lez_04

January 21, 2026

1 Collezioni (Insiemi e Dizionari)

Nella scorsa lezione abbiamo iniziato a parlare di "collezioni" e abbiamo imparato ad utilizzare tuple e vettori. In questa lezione passeremo ad altri due tipi di variabili collezione: gli **insiemi** e i **dizionari**.

1.1 Insiemi

Gli insiemi in python sono delimitati da **parentesi graffe** '{ }' e si comportano in modo particolare rispetto alle altre variabili collezione.

Questi infatti sono estremamente simili a quello che intendiamo in matematica quando parliamo di insiemi, e in particolare:

- Un insieme non contiene elementi duplicati;
- Un insieme non è ordinato (vedremo cosa significa a breve);
- È possibile definire sulle variabili *insieme* le operazioni normalmente definite negli insiemi matematici (unione, intersezione, ecc..)

Ecco alcuni esempi di insieme che ci aiutano a capire le caratteristiche di questo tipo di variabili:

[2]:
```
cesta = {'banana', 'mela', 'arancia', 'mela', 'kiwi', 'banana'}
print(cesta)
print(type(cesta))
```

```
{'banana', 'arancia', 'kiwi', 'mela'}
<class 'set'>
```

Abbiamo definito una variabile insieme e abbiamo specificato quali elementi questa avrebbe contenuto. Gli elementi 'mela' e 'banana' erano stati inseriti **più volte** in fase di definizione, tuttavia quando abbiamo stampato con *print()* l'insieme li conteneva una volta sola. Quindi l'insieme *scarta automaticamente i duplicati.*

Questa peculiarità degli insiemi è comoda se ad esempio abbiamo una lista dal quale vogliamo eliminare gli elementi duplicati. Possiamo infatti trasformarlo in un insieme con la funzione **set()**:

[3]:
```
#definisco una lista contenente valori duplicati
lista_con_duplicati = [1, 2, 3, 1, 1, 4, 5, 3, 3, 6, 5]
#trasformo la lista in un insieme (questo rimuoverà i duplicati)
insieme_ottenuto_da_lista = set(lista_con_duplicati)
#trasformo nuovamente l'insieme in una lista
```

```
lista_senza_duplicati = list(insieme_ottenuto_da_lista)
print(lista_senza_duplicati)

vettore = [1, 2, 3, 1, 1, 4, 1, 5]
vettore_pulito = list(set(vettore))
print(vettore)
print(vettore_pulito)
```

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

L'altra caratteristica degli insiemi è quella di non essere ordinati. Ma cosa significa di preciso?

Significa che Python internamente non tiene traccia dell'ordine in cui si trovano i valori all'interno di un insieme. Questo ha come conseguenza principale che **non** possiamo accedere agli elementi di un insieme utilizzando degli indici:

[5]:
```
insieme = {1, 2, 3, 4}
print(insieme[2])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_7043/154105990.py in <module>
      1 insieme = {1, 2, 3, 4}
----> 2 print(insieme[2])

TypeError: 'set' object is not subscriptable
```

Tuttavia questo non ci impedisce di stampare tutti gli elementi contenuti nell'insieme attraverso un ciclo for:

[6]:
```
for element in insieme:
    print(element)
```

```
1
2
3
4
```

Vediamo ora come aggiungere o rimuovere elementi da un set (da un insieme)

1.1.1 Aggiungere elementi ad un insieme

Possiamo utilizzare due metodi per aggiungere elementi ad un insieme:

- *insieme.add(elemento)* : Aggiunge l'elemento passato come parametro all'insieme
- *insieme.update(iterabile)* : Aggiunge tutti gli elementi di un iterabile all'insieme

Il primo è quindi equivalente al metodo append() nelle liste (con la differenza che per gli insieme non ha senso dire che l'elemento viene aggiunto *in fondo* all'insieme. Il secondo invece è equivalente

al metodo extend() nelle liste. Vediamo ora due esempi banali:

[7]:
```
cesta = {'mela', 'banana'}
print(cesta)
#aggiungo un elemento con add():
cesta.add('kiwi')
print(cesta)
#definisco due "gruppi" di elementi:
lista_frutta = ['pera', 'mango']
tupla_frutta = ('arancia', 'uva')
#aggiungo la lista:
cesta.update(lista_frutta)
print(cesta)
#aggiungo la tupla:
cesta.update(tupla_frutta)
print(cesta)
```

```
{'mela', 'banana'}
{'mela', 'kiwi', 'banana'}
{'mela', 'banana', 'pera', 'kiwi', 'mango'}
{'mela', 'uva', 'banana', 'arancia', 'pera', 'kiwi', 'mango'}
```

1.1.2 Rimuovere elementi da un insieme

Anche per rimuovere elementi da un set vedremo due metodi, tra i quali esiste una singola differenza fondamentale: il primo da errore se non trova l'elemento da rimuovere, il secondo no.

> **NOTA**: la domanda che potrebbe sorgere spontanea è la seguente: "*Perché dovrei utilizzare un metodo che blocca potenzialmente il programma in caso di errore, quando ho a disposizione un metodo che non mi restituisce nessun errore?*". La risposta è che a volte abbiamo bisogno di sapere quando qualcosa va storto. Ad esempio potrebbe essere importante sapere che quell'elemento è mancante all'interno del nostro set, e un errore non deve necessariamente bloccare l'esecuzione di un programma, poiché può essere gestito. Parleremo della sintassi try/except più avanti nel corso.

I due metodi sono i seguenti: - *set.remove(elemento)* : Rimuove l'elemento passato come argomento e genera un errore se non lo trova - *set.discard(elemento* : Rimuove l'elemento passato come argomento, senza generare errori in caso di elemento mancante

Qui sotto, due banali esempi del loro funzionamento:

[11]:
```
cesta = {'arancia', 'uva', 'banana', 'kiwi', 'pera', 'mela', 'mango'}
#rimuovo l'arancia con discard():
cesta.discard('arancia')
print(cesta)
#Ora non ci sono più arance, ma provo a rimuoverne una lo stesso:
print('Rimuovo l\'arancia che non c\'è con discard()')
cesta.discard('arancia')
print('Visto? Nessun errore')
print('Ora provo a rimuoverla con il metodo remove()')
```

```
try:
    cesta.remove('arancia')
except KeyError:
    print('L\'arancia non cè!')
print('Il programma continua')
```

```
{'mela', 'banana', 'pera', 'kiwi', 'mango', 'uva'}
Rimuovo l'arancia che non c'è con discard()
Visto? Nessun errore
Ora provo a rimuoverla con il metodo remove()
Il programma continua
```

Giusto per soddisfare eventuali curiosità, vediamo qui brevemente come poteva essere gestito l'errore prodotto da metodo remove() quando l'elemento da rimuovere non è presente nell'insieme.

Per gestire l'eccezione abbiamo bisogno di sapere che tipo di errore viene generato, e questo possiamo vederlo nell'output generato dal codice qui sopra. La parte che ci interessa è **KeyError**.

Per gestire l'errore utilizziamo la sintassi **try/except**, che in soldoni si può tradurre così:

Prova a fare questo, a meno che non incontri questo errore. In quel caso, fai quest'altro.

Vediamo ora Try/Except in azione:

[13]:
```
set = {'mela', 'arancia'}
#proviamo a rimuovere un elemento non presente con remove()

try: #prova
    set.remove('banana') #a rimuovere questo elemento
except KeyError: #a meno che non incontri questo errore
    print('Elemento non trovato') #in quel caso, dimmi cosa succede

print('Anche in caso di errore, il codice può proseguire!')
```

```
Elemento non trovato
Anche in caso di errore, il codice può proseguire!
```

1.1.3 Operazioni tra insiemi

Le operazioni definite sugli insiemi, come già detto, sono quelle che troviamo negli insiemi intesi come entità matematiche, e in particolare troviamo:

- **Unione** *insieme1* | *insieme2* : Restituisce un insieme contentente gli elementi di entrambi gli insiemi
- **Intersezione** *insieme1* & *insieme2* : Restituisce un insieme contenente gli elementi in comune tra i due insiemi
- **Differenza** *insieme1* - *insieme2* : Restituisce gli elementi dell'insieme 1 che non sono nell'insieme 2
- **Differenza Simmetrica** *insieme1* ^ *insieme2* : Restituisce gli elementi unici, ossia gli elementi non in comune (Il contrario dell'intersezione)

Vediamo alcuni esempi di queste operazioni:

[12]:
```
insieme1 = {'mela', 'banana', 'kiwi', 'arancia', 'mango'}
insieme2 = {'mandarino', 'banana', 'uva', 'kiwi', 'limone'}

unione = insieme1 | insieme2
print(unione)
intersezione = insieme1 & insieme2
print(intersezione)
differenza = insieme1 - insieme2
print(differenza)
diff_simmetrica = insieme1 ^ insieme2
print(diff_simmetrica)
```

```
{'mela', 'uva', 'mandarino', 'banana', 'arancia', 'limone', 'kiwi', 'mango'}
{'banana', 'kiwi'}
{'mela', 'mango', 'arancia'}
{'uva', 'limone', 'mela', 'arancia', 'mango', 'mandarino'}
```

1.1.4 Metodi Booleani sugli insiemi

Per concludere il paragrafo sugli insiemi, diamo un'occhiata ai metodi booleani definiti su essi.

Per metodi booleani intendiamo dei metodi che ritornino True oppure False, rispondono quindi a una domanda *binaria*.

Con questi metodi chiediamo al programma di dirci se due insiemi rispettano o meno una data condizione, come ad esempio essere disgiunti, o se l'insieme2 è un sottoinsieme dell'insieme1. Qui l'elenco dei metodi che utilizzeremo:

- *insieme1.isdisjoint(insieme2)* : Ci dice se l'insieme1 è disgiunto dall'insieme2 dato come parametro (quindi se non hanno elementi in comune)
- *insieme1.issubset(insieme2)* : Ci dice se l'insieme1 è un sottoinsieme dell'insieme 2 (se è contenuto in esso)
- *insieme1.issuperset(insieme2)* : Ci dice se l'insieme1 contiene l'insieme2

Vediamo i metodi in azione

[14]:
```
insieme1 = {'arancia', 'mela', 'banana', 'kiwi', 'uva'}
insieme2 = {'mango', 'papaya', 'melone'}
print(insieme1, insieme2)
#controllo se i due insiemi hanno elementi in comune
if insieme1.isdisjoint(insieme2):
    print("L'insieme1 e l'insieme2 sono disgiunti")
else:
    print("I due insiemi hanno elementi in comune")

#definisco un nuovo insieme
insieme3 = {'arancia', 'banana', 'mandarino'}
```

```
#controllo se ha elementi in comune con l'insieme3
print("L'insieme1 e l'insieme3 sono disgiunti") if insieme1.
 ↪isdisjoint(insieme3) else print('I due insiemi hanno elementi in comune')

#definisco un nuovo insieme:
insieme4 = {'arancia', 'mela'}
#controllo se l'insieme 4 è un sottoinsieme dell'insieme 1
if insieme4.issubset(insieme1): print("L'insieme4 è sottoinsieme dell'insieme1")
#controllo se l'insieme 1 è contenuto nell'insieme 4
if insieme1.issuperset(insieme4): print("L'insieme1 contiene l'insieme4")
```

```
{'mela', 'banana', 'kiwi', 'arancia', 'uva'} {'papaya', 'mango', 'melone'}
L'insieme1 e l'insieme2 sono disgiunti
I due insiemi hanno elementi in comune
L'insieme4 è sottoinsieme dell'insieme1
L'insieme1 contiene l'insieme4
```

Passiamo ora ad un tipo di variabile collezione (nonché struttura dati) che utilizzeremo molto quando andremo a vedere qualche applicazione reale del linguaggio Python: i **dizionari**.

1.2 Dizionari

I dizionari sono un tipo di collezione caratterizzata dal fatto che ciascun elemento contenuto in essi è definito da una coppia **(chiave : valore)**.

Anche i dizionari sono delimitati da parentesi graffe '**{ }**' come gli insiemi, tuttavia sono facilmente distinguibili da questi ultimi per il loro contenuto. Di seguito definiamo un piccolo dizionario:

[19]:
```
capitali = {
    'Inghilterra' : 'Londra',
    'Scozia' : 'Edinburgo',
    'Galles' : 'Cardiff'
}

print(capitali)
print(capitali['Scozia'])
print(type(capitali))
```

```
{'Inghilterra': 'Londra', 'Scozia': 'Edinburgo', 'Galles': 'Cardiff'}
Edinburgo
<class 'dict'>
```

Possiamo vedere un dizionario come una lista dove gli indici numerici sono sostituiti dalle chiavi.

Pertanto per accedere al singolo valore che ci interessa in un dizionario utilizzeremo la notazione **dizionario[chiave]**, esattamente come per i vettori e per le tuple ma in quel con indici numerici:

[17]:
```
print('La capitale del Galles è', capitali['Galles'])
capitale_inghilterra = capitali['Inghilterra']
print('La capitale dell\'inghilterra è', capitale_inghilterra)
```

```
La capitale del Galles è Cardiff
La capitale dell'inghilterra è Londra
```

Essendo il dizionario un **iterabile**, possiamo scorrere i suoi elementi con un ciclo for. Tuttavia l'output potrebbe essere inaspettato ad un primo sguardo:

[20]:
```
for i in capitali:
    print(i)
```

```
Inghilterra
Scozia
Galles
```

Come possiamo vedere dall'output, quando il dizionario viene *interrogato* con un ciclo for, quello che ci restituisce sono le chiavi che contiene. Quindi se siamo interessati ai valori o alle coppie chiave:valore, dobbiamo aggiungere un po' di fantasia e di lavoro extra:

[21]:
```
#stampare i valori di un dizionario con un ciclo for
for key in capitali:
    print(capitali[key])

#stampare le coppie chiave-valore di un dizionario con un ciclo for

for key in capitali:
    print(key, '::', capitali[key])
```

```
Londra
Edinburgo
Cardiff
Inghilterra :: Londra
Scozia :: Edinburgo
Galles :: Cardiff
```

1.2.1 Definire un dizionario a partire da un altro iterabile

Un altro modo per definire un dizionario è quello di farlo a partire da un altro iterabile (come una lista o una tupla) attraverso la funzione **dict()**.

ATTENZIONE: è chiaro come la funzione *dict()* debba poter stabilire quali siano le chiavi e quali i valori, pertanto non possiamo utilizzare un iterabile qualsiasi per trasformarlo in un dizionario.

In particolare, è importante che l'iterabile scelto abbia la seguente struttura:

`iterabile = [(chiave1, valore1), (chiave2, valore2,), ..., (chiaveN, valoreN)]`

Quindi l'iterabile deve contenere al suo interno **altri N iterabili** composti da due elementi: Il primo elemento sarà considerato come chiave da dict(), il secondo come valore associato a quella chiave.

Vediamo un esempio:

[22]:
```
#lista di tuple da trasformare in un dizionario
iterabile_capitali = [('Regno Unito', 'Londra'),
                      ('Germania', 'Berlino'),
                      ('Spanga', 'Madrid')]
#il primo elemento di ogni tupla diventerà una chiave, il secondo il valore

dizionario_capitali = dict(iterabile_capitali)
for key in dizionario_capitali:
    print(key, dizionario_capitali[key])

print(dizionario_capitali)
```

```
Regno Unito Londra
Germania Berlino
Spanga Madrid
{'Regno Unito': 'Londra', 'Germania': 'Berlino', 'Spanga': 'Madrid'}
```

> **NOTA**: La scelta di utilizzare una lista di tuple è puramente arbitraria. Una lista di liste o una tupla di tuple o una tupla di liste sarebbero state perfettamente equivalenti, purché di dimensione N*2

Vediamo ora come aggiungere o rimuovere elementi da un dizionario.

1.2.2 Aggiungere o rimuovere elementi

Aggiungere un elemento ad un dizionario è abbastanza banale. È infatti sufficiente specificare una nuova chiave tra parentesi quadre e assegnarle un valore:

[25]:
```
capitali = {
    'Regno Unito' : 'Londra',
    'Spagna' : 'Madrid'
}

print(capitali)
print('Aggiungo la Germania...')
capitali['Germania'] = 'Berlino'
print(capitali)
capitali['Birmania'] = 'Myanmar'
print(capitali)
capitali['Germania'] = 'Bu'
print(capitali)
```

```
{'Regno Unito': 'Londra', 'Spagna': 'Madrid'}
Aggiungo la Germania…
{'Regno Unito': 'Londra', 'Spagna': 'Madrid', 'Germania': 'Berlino'}
{'Regno Unito': 'Londra', 'Spagna': 'Madrid', 'Germania': 'Berlino', 'Birmania':
'Myanmar'}
{'Regno Unito': 'Londra', 'Spagna': 'Madrid', 'Germania': 'Bu', 'Birmania':
'Myanmar'}
```

Mentre, se volessimo aggiungere più elementi alla volta, possiamo utilizzare il metodo *dict.update(iterabile)*.

Di nuovo, l'iterabile può essere sia un dizionario che un'iterabile generico, purché rispetti le condizioni necessarie, ossia che abbia dimensione N*2:

[27]:
```
prima_aggiunta = {
    'Italia' : 'Roma',
    'Grecia' : 'Atene'
 }

#aggiungo un dizionario
capitali.update(prima_aggiunta)
print(capitali)

#aggiungo un generico iterabile di dimensione N*2
seconda_aggiunta = [('Portogallo', 'Lisbona'), ('Austria', 'Vienna')]

capitali.update(seconda_aggiunta)
print(capitali)
```

```
{'Regno Unito': 'Londra', 'Spagna': 'Madrid', 'Germania': 'Bu', 'Birmania':
'Myanmar', 'Italia': 'Roma', 'Grecia': 'Atene', 'Portogallo': 'Lisbona',
'Austria': 'Vienna'}
{'Regno Unito': 'Londra', 'Spagna': 'Madrid', 'Germania': 'Bu', 'Birmania':
'Myanmar', 'Italia': 'Roma', 'Grecia': 'Atene', 'Portogallo': 'Lisbona',
'Austria': 'Vienna'}
```

Per quanto riguarda invece la *rimozione* di elementi da un dizionario, ho due opzioni: - Il metodo *dict.pop(chiave)* : Rimuove la chiave indicata e l'elemento corrispondente dal dizionario - La keyword **del** *dizionario[chiave]* : Stesso risultato di *dict.pop()*

La differenza sostanziale tra questi due approcci, è che pop() **ritorna l'elemento rimosso**, mentre **del** si limita a rimuoverlo:

[28]:
```
elemento_rimosso_con_pop = capitali.pop('Portogallo')
print(elemento_rimosso_con_pop)
print(capitali)

del capitali['Italia']
print(capitali)
```

```
Lisbona
{'Regno Unito': 'Londra', 'Spagna': 'Madrid', 'Germania': 'Bu', 'Birmania':
'Myanmar', 'Italia': 'Roma', 'Grecia': 'Atene', 'Austria': 'Vienna'}
{'Regno Unito': 'Londra', 'Spagna': 'Madrid', 'Germania': 'Bu', 'Birmania':
'Myanmar', 'Grecia': 'Atene', 'Austria': 'Vienna'}
```

1.2.3 Metodi per estrarre informazioni

I seguenti metodi ci permettono di estrarre le chiavi, i valori, o le coppie chiave-valore contenute in un dizionario.

Tutti e tre i metodi ritornano degli iterabili, che possono essere trasformati in tuple, liste o insiemi per poi essere utilizzate in altri contesti.

I metodi che presenteremo sono i seguenti: - *dict.keys()* : Restituisce un iterabile contenente le chiavi del dizionario - *dict.values()* : Restituisce un iterabile contenente i valori del dizionario - *dict.items()* : Restituisce un iterabile contenente le coppie chiave:valore del dizionario

[34]:
```
chiavi = capitali.keys()
print('Ecco le chiavi estratte con keys()')
print(chiavi, type(chiavi))

valori = capitali.values()
print('Ecco i valori estratti con values()')
print(valori, type(valori))

coppie = capitali.items()
print('Ecco le coppie chiave:valore estratte con items()')
print(coppie, type(coppie))
```

```
Ecco le chiavi estratte con keys()
dict_keys(['Regno Unito', 'Spagna', 'Germania', 'Birmania', 'Grecia',
'Austria']) <class 'dict_keys'>
Ecco i valori estratti con values()
dict_values(['Londra', 'Madrid', 'Bu', 'Myanmar', 'Atene', 'Vienna']) <class
'dict_values'>
Ecco le coppie chiave:valore estratte con items()
dict_items([('Regno Unito', 'Londra'), ('Spagna', 'Madrid'), ('Germania', 'Bu'),
('Birmania', 'Myanmar'), ('Grecia', 'Atene'), ('Austria', 'Vienna')]) <class
'dict_items'>
```

Come vediamo dall'output, ciascuno di questi iterabili ha un tipo a sé stante.

Per poterci lavorare meglio potrebbe essere conveniente trasformarli in liste con la funzione *list()* che già conosciamo:

[35]:
```
lista_chiavi = list(chiavi)
lista_valori = list(valori)
lista_oggetti = list(coppie)

print(lista_chiavi)
print(lista_valori)
print(lista_oggetti)
```

```
['Regno Unito', 'Spagna', 'Germania', 'Birmania', 'Grecia', 'Austria']
['Londra', 'Madrid', 'Bu', 'Myanmar', 'Atene', 'Vienna']
[('Regno Unito', 'Londra'), ('Spagna', 'Madrid'), ('Germania', 'Bu'),
```

```
('Birmania', 'Myanmar'), ('Grecia', 'Atene'), ('Austria', 'Vienna')]
```

1.3 Esercizi

1. Dati i due vettori qui sotto, utilizzare gli elementi del primo vettore come chiavi e quelli del secondo vettore come valori per creare un dizionario:

```
vett1 = ['Cane', 'Gatto', 'Topo', 'Maiale'] vett2 = ['Bau', 'Miao', 'Squit',
'Oink']
```

2. Utilizzare il metodo format() per stampare le coppie chiave valore del dizionario nel seguente formato:

```
Chiave1 -> Valore1 Chiave2 -> Valore2 ..... Chiave3 -> Valore3
```

3. Dato il seguente vettore, eliminare gli elementi presenti più volte al suo interno:

```
vett = [1, 2, 3, 4, 1, 5, 5, 6, 3, 7, 1, 8]
```

4. Dati i due vettori seguenti, generare un vettore v3 che contenga solo gli elementi che i due vettori hanno in comune:

```
v1 = [1, 2, 3, 4, 5, 6, 7, 8, 9] v2 = [1, 3, 5, 7, 9, 11, 13,]
```

5. Generare un vettore contenente 50 numeri casuali compresi tra 0 e 10, eliminare i duplicati, ri-ordinare gli elementi in ordine decrescente e stampare il vettore risultante. *Suggerimento:* ad un certo punto è consigliabile cercare online come risolvere uno di questi passaggi.

1.4 Soluzioni

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

1.5 Esercizio 1

Per il primo esercizio vengono proposte due soluzioni.

La prima utilizza un ciclo for per riempire un vettore nel modo appropriato, affinché possa essere usato per costruire un dizionario con dict().

Il secondo metodo utilizza la funzione ***zip(v1, v2)***, che prende due vettori di dimensioni uguali e genera un vettore di tuple così strutturato:

`a = [a1, a2, a3, ..., aN b = [b1, b2, b3, ..., bN zippato = zip(a, b) --->  [(a1, b1), (a2, b2), (a3, b3), ..., (aN, bN]`

[43]:
```
#soluzione 1

vett1 = ['Cane', 'Gatto', 'Topo', 'Maiale']
vett2 = ['Bau', 'Miao', 'Squit', 'Oink']
lista_per_dizionario = [] #lista vuota da riempire con le coppie
for i in range(len(vett1)):
    tupla_coppia = (vett1[i], vett2[i]) #genero la tupla (chiave, valore)
    lista_per_dizionario.append(tupla_coppia) #aggiungo la coppia alla lista
dizionario = dict(lista_per_dizionario) #creo il dizionario con dict()
print(dizionario)
```

```
{'Cane': 'Bau', 'Gatto': 'Miao', 'Topo': 'Squit', 'Maiale': 'Oink'}
```

[44]:
```
#soluzione 2 - funzione zip()

vett1 = ['Cane', 'Gatto', 'Topo', 'Maiale']
vett2 = ['Bau', 'Miao', 'Squit', 'Oink']
lista_per_dizionario = zip(vett1, vett2)

#vediamo cosa ha generato zip:
for entry in lista_per_dizionario:
    print(entry)

#per scrivere il tutto in modo più compatto:

dizionario = dict(zip(vett1, vett2))
print(dizionario)
```

```
('Cane', 'Bau')
('Gatto', 'Miao')
('Topo', 'Squit')
('Maiale', 'Oink')
{'Cane': 'Bau', 'Gatto': 'Miao', 'Topo': 'Squit', 'Maiale': 'Oink'}
```

1.6 Esercizio 2

[45]:
```
for key in dizionario:
    print('{} -> {}'.format(key, dizionario[key]))
```

```
Cane -> Bau
Gatto -> Miao
Topo -> Squit
Maiale -> Oink
```

1.7 Esercizio 3

[ ]:
```
vett = [1, 2, 3, 4, 1, 5, 5, 6, 3, 7, 1, 8]
vettore_pulito = list(set(vett))
print(vettore_pulito)
```

1.8 Esercizio 4

[ ]:
```
v1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
v2 = [1, 3, 5, 7, 9, 11, 13,]

vettore_intersezione = list(set(v1) & set(v2))
print(vettore_intersezione)
```

1.9 Esercizio 5

[ ]:
```
from random import randint

vett = []
for i in range(50):
    vett.append(randint(0, 10))

insieme = set(vett)
vett = list(insieme)
vett.sort(reverse = True)
print(vett)
```