riforma qualsiasi array ad un vettore 1D
  - se riesce a preservare la continguità in memoria non crea una copia
  - per forzare la creazione di una copia, usare invece `x.flatten()`
  - in pratica `x.ravel() == x.reshape((-1, ))`

```
>>> x.ravel()  →  array([1, 2, 3, 4, 5, 6])
```

Conversione da Array ad Oggetti Python

strutture dati di python (list, tuple, set, dictionary)

- I `ndarray` possono essere trattati come collezioni Python
  - possono essere iterati con `for`, convertiti in liste o altri tipi, …
- Iterando un vettore, si ottengono i suoi valori in sequenza

```
>>> x = np.array([1, 2, 3])
>>> list(x)
[1, 2, 3]
```

un vettore (1D) è convertito in una lista di valori scalari (0D)

- Iterando una matrice, se ne ottengono le singole righe

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> list(x)
[array([1, 2, 3]), array([4, 5, 6])]
```

una matrice (2D) è convertita in una lista di vettori (1D)

  - si possono ottenere i singoli valori iterando su `x.flat`
- In generale: `list(x) == [x[0], x[1], …]`

Concatenazione di Array

- Si può usare `concatenate` per concatenare vettori tra loro

```
>>> np.concatenate([np.arange(3), np.arange(4)])
array([0, 1, 2, 0, 1, 2, 3])
```

- Se si concatenano matrici, si può specificare lungo quale dimensione vanno concatenate (default = 0, righe)

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
```

1	2
3	4

5	6
7	8

```
>>> np.concatenate([a, b])
array([[1, 2], [3, 4], [5, 6], [7, 8]])
```

1	2
3	4
5	6
7	8

```
>>> np.concatenate([a, b], 1)
array([[1, 2, 5, 6], [3, 4, 7, 8]])
```

1	2	5	6
3	4	7	8

Operazioni tra Componenti di Array

- Le operazioni aritmetiche tra numeri (+, -, *, …) sono definite anche tra ndarray
- Operazioni tra array di pari forma son sempre eseguite elemento per elemento
  - es. con x e y array e z=x+y si ha z[i]==x[i]+y[i] per ogni indice i
  - lo stesso vale per -, *, /, …
- Con l'operatore + si esegue quindi la somma canonica tra vettori o tra matrici

```
>>> x = np.array([[1, 2],
                  [3, 4]])
>>> y = np.array([[2, 3],
                  [4, 5]])
>>> x + y
array([[3, 5],
       [7, 9]])
>>> x * y
array([[ 2,  6],
       [12, 20]])
```

Operazioni tra Array e Scalari

- Le stesse operazioni base si possono anche applicare tra un `ndarray` ed un valore singolo (*scalare*)
- In questo caso, si ottiene un array della stessa forma di quello usato, col risultato dell'operazione applicata a ciascun elemento
- Con l'operatore `*` si ottiene quindi il prodotto canonico tra scalare e vettore

```
>>> x = np.array([[1, 2],
                  [4, 8]])
>>> x ** 2 # potenza di 2
array([[ 1,  4],
       [16, 64]])
>>> 1 / x
array([[1.   , 0.5  ],
       [0.25 , 0.125]])
```

Array Binari e Booleani

- Esistono diversi casi in cui si usano vettori e matrici *binari* in cui tutti i valori sono 0 o 1
  - ad es. nell'e-commerce possiamo codificare un ordine con un vettore di 1 e 0 per ogni prodotto acquistato e non tra quelli esistenti
  - per indicare un'opzione tra *N* possibili è comune usare un vettore binario con *N* elementi di cui uno solo è 1 (detto vettore *one-hot*)
- Possono essere rappresentati in NumPy da array di tipo `bool`
  - gli 1 sono rappresentati come `True` e gli 0 come `False`

```
x = np.array([True, False, True, False])
```

- Con `astype`, si può convertire tra valori `bool` e `int`
  - `astype` restituisce sempre una copia dell'array

```
>>> x.astype(np.int)  →  array([1, 0, 1, 0])
```

Selezione di Elementi con Array Booleani

- Usando un array di `bool` come indice di un array *x*, si ottiene una copia di *x* con i soli elementi corrispondenti a valori `True`

```
>>> x =     np.array([  -1,    2,    3,   -4])
>>> flags = np.array([ True, False,  True, False])
>>> x[flags]   # estraggo da x valori True in flags
array([-1, 3])
```

- Un array di `bool` può essere ottenuto dagli stessi array, in modo da selezionare elementi che soddisfino una condizione

```
>>> x > 0      # quali valori di x sono positivi?
array([False,  True,  True, False])
>>> x[x > 0]   # estraggo da x i valori positivi
array([2, 3])
```

Operatori su Array Booleani

- Gli array booleani supportano gli operatori &, |, ^ e ~ per calcolare AND, OR, XOR e NOT elemento per elemento
  - si possono applicare anche ad array di interi per operazioni bit a bit

```
>>> np.array([False, True]) & np.array([True, True])
array([False,  True])
>>> ~np.array([True, True, False])
array([False, False,  True])
```

- Si possono usare per selezionare valori di array componendo diverse condizioni

```
>>> x = np.array([21, 13, 8, 18, 25, 14])
>>> x[(x >= 10) & (x <= 20)]  # elementi tra 10 e 20
array([13, 18, 14])
```

In espressioni come questa **le condizioni devono essere scritte tra parentesi** per applicare la corretta precedenza tra gli operatori

Operazioni tra Array di Forme Diverse

- Applicando un'operazione tra due array di forma diversa, NumPy tenta di conformarli per eseguirla (*broadcasting*)

1. Se gli array hanno un numero di dimensioni diverse allora a quello che ne ha meno aggiunge dimensioni di lunghezza 1
   - es. un vettore con *n* elementi è conformabile ad una matrice 1×*n*
2. Per ciascuna dimensione, se ha lunghezza 1 in un array e *n* nell'altro, i valori del primo sono ripetuti *n* volte
   - es. una matrice 1×*n*, per essere sommata ad una *m*×*n*, è convertita anch'essa ad una *m*×*n* copiando *m* volte la stessa riga di *n* elementi

- Se gli array non sono conformabili secondo queste regole allora l'operazione produce errore

Il broadcasting in NumPy è il meccanismo che consente di eseguire operazioni tra array di forme diverse "estendendo" automaticamente (senza copiare i dati) gli array più piccoli lungo le dimensioni mancanti, in modo da allinearne le forme. In pratica, quando due array hanno un numero diverso di dimensioni, NumPy aggiunge automaticamente dimensioni di lunghezza 1 (virtuali) all'inizio delle forme più piccole e replica i valori lungo quegli assi per renderli compatibili con l'altro array. Se poi le dimensioni risultanti sono uguali o pari a 1, l'operazione viene eseguita correttamente

Esempio di Broadcasting

- Ad esempio, come è calcolata la somma tra questi array?

```
>>> a = np.array([10, 20, 30])                 #   3
>>> b = np.array([ [1, 2, 3], [4, 5, 6] ])    # 2x3
```

1 • Ad a è aggiunta una dimensione per conformità con b
  - da vettore con 3 elementi a matrice 1×3

2 • a e b hanno rispettivamente 1 e 2 righe → l'unica riga di a è replicata 2 volte

10	20	30	+	1	2	3	
				4	5	6	
			=				

- a e b hanno entrambe 3 colonne → nessuna ulteriore azione è necessaria

10	20	30	+	1	2	3	
10	20	30		4	5	6	
			=				
11	22	33					
14	25	36					

```
>>> a + b
array([[11, 22, 33],
       [14, 25, 36]])
```

Aggiungere una Dimensione ad un Array

- Di un array si può ottenere una vista "espansa" con una dimensione aggiunta (o più d'una), mantenendone inalterati i valori
  - ad esempio un vettore (1D) può essere trasformato in una matrice (2D) con una singola riga o una singola colonna
  - questo può essere utile in operazioni tra array di forma diversa se il broadcasting non da il risultato voluto

- Per far ciò, usiamo la selezione ad intervalli aggiungendo None in corrispondenza della dimensione da aggiungere

```
>>> x = np.array([1,2,3])
>>> x[None, :]
array([[1, 2, 3]])
>>> x[:, None]
array([[1],
       [2],
       [3]])
```

aggiungo la prima dimensione (righe) -> ottengo una matrice riga 1x3

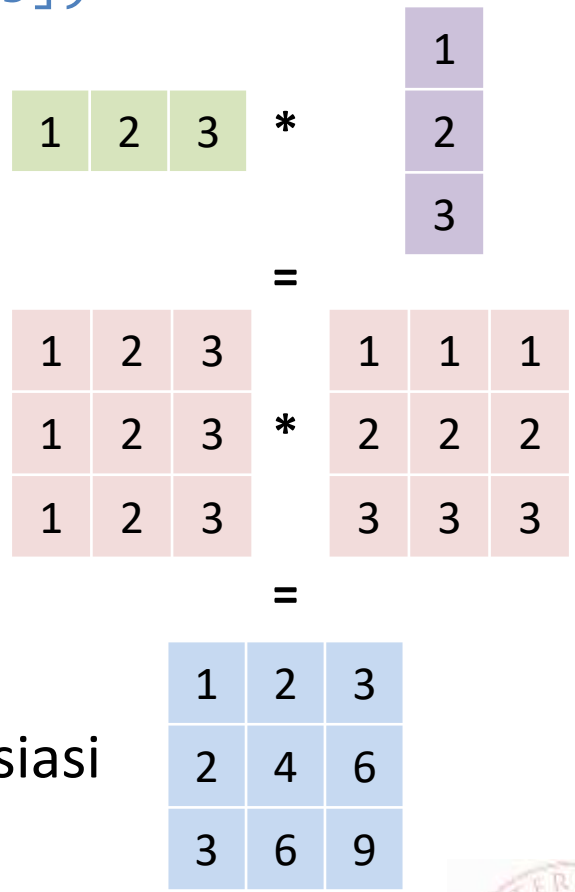
aggiungo la seconda dimensione (colonne) -> ottengo una matrice colonna 3x1

significa che, quando il comportamento automatico di broadcasting non allinea gli assi nel modo che serve per un'operazione elemento per elemento, possiamo forzare manualmente l'aggiunta di dimensioni di lunghezza 1 per garantire che gli array combacino esattamente lungo gli assi desiderati. Questo evita errori o risultati inattesi senza dover ricorrere a reshape più complessi o a copie dei dati.

Esempio di Operazione su Array Espansi

- Sia dato il vettore `x = np.array([1, 2, 3])`
- Come ottenere una matrice 3x3 **A** dove ogni elemento $a_{i,j}$ è il prodotto tra l'*i*-esimo e il *j*-esimo elemento di **x** ?
- Moltiplico **x** espanso a matrice riga (1x3) per **x** espanso a matrice colonna (3x1)

```
>>> x[None, :] * x[:, None]
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
```



- Lo stesso principio si può applicare su qualsiasi operazione, anche su due vettori differenti

Funzioni Universali

- Le *funzioni universali* di NumPy applicano una funzione a tutti gli elementi di uno o più array
  - un parametro – le funzioni unarie includono `abs` (valore assoluto), `sqrt` (radice quadrata), `log` (logaritmo), `floor` (arrotondamento per difetto), …
  - due parametri – le funzioni binarie includono `maximum` (elemento maggiore tra i due array element-wise), `minimum` (elemento minore tra i due array), …
    - se gli array hanno forme diverse, si applica il broadcasting

```
>>> np.abs(np.array([-1, 2, -3, 4]))
array([1, 2, 3, 4])
>>> np.maximum(np.array([[4, 3], [2, 1]]),
               np.array([[1, 2], [3, 4]]))
array([[4, 3],
       [3, 4]])
```

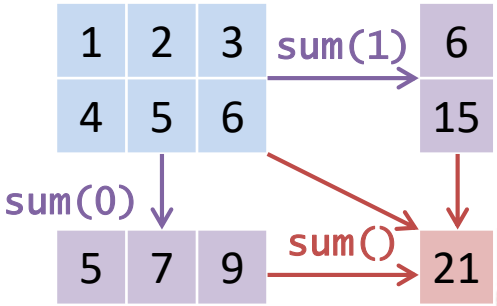
Funzioni Aggregate

- `ndarray` offre diversi metodi per ottenere statistiche aggregate dai valori dell'array: `min`, `max`, `sum`, `mean`, …
- Senza specificare argomenti, tutti i valori dell'array sono aggregati insieme in un unico valore scalare

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> x.sum()     # somma di tutti i valori
21
```

- In alternativa, si può aggregare lungo una dimensione
  - nelle matrici, 0 aggrega le righe e 1 le colonne

```
>>> x.sum(0) # somma delle righe
array([5, 7, 9])
>>> x.sum(1) # somma delle colonne
array([6, 15])
```



Lista di Funzioni Aggregate Comuni

- `sum` / `prod` / `mean` = somma / prodotto / media dei valori
- `cumsum` / `cumprod` = somma / prodotto cumulativi
  - es. cumsum([1, 2, 3]) = [1, 1+2, 1+2+3] = [1, 3, 6]
- `var` / `std` = varianza / deviazione standard
  - di default sulla popolazione (denominatore N), per la stima su un campione (denominatore N-1) specificare parametro `ddof=1`
- `min` / `max` = valore minimo / massimo
- `argmin` / `argmax` = indice del valore minimo / massimo
  - es. argmax([23, 8, 64, 12, 53, 23]) = 2
- `all` / `any` = `True` se tutti / almeno un elemento sono `True`
  - si ricordi che in Python sono `True` tutti i numeri diversi da 0

significa precisamente:
- Se non si specifica nulla, NumPy calcola varianza usando N elementi (varianza di popolazione).
- Se si sta lavorando con un campione di dati e si vuole stimare la varianza della popolazione da cui il campione proviene, si passa ddof=1 affinché il denominatore diventi N−1, evitando così di sottostimare la variabilità del campione

Ordinamento dei Valori di un Vettore

- La funzione `sort` applicata su un array ne restituisce uno nuovo con i valori in ordine crescente

```
>>> x = np.array([32, 8, 2, 4, 16, 64, 1])
>>> np.sort(x)
array([ 1,  2,  4,  8, 16, 32, 64])
```

- Il metodo `sort` riordina i valori di un vettore modificando il vettore stesso ("in place") x.sort()
- Il metodo `argsort` restituisce gli indici dell'array (da 0 a N-1) ordinati secondo l'ordine crescente dei valori in esso
  - per qualsiasi vettore `x` si ha `x[x.argsort()] == np.sort(x)`

```
>>> x.argsort()
array([6, 2, 3, 1, 4, 0, 5])
```

significa che l'elemento più piccolo dell'array è quello di indice 6, il secondo più piccolo è quello di indice 2 e così via

Algebra Lineare in NumPy

- Abbiamo visto come creare array per rappresentare vettori e matrici e come eseguire semplici operazioni su di essi
  - lettura / modifica di elementi singoli o multipli
  - applicazione di operazioni o funzioni su ciascun elemento
  - calcolo di statistiche (minimo, massimo, media, …)
- Abbiamo visto come svolgere operazioni di base tra vettori e matrici con gli operatori standard Python
  - la somma tra vettori o matrici di pari dimensioni `a` e `b` è data da `a+b`
  - il prodotto tra un vettore o matrice `a` e uno scalare `c` è dato da `c*a`
  - NB: il prodotto tra vettori o tra matrici è diverso! (lo vedremo a breve)
- Nel seguito vediamo operazioni più avanzate, come si svolgono in NumPy e come si possono usare nella pratica

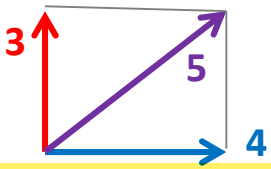
Norma di un Vettore

- La norma *euclidea* (o norma 2) di un vettore indica intuitivamente la sua "lunghezza"

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$$

- In NumPy si ottiene con la funzione `norm` del modulo `linalg`

```
>>> x = np.array([4, 3])
>>> np.linalg.norm(x)  →  5.0
```



$$\|\mathbf{x}\|_2 = \sqrt{4^2 + 3^2} = 5$$

- Passando una matrice `X`, di default viene calcolata la *norma di Frobenius*, equivalente alla norma di `X.flatten()`
- In alternativa, indicando `axis=0` o `axis=1`, il risultato è un vettore con le norme euclidee di ciascuna colonna o riga

```
>>> X = np.array([[ 4,  3],
                  [ 5, 12]])
>>> np.linalg.norm(X, axis=1)  →  array([5, 13])
```

Vettori Unitari e Normalizzazione

- Un *vettore unitario* (*versore*) è un vettore con norma 1
- Qualsiasi vettore non nullo può essere diviso per la sua norma per ottenerne uno *normalizzato* con norma 1
- Intuitivamente, con la normalizzazione si estrae la "direzione" di un vettore, eliminando l'informazione sulla lunghezza
  - ciò è utile ad esempio nell'analisi di testi, come vedremo
- In NumPy non c'è una funzione apposita, ma si può creare…

```
>>> def normalize(v):    # si assume v non sia nullo!
...     return v / np.linalg.norm(v)
>>> normalize(np.array([4, 3]))
array([0.8, 0.6])
>>> np.linalg.norm(normalize(np.array([4, 3])))
1.0
```

vettore di due dimensioni

norma 2 di [4,3] è 5
[4/5 e 3/5] è [0.8,0.6] (vettore normalizzato)

quindi il vettore normalizzato di [4,3] estrae la direzione riducendo la lunghezza ad 1

Prodotto Scalare tra Vettori

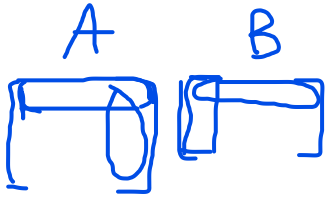
- Il *prodotto scalare* (*dot product*), definito tra vettori di pari lunghezza, è pari alla somma dei prodotti tra le componenti corrispondenti dei due vettori

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i \cdot b_i$$

  - in termini NumPy: `(a*b).sum()`
- In NumPy, si può eseguire indifferentemente tramite la funzione `dot`, il metodo `dot` degli `ndarray` o l'operatore `@`

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([4, 2, 1])
>>> np.dot(x, y)  → 11
>>> x.dot(y)      → 11
>>> x @ y         → 11
```

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} \cdot \begin{bmatrix}4 & 2 & 1\end{bmatrix} = 1 \cdot 4 + 2 \cdot 2 + 3 \cdot 1 = 11$$



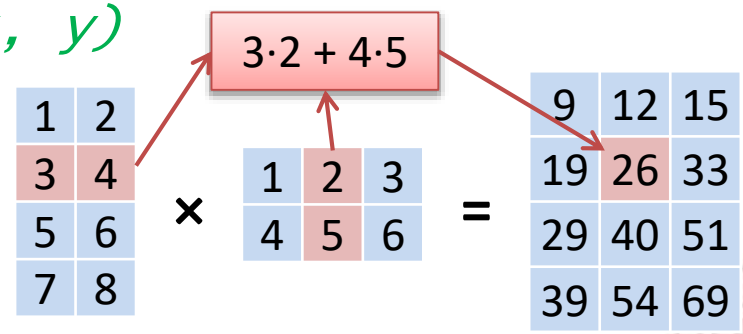
Prodotto tra Matrici

- Sia **A** una matrice *n*×*m* e **B** una matrice *m*×*p*
  - le colonne di **A** devono essere quante le righe di **B**

$$(\mathbf{AB})_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}$$

- Il prodotto **AB** è la matrice *n*×*p* in cui ciascun elemento *i*, *j* è il prodotto scalare tra la riga *i* di **A** e la colonna *j* di **B**
  - per il prodotto tra matrici **non** vale la proprietà commutativa (**AB** ≠ **BA**)
- In NumPy, si esegue anch'esso con `dot` o con `@`

```
>>> x = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> y = np.array([[1, 2, 3], [4, 5, 6]])
>>> x.dot(y) # oppure np.dot(x, y)
array([[ 9, 12, 15],
       [19, 26, 33],
       [29, 40, 51],
       [39, 54, 69]])
```



Un Esempio Pratico …

- Molti problemi pratici possono essere risolti rappresentando i dati in forma di matrici ed usando operazioni come il prodotto
- ***Esempio:*** tre aziende P_1, P_2, P_3 hanno bisogno di comprare prodotti nelle quantità indicate nella tabella a fianco…

	roll	bun	cake	bread
P_1	6	5	3	1
P_2	3	6	2	2
P_3	3	4	3	1

- …nelle vicinanze esistono due fornitori S_1, S_2 che vendono questi articoli, ai prezzi indicati nella tabella a fianco…

	S_1	S_2
roll	1.50	1.00
bun	2.00	2.50
cake	5.00	4.50
bread	16.00	17.00

- …qual è la spesa minima di ciascuna azienda, assumendo che ciascuna acquisti tutti i suoi prodotti dallo stesso fornitore?
- Di solito scriveremmo un programma che ottiene la spesa totale per ogni azienda presso ogni fornitore tramite cicli for

Soluzione "Classica" in Python

```
# DATI
P1 = {"roll": 6, "bun": 5, "cake": 3, "bread": 1}
P2 = {"roll": 3, "bun": 6, "cake": 2, "bread": 2}
P3 = {"roll": 3, "bun": 4, "cake": 3, "bread": 1}
S1 = {"roll": 1.5, "bun": 2, "cake": 5, "bread": 16}
S2 = {"roll": 1, "bun": 2.5, "cake": 4.5, "bread": 17}
# SOLUZIONE
result = [] # minimi per azienda
for needs in [P1, P2, P3]:
    min_cost = None
    for supplier in [S1, S2]:
        cost = 0
        for item, count in needs.items():
            cost += count * supplier[item]
        if min_cost is None or cost < min_cost:
            min_cost = cost # salva prezzo minore
    result.append(min_cost) # aggiorna risultati
```

per prodotto

per fornitore

per azienda

Soluzione in Python basata su *Comprehensions*

```
# DATI
P1 = {"roll": 6, "bun": 5, "cake": 3, "bread": 1}
P2 = {"roll": 3, "bun": 6, "cake": 2, "bread": 2}
P3 = {"roll": 3, "bun": 4, "cake": 3, "bread": 1}
S1 = {"roll": 1.5, "bun": 2, "cake": 5, "bread": 16}
S2 = {"roll": 1, "bun": 2.5, "cake": 4.5, "bread": 17}
# SOLUZIONE (istruzione singola!)
result = [
    min(
        sum(
            count * supplier[item]
            for item, count in needs.items()
        ) for supplier in [S1, S2]
    ) for needs in [P1, P2, P3]
]
```

per prodotto

per fornitore

per azienda

Soluzione con Calcolo di Matrici in NumPy

- Definendo una matrice **Q** con le quantità di prodotti richiesti e una matrice **P** dei prezzi unitari dei prodotti…

3 aziende × 4 prodotti

$$\mathbf{Q} = \begin{bmatrix} 6 & 5 & 3 & 1 \\ 3 & 6 & 2 & 2 \\ 3 & 4 & 3 & 1 \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} 1.5 & 1 \\ 2 & 2.5 \\ 5 & 4.5 \\ 16 & 17 \end{bmatrix}$$

4 prodotti × 2 fornitori

- … la spesa di un'azienda da un forniore è il prodotto scalare tra la riga delle quantità in **Q** e la colonna dei prezzi in **P**
- Possiamo quindi calcolare il prodotto tra **Q** e **P** con tutti i totali e selezionare quello minimo da ogni riga (azienda)

$$\mathbf{R} = \mathbf{QP} = \begin{bmatrix} 50 & 49 \\ 58.5 & 61 \\ 43.5 & 43.5 \end{bmatrix}$$

3 aziende × 2 fornitori

- In NumPy, una volta definite **Q** e **P**, basta quindi scrivere:

```
result = (Q @ P).min(1)
```

Matrici Trasposte

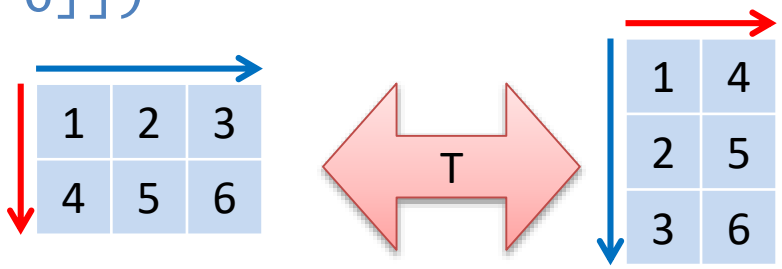
- Data una matrice **A** *m*×*n*, la sua *trasposta* $\mathbf{A}^T$ è la matrice *n*×*m* in cui righe e colonne sono scambiate
- Per le matrici trasposte si verificano le proprietà:

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T \qquad (\mathbf{A}+\mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$

- In NumPy, la trasposta di una matrice è data dall'attributo `T`
  - `T` è una vista sull'array d'origine, non una copia

```
>>> x = np.array([[1, 2, 3],
                  [4, 5, 6]])

>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
```



Matrici Quadrate

- Una matrice si dice *quadrata* se ha tante righe quante colonne
  - il numero di righe e colonne è detto *ordine* della matrice
- Una matrice quadrata **A** è
  - *simmetrica* se è uguale alla sua trasposta ($\mathbf{A} = \mathbf{A}^T$)

es: 2 righe 3 colonne
ordine: (2,3)

es.:

1	-2	3
-2	5	4
3	4	6

  - *antisimmetrica* se è uguale all'opposto della sua trasposta ($\mathbf{A} = -\mathbf{A}^T$)

es.:

1	2	-3
-2	5	-4
3	4	6

Matrici Diagonali

- La *diagonale principale* di una matrice quadrata è il vettore degli elementi lungo la diagonale da 1,1 a *n*,*n*
- Una matrice quadrata è *diagonale* se tutti gli elementi al di fuori della diagonale principale sono 0
- Con la funzione `diag` si può
  - estrarre la diagonale di una matrice quadrata
  - creare una matrice diagonale da un vettore con i valori della diagonale

```
>>> x = np.array(
   [[1, 2, 3],
    [4, 5, 6]
    [7, 8, 9]])
>>> np.diag(x)
array([1, 5, 9])
```

1	2	3
4	5	6
7	8	9

(alternativa: `x.diagonal()`)

```
>>> np.diag([1, 2, 3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

1	0	0
0	2	0
0	0	3

Matrice Identità

- La matrice *identità* di ordine *n*, indicata con $\mathbf{I}_n$, è la matrice diagonale *n*×*n* con tutti gli elementi della diagonale pari ad 1

$$\text{es.}: \mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- La matrice identità costituisce l'elemento neutro della moltiplicazione tra matrici

$$\mathbf{I}_m \mathbf{A} = \mathbf{A} \mathbf{I}_n = \mathbf{A}$$ per qualsiasi matrice *m*×*n* **A**

- Una matrice identità si può creare in NumPy con le funzioni `identity` oppure `eye`, indicando l'ordine della matrice

```
>>> np.eye(3)    # matrice identità di ordine 3
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Matrici Inverse

- Una matrice quadrata $n \times n$ $\mathbf{A}$ è *invertibile* se esiste una matrice *inversa* $\mathbf{A}^{-1}$ di pari dimensioni tale che $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}_n$
- Una matrice quadrata *non* invertibile è detta *singolare*
- Per matrici invertibili valgono le proprietà:

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1} \qquad (\mathbf{A}^{-1})^{T} = (\mathbf{A}^{T})^{-1}$$

- Per l'inversa in NumPy, usare la funzione `inv` di `linalg`

```
>>> x=np.array([[1, 2],
                [3, 4]])
>>> np.linalg.inv(x)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> x.dot(np.linalg.inv(x))
array([[1.0000000e+00, 0.0000000e+00],
       [8.8817842e-16, 1.0000000e+00]])
```

Non si ottiene esattamente $\mathbf{I}_2$ (ma una sua buona approssimazione) per perdite di precisione durante il calcolo

Sistemi di Equazioni Lineari in Forma di Matrici

- Un sistema di *m* equazioni lineari in *n* incognite può essere scritto in forma di matrici come **A x** = **b**

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n = b_2 \\ \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n = b_m \end{cases}$$

  - con **A** matrice *m*×*n*, **b** matrice *m*×1 e **x** matrice *n*×1
- Se la matrice **A** è quadrata (*m*=*n*) e invertibile, i valori delle incognite possono essere trovati tramite calcolo tra matrici:

$$\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}$$

- In NumPy diventa:

```
>>> x = np.linalg.inv(A).dot(b)
```

- … oppure tramite l'apposita funzione `solve`:

```
>>> x = np.linalg.solve(A, b)
```

Risoluzione di Sistemi Lineari con NumPy: Esempio

- **Quesito:** In un totale di 7 monete da 5 e 10 centesimi il cui valore è 55 centesimi, quante sono le monete dei due tagli ?
- **Sistema di equazioni**
  - sia *x* = num. monete da 5 centesimi e *y* = num. monete da 10

$$\begin{cases} x + y = 7 \\ 5x + 10y = 55 \end{cases}$$

- **Risoluzione in NumPy**

```
>>> A = np.array([ [1, 1], [5, 10] ])
>>> b = np.array([7, 55])
>>> np.linalg.solve(A, b)
array([3., 4.])
```

x = 3 monete da 5c
y = 4 monete da 10c