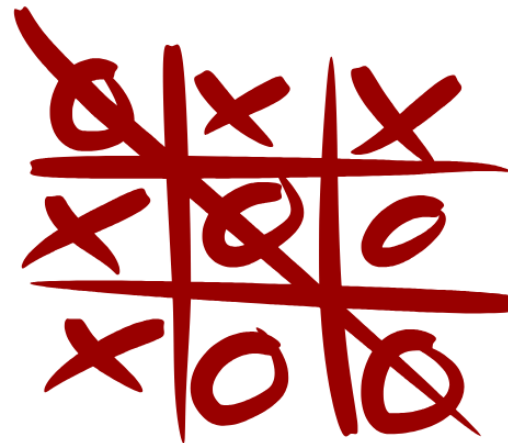




UNIVERSITÀ
DI PARMA

strutture dati

informatica e laboratorio di programmazione



- *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>

- **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*

```
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
```

- *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		
• Farina Manitoba 200 g	• Farina 00 300 g	• Acqua 300 ml
• Olio extravergine d'oliva 35 g	• Sale fino 10 g	• Lievito di birra fresco 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')]
```

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

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

```
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(sqrt, range(5))))
#[0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
```

risultati in lista, solo per visualizzarli

```
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** (*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 ...)




```
#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])
```

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

MATRICI

```
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
```

```
# conversione 2D -> 1D (dalla matrice a alla matrice b)  
indice = riga * num_colonne + colonna
```

```
# conversione 1D -> 2D (dalla matrice b alla matrice a)  
riga = indice // num_colonne  
colonna = indice % num_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)
```

```
matrix = [2, 4, 3, 8,  
          9, 3, 2, 7,  
          5, 6, 9, 1]  
rows = 3 # dato non ricavabile dalla pseudo-matrice  
cols = len(matrix) // rows  
  
for x in range(cols):  
    total = 0  
    for y in range(rows):  
        val = matrix[y * cols + x] # 2D -> 1D  
        total += val  
    print("Col #", x, "sums to", total)
```

```
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)
```

strutture dati
esercizi



○ 10.1 file CSV

- leggere una **matrice di interi** da un file testuale **CSV**
 - file CSV \Rightarrow **Comma-Separated Values**: valori riga per riga, separati da virgola
- memorizzare i dati in una lista semplice (**pseudo-matrice**)
- inferire le **dimensioni** della matrice (rows×cols) in base a:
 - numero di righe del file
 - numero di valori in una riga
- da angolo in basso a destra, **sommare sulla diagonale**

5,7,2,11
1,3,12,9
4,6,10,8

nell'esempio, sommare: $8 + 12 + 7$ (celle dove $cols - x == rows - y$)

- **10.2 incolonnamento dati**
 - visualizzare due *tabelle* con i caratteri *ASCII*
 - 4 righe x 24 colonne, codici da 32 a 126
 - tabella 1: mostrare in *ordine* i caratteri, *colonna per colonna*
 - tabella 2: mostrare in *ordine* i caratteri, *riga per riga*

```
$ ( , 048 < @DHLPTX \ ` dhlptx |  
! % ) - 159 = AEIMQUY ] aeimquy }  
" & * . 26 : > BFJNRVZ ^ bfjnr vz ~  
# ' + / 37 ; ? CGKOSW [ _ cgkosw {
```

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7  
8 9 : ; < = > ? @ A B C D E F G H I J K L M N O  
P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g  
h i j k l m n o p q r s t u v w x y z { | } ~
```

*usare sempre due cicli for annidati: esterno su y, interno su x
in ogni posizione, calcolare il carattere da visualizzare: $x * ROWS + y...$*

- 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



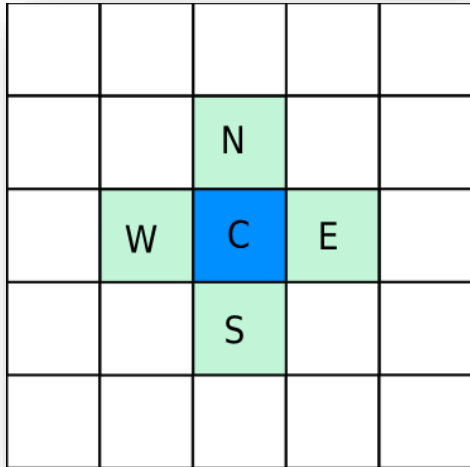
○ *10.3 scitola spartana*

- leggere un intero file di testo
- 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)



- **10.4 funzione di smooth**
 - scrivere una **funzione smooth**
 - **parametro**: matrice iniziale, di float
 - **risultato**: nuova matrice con smooth
 - matrici rappresentate come liste di liste
 - **smooth**: per ogni cella in matrice iniziale
 - il risultato è la media dell'intorno
 - 5 valori: cella stessa e 4 adiacenti
 - attenzione alle celle esterne
 - sommare e contare solo i valori disponibili
 - 4 valori ai bordi, 3 valori agli angoli
 - verificare la funzione con alcune matrici di test



		N		
	W	C	E	
		S		

○ 10.5 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 (Δy , Δx)
avanzare fino al bordo o ad una cella già visitata,
poi cambiare la direzione in senso orario*

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

