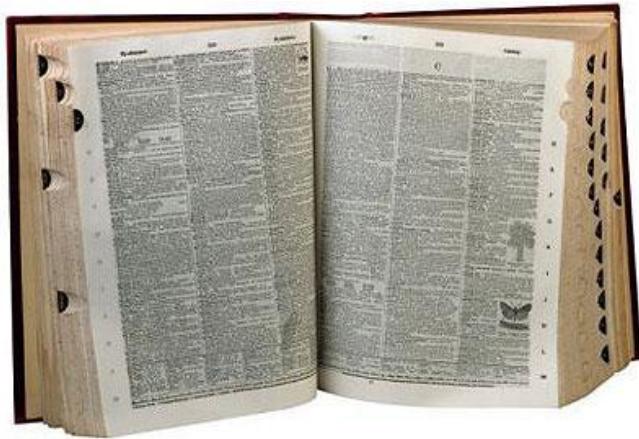


# *dizionari e matrici*



$$A = \begin{pmatrix} [A]_{1,1} & [A]_{1,2} & \cdots & [A]_{1,n} \\ [A]_{2,1} & [A]_{2,2} & \cdots & [A]_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ [A]_{m,1} & [A]_{m,2} & \cdots & [A]_{m,n} \end{pmatrix}$$

## *dizionari e matrici*

- approfondimenti
  - list comprehension
  - enumerate
  - zip map
- dizionari
- matrici

SUMMARY



## *list comprehension*

- *list comprehension*  $\Rightarrow$  modo **conciso** per creare una lista
- ogni elemento è il **risultato di una operazione** su un membro di un oggetto **iterabile**
- è possibile imporre una **condizione** sugli elementi (*opzionale*)

```
# quadrati dei numeri da 1 a 10
lista_quadrati = [x ** 2 for x in range(1,11)]
# codice equivalente
# lista_quadrati = []
# for x in range(1,11):
#     lista_quadrati.append(x ** 2)
```

```
# numeri dispari da 1 a 19
dispari = [str(x) for x in range(1,20) if (x % 2) != 0]
# codice equivalente
# dispari = []
# for x in range(1,20):
#     # if x%2 != 0:
#     # dispari.append(str(x))
```

<https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions>

## *enumerate*

- ***accoppia*** ciascun ***valore*** di una sequenza ad un ***indice*** crescente
- genera una sequenza di ***tuple*** (coppie)
  - spesso si usa nei cicli for, quando serve *sia il valore che l'indice*

```
enumerate(iterable, start=0)
```

*restituisce un oggetto enumerate*  
*iterable deve essere un oggetto che supporta l'iterazione*



*lazy evaluation (valutazione pigra) consiste nel ritardare una computazione finché il risultato non è richiesto effettivamente*

## *enumerate (esempi)*

```
colori = ["rosso", "verde", "giallo", "nero"]  
for colore in enumerate(colori):  
    print(colore, end=" ")  
# (0, 'rosso') (1, 'verde') (2, 'giallo') (3, 'nero')
```

```
# conversione in lista  
lista_colori=list(enumerate(colori))  
print(lista_colori)  
# [(0, 'rosso'), (1, 'verde'), (2, 'giallo'), (3, 'nero')]
```

```
for i, val in enumerate(colori):  
    print(i, val)  
# 0 rosso  
# 1 verde  
# 2 giallo  
# 3 nero
```

*zip*

- ***accoppia*** gli elementi di due sequenze
- genera una sequenza di tuple (***coppie***)
- il risultato ha la lunghezza della sequenza più breve

Ingredienti per 2 pizze di 28 cm di diametro		
• <a href="#">Farina Manitoba</a> 200 g	• <a href="#">Farina 00</a> 300 g	• <a href="#">Acqua</a> 300 ml
• <a href="#">Olio extravergine d'oliva</a> 35 g	• <a href="#">Sale fino</a> 10 g	• <a href="#">Lievito di birra fresco</a> 5 g

```
ingredienti = ["farina manitoba", "farina", "acqua", "olio", "sale", "lievito"]
quantità = ["200 g", "300 g", "300 ml"]

print(list(zip(ingredienti, quantità)))
# [('farina manitoba', '200 g'), ('farina', '300 g'), ('acqua', '300 ml')]
```

*funzione di ordine superiore (higher-order function) prende altre funzioni come parametri e/o restituisce funzioni come risultato*

*map*

- prende come **parametri** una **funzione** ed una **sequenza**
  - *funzione di ordine superiore*
- **applica la funzione** a ciascuno dei valori
- restituisce la **sequenza** di risultati

```
from math import sqrt
valori = [0, 1, 2, 3, 4]
print(list(map(sqrt, valori)))
#[0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0]

print(list(map(bin, range(8))))
#[ '0b0', '0b1', '0b10', '0b11', '0b100', '0b101', '0b110', '0b111' ]
```

*risultati in lista, solo per visualizzarli*

## *ordinare – rovesciare liste*

```
ingredienti = ["farina", "acqua", "olio", "sale", "lievito"]
print(ingredienti)    #['farina', 'acqua', 'olio', 'sale', 'lievito']

ingredienti.reverse()    # in place
print(ingredienti)    #['lievito', 'sale', 'olio', 'acqua', 'farina']

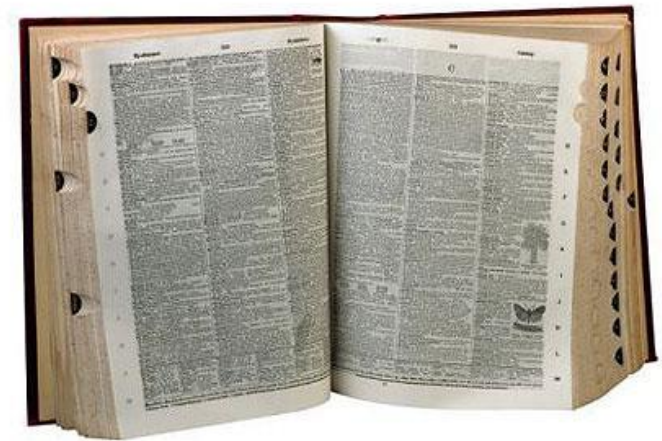
ingredienti.sort()    # in place
print(ingredienti)    #['acqua', 'farina', 'lievito', 'olio', 'sale']

ingredienti = ["farina", "acqua", "olio", "sale", "lievito"]
ingr_rev = sorted(ingredienti, reverse=True) # not in place
print(ingredienti)    #['farina', 'acqua', 'olio', 'sale', 'lievito']
print(ingr_rev)        #['sale', 'olio', 'lievito', 'farina', 'acqua']
ingr_lun = sorted(ingredienti, key=len)
print(ingr_lun)        #['olio', 'sale', 'acqua', 'farina', 'lievito']
```



## *dizionario*

- ***dizionario*** (*mappa, array associativo*)
- insieme non ordinato di ***coppie chiave / valore***
- le ***chiavi*** sono ***uniche***: hanno la funzione di ***indice*** per accedere al valore corrispondente
- le ***chiavi*** possono essere un qualsiasi ***tipo immutabile*** (int, str ...)



## *dizionario funzionalità*

```
#definizione di dizionario {}
tel = {"john": 4098, "terry": 4139}
#accesso a un elemento
print(tel["john"])    #4098
#aggiunta di una coppia chiave/valore
tel["graham"] = 4127
#visualizzazione dizionario
print(tel)             #{"graham": 4127, "terry": 4139, "john": 4098}
#visualizzazione chiavi
print(list(tel))       #['john', 'terry', 'graham']
#lista di coppie
print(list(tel.items()))
#[( 'john', 4098), ( 'terry', 4139), ( 'graham', 4127)]
#sequenza di elementi
for k in tel.keys():
    print(k, tel[k])
```

# *matrici*

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{pmatrix}$$

## *liste multidimensionali*

```
a = [['A', 'B', 'C', 'D'],  
      ['E', 'F', 'G', 'H'],  
      ['I', 'L', 'M', 'N']]          # 2D
```

```
b = ['A', 'B', 'C', 'D',  
      'E', 'F', 'G', 'H',  
      'I', 'L', 'M', 'N']          # 1D
```

## *matrice - somma colonne*

```
matrix = [[2, 4, 3, 8],  
          [9, 3, 2, 7],  
          [5, 6, 9, 1]]  
rows = len(matrix)  
cols = len(matrix[0])  
  
for x in range(cols):  
    total = 0  
    for y in range(rows):  
        val = matrix[y][x]  
        total += val  
    print("Col #", x, "sums to", total)
```

## *matrici - inizializzazione*

```
cols = 3      #dato noto
rows = 4      #dato noto
#inizializzazione di tutti gli elementi a ' '
matrix = [[' ' for x in range(cols)] for y in range(rows)]
```

```
#metodo alternativo
matrix = []
for y in range(rows):
    new_row = []
    for x in range(cols):
        new_row.append(' ')
    matrix.append(new_row)
```

# *esercizi*



## *~400 a.c. – scitala spartana*

- *una **scitala** (dal greco σκυτάλη = bastone) era una piccola bacchetta utilizzata dagli Spartani per trasmettere messaggi segreti*
- *il messaggio veniva scritto su di una striscia di pelle arrotolata attorno alla scitala, come se fosse stata una superficie continua*
- *una volta srotolata e tolta dalla scitala la striscia di pelle, era impossibile capire il messaggio*
- *la decifrazione era invece possibile se si aveva una bacchetta identica alla scitala del mittente: vi si arrotolava nuovamente la striscia di pelle ricostruendo la primitiva posizione*
- *si tratta del più antico metodo di crittografia per trasposizione conosciuto*





## *esercizi (1)*

- ***scitala spartana***

- richiedere una stringa di caratteri
- inserire in una matrice i primi  $W \times H$  caratteri
  - $W$  colonne  $\times$   $H$  righe, valori prefissati
  - riempire una riga della matrice dopo l'altra
  - da destra a sinistra, una riga alla volta ( $\rightarrow$ ,  $\downarrow$ )
- scrivere il contenuto della matrice su console
  - scrivere una colonna della matrice dopo l'altra
  - prima riga su console = prima colonna della matrice...
  - dall'alto verso il basso, una colonna alla volta ( $\downarrow$ ,  $\rightarrow$ )



*usare una lista di liste (con dimensioni predefinite)*

## *esercizi (2)*

- ***spirale***

- scrivere una funzione per riempire di ***numeri crescenti*** una ***matrice*** quadrata (o rettangolare)
- seguire il percorso a spirale suggerito nella figura a fianco
  - dimensioni della matrice indicate dall'utente a runtime

*tenere traccia della direzione attuale ( $\Delta y$ ,  $\Delta x$ )  
avanzare fino al bordo o ad una cella già visitata,  
poi cambiare la direzione in senso orario*

*coordinate raster, rotazione oraria di  $90^\circ$ :  $(x', y') = (-y, x)$   
in generale:  $(x', y') = (x \cdot \cos(\theta) - y \cdot \sin(\theta), x \cdot \sin(\theta) + y \cdot \cos(\theta))$*

