lez_03

January 21, 2026

1 Collezioni (Tuple e Liste)

Le collezioni comprendono diversi tipi di variabili che ci permettono di lavorare con insiemi di valori. L'esempio più banale sono i vettori, che in python prendono il nome di liste, ma troviamo anche tuple, insiemi e dizionari.

Passeremo in rassegna i diversi tipi di variabili *collezione*, iniziando dalle tuple.

1.1 Tuple

Le **tuple** sono collezioni di dati definite attraverso le parentesi tonde '( )' Qui vediamo una semplice tupla:

[2]:
```
tup = (1, 2, 7.4, 'gatto', (1, 2, 3), 4 > 2)
print(tup)
print(tup[0])
print(type(tup[0]), type(tup[5]))
```

```
(1, 2, 7.4, 'gatto', (1, 2, 3), True)
1
<class 'int'> <class 'bool'>
```

Dal codice qui sopra possiamo fare alcune considerazioni iniziali sulle tuple: - Possono contenere tipi di variabili diversi; - Possono essere stampate; - Si può accedere sequenzialmente ai singoli elementi con indici tra parentesi quadre

In questo sono molto simili ai vettori, ma la loro particolarità è quella di essere (come le stringhe) **immutabili**. Questo significa che non posso modificare una tupla una volta che è stata definita.

In che misura questo è utile? È utile se vogliamo una collezione di dati che non deve essere modificata per nessun motivo, ad esempio da una funzione o manualmente, e utilizzando una tupla possiamo evitare che questa sia modificata accidentalmente:

[2]:
```
tup[2] = 'Miao'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-1d5015274d34> in <module>
----> 1 tup[2] = 'Miao'
```

```
TypeError: 'tuple' object does not support item assignment
```

Le tuple sono **iterabili**, questo significa che possiamo *scorrerle* con un ciclo for. In alternativa, utilizzando sempre il ciclo for con la funzione *range()*, possiamo scorrere gli elementi della tupla accedendo con gli indici. I due esempi sono riportati qui sotto:

[3]:
```
for pippo in tup:
    print(pippo, type(pippo))
```

```
1 <class 'int'>
2 <class 'int'>
7.4 <class 'float'>
gatto <class 'str'>
(1, 2, 3) <class 'tuple'>
True <class 'bool'>
```

[4]:
```
for i in range(0, len(tup)):
    print(tup[i], type(tup[i]))
```

```
1 <class 'int'>
2 <class 'int'>
7.4 <class 'float'>
gatto <class 'str'>
(1, 2, 3) <class 'tuple'>
True <class 'bool'>
```

Come vediamo, i due diversi metodi producono lo stesso risultato, tuttavia per compattezza il primo è sicuramente migliore.

> **NOTA**: Inoltre il primo metodo, per varie ragioni, è anche più veloce ed efficiente in termini di esecuzione.

Infine, possiamo trasformare un altra variabile collezione (come una lista) in una tupla, usando la funzione **tuple()**:

[5]:
```
lista = [1, 2, 3, 4]
print(lista, type(lista))
tupla = tuple(lista)
print(tupla, type(tupla))
```

```
[1, 2, 3, 4] <class 'list'>
(1, 2, 3, 4) <class 'tuple'>
```

[8]:
```
stringa = 'abcdefg'
tupla_da_stringa = tuple(stringa)
print(tupla_da_stringa)
```

```
('a', 'b', 'c', 'd', 'e', 'f', 'g')
```

1.2 Liste

Le liste sono **vettori**. Come le tuple sono collezioni di elementi, con la differenza che le liste sono **modificabili**. Sono delimitate da parentesi quadre [ ] e come le tuple possono contenere variabili di tipo diverso.

[6]:
```
lista = [1, 5.72, 'pippo', (1, 2, 3)]
for element in lista:
    print(element, type(element))
```

```
1 <class 'int'>
5.72 <class 'float'>
pippo <class 'str'>
(1, 2, 3) <class 'tuple'>
```

Anche le liste sono **iterabili**, e come le tuple possiamo scorrere i loro elementi con un ciclo for.

Possiamo accedere sequenzialmente agli elementi di una lista con gli indici interi:

[7]:
```
print(lista[0])
print(lista[len(lista) - 1])

for i in range(0, len(lista)):
    print(lista[i])
```

```
1
(1, 2, 3)
1
5.72
pippo
(1, 2, 3)
```

1.2.1 Aggiungere elementi ad una lista

Essendo una lista modificabile, possiamo eseguire delle operazioni che modifichino i suoi elementi, o che aggiungano o rimuovano elementi da essa.

Di seguito alcuni metodi per aggiungere elementi ad una lista:

- *list.append(elemento)* : Aggiunge un elemento in fondo alla lista
- *list.extend(iterabile)* : Aggiunge tutti gli elementi di un iterabile in fondo alla lista
- *list.insert(indice, elemento)* : Aggiunge un elemento nella posizione specificata come argomento

[16]:
```
pop_stars = ['Lady Gaga', 'Rihanna']
print(pop_stars)
print('Adding Ariana with append():')
pop_stars.append('Ariana Grande')
print(pop_stars)
print('Adding some people with extend():')
more_pop_stars = ['Beyoncé', 'Giorgio Mastrota']
```

```
pop_stars.extend(more_pop_stars)
print(pop_stars)
print('Adding a pop star in first position with insert():')
pop_stars.insert(0, 'Cardi B')
print(pop_stars)
```

```
['Lady Gaga', 'Rihanna']
Adding Ariana with append():
['Lady Gaga', 'Rihanna', 'Ariana Grande']
Adding some people with extend():
['Lady Gaga', 'Rihanna', 'Ariana Grande', 'Beyoncé', 'Giorgio Mastrota']
Adding a pop star in first position with insert():
['Cardi B', 'Lady Gaga', 'Rihanna', 'Ariana Grande', 'Beyoncé', 'Giorgio
Mastrota']
```

1.2.2 Rimuovere elementi da una lista

Vediamo due dei metodi che ci permettono di rimuovere degli elementi da una lista:

- *list.pop()* : Rimuove l'ultimo elemento di una lista
- *list.pop(indice)* : Rimuove l'elemento della lista nella posizione specificata
- *list.remove(elemento)* : Rimuove la prima occorrenza dell'elemento indicato

list.pop() ritorna l'elemento rimosso, mentre remove() non ritorna nulla

[18]:
```
['Cardi B', 'Lady Gaga', 'Rihanna', 'Ariana Grande', 'Beyoncé', 'Giorgio␣
↪Mastrota']
rimosso = pop_stars.pop()
print('Removed element: ', rimosso)
print(pop_stars)
rimosso2 = pop_stars.pop(3)
print('Removed element number 4: ', rimosso2)
print(pop_stars)
print('Removing Lady Gaga :(')
pop_stars.remove('Lady Gaga')
print(pop_stars)
```

```
Removed element:  Giorgio Mastrota
['Cardi B', 'Lady Gaga', 'Rihanna', 'Ariana Grande', 'Beyoncé']
Removed element number 4:  Ariana Grande
['Cardi B', 'Lady Gaga', 'Rihanna', 'Beyoncé']
Removing Lady Gaga :(
['Cardi B', 'Rihanna', 'Beyoncé']
```

1.2.3 list() e List Comprehension

Oltre alla definizione "manuale", possiamo ottenere una lista trasformando un altro iterabile (come una tupla), oppure possiamo usare una feature di python nota come ***list comprehension***: si tratta di un one-liner in grado di creare una lista partendo da un iterabile come l'output della funzione *range()*:

[23]:
```
stringa = 'ATCGACGCTGAGCTAGCT'
tupla = (1, 2 ,3 ,4 ,5)
lista1 = list(tupla)
lista2 = list(stringa)
print(lista1, type(lista1))
print(lista2, type(lista2))

vettore_numeri = [1, 2, 3, 4, 5, 6, 7, 8]
vettore_doppio = [x*2 for x in vettore_numeri]
vettore_doppio2 = []
for numero in vettore_numeri:
    vettore_doppio2.append(numero * 2)
print(vettore_doppio2)
print(vettore_numeri)
print(vettore_doppio)

#list comprehension:

lista2 = [x for x in range(15)]
print(lista2)

iterabile = (1, 5, 3, 5, 6, 7, 3)
lista3 = [elemento for elemento in iterabile]
print(lista3)
```

```
[1, 2, 3, 4, 5] <class 'list'>
['A', 'T', 'C', 'G', 'A', 'C', 'G', 'C', 'T', 'G', 'A', 'G', 'C', 'T', 'A', 'G',
'C', 'T'] <class 'list'>
[2, 4, 6, 8, 10, 12, 14, 16]
[1, 2, 3, 4, 5, 6, 7, 8]
[2, 4, 6, 8, 10, 12, 14, 16]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
[1, 5, 3, 5, 6, 7, 3]
```

Quello che fa list() è abbastanza evidente.

Per quanto riguarda la sintassi della list comprehension, proviamo a spiegarla brevemente: > Prendi ciascun elemento contenuto in quell'iterabile e aggiungilo ad una lista.

L'elemento può essere preso così com'è, oppure può essere modificato da un'operazione, una funzione o altro, come vediamo nei prossimi due esempi:

[24]:
```
lista = [x for x in range(15)]
lista_al_quadrato = [x**2 for x in range(15)]

#definiamo una breve funzione che raddoppia un valore

def raddoppia(n):
    return n*2
```

```
lista_raddoppiata_da_funzione = [raddoppia(x) for x in range(15)]

print(lista)
print(lista_al_quadrato)
print(lista_raddoppiata_da_funzione)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

La list comprehension è molto comoda e molto usata, pertanto è importante conoscere questa notazione.

Vediamo ora un'ultima funzione che riguarda le liste per poi passare a degli esercizi. La funzione che guarderemo ora è enumerate(), e vedremo prima del codice per poi spiegare cosa fa.

1.2.4 enumerate()

[38]:
```
lista = 'Cane,Gatto,Topo,Pesce'.split(',')
print(lista)
lista_numerata = enumerate(lista)
for element in lista_numerata:
    print(element)
print(lista_numerata)

vettore_di_vettori = [[0, 'Cane', 'Lorenzo'], [1, 'Gatto', 'Giorgio'], [2,␣
 ↪'Topo', 'Dario'], [3, 'Pesce', 'Sara']]
sequenze = [('283718', 'ACAGCTACTAGCTAGCTG', 'H. Sapiens'), ('243823718',␣
 ↪'GCTAGCTGCTG', 'A. Thaliana')]
for accession, seq, orgn in sequenze:
    print(f'>{accession} | {orgn}\n{seq}\n')

for elemento in vettore_di_vettori:
    print(elemento)

for indice, animale, nome in vettore_di_vettori:
    print(indice + 100)
    print(animale)
    print(f'Il proprietario di {animale} è {nome}')
```

```
>283718 | H. Sapiens
ACAGCTACTAGCTAGCTG

>243823718 | A. Thaliana
GCTAGCTGCTG

[0, 'Cane', 'Lorenzo']
[1, 'Gatto', 'Giorgio']
```

```
[2, 'Topo', 'Dario']
[3, 'Pesce', 'Sara']
100
Cane
Il proprietario di Cane è Lorenzo
101
Gatto
Il proprietario di Gatto è Giorgio
102
Topo
Il proprietario di Topo è Dario
103
Pesce
Il proprietario di Pesce è Sara
```

enumerate() prende come argomento una lista, estrae ciascun elemento della lista e crea un oggetto *enumerate*.

Questo oggetto è un iterabile che contiene lo stesso numero di elementi della lista di partenza, ed ogni elemento dell'oggetto enumerate è una **tupla** che contiene due elementi: la posizione dell'elemento nella lista originaria e l'elemento nella lista originaria.

Quando abbiamo un vettore che contiene collezioni al suo interno (se queste contengono tutte lo stesso numero di elementi), possiamo scorrere gli elementi delle collezioni con un solo ciclo for in questo modo:

[36]:
```
for indice, contenuto in enumerate(lista):
    print(indice, contenuto)
```

```
0 Cane
1 Gatto
2 Topo
3 Pesce
```

Dove la spiegazione ha probabilmente fallito, spero che il codice sia riuscito a chiarire i dubbi!

1. Il ciclo for passa in rassegna ogni elemento dell'oggetto enumerate;
2. Ciascun elemento dell'oggetto enumerate è una tupla di due elementi (indice, contenuto);
3. L'istruzione nel corpo del ciclo for viene eseguita, e vengono stampati i due elementi di ciascuna tupla

Nel disperato tentativo di rendere questo concetto ancora più chiaro, consideriamo il seguente esempio, nel quale abbiamo un vettore che contiene 3 iterabili da 3 elementi ciascuno:

[38]:
```
vett = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
print(len(vett))
for entry in vett:
    print(len(entry))
```

```
3
3
```

```
3
3
```

In generale, quando abbiamo un vettore di dimensione NxM (dove N è il numero di elementi del vettore, ed M è il numero di elementi contenuto da ciascun elemento del vettore), possiamo utilizzare la seguente notazione:

```
for x1, x2, x3, ..., xM in vettore:     do_something()
```

Nel nostro caso:

[39]:
```
for x1, x2, x3 in vett:
    print(x1, x3)
```

```
1 3
4 6
7 9
```

E per fare un esempio ancora più elementare, consideriamo un altro vettore che contiene 3 tuple: ogni tupla contiene 3 elementi, siano (Nome, Cognome, Età). Ecco come possiamo scorrere il vettore con un ciclo for:

[40]:
```
lista_anagrafe = (('Mario', 'Rossi', 32), ('Maria', 'Verdi', 31), ('Giancane',␣
 ↪'Sebastianinolelli', 19429))

for nome, cognome, eta in lista_anagrafe:
    print('{} {} ha {} anni'.format(nome, cognome, eta))
```

```
Mario Rossi ha 32 anni
Maria Verdi ha 31 anni
Giancane Sebastianinolelli ha 19429 anni
```

1.3 Esercizi

1. Popolare una lista di elementi con i primi n numeri pari utilizzando un ciclo for e la funzione range
2. Ripetere l'esercizio con la list comprehension
3. Prendere in input un elenco di elementi separati da una virgola (o da un altro spaziatore qualsiasi) e inserirli in una lista (es. 'Cane,Gatto,Topo' -> ['Cane', 'Gatto', 'Topo']
4. Stampare gli elementi dela lista così ottenuta con i relativi indici senza utilizzare enumerate. L'output dovrebbe avere questa forma: - 0 elemento0 - 1 elemento1 - ……. - N elementoN
5. Ripetere l'esercizio 4 utilizzando enumerate

..

..

..

..

..

..

..

..

..

..

..

1.4 Soluzioni

1.4.1 Esercizio 1

[41]:
```
n = 20
lista_pari = []
for i in range(n):
    lista_pari.append(i*2)
print(lista_pari, len(lista_pari))
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38] 20
```

1.4.2 Esercizio 2

[42]:
```
n = 20
lista_pari = [x*2 for x in range(n)]
print(lista_pari, len(lista_pari))
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38] 20
```

1.4.3 Esercizio 3

[43]:
```
stringa = input('Dammi elementi separati da virgola:')
lista = stringa.split(',')
print(lista)
```

```
Dammi elementi separati da virgola: cane,gatto,topo,pesce
```

```
['cane', 'gatto', 'topo', 'pesce']
```

1.4.4 Esercizio 4

[44]:
```
for i in range(len(lista)):
    print(i, lista[i])
```

```
0 cane
1 gatto
2 topo
3 pesce
```

1.4.5 Esercizio 5

[45]:
```
for index, element in enumerate(lista):
    print(index, element)
```

```
0 cane
1 gatto
2 topo
3 pesce
```