

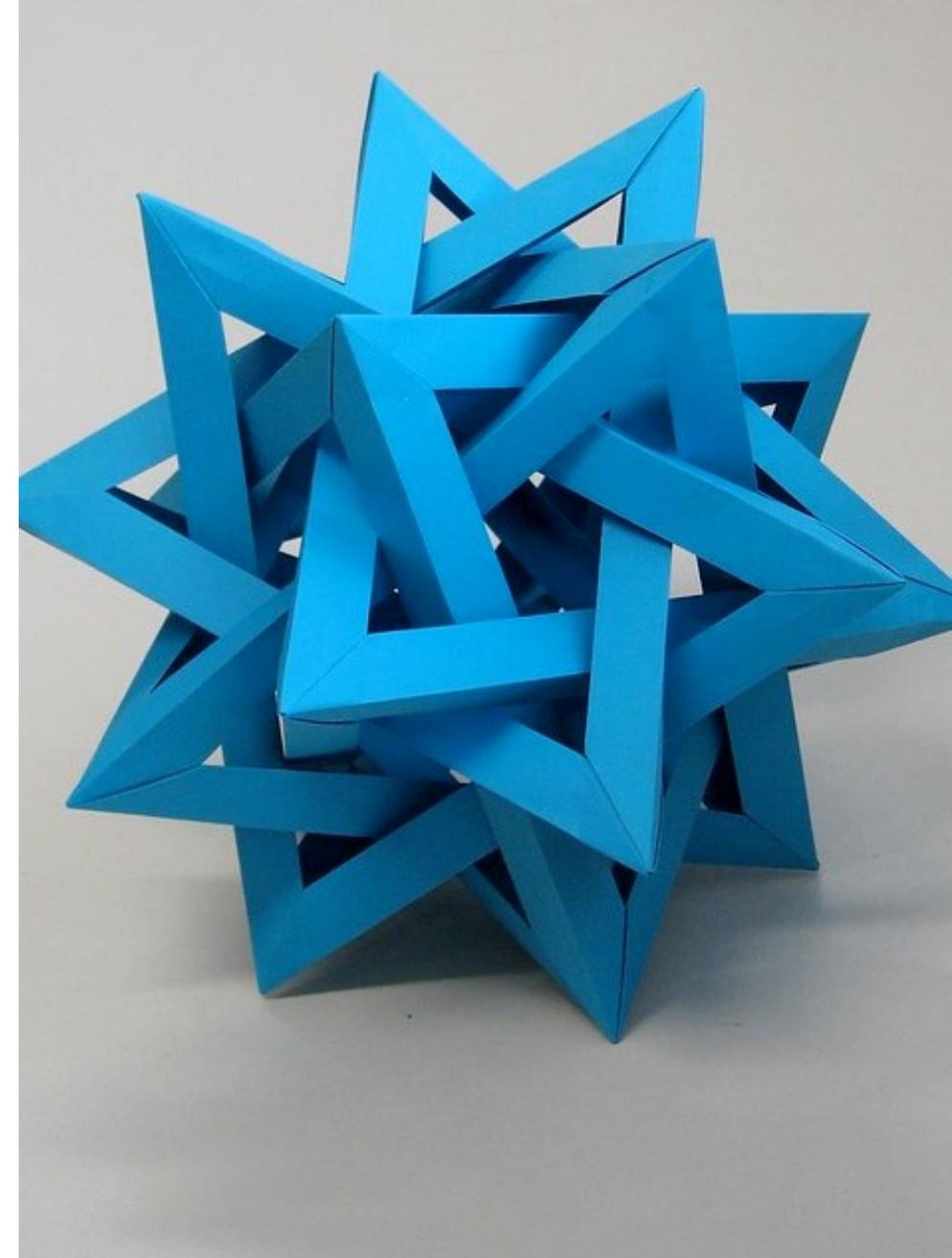


# Unità P8: Strutture dati complesse

INSIEMI, DIZIONARI E COMBINAZIONI DI  
STRUTTURE DATI



Capitolo 8



# Obiettivi dell'unità

- Costruire ed utilizzare contenitori di tipo **insieme (set)**
- Conoscere le operazioni più comuni sugli insiemi
- Costruire ed utilizzare contenitori di tipo **dizionario (dict)**
- Usare i dizionari come tabelle di **ricerca**

In questa unità impareremo come lavorare con due nuovi tipi di contenitori (insiemi e dizionari), e come combinare tra loro diversi contenitori per rappresentare strutture complesse.

# Indice

- Insiemi
- Dizionari
- Strutture complesse

# Insiemi



8.1

---

CONTENITORI MUTABILI NON ORDINATI DI ELEMENTI UNIVOCI  
IMMUTABILI

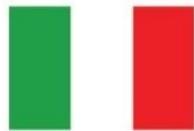
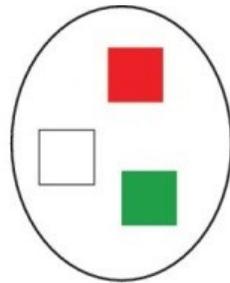
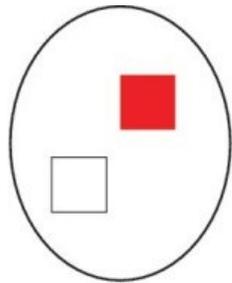
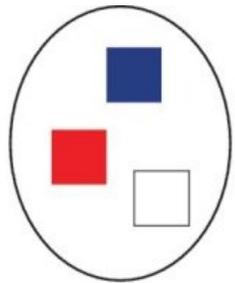
<https://realpython.com/python-sets/>

# Insiemi

- Un insieme (**set**) è un tipo contenitore che memorizza una **raccolta di valori univoci**
  - Per definizione un insieme **non** può avere elementi **duplicati**
- A differenza di una lista, gli elementi (o membri) di un insieme **non** sono memorizzati in alcun **ordine** particolare e **non** vi si può accedere per **posizione**
- Le **operazioni** permesse sono le stesse definite sugli **insiemi matematici**
- Poiché gli insiemi non devono mantenere un ordine particolare, le operazioni su di essi sono molto **più veloci** delle equivalenti sulle liste.

# Esempio di insiemi

- Questi insiemi contengono tre colori: i colori che compongono le bandiere britannica, canadese ed italiana
- In ciascun insieme, l'ordine non è importante ed i colori non sono mai duplicati

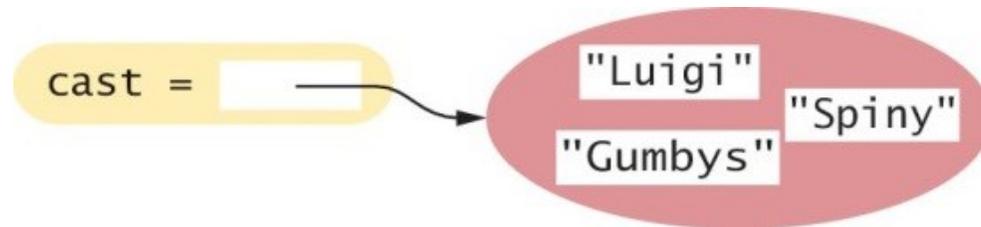


```
uk_flag = { 'blue', 'red', 'white' }  
ca_flag = { 'red', 'white' }  
it_flag = { 'green', 'red', 'white' }
```

# Creare ed usare insiemi

- Per creare un insieme con alcuni elementi iniziali, si possono specificare racchiusi tra parentesi graffe, come in matematica

```
cast = { "Luigi", "Gumbys", "Spiny" }
```



- In alternativa, si può usare la funzione `set()` per convertire una qualunque `sequenza(*)` in un insieme:

```
names = ["Luigi", "Gumbys", "Spiny"]  
cast = set(names)
```

(\*) sequenza: stringa, lista, tupla, range, insieme, ...

# Valori ammessi negli insiemi

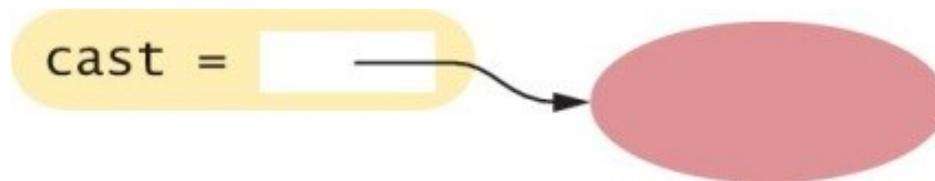
- Per ragioni tecniche legate all'implementazione del linguaggio, i **valori memorizzati** in un insieme (e le chiavi di un dizionario, v. oltre) **devono** essere:
  - Immutabili
  - Comparabili
- **Sì**: numeri (interi o float), stringhe, tuple, oggetti
- **No**: liste, dizionari, file, insiemi
- Per approfondimenti vedere «hashable»:  
<https://docs.python.org/3/glossary.html#term-hashable>

# Creare un insieme vuoto

- *Non* è possibile usare `{}` per creare un insieme vuoto in Python (tale sintassi crea invece un dizionario vuoto)
- Si usa semplicemente la funzione `set()` senza argomenti:

```
cast = set()
```

- Come per gli altri contenitori, si può usare la funzione `len()` per conoscere il numero di elementi in un insieme:



```
number_of_movie_characters = len(cast) # In this case it's zero
```

# Appartenenza ad un insieme: `in`

- Per scoprire se un elemento è contenuto in un insieme, si usa l'operatore `in` oppure il suo inverso, `not in`:

```
if "Luigi" in cast :  
    print("Luigi is a character in Monty Python's Flying Circus.")  
else:  
    print("Luigi is not a character in the show.")
```

# Accedere agli elementi di un insieme

- Si può usare un ciclo **for** ed iterare su ciascun elemento:

```
print("The cast of characters includes:")
for character in cast :
    print(character)
```

- Poiché gli insiemi sono **non ordinati**, **non** si può accedere agli elementi per **posizione** (come si farebbe con una lista)

```
for i in range(len(cast)):
    print(cast[i])
```

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

⊘ cast[i] ⊘

# Gli insiemi non sono ordinati

- Si noti che l'**ordine** in cui gli elementi vengono visitati (in un ciclo o in una stampa) dipende da come essi sono memorizzati internamente – è **imprevedibile** ed è **diverso** dall'ordine di inserimento, e può essere diverso cambiando la versione di Python

```
uk_flag = { 'blue', 'red', 'white' }  
ca_flag = { 'red', 'white' }  
it_flag = { 'green', 'red', 'white' }  
  
print(uk_flag)  
print(ca_flag)  
print(it_flag)
```

```
{'blue', 'white', 'red'}  
{'white', 'red'}  
{'green', 'white', 'red'}
```

# Gli insiemi non sono ordinati

- Per esempio, il ciclo precedente stampa:

The cast of characters includes:

Gumbys

Spiny

Luigi

- Si noti che l'ordine degli elementi nell'output è diverso dall'ordine in cui era stato creato l'insieme

```
print("The cast of characters includes:")  
for character in cast :  
    print(character)
```

```
cast = { "Luigi", "Gumbys", "Spiny" }
```

# Visualizzare un insieme in modo ordinato

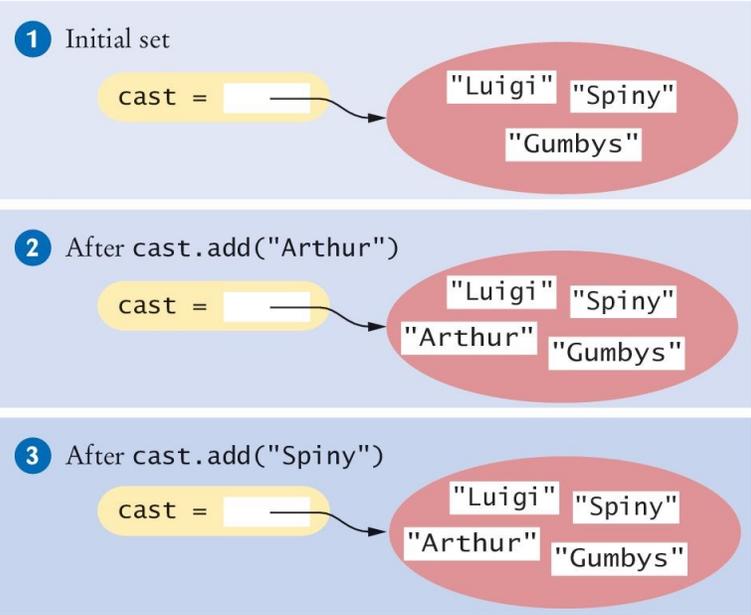
- Si può usare la funzione `sorted()`, che restituisce una **lista** (**non un set**) degli elementi, elencati secondo un ordinamento definito (numeri crescenti/decrescenti, stringhe in ordine alfabetico, ...)
- Il ciclo seguente stampa il cast in ordine alfabetico crescente:

```
for actor in sorted(cast):  
    print(actor)
```

# Aggiungere elementi: add

- Gli insiemi sono collezioni **mutabili**, quindi si possono **aggiungere elementi** usando il metodo **add()**:

```
cast = set(["Luigi", "Gumbys", "Spiny"]) #1
cast.add("Arthur") #2
cast.add("Spiny") #3
```



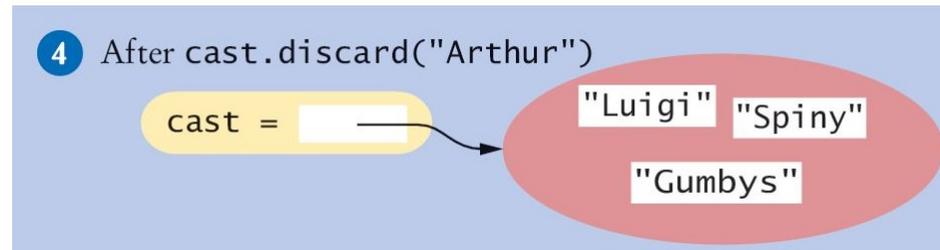
"Arthur" non è nel set, quindi viene aggiunto e la dimensione del set si incrementa di 1

"Spiny" è già nel set, quindi non vi è alcuna modifica al set stesso

# Cancellare elementi: `discard`

- Il metodo `discard()` rimuove un elemento (se tale elemento esiste):

```
cast.discard("Arthur") #4
```



- Non ha alcun effetto se l'elemento non era presente

```
cast.discard("The Colonel") # Has no effect
```

# Cancellare elementi: `remove`

- Esiste anche il metodo `remove()`, che invece
  - Rimuove un elemento da un set, se esso esiste esiste
  - Solleva un'eccezione se l'elemento non fa parte del set:

```
cast.remove("The Colonel") # Raises an exception
```

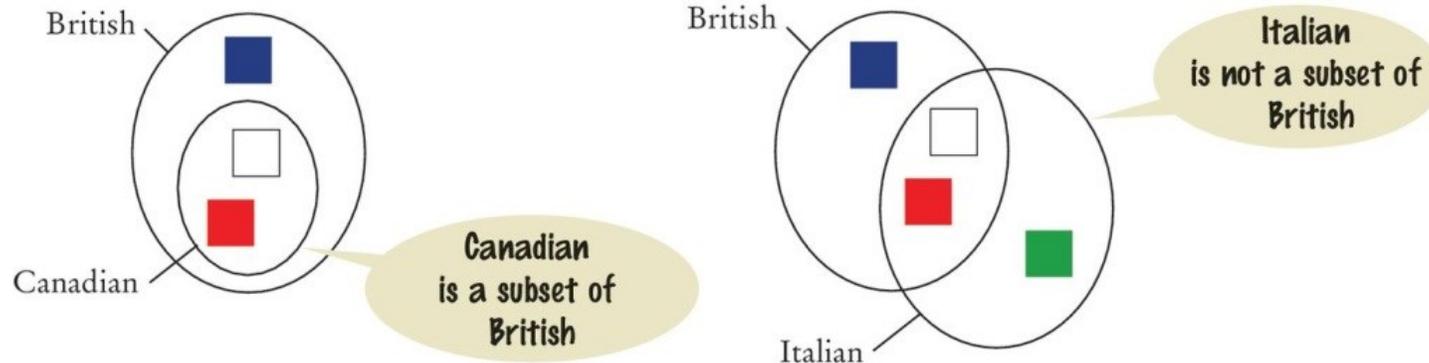
# Cancellare elementi: `clear`

- Il metodo `clear()` cancella tutti gli elementi del set, lasciando l'insieme vuoto:

```
cast.clear() # cast now has size 0
```

# Sottoinsiemi

- Un insieme si dice **sottoinsieme** di un altro insieme, se e solo se **ciascun** elemento del primo insieme è **anche** elemento del secondo
- Nella figura, i colori della bandiera canadese sono un sottoinsieme dei colori britannici
- I colori italiani non sono un sottoinsieme



# Il metodo `issubset`

- Il metodo `issubset()` restituisce `True` o `False` sulla base del fatto che un insieme sia sottoinsieme di un altro:

```
canadian = { "Red", "White" }
british = { "Red", "Blue", "White" }
italian = { "Red", "White", "Green" }

# True
if canadian.issubset(british) :
    print("All Canadian flag colors occur in the British flag.")

# True
if not italian.issubset(british) :
    print("At least one of the colors in the Italian flag does
not.")
```

# Uguaglianza o disuguaglianza di insiemi

- L'uguaglianza tra due insiemi si verifica semplicemente con gli operatori "==" e "!="
- Due insiemi sono uguali se e solo se hanno esattamente gli stessi elementi

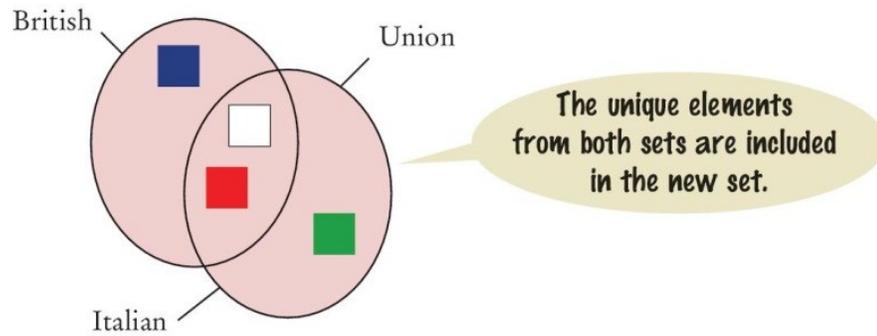
```
french = { "Red", "White", "Blue" }  
if british == french :  
    print("The British and French flags use the same colors.")
```

# Unione di insiemi: `union`

- L'**unione** di due insiemi contiene tutti gli elementi presenti nei due insiemi, escludendo i duplicati

```
# in_either: The set {"Blue", "Green", "White", "Red"}  
in_either = british.union(italian)
```

- Gli insiemi britannico ed italiano contengono entrambi rosso e bianco, ma l'unione è un insieme e quindi contiene una sola copia di ciascun colore

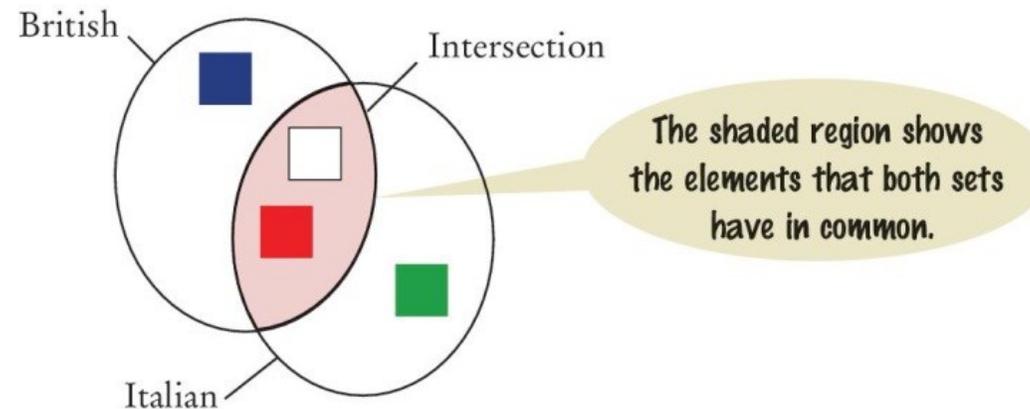


Si noti che il metodo `union()` restituisce un nuovo insieme. Infatti non modifica nessuno dei due insiemi coinvolti

# Intersezione di insiemi : **intersection**

- L'**intersezione** di due insiemi contiene tutti gli elementi presenti in entrambi gli insiemi

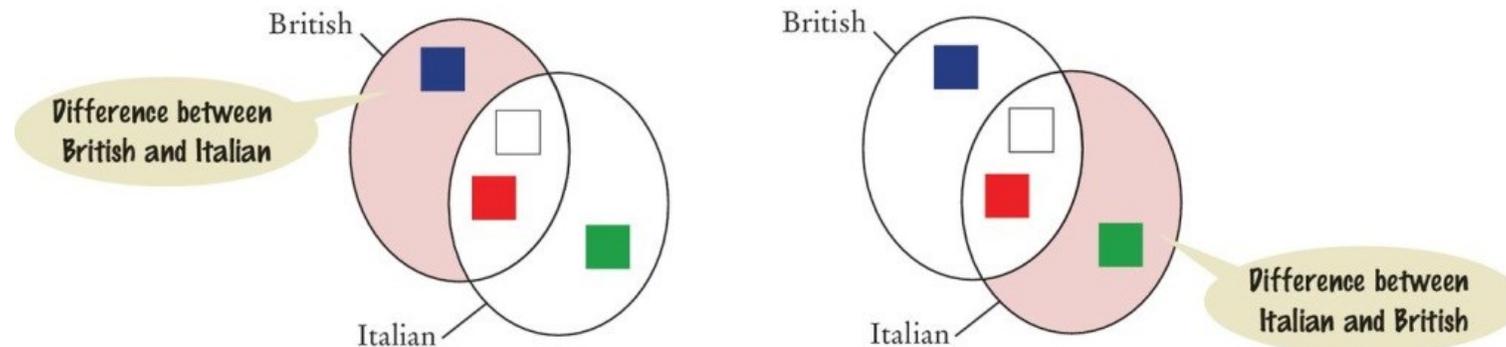
```
# in_both: The set {"White", "Red"}  
in_both = british.intersection(italian)
```



# Differenza tra due insiemi: `difference`

- La **differenza** di due insiemi genera un nuovo insieme che contiene gli elementi del primo insieme **che non sono** nel secondo

```
print("Colors that are in the Italian flag but not the  
British:")  
print(italian.difference(british) # Prints {'Green'}
```



# Operazioni frequenti sugli insiemi

Operazione	Descrizione
<code>s = set()</code> <code>s = set(seq)</code> <code>s = {e<sub>1</sub>, e<sub>2</sub>, ..., e<sub>n</sub>}</code>	Crea un nuovo insieme, rispettivamente: vuoto; che contiene una copia della sequenza <i>seq</i> ; che contiene gli elementi indicati.
<code>len(s)</code>	Restituisce il numero di elementi presenti nell'insieme <i>s</i> .
<code>elemento in s</code> <code>elemento not in s</code>	Determina se <i>elemento</i> appartiene all'insieme <i>s</i> .
<code>s.add(elemento)</code>	Aggiunge un nuovo <i>elemento</i> all'insieme <i>s</i> . Se l'elemento è già presente, non succede nulla.
<code>s.discard(elemento)</code> <code>s.remove(elemento)</code>	Elimina <i>elemento</i> dall'insieme <i>s</i> . Se l'elemento non appartiene all'insieme, <code>discard</code> non ha alcun effetto, mentre <code>remove</code> solleva un'eccezione.
<code>s.clear()</code>	Elimina tutti gli elementi dall'insieme <i>s</i> .
<code>s.issubset(t)</code>	Determina se l'insieme <i>s</i> è un sottoinsieme dell'insieme <i>t</i> .
<code>s == t</code> <code>s != t</code>	Determina se l'insieme <i>s</i> è uguale all'insieme <i>t</i> oppure se sono diversi.
<code>s.union(t)</code>	Restituisce un nuovo insieme che contiene gli elementi che appartengono a <i>s</i> oppure a <i>t</i> (oppure a entrambi).
<code>s.intersection(t)</code>	Restituisce un nuovo insieme che contiene gli elementi che appartengono sia a <i>s</i> sia a <i>t</i> .
<code>s.difference(t)</code>	Restituisce un nuovo insieme che contiene gli elementi che appartengono a <i>s</i> ma non a <i>t</i> .

# Operazioni sugli insiemi

Function	Description
<code>all()</code>	Returns <code>True</code> if all elements of the set are true (or if the set is empty).
<code>any()</code>	Returns <code>True</code> if any element of the set is true. If the set is empty, returns <code>False</code> .
<code>enumerate()</code>	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
<code>len()</code>	Returns the length (the number of items) in the set.
<code>max()</code>	Returns the largest item in the set.
<code>min()</code>	Returns the smallest item in the set.
<code>sorted()</code>	Returns a new sorted list from elements in the set(does not sort the set itself).
<code>sum()</code>	Returns the sum of all elements in the set.

Method	Description
<code>add()</code>	Adds an element to the set
<code>clear()</code>	Removes all elements from the set
<code>copy()</code>	Returns a copy of the set
<code>difference()</code>	Returns the difference of two or more sets as a new set
<code>difference_update()</code>	Removes all elements of another set from this set
<code>discard()</code>	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
<code>intersection()</code>	Returns the intersection of two sets as a new set
<code>intersection_update()</code>	Updates the set with the intersection of itself and another
<code>isdisjoint()</code>	Returns <code>True</code> if two sets have a null intersection
<code>issubset()</code>	Returns <code>True</code> if another set contains this set
<code>issuperset()</code>	Returns <code>True</code> if this set contains another set
<code>pop()</code>	Removes and returns an arbitrary set element. Raises <code>KeyError</code> if the set is empty
<code>remove()</code>	Removes an element from the set. If the element is not a member, raises a <code>KeyError</code>
<code>symmetric_difference()</code>	Returns the symmetric difference of two sets as a new set
<code>symmetric_difference_update()</code>	Updates a set with the symmetric difference of itself and another
<code>union()</code>	Returns the union of sets in a new set
<code>update()</code>	Updates the set with the union of itself and others

<https://www.programiz.com/python-programming/dictionary>

# ⚡ Scorciatoie di Python: operatori abbreviati

L'operazione (usando un metodo)	È equivalente a (usando un operatore)
<code>x1.union(x2)</code>	<code>x1   x2</code>
<code>x1.intersection(x2)</code>	<code>x1 &amp; x2</code>
<code>x1.difference(x2[, x3 ...])</code>	<code>x1 - x2</code>
<code>x1.symmetric_difference(x2)</code>	<code>x1 ^ x2</code>
<code>x1.issubset(x2)</code>	<code>x1 &lt;= x2</code>
	<code>x1 &lt; x2</code>
<code>x1.issuperset(x2)</code>	<code>x1 &gt;= x2</code>
	<code>x1 &gt; x2</code>

⚠ Più brevi e compatti, ma meno leggibili

# Suggerimento

- Nei programmi che devono gestire raccolte di elementi **univoci**, gli **insiemi** sono molto più efficienti delle liste
- Alcuni programmatori sono abituati ad usare le liste, ed invece di:

```
item_set.add(item)
```

- usano:

```
if (item not in item_list)  
    item_list.append(item)
```

- Il programma in questo modo è molto più lento
  - Fino a 10 volte...!
  - Ripetere l'operazione 'not in' implica la scansione ripetuta degli elementi della lista...

# Esempio: Conteggio di parole uniche

---

# Definizione del problema

- Vogliamo contare il numero di parole uniche (distinte tra loro) in un documento di testo
  - Es. il testo di “Mary had a little lamb” contiene 57 parole uniche
- Il nostro compito è scrivere un programma che legge un documento di testo da un file e determina il numero di parole uniche presenti nel documento

# Passo 1: Capire il problema

- Per contare il numero di parole uniche nel documento, dobbiamo saper determinare se una parola sia già stata incontrata prima nel documento
  - Solo la *prima occorrenza* di una parola deve essere *contata*
- Il modo più semplice di farlo è leggere una parola per volta dal file ed aggiungerla ad un insieme
  - Poiché un insieme non può contenere duplicati. Possiamo usare il metodo `add`
  - Il metodo `add` impedirà ad una parola che era già stata incontrata di essere aggiunta una seconda volta
- Dopo l'elaborazione di tutte le parole del documento, la dimensione dell'insieme sarà uguale al numero di parole uniche contenute nel documento

# Passo 2: Decomporre il problema

- Creare un insieme vuoto
- Per ogni parola nel documento
  - Aggiungi la parola all'insieme
- Numero di parole uniche = dimensione dell'insieme
  
- Creare un insieme vuoto, aggiungere un elemento all'insieme, determinare la dimensione dell'insieme: tutte operazioni standard sugli insiemi
- Leggere le parole dal file può essere gestito da una funzione a parte

# Passo 3: Costruire l'insieme

- Dobbiamo leggere le singole parole del file. Per semplicità usiamo una costante letterale come nome del file

```
input_file = open('nurseryrhyme.txt', 'r')
```

```
for line in input_file :
```

```
    the_words = line.split()
```

```
    for words in the_words :
```

```
        Process word
```

- Per contare le parole uniche dovremo eliminare tutti i **caratteri non letterali** e ignorare le **maiuscole**
- Costruiremo una funzione per «pulire» le parole prima di aggiungerle all'insieme

# Passo 4: «Pulire» le parole

- Per eliminare tutti i caratteri che non sono lettere, possiamo iterare sulla stringa, un carattere alla volta, e costruire una nuova parola «pulita»

```
def clean(string) :  
    result = ''  
    for char in string :  
        if char.isalpha() :  
            result = result + char  
    return result.lower()
```

# Passo 5: Mettiamo tutto insieme

- Implementiamo la funzione `main()` e combiniamola con le altre funzioni
- Aprire il file: `countwords.py`

 `countwords.py`

# Esempio: Controllo ortografico

---

# Esempio: Controllo ortografico

- Controllare il corretto 'spelling' di un documento, stampando tutte le parole che non compaiono in un apposito dizionario
- Dati:
  - `words.txt`: file di testo contenente le parole corrette (una per riga)
  - `story.txt`: file di testo libero contenente un racconto (in questo caso il libro Alice nel Paese delle Meraviglie)
- Stampare tutte le parole del documento che non si trovano tra le parole corrette

# Progettazione

- Leggere il contenuto del dizionario (una parola per riga) in un insieme di stringhe
- Leggere il contenuto del file di testo contenente il racconto
  - Separare le singole parole, isolandole dagli spazi e dai segni di punteggiatura
  - Memorizzare le parole in un secondo insieme
- Attenzione alla differenza maiuscole/minuscole
- Calcolare la differenza tra i due insiemi: parole che sono presenti nel testo, ma non sono presenti nel dizionario
- Stampare tale differenza

# Esempio: spellcheck.py (1)

```
WORDS_FILENAME = 'words.txt'
STORY_FILENAME = 'story.txt'

def main():
    words = read_words(WORDS_FILENAME) # parole del dizionario

    story = read_words(STORY_FILENAME) # parole della storia

    # se non ci sono errori => storia è un sottoinsieme del dizionario
    # se ci sono errori => errori = storia - dizionario

    if story.issubset(words):
        print("La storia non ha errori lessicali")
    else:
        misspelled_words = story.difference(words)
        print("Le parole sbagliate sono:")
        print(sorted(misspelled_words))
```



 spellcheck.py

# Esempio: spellcheck.py (2)

```
def read_words(filename):
```

*Dato il nome di un file, restituisce un \*\*insieme\*\* che contiene le parole uniche presenti nel file  
Attenzione: rimuove tutti i segni di punteggiatura*

*:param filename: nome del file da leggere  
:return: insieme contenente tutte le parole singole*

```
    try:
        file = open(filename, 'r')
    except IOError:
        print(f"Errore nell'apertura del file {filename}")
        exit()

    insieme = set()

    for line in file:
        parole = line.rstrip().replace('-', ' ').split()
        for parola in parole:
            clean_word = only_alphabetic(parola)
            if clean_word != '':
                insieme.add(clean_word.lower())

    file.close()
    return insieme
```



 spellcheck.py

# Esempio: spellcheck.py (3)

```
def only_alphabetic(parola):
```

*rimuove tutti i caratteri INIZIALI e FINALI non alfabetici da una stringa*

*:param parola: una qualsiasi parola*

*:return: la stessa parola, a cui sono stati rimossi segni non alfabetici iniziali e finali*

```
    pulita = parola
```

```
    while len(pulita) > 0 and not pulita[0].isalpha():  
        pulita = pulita[1:]
```

```
    while len(pulita) > 0 and not pulita[-1].isalpha():  
        pulita = pulita[:-1]
```

```
    return pulita
```



 spellcheck.py

# Esecuzione: spellcheck.py

```
...  
champaign  
chatte  
clamour  
comfits  
conger  
croqueted  
croqueting  
cso  
daresay  
dinn  
dir  
draggled  
dutchess  
...
```

# Dizionari



8.2

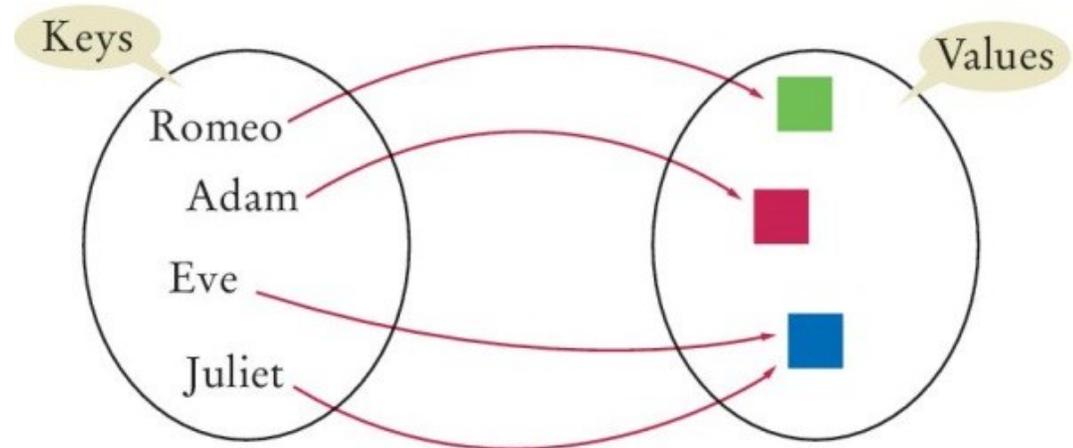
---

CONTENITORI MUTABILI DI COPPIE CHIAVE-VALORE, UTILIZZABILI  
COME ARRAY ASSOCIATIVI O COME RECORD DATI

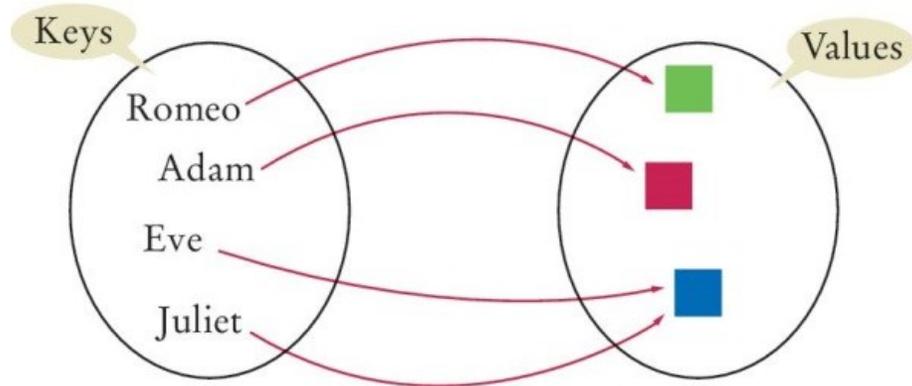
<https://realpython.com/python-dicts/>

# Dizionari

- Un dizionario è un contenitore che memorizza delle associazioni tra **chiavi** e **valori**
  - Viene talvolta chiamato *array associativo* o *mappa*
- Ogni **chiave** nel dizionario ha un **valore** ad essa associato
- Le **chiavi** sono **uniche**, ma un **valore** può essere associato a **diverse chiavi**
- Esempio (la corrispondenza chiave-valore è indicata dalla freccia):



# Sintassi per creare dizionari



```
{  
  "Romeo": "green",  
  "Adam": "purple",  
  "Eve": "blue",  
  "Juliet": "blue"  
}
```

# Nota

## Ricordiamo...

### Valori ammessi negli insiemi

- Per ragioni tecniche legate all'implementazione del linguaggio, i **valori memorizzati** in un insieme (e le chiavi di un dizionario, v. oltre) **devono** essere:
  - Immutabili
  - Comparabili
- **Si**: numeri (interi o float), stringhe, tuple, oggetti
- **No**: liste, dizionari, file
  
- Per approfondimenti vedere «hashable»:  
<https://docs.python.org/3/glossary.html#term-hashable>

- Solitamente le **chiavi** di un dizionario sono
  - Stringhe (nomi di campi, codici identificativi, ...)
  - Numeri (matricole, identificatori numerici, ...)
  
- Non c'è alcun limite al tipo di dati che può assumere il campo **valore** di un dizionario

# Sintassi: Insiemi e Dizionari

## Esempio

Gli elementi di insiemi e dizionari sono racchiusi tra parentesi graffe.

I dizionari contengono coppie chiave/valore.

Un insieme

```
colors = { "Red", "Green", "Blue" }
```

Chiave

Valore

```
favoriteColors = { "Romeo": "Green", "Adam": "Red" }
```

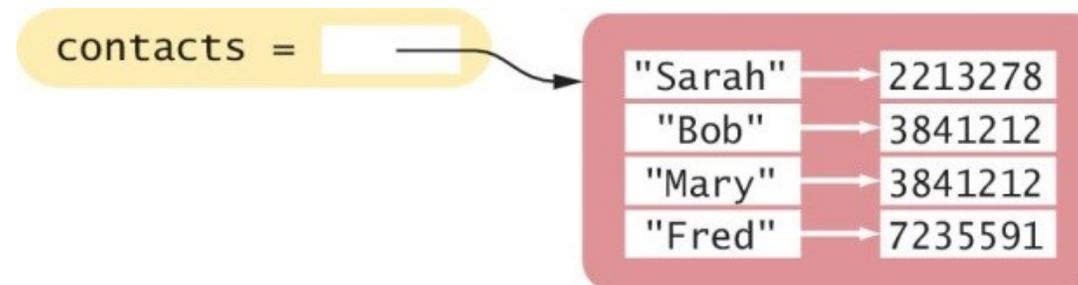
Una coppia di parentesi graffe vuota è un dizionario.

```
emptyDict = {}
```

# Creare dizionari

- Supponiamo di voler creare un programma che ricerchi il numero di telefono di una persona, come nella rubrica telefonica
- Potremmo usare un dizionario in cui i nomi sono le chiavi ed i numeri di telefono sono i valori

```
contacts = { "Fred": 7235591, "Mary": 3841212, "Bob":  
            3841212, "Sarah": 2213278 }
```



# Duplicare dizionari: `dict()`

- Si può creare una **copia** (un **duplicato**) di un dizionario usando la funzione `dict()` :

```
copyOfContacts = dict(contacts)
```

- Come caso particolare, si può creare un dizionario vuoto

```
contacts = dict()  
contacts = {}
```

# Duplicare dizionari: `dict()`

- Si può creare una **copia** (un **duplicato**) di un dizionario usando la funzione `dict()` :

```
copyOfContacts = dict(contacts)
```

- Come caso particolare, si può creare un

```
contacts = dict()  
contacts = {}
```



Attenzione: una semplice assegnazione  
`contacts2 = contacts`  
Non crea una copia, ma semplicemente un  
«**alias**», ossia una nuova variabile che si  
riferisce **allo stesso dizionario**

Provare su PythonTutor!

# Accedere ai valori di un dizionario: [ ]

- L'operatore di indicizzazione [ ] viene usato per restituire il valore associato ad una chiave data
  - Sintassi: `dictionary [ key ]`
- La chiave può essere una costante, una variabile o un'espressione
- Si noti che il dizionario **non** è un contenitore di tipo **sequenziale** (come una lista)
  - Non si può accedere agli elementi per indice o posizione
  - Si può accedere ad un valore **solamente usando la sua chiave associata**

```
print("Fred's number is",  
      contacts["Fred"])  
# stampa 7235591
```

**⚠** *La chiave usata nell'operatore di indicizzazione deve essere una chiave **esistente** nel dizionario, altrimenti si verificherà un'eccezione **KeyError***

# Dizionari: verifica dell'esistenza di una chiave

- Per scoprire **se una chiave è presente** nel dizionario, si usa l'operatore **in** (oppure **not in**)
  - In questo modo si evitano le eccezioni nel caso in cui la chiave non esista

```
if "John" in contacts :  
    print("John's number is", contacts["John"])  
else :  
    print("John is not in my contact list.")
```

# Chiavi mancanti

- Cercando di accedere con una chiave non esistente, si ottiene un **KeyError**
- Ricordarsi di controllare l'esistenza di una chiave prima di accedervi

```
if "Tony" in contacts:  
    print(contacts["Tony"])  
else:  
    print("Missing")
```

# Valori di default

- Come scorciatoia, si può usare il metodo `get` per accedere al dizionario, che permette di specificare anche un **valore di default**
  - Sintassi: `dictionary.get(key, default)`
- Nel caso in cui **non** ci sia una chiave corrispondente, viene restituito il valore di **default** (secondo argomento)

```
number = contacts.get("Tony", "missing")
```

*Equivalente a ...*

```
if "Tony" in contacts:  
    number = contacts["Tony"]  
else:  
    number = "missing"
```

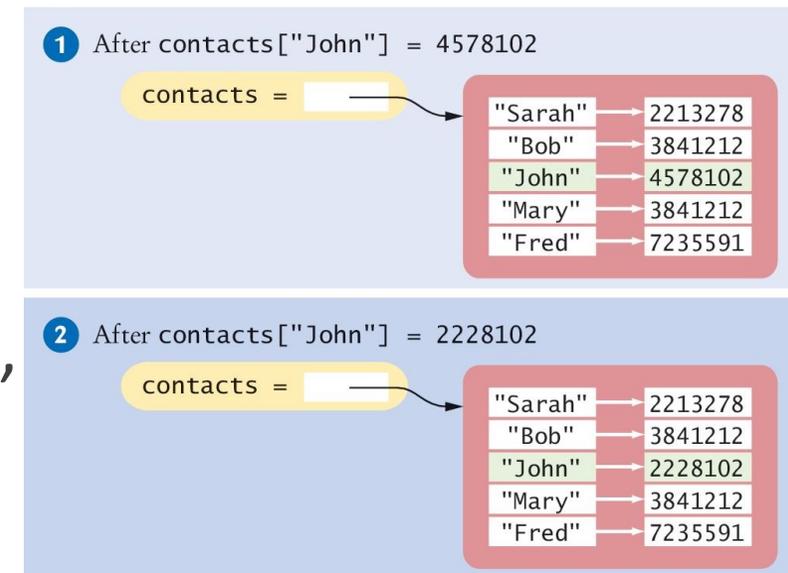
# Aggiungere o modificare elementi

- Un dizionario è un contenitore *mutabile*
- Si può **aggiungere** un nuovo elemento usando l'operatore di indicizzazione `[]` (non si può fare con le liste!)

```
contacts["John"] = 4578102 #1
```

- Per **modificare** il valore associato ad una data chiave, basta impostare un nuovo valore con l'operatore `[]` sulla **chiave esistente**

```
contacts["John"] = 2228102 #2
```



# Aggiungere dinamicamente nuovi elementi

- Spesso non si conosce il contenuto del dizionario nel momento in cui lo creiamo
  - I dati verranno acquisiti dall'utente o da un file...
- Possiamo partire creando un dizionario vuoto...

```
favoriteColors = {}
```

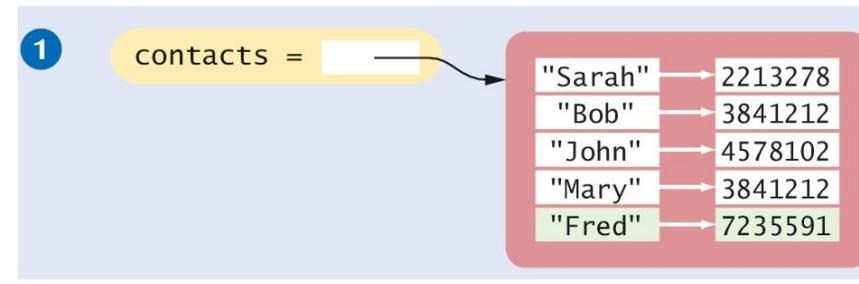
- ...e poi aggiungere gli elementi, quando li conosceremo

```
favoriteColors["Juliet"] = "Blue"  
favoriteColors["Adam"] = "Red"  
favoriteColors["Eve"] = "Blue"  
favoriteColors["Romeo"] = "Green"
```

# Cancellare elementi

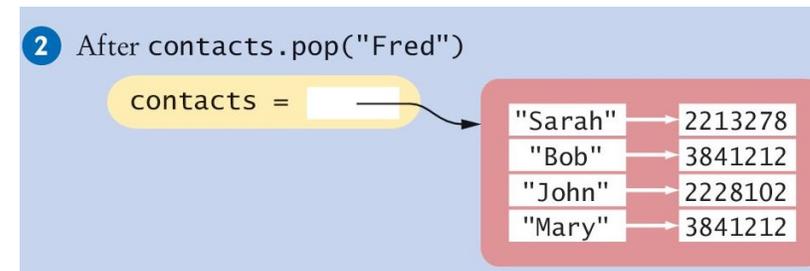
- Per eliminare un elemento da un dizionario, chiamare il metodo `pop()` con la chiave da eliminare come argomento:

```
contacts = { "Fred":  
            7235591, "Mary": 3841212,  
            "Bob": 3841212, "Sarah":  
            2213278 }
```



- Questo metodo cancella **l'intero elemento**, sia la chiave che il suo valore associato

```
contacts.pop("Fred")
```



# Ricordarsi l'elemento cancellato

- Il metodo `pop()` restituisce il valore dell'elemento cancellato, ne possiamo approfittare per memorizzarlo in una variabile:

```
freds_number = contacts.pop("Fred")
```

- Nota: Se la chiave non esiste nel dizionario, il metodo `pop` solleva un'eccezione `KeyError`
  - Per evitare l'eccezione, occorre controllare l'esistenza della chiave nel dizionario:

```
if "Fred" in contacts :  
    contacts.pop("Fred")
```

# Attraversare un dizionario

- È possibile **iterare** su tutte le **chiavi** di un dizionario usando un semplice ciclo `for-in`:

```
print("My Contacts:")  
for key in contacts :  
    print(key)
```

- Questo codice genera:

My Contacts:

Fred

Mary

Bob

Sarah

Il dizionario memorizza le chiavi nell'**ordine di inserimento**.

Nota: nelle versioni di Python < 3.6 questo ordine non era garantito

# Attraversare un dizionario in ordine di valore

- Per iterare sulle chiavi **in ordine di valore della chiave (alfabetico o numerico)**, si può usare la funzione `sorted()` nel costruire il ciclo `for` :
- In questo caso i contatti saranno stampati ordinati alfabeticamente per nome:

```
My Contacts:  
Bob 3841212  
Fred 7235591  
John 4578102  
Mary 3841212  
Sarah 2213278
```

```
print("My Contacts:")  
for key in sorted(contacts) :  
    print(key, contacts[key])
```

# Iterare in modo più efficiente su un dizionario

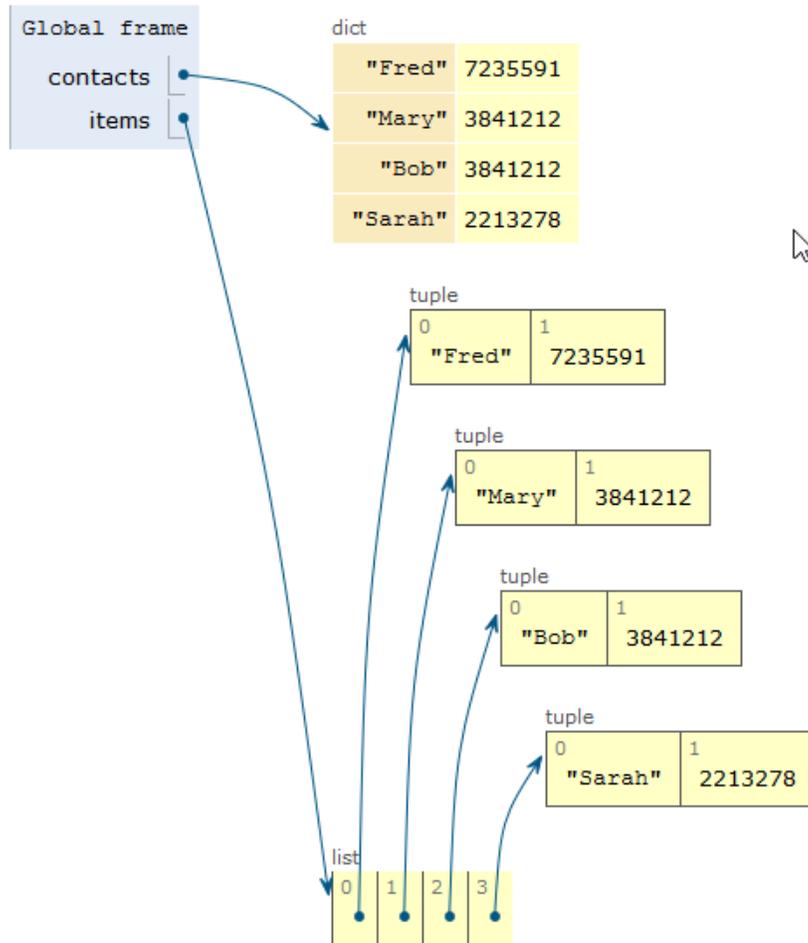
- Python permette di **iterare sugli elementi (chiave+valore)** in un dizionario usando il metodo `items()`
  - È più efficiente rispetto ad iterare sulle chiavi, e fare molti accessi al dizionario per ottenere i valori corrispondenti a ciascuna chiave
- Il metodo `items()` restituisce una **sequenza di *tuple*** che contengono le chiavi ed i valori degli elementi
  - La variabile `item` del ciclo for viene assegnata ad una ***tupla***, che contiene come primo campo [0] la **chiave** e come secondo [1] il **valore**

```
for item in contacts.items() :  
    print(item[0], item[1])
```

```
for (key, val) in contacts.items() :  
    print(key, val)
```

👁️ assegnare dei nomi alle componenti della tupla migliora di molto la leggibilità

# Esempio



```
for item in contacts.items() :  
    print(item[0], item[1])
```

```
for (key, val) in contacts.items() :  
    print(key, val)
```

# Operazioni frequenti con i dizionari

Operazione	Descrizione
$d = \text{dict}()$ $d = \text{dict}(c)$	Crea un nuovo dizionario vuoto o contenente una copia del dizionario $c$ .
$d = \{\}$ $d = \{k_1: v_1, k_2: v_2, \dots, k_n: v_n\}$	Crea un nuovo dizionario vuoto o contenente inizialmente le coppie specificate, ognuna delle quali è costituita da una chiave ( $k$ ) e un valore ( $v$ ), separati da un carattere “due punti”.
$\text{len}(d)$	Restituisce il numero di coppie presenti nel dizionario $d$ .
$\text{chiave in } d$ $\text{chiave not in } d$	Determina se la <i>chiave</i> appartiene o non appartiene al dizionario $d$ .
$d[\text{chiave}] = \text{valore}$	Aggiunge una nuova coppia chiave/valore al dizionario $d$ , se <i>chiave</i> non vi è già presente, altrimenti modifica il valore associato alla <i>chiave</i> .
$x = d[\text{chiave}]$	Restituisce il valore associato alla <i>chiave</i> data, che deve essere presente nel dizionario $d$ , altrimenti viene sollevata un’eccezione.
$d.\text{get}(\text{chiave}, \text{predefinito})$	Restituisce il valore associato alla <i>chiave</i> data, oppure il valore <i>predefinito</i> se la <i>chiave</i> non è presente nel dizionario $d$ .
$d.\text{pop}(\text{chiave})$	Elimina dal dizionario $d$ la <i>chiave</i> e il valore ad essa associato; se la <i>chiave</i> non è presente nel dizionario, solleva un’eccezione.
$d.\text{values}()$	Restituisce una sequenza contenente tutti i valori presenti nel dizionario.

# Funzioni e metodi per i dizionari

Function	Description
<code>all()</code>	Return <code>True</code> if all keys of the dictionary are True (or if the dictionary is empty).
<code>any()</code>	Return <code>True</code> if any key of the dictionary is true. If the dictionary is empty, return <code>False</code> .
<code>len()</code>	Return the length (the number of items) in the dictionary.
<code>cmp()</code>	Compares items of two dictionaries. (Not available in Python 3)
<code>sorted()</code>	Return a new sorted list of keys in the dictionary.

Method	Description
<code>clear()</code>	Removes all items from the dictionary.
<code>copy()</code>	Returns a shallow copy of the dictionary.
<code>fromkeys(seq[, v])</code>	Returns a new dictionary with keys from <code>seq</code> and value equal to <code>v</code> (defaults to <code>None</code> ).
<code>get(key[,d])</code>	Returns the value of the <code>key</code> . If the <code>key</code> does not exist, returns <code>d</code> (defaults to <code>None</code> ).
<code>items()</code>	Return a new object of the dictionary's items in (key, value) format.
<code>keys()</code>	Returns a new object of the dictionary's keys.
<code>pop(key[,d])</code>	Removes the item with the <code>key</code> and returns its value or <code>d</code> if <code>key</code> is not found. If <code>d</code> is not provided and the <code>key</code> is not found, it raises <code>KeyError</code> .
<code>popitem()</code>	Removes and returns an arbitrary item ( <b>key, value</b> ). Raises <code>KeyError</code> if the dictionary is empty.
<code>setdefault(key[,d])</code>	Returns the corresponding value if the <code>key</code> is in the dictionary. If not, inserts the <code>key</code> with a value of <code>d</code> and returns <code>d</code> (defaults to <code>None</code> ).
<code>update([other])</code>	Updates the dictionary with the key/value pairs from <code>other</code> , overwriting existing keys.
<code>values()</code>	Returns a new object of the dictionary's values

<https://www.programiz.com/python-programming/dictionary>

# Dizionari come «array associativi»

---



# Array Associativo

	Lista (Array)	Dizionario (Array Associativo)
Tipo di struttura dati	Sequenza	Mapping
Informazioni memorizzate	Sequenza di valori	Insieme di coppie (chiave, valore)
Chiavi/indici	Indice intero tra $0$ e $\text{len}() - 1$	Valore chiave immutabile: numero, stringa, tupla
Valori	Qualsiasi	Qualsiasi

- Un dizionario è un buon candidato a memorizzare informazioni in cui ciascun **elemento** sia identificabile da un **valore-chiave univoco**
- Lo possiamo immaginare come una lista, in cui l'**indice** non sia necessariamente numerico (può essere stringa, tupla, ...)

# Esempio (1)

- Calcoliamo la frequenza con cui compaiono dei nomi in un elenco di persone
- *Chiave*: nome di una persona (stringa)
- *Valore*: numero di volte in cui compare (intero)

```
count = {  
    'Jimmy': 3,  
    'Timmy': 2,  
    'Tommy': 4  
}
```

```
count[nome] = count[nome]+1
```

## Esempio (2)

- Ciascuno studente ha una matricola. A tale matricola è associato il suo nome reale
- *Chiave*: numero di matricola (intero)
- *Valore*: nome dello studente (stringa)
- Nota: le chiavi sono numeri interi, ma non consecutivi, né compresi tra 0 e len()-1

```
studenti = {  
    123456: 'Mario Rossi',  
    211003: 'Paolo Bianchi',  
    234321: 'Anna Verdi'  
}
```

# Esempio (2 bis)

- Ciascuno studente ha una matricola. A tale matricola sono associati il suo nome ed il suo cognome
- *Chiave*: numero di matricola (intero)
- *Valore*: cognome e nome dello studente (lista, con 2 elementi di tipo stringa)

```
studenti = {  
    123456: ['Mario', 'Rossi'],  
    211003: ['Paolo', 'Bianchi'],  
    234321: ['Anna', 'Verdi']  
}
```

# Esempio (3)

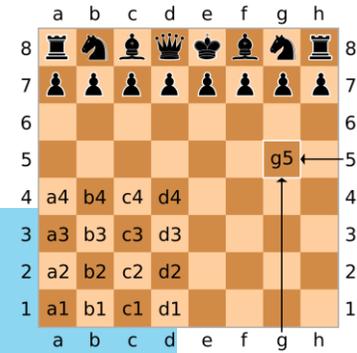
- In una partita a scacchi, diversi pezzi sono posizionati sulle diverse caselle
- *Chiave*: coordinate della casella (tupla contenente due interi: riga e colonna)
- *Valore*: tipo di pezzo (stringa)

```
pezzi = {  
    (1, 3): 'Re',  
    (2, 2): 'Pedone',  
    (8, 8): 'Torre'  
}
```

# Esempio (3 bis)

- In una partita a scacchi, diversi pezzi sono posizionati sulle diverse caselle
  - Utilizziamo la notazione standard per le caselle
  - Utilizziamo la notazione standard per i pezzi
- *Chiave*: coordinate della casella (tupla contenente una riga: stringa e una colonna: intero)
- *Valore*: pezzo (tupla contenente il tipo: stringa ed il colore: stringa)

```
pezzi = {  
    ('b', 3): ('R', 'B'),  
    ('c', 2): ('P', 'N'),  
    ('h', 8): ('T', 'N')  
}
```



[https://en.wikipedia.org/wiki/Algebraic\\_notation\\_\(chess\)](https://en.wikipedia.org/wiki/Algebraic_notation_(chess))

# Dizionari come «record dati»

---



Argomenti  
avanzati  
8.3

# Memorizzare record di dati

- I **record di dati**, in cui ciascun record è composto da **diversi campi**, sono strutture molto frequenti
- Talvolta i singoli campi si possono memorizzare come elementi di una lista  

```
person = [ 'John', 'Doe', 1971, 'New York', 'Student' ]
```
- Ciò però richiederebbe di ricordare in quale posizione della lista è memorizzato ciascun campo
  - Può essere causa di errori di esecuzione se si usa l'elemento sbagliato nell'elaborare il record
  - `person[1]` è il cognome... o era `person[2]`?
- In Python, è frequente **usare un dizionario per memorizzare record dati**
  - Simile a quanto altri linguaggi fanno con «strutture» (C/C++) o «oggetti» (JS, Java)

# Dizionari come record dati

- Si crea un dizionario per ciascun record dati. Nel dizionario, la **chiave** è il **nome** del campo, ed il **valore** contiene i **dati** associati a tale campo.

- Esempio (dati di uno studente):

Ricorda che la chiave deve essere una stringa, non dimenticare le virgolette

```
person1 = {  
    'firstName': 'John',  
    'lastName': 'Doe',  
    'birthdate': 1971,  
    'city': 'New York',  
    'profession': 'Student' }
```

- A questo punto potremo creare una lista contenente tali studenti

```
people = [ person1, person2, person3, ... ]
```

# Lettura da file contenenti record dati

- Per estrarre i record dati da un file, possiamo definire una funzione che legga un singolo record e restituisca un dizionario appositamente creato
- Es.: se il file contiene record fatti da nazione e popolazione separati da due-punti

```
Afghanistan:32738376  
Akrotiri:15700  
Albania:3619778  
Algeria:33769669  
American Samoa:57496  
Andorra:72413  
Angola:12531357  
Anguilla:14108
```

```
def extract_record(infile):  
    record = {}  
    line = infile.readline()  
    if line != "":  
        fields = line.split(":")  
        record["country"] = fields[0]  
        record["population"] = int(fields[1])  
    return record
```

# Lettura da file contenenti record dati (2)

- Il dizionario creato dalla funzione `extract_record` ha due elementi, uno con chiave "country" e l'altro con chiave "population"
- Il risultato di questa funzione si può usare per stampare i record sul terminale

 world\_population.py

```
infile = open("populations.txt", "r")
record = extract_record(infile)
while len(record) > 0:
    print(f"{record['country']:20} {record['population']:10}")
    record = extract_record(infile)
```

- Oppure per memorizzarli in una lista (`lista.append(record)`)

# Esempio: leggiamo i dati degli studenti

MATRICOLA,COGNOME,NOME,EMAIL,COD.INS. E CREDITI,CDS STUDENTE,FREQUENZA,ESAME,ATHOME,SDSS,DUMMY,INFO MOBILITA,STUDENTE ALL'ESTERO,INS IN L.A.,ANNO ISCRIZ.,CONDIZIONE

293374,LA GALA,VITO,s297372@studenti.polito.it,14BHDMK (8),ELN1T3,Frequentato nel 2020/2021, - , - , - , - , - ,N,N,2021/22,ATTIVO

296912,VULTAGGIO,MATTIA,s293359@studenti.polito.it,14BHDPPI (8),BIO1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

286405,GURGIGNO,FRANCESCA,s298343@studenti.polito.it,14BHDMQ (8),CIN1T3,Da frequentare (può sost. esami), - , - ,N, - , - ,N,N,2021/22,ATTIVO

296327,LEONE,DAVIDE,s294681@studenti.polito.it,14BHDPPI (8),ENE1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2020/21,ATTIVO

298449,CAULA,MATTEO,s285164@studenti.polito.it,14BHDPD (8),MEC1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2020/21,ATTIVO

297084,MAROVINO,ANDREA REMI,s287096@studenti.polito.it,14BHDMB (8),ELN1T3,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2020/21,ATTIVO

297153,LAPENNA,NICOLA,s285076@studenti.polito.it,14BHDOA (8),PRO1B1,Da frequentare (può sost. esami), - , - ,N, - , - ,N,N,2020/21,ATTIVO

298369,HU,NIKOLO,s294429@studenti.polito.it,14BHDLZ (8),MEC1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

300685,IOSSA,EDOARDO,s297226@studenti.polito.it,14BHDMN (8),ENE1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2020/21,ATTIVO

284268,LASAGNO,FRANCESCO,s285100@studenti.polito.it,14BHDMK (8),INF1T3,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2020/21,ATTIVO

285922,CARAMAZZA,FRANCESCO,s295497@studenti.polito.it,14BHDLZ (8),EDI1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

284777,CEKAJ,MAURO,s283627@studenti.polito.it,14BHDLN (8),AER1T1,Da frequentare (può sost. esami), - , - ,N, - , - ,N,N,2021/22,ATTIVO

281432,LICITRA,LUCA,s293415@studenti.polito.it,14BHDPPI (8),AER1T1,Frequentato nel 2020/2021, - , - , - , - , - ,N,N,2020/21,ATTIVO

298032,LISA,ANDREA,s300650@studenti.polito.it,14BHDMN (8),MEC1T1,Frequentato nel 2020/2021, - , - , - , - , - ,N,N,2021/22,ATTIVO

297155,LASORSA,EMILIO,s294183@studenti.polito.it,14BHDMB (8),ENE1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

282841,LANDINI,MATTIA,s283587@studenti.polito.it,14BHDDOD (8),FIS1T3,Da frequentare (può sost. esami), - , - ,N, - , - ,N,N,2021/22,ATTIVO

283428,JUGET,NOEMI,s297302@studenti.polito.it,14BHDLZ (8),PRO1B1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2020/21,ATTIVO

295286,ALBANI,LUCIA,s299990@studenti.polito.it,14BHDLZ (8),AER1T1,Da frequentare (può sost. esami), - , - ,N, - , - ,N,N,2021/22,ATTIVO

299050,HIRJEU,ALESSIA,s298194@studenti.polito.it,14BHDMO (8),EDI1T1,Da frequentare (può sost. esami), - , - ,N, - , - ,N,N,2020/21,ATTIVO

285380,LANZA,SIMONE FRANCESCO,s298021@studenti.polito.it,14BHDPD (8),EDI1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2020/21,ATTIVO

299212,CHEN,IRENE,s298905@studenti.polito.it,14BHDLZ (8),CIV1T1,Frequentato nel 2020/2021, - , - ,N, - , - ,N,N,2020/21,ATTIVO

300794,HENRIOD,NICCOLO,s284736@studenti.polito.it,14BHDMN (8),MEC1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

299977,CARTA,DALIA,s295758@studenti.polito.it,14BHDOA (8),MEC1T1,Da frequentare (può sost. esami), - , - ,N, - , - ,N,N,2021/22,ATTIVO

248795,IVALDI,MICHELE,s281248@studenti.polito.it,14BHDMQ (8),INF1T3,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

299307,KAUR,GAIA,s299135@studenti.polito.it,14BHDMN (8),BIO1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

299135,HASSAN,VINCENZO,s284323@studenti.polito.it,14BHDMN (8),AUT1T1,Frequentato nel 2020/2021, - , - , - , - , - ,N,N,2021/22,ATTIVO

295986,DUMITRIU,GIORGIA,s284005@studenti.polito.it,14BHDLZ (8),CIV1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

294183,LAURORA,TOMMASO,s299015@studenti.polito.it,14BHDLN (8),ELN1T3,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

298194,LANZAFAME,LEONARDO,s297155@studenti.polito.it,14BHDLX (8),AER1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

283794,CARRER,GABRIELE,s300069@studenti.polito.it,14BHDMX (8),CHI1T1,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

298021,JARNIJA,JOSE ANTONIO,s284268@studenti.polito.it,14BHDMB (8),INF1T3,Da frequentare (può sost. esami), - , - , - , - , - ,N,N,2021/22,ATTIVO

297537,INFANTE,MICHELE,s298838@studenti.polito.it,14BHDMK (8),PRO1A1,Frequentato nel 2020/2021, - , - , - , - , - ,N,N,2021/22,ATTIVO

299347,LACINAJ,BJORDI,s295495@studenti.polito.it,14BHDMH (8),FIS1T3,Frequentato nel 2020/2021, - , - ,N, - , - ,N,N,2021/22,ATTIVO

**NOTA: i nomi non corrispondono a studenti reali, in quanto matricole, nomi e cognomi sono stati rimescolati in ordine casuale. Il formato del file, invece, è corretto.**

# Da file di record a lista di dizionari



```
FILENAME = '14BHDLZ_2022_shuffled.csv'

file = open(FILENAME, 'r')

studenti = []
prima = True

for line in file:
    if not prima:
        record = line.rstrip().split(',')
        studenti.append({
            'matricola': int(record[0]),
            'cognome': record[1],
            'nome': record[2]
        })
    else:
        prima = False # La prima linea contiene i nomi dei campi
file.close()
```

 iscritti\_corso.py

# Da file di record a lista di dizionari

```
FILENAME = '14BHDLZ_2022_shuffled.csv'
```

```
file = open(FILENAME, 'r')
```

```
studenti = []
```

```
prima = True
```

```
for line in file:
```

```
    if not prima:
```

```
        record = line.rstrip().split(',')
```

```
        studenti.append({
```

```
            'matricola': int(record[0]),
```

```
            'cognome': record[1],
```

```
            'nome': record[2]
```

```
        })
```

```
    else:
```

```
        prima = False # la prima
```

```
file.close()
```



```
students = {list: 310} [{'matricola': 293374, 'cognome': 'LA GALA', 'nome': 'VITO'}, {'matricola': 296912, 'cognome': 'VULTAGGIO', 'nome': 'MATTIA'}, {'matricola': 286405, 'cognome': 'GURGIGNO', 'nome': 'FRANCESCA'}, {'matricola': 296327, 'cognome': 'LEONE', 'nome': 'DAVIDE'}, {'matricola': 298449, 'cognome': 'CAULA', 'nome': 'MATTEO'}, {'matricola': 297084, 'cognome': 'MAROVINO', 'nome': 'ANDREA REMI'}, {'matricola': 297153, 'cognome': 'LAPENNA', 'nome': 'NICOLA'}, {'matricola': 298369, 'cognome': 'HU', 'nome': 'NIKOLO'}]
```

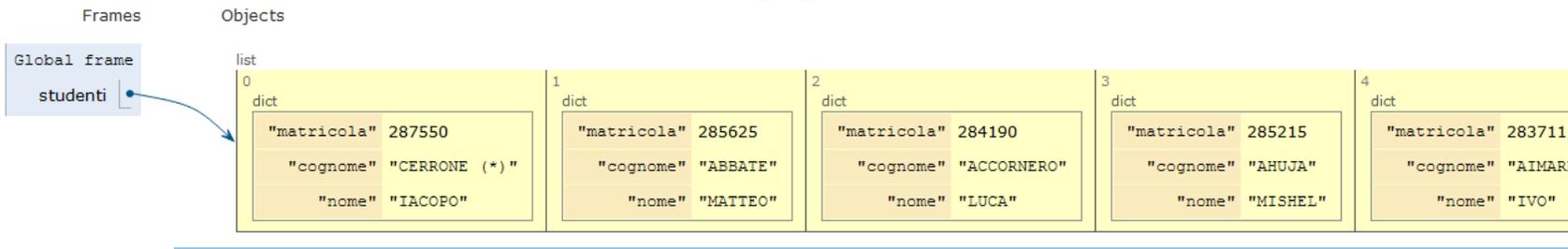
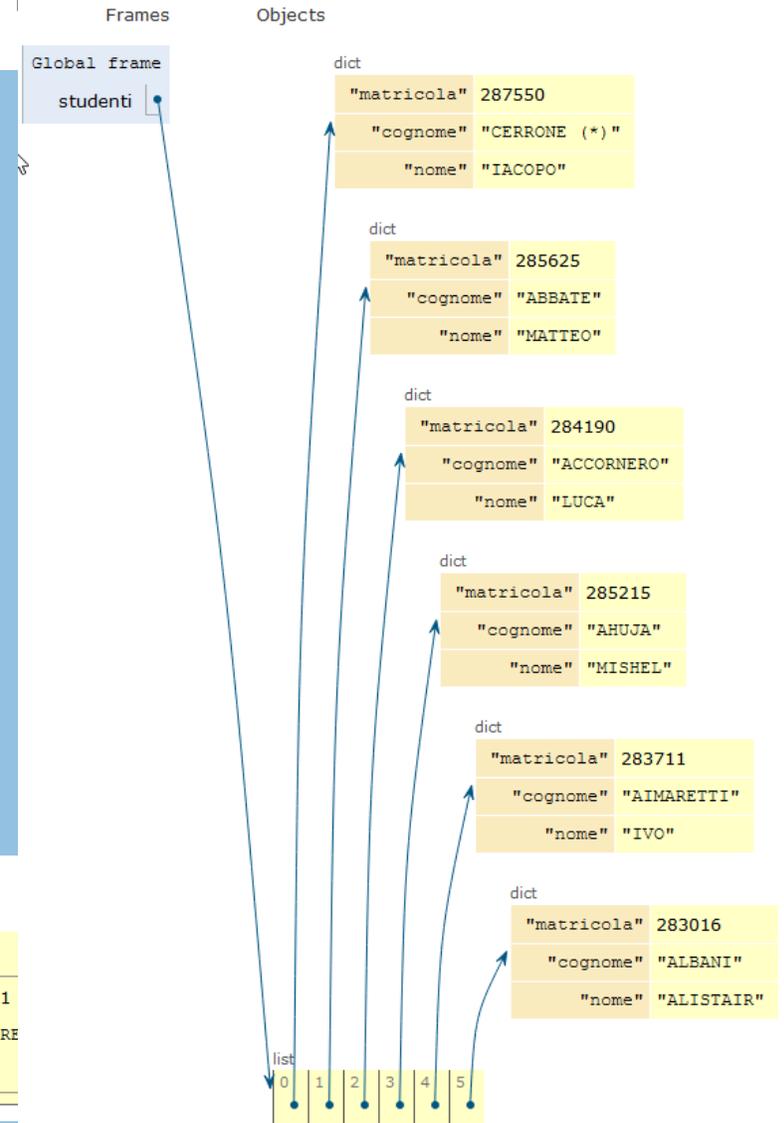
# Da file di record a lista di dizionari

```
FILENAME = '14BHDLZ_2022_shuffled.csv'

file = open(FILENAME, 'r')

studenti = []
prima = True

for line in file:
    if not prima:
        record = line.rstrip().split(',')
        studenti.append({
            'matricola': int(record[0]),
            'cognome': record[1],
            'nome': record[2]
```



# Leggere i record di un file CSV come dizionari

```
class csv.DictReader(f, fieldnames=None, restkey=None, restval=None, dialect='excel',  
*args, **kwargs)
```

Create an object that operates like a regular reader but maps the information in each row to a `dict` whose keys are given by the optional `fieldnames` parameter.

The `fieldnames` parameter is a `sequence`. If `fieldnames` is omitted, the values in the first row of file `f` will be used as the fieldnames. Regardless of how the fieldnames are determined, the dictionary preserves their original ordering.

```
>>> import csv  
>>> with open('names.csv', newline='') as csvfile:  
...     reader = csv.DictReader(csvfile)  
...     for row in reader:  
...         print(row['first_name'], row['last_name'])  
...  
Eric Idle  
John Cleese  
  
>>> print(row)  
{'first_name': 'John', 'last_name': 'Cleese'}
```

# DictReader del modulo `csv`

- Quando leggiamo un file in formato CSV, possiamo leggere le righe con l'ausilio di
  - `csv.reader`, che restituisce i campi di una riga sotto forma di lista (v. unità P7)
  - `csv.DictReader`, che restituisce i campi di una riga sotto forma di dizionario
  - Le chiavi del dizionario sono tratte dalla prima riga del file, oppure dal parametro opzionale `fieldnames`

# Da file CSV a lista di dizionari

```
import csv

file = open(FILENAME, 'r')
reader = csv.DictReader(file)

studenti = []
for record in reader:
    studenti.append(record)
file.close()
```



# Da file CSV a lista di dizionari

```
import csv

file = open(FILENAME, 'r')
reader = csv.DictReader(file)
```

```
studenti = []
for record in reader:
    studenti.append(record)
file.close()
```



```
students = {list: 310} [{"MATRICOLA": '293374', 'COGNOME': 'LA GALA', 'NOME': 'VITO', 'EMAIL': 's297372@studenti.polito.it', 'COD.INS. E CREDI
000 = {dict: 16} {"MATRICOLA": '293374', 'COGNOME': 'LA GALA', 'NOME': 'VITO', 'EMAIL': 's297372@studenti.polito.it', 'COD.INS. E CRED
01 'MATRICOLA' = {str} '293374'
01 'COGNOME' = {str} 'LA GALA'
01 'NOME' = {str} 'VITO'
01 'EMAIL' = {str} 's297372@studenti.polito.it'
01 'COD.INS. E CREDITI' = {str} '14BHDMK (8)'
01 'CDS STUDENTE' = {str} 'ELN1T3'
01 'FREQUENZA' = {str} 'Frequentato nel 2020/2021'
01 'ESAME' = {str} '- '
01 'ATHOME' = {str} '- '
01 'SDSS' = {str} '- '
01 'DUMMY' = {str} '- '
01 'INFO MOBILITA' = {str} '- '
01 "STUDENTE ALL'ESTERO" = {str} 'N'
01 'INS IN L.A.' = {str} 'N'
01 'ANNO ISCRIZ.' = {str} '2021/22'
01 'CONDIZIONE' = {str} 'ATTIVO'
01 '__len__' = {int} 16
> 001 = {dict: 16} {"MATRICOLA": '296912', 'COGNOME': 'VULTAGGIO', 'NOME': 'MATTIA', 'EMAIL': 's293359@studenti.polito.it', 'COD.INS. E
> 002 = {dict: 16} {"MATRICOLA": '286405', 'COGNOME': 'GURGIGNO', 'NOME': 'FRANCESCA', 'EMAIL': 's298343@studenti.polito.it', 'COD.IN
> 003 = {dict: 16} {"MATRICOLA": '296327', 'COGNOME': 'LEONE', 'NOME': 'DAVIDE', 'EMAIL': 's294681@studenti.polito.it', 'COD.INS. E CRED
> 004 = {dict: 16} {"MATRICOLA": '298449', 'COGNOME': 'CAULA', 'NOME': 'MATTEO', 'EMAIL': 's285164@studenti.polito.it', 'COD.INS. E CRED
```

Tutti i campi vengono  
letti in automatico

# Ordinare una lista di dizionari?

- La lista viene riempita con lo stesso ordine in cui gli elementi compaiono nel file
- Come ordinare l'elenco degli studenti per nome? Per matricola? Per cognome?
- Non funziona: `studenti.sort()`  
`TypeError: '<' not supported between instances of 'dict' and 'dict'`
  - Non è possibile confrontare tra loro due `dict`, quindi il metodo `sort` non sa come comportarsi

Approfondiamo il funzionamento degli ordinamenti...

# Ordinamenti avanzati

---

COME ORDINARE LISTE DI DIZIONARI E LISTE DI LISTE O TUPLE  
USANDO IL PARAMETRO *KEY=*

Sorting HOW TO

<https://docs.python.org/3/howto/sorting.html>

<https://realpython.com/python-sort/>

<https://realpython.com/sort-python-dictionary/>

# Premessa

- Una lista può essere ordinata:
  - con il metodo `lista.sort()` – modifica l'ordine degli elementi nella lista stessa
  - tramite la funzione `sorted(lista)` – restituisce una nuova lista, ordinata, senza modificare la lista di partenza
- Gli algoritmi di ordinamento, internamente, utilizzano più volte un confronto del tipo
  - `lista[i] < lista[j]`
- Il tipo di ordinamento dipende quindi dal comportamento dell'operatore di confronto '`<`'
  - Questo comportamento dipende, a sua volta, dal **tipo di dato contenuto** nella lista

# Premessa

- Una lista può essere ordinata:
  - con il metodo `lista.sort()` – modifica l'ordine degli elementi nella lista stessa
  - tramite la funzione `sorted(lista)` senza modificare la lista di partenza
- Gli algoritmi di ordinamento, intere confronti del tipo
  - `lista[i] < lista[j]`
- Il tipo di ordinamento dipende quindi dal comportamento dell'operatore di confronto '`<`'
  - Questo comportamento dipende, a sua volta, dal **tipo di dato contenuto** nella lista

Cosa succede se la mia lista contiene dei dati complessi?

Come si ordina una lista di liste (tabella)?  
Una lista di dizionari?



# Criteri di ordinamento

Tipo di dato contenuto nella lista	Ordinamento risultante
int, float	<b>Numerico</b> crescente
complex	<b>Impossibile</b> – i numeri complessi non sono ordinabili TypeError: '<' not supported between instances of 'complex' and 'complex'
str	<b>Lessicografico</b> crescente, basato sui codici Unicode (“circa” alfabetico)
list, tuple	Secondo il criterio di <b>confronto crescente tra liste</b> (salta gli elementi uguali e confronta il primo elemento diverso) – vedi unità P6
dict	<b>Impossibile</b> – I dizionari non sono confrontabili tra loro TypeError: '<' not supported between instances of 'dict' and 'dict'
set	<b>Imprevedibile</b> – non genera errori, ma il risultato è inutile, a meno che i diversi set siano sottoinsiemi l’uno dell’altro (infatti < è sinonimo di <code>issubset</code> )

Algoritmo di confronto tra liste:  $a < b$

- Partiamo da  $i = 0$
- Confrontiamo  $a[i]$  con  $b[i]$ 
  - Se  $a[i]=b[i]$ , allora incrementa  $i$  e ripeti
  - Se  $a[i]<b[i]$ , allora  $a<b$  sarà True
  - Se  $a[i]>b[i]$ , allora  $a<b$  sarà False
  - Se una lista finisce prima dell'altra, viene considerata minore
- Si confrontano uno ad uno gli elementi, alla ricerca della prima coppia di elementi diversi, o della fine di una delle due liste

# Criteri di ordinamento

Tipo di dato contenuto nella lista	Ordinamento risultante
int, float	<b>Numerico crescente</b>
complex	<b>Impossibile</b> – i numeri complessi non sono ordinabili TypeError: '<' not supported between instances of 'complex' and 'complex'
str	<b>Lessicografico crescente</b> , basato sui codici Unicode (“circa” alfabetico)
list, tuple	Secondo il criterio di <b>confronto crescente tra liste</b> (salta gli elementi uguali e confronta il primo elemento diverso) – vedi unità P6
dict	Dizionari non sono confrontabili tra loro TypeError: '<' not supported between instances of 'dict' and 'dict'
	Non genera errori, ma il risultato è inutile, a meno che i dizionari non siano uguali o l'uno sia sottinsieme dell'altro (infatti < è sinonimo di >)

Se volessimo un ordinamento **decescente**, è sufficiente aggiungere il parametro **reverse=True** come argomento di sort/sorted

Algoritmo di confronto tra liste:  $a < b$

- Partiamo da  $i = 0$
- Confrontiamo  $a[i]$  con  $b[i]$ 
  - Se  $a[i] = b[i]$ , allora incrementa  $i$  e ripeti
  - Se  $a[i] < b[i]$ , allora  $a < b$  sarà True
  - Se  $a[i] > b[i]$ , allora  $a < b$  sarà False
  - Se una lista finisce prima dell'altra, viene considerata minore
- Si confrontano uno ad uno gli elementi, alla ricerca della prima coppia di elementi diversi, o della fine di una delle due liste

# Chiavi di ordinamento (I)

- È possibile specificare alle funzioni `sort/sorted` quale criterio di ordinamento devono utilizzare, creando quindi degli ordinamenti “personalizzati”
  - Utili per ordinare liste di dizionari
  - Utili per modificare l’ordinamento ‘naturale’ per liste di liste
- In questi casi bisogna fornire una “**chiave di ordinamento**”, usando il parametro `key`
  - `studenti.sort(key=...)`

# Chiavi di ordinamento (II)

```
lista.sort(key=...)
```

- La **chiave**, *in generale*, è una **funzione**, calcolabile per ogni singolo record (dizionario), il cui valore dipende solo dai dati in esso memorizzati
  - **Non** ci occupiamo del caso generale, che richiede i costrutti 'lambda'
- La funzione `sort` confronterà gli elementi sulla base del valore della funzione chiave
  - Senza chiave: `lista[i] < lista[j]`
  - Con chiave: `key(lista[i]) < key(lista[j])`
- L'ordinamento avverrà quindi per valori crescente della chiave di ordinamento

## Caso particolare 1: Ordinare una lista di **dizionari** in base al valore di un campo

```
lista.sort(key=...)
```

```
key=itemgetter('campo')
```

- Consideriamo una lista di dizionari
- Nei casi più semplici, la chiave di ordinamento corrisponde ad **uno dei campi** del dizionario
- Esempi:
  - Ordinare per cognome
  - Ordinare per matricola
- In questo caso esiste la funzione `operator.itemgetter()` che implementa una chiave di ordinamento per selezionare il campo di interesse, specificando il nome del campo come argomento:
  - `key=operator.itemgetter('nome campo')`

# Esempio: Ordinare una lista di dizionari



```
from operator import itemgetter

# ordine di lettura dal file
print(studenti[0], studenti[1], studenti[2])

# ordina per matricola
studenti.sort(key=itemgetter('matricola'))
print(studenti[0], studenti[1], studenti[2])

# ordina per nome
studenti.sort(key=itemgetter('nome'))
print(studenti[0], studenti[1], studenti[2])

# ordina per cognome
studenti.sort(key=itemgetter('cognome'))
print(studenti[0], studenti[1], studenti[2])
```

## Caso particolare 2: Ordinare una lista di **liste** in base al valore di un campo

```
lista.sort(key=...)
```

```
key=itemgetter(indice)
```

- Consideriamo una lista di liste (tabella)
  - Identico ragionamento vale per una lista di tuple
- Nei casi più semplici, la chiave di ordinamento corrisponde ad **una delle colonne** della tabella
- Esempi:
  - Ordinare secondo il valore della prima colonna
  - Ordinare secondo il valore della quinta colonna
- La funzione `operator.itemgetter()` può essere utilizzata anche in questo caso, specificando il numero intero (indice) della colonna su cui ordinare:
  - `key=operator.itemgetter(num_colonna)`

# Esempio: Ordinare una lista di liste

- L'utilizzo di `itemgetter` è possibile anche se la lista contiene *liste* oppure *tuple*
- In questo caso l'argomento di `itemgetter` è **l'indice** della posizione rispetto a cui ordinare

```
[['A', '2'],  
 ['A', '4'],  
 ['A', '3'],  
 ['C', '4'],  
 ['D', '4'],  
 ['C', '1'],  
 ['B', '4'],  
 ['D', '1'],  
 ['D', '3'],  
 ['B', '1'],  
 ['C', '2'],  
 ['A', '1'],  
 ['C', '3'],  
 ['B', '2'],  
 ['B', '3'],  
 ['D', '2']]
```

```
caselle.sort(key=itemgetter(0))  
# ordina in base al primo  
elemento
```

```
caselle.sort(key=itemgetter(1))  
# ordina in base al secondo  
elemento
```

```
[['A', '1'],  
 ['A', '4'],  
 ['A', '2'],  
 ['A', '3'],  
 ['B', '2'],  
 ['B', '4'],  
 ['B', '3'],  
 ['B', '1'],  
 ['C', '2'],  
 ['C', '4'],  
 ['C', '3'],  
 ['C', '1'],  
 ['D', '1'],  
 ['D', '2'],  
 ['D', '4'],  
 ['D', '3']]
```

```
[['D', '1'],  
 ['C', '1'],  
 ['A', '1'],  
 ['B', '1'],  
 ['D', '2'],  
 ['B', '2'],  
 ['C', '2'],  
 ['A', '2'],  
 ['B', '3'],  
 ['D', '3'],  
 ['A', '3'],  
 ['C', '3'],  
 ['C', '4'],  
 ['A', '4'],  
 ['B', '4'],  
 ['D', '4']]
```

## Caso particolare 3: Utilizzare **più criteri** di ordinamento in cascata

```
lista.sort(key=...)
```

```
key=itemgetter(a,b,c)
```

- Talvolta, un singolo campo (o una singola colonna) non sono sufficienti a specificare il criterio di ordinamento completo
- Esempi:
  - Ordinare gli studenti per cognome, e a parità di cognome per nome
  - Ordinare le coordinate per ascissa, e a parità di ascissa per ordinata
- Si può sfruttare il fatto che `itemgetter` può ricevere più parametri, che corrispondono ai campi (per dict) o indici (per list) rispetto i quali ordinare
  - `lista.sort(key=operator.itemgetter('cognome', 'nome'))`
  - `coordinate.sort(key=operator.itemgetter(0, 1))`

# Ordinamento per criteri multipli

- Qualora il criterio di ordinamento coinvolga più campi, o combinazioni di chiavi più complesse, si può adottare un approccio alternativo:
  - **Ordinare più volte** la lista, con **chiavi** di ordinamento diverse, **partendo** dalle chiavi 'meno importanti' per **finire** con quella 'più importante'
- **Esempio:** ordinare per *cognome*, e a parità di *cognome*, per nome
  - `studenti.sort(key=itemgetter('nome'))` # chiave secondaria
  - `studenti.sort(key=itemgetter('cognome'))` # chiave primaria
  - Questo metodo funziona perché l'algoritmo di sort è “*stabile*” (ossia non scambia di posizione elementi che hanno lo stesso valore di chiave) – v.  
<https://docs.python.org/3/library/stdtypes.html#list.sort> e  
<https://docs.python.org/3/howto/sorting.html#sort-stability-and-complex-sorts>

## Caso particolare 4: Utilizzare **funzioni predefinite**

```
lista.sort(key=...)
```

```
key=len
```

- È possibile che si possano utilizzare delle **funzioni predefinite** di Python per estrarre il valore della chiave da una lista
  - sum, min, max, len, abs, round... (si veda l'unità P5)
- Esempi:
  - Ordinare una lista di insiemi, in base al **numero** crescente di **elementi** → la funzione chiave è **len()**
  - Ordinare una lista di liste (tabella), in base alla **somma** dei valori di ciascuna riga → la funzione chiave è **sum()**
- Soluzione:
  - `insiemi.sort(key=len)`
  - `tabella.sort(key=sum)`
- ⚠ Attenzione! ⚠  
Occorre indicare come chiave il **nome** della funzione, **senza le parentesi tonde**. Infatti la funzione non la dobbiamo invocare noi, la invocherà l'algoritmo di sort.

# Caso generale: costruirsi una funzione personalizzata

```
lista.sort(key=...)
```

```
key=my_function
```

- Nel caso generale, la funzione per il calcolo della chiave può essere personalizzata.
- Definire una funzione che:
  - Riceve un argomento (che sarà un elemento della lista)
  - Restituisce un valore (che sarà il valore chiave utilizzato)
- **Esempio:** ordinare delle coordinate sulla base della somma “ascissa + ordinata”

```
coordinate=[[2, 3], [3, 4], [-1, 0]]
```

```
def somma_xy(p): # calcola chiave ordinamento  
    return p[0]+p[1]  
    # p corrisponde ad un elemento della lista  
    # ad esempio [3, 4]
```

```
coordinate.sort(key=somma_xy)
```

# Scorciatoia: usare funzioni `lambda`

- Quando definisco una funzione di ordinamento, dovrò costruire una funzione nuova, assegnare un nome, e questa funzione avrà solitamente solo una istruzione di `return`
- Come scorciatoia, posso definire una funzione “anonima in-line” con il costrutto `lambda`

```
def somma_xy(p): # chiave
    return p[0]+p[1]
coordinate.sort(key=somma_xy)
```



```
coordinate.sort(key=lambda p: p[0]+p[1])
```

Parametro  
(elemento della lista)

Espressione che  
calcola la chiave  
(il `return` è sottinteso)

# Generalizzazione (1)

- L'attributo `key=` è presente anche per le funzioni `min` e `max`
  - `max( esami, key=itemgetter('voto') )`  
# trova l'esame di voto massimo in una lista di esami
  - `min( caselle, key=itemgetter(0) )`  
# trova la casella con il primo elemento minimo : ['A', '1']
- ⚡ Nota: viene restituito *l'intero elemento* (dizionario o lista) che corrisponde al max/min di un *singolo campo*
  - Non è la stessa cosa che trovare il valore numerico del voto massimo... avrei un numero (30) ma non saprei a quale esame corrisponde

# Generalizzazione (2)

- È utile combinare `itemgetter` con `enumerate`, ad esempio per trovare in un solo passaggio il *valore* del massimo e la sua *posizione*

```
> casuali
```

```
[5, 17, 1, 14, 4, 13, 8, 15, 18, 9, 10, 0, 19, 7, 3, 2, 11, 12, 16, 6]
```

```
> max(casuali)
```

```
19
```

```
> list(enumerate(casuali))
```

```
[(0, 5), (1, 17), (2, 1), (3, 14), (4, 4), (5, 13), (6, 8), (7, 15), (8, 18), (9, 9), (10, 10), (11, 0), (12, 19), (13, 7), (14, 3), (15, 2), (16, 11), (17, 12), (18, 16), (19, 6)]
```

```
> max(enumerate(casuali), key=itemgetter(1))
```

```
(12, 19) # posizione e valore del max
```

# Strutture dati complesse

---



8.3

# Strutture dati complesse

- I **contenitori** sono molto utili per memorizzare **gruppi di valori**
  - In Python, i contenitori **lista** e **dizionario** possono contenere qualsiasi tipo di dato, **compresi altri contenitori**
- Alcuni tipi di dati potrebbero richiedere organizzazioni più complesse dei dati
  - Vedremo alcuni esempi che richiedono strutture dati più complesse
  - Si otterranno combinando opportunamente i tipi di strutture noti
    - Liste
    - Insiemi
    - Dizionari
    - Tuple

# Esempio: indice analitico

- L'**indice analitico** di un libro specifica su quali pagine compaia un determinato termine
- Vogliamo costruire un indice analitico conoscendo i numeri di pagina in cui i termini sono presenti. Supponiamo di partire da un file di testo nel seguente formato:

```
6:type  
7:example  
7:index  
7:program  
8:type  
10:example  
11:program  
20:set
```

Notare che lo stesso termine può comparire più volte, in pagine diverse

# Esempio: indice analitico

- Il file riporta ogni occorrenza di ciascun termine da includere nell'indice analitico e la pagina in cui tale termine compare
- Se un termine compare più di una volta nella stessa pagina, l'indice analitico dovrà inserire tale pagina una sola volta
  - Se le pagine sono diverse, dovranno essere tutte elencate

# Esempio: indice analitico

- L'output del programma dovrà essere una lista di termini, in ordine alfabetico, seguito dai numeri di pagina in cui tale termine compare, separati da virgole:

example: 7, 10

index: 7

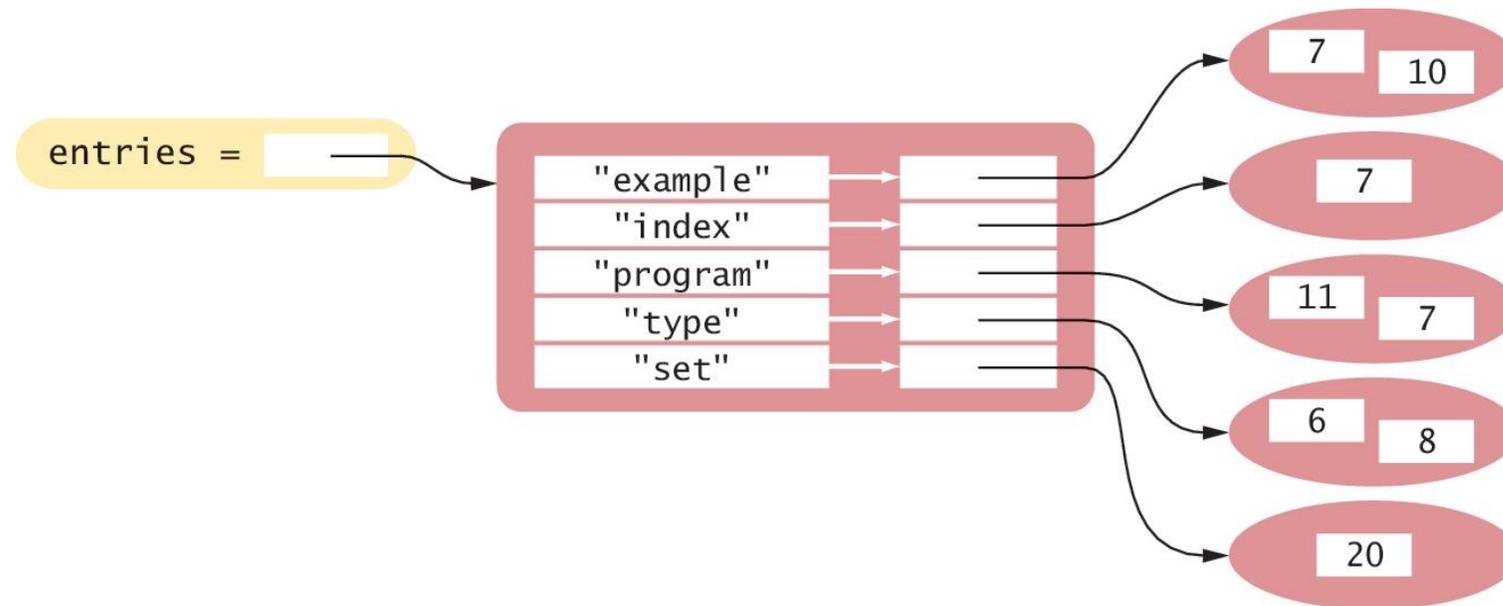
program: 7, 11

type: 6, 8

set: 20

# Esempio: indice analitico

- Per questo problema risulta utile costruire **un dizionario di insiemi**
- Ogni chiave del dizionario è un termine dell'indice analitico
- Il valore corrispondente sarà un insieme contenente i numeri di pagina



# Perché usiamo un dizionario?

- I **termini** nell'indice analitico devono essere **unici**
  - Rendendo ciascun termine **la chiave di un dizionario**, siamo certi che ci sarà una sola istanza di ciascuno di essi
- L'indice analitico dovrà comparire in ordine **alfabetico** per termine
  - Possiamo **iterare sulle chiavi in ordine**, quando produrremo il listato
- I **numeri** di pagina **duplicati** per un termine devono essere inclusi **una sola volta**
  - Mettendo i numeri di pagina in un **insieme**, è garantito che non si aggiungano duplicati
  - **Non serve un dizionario** per i numeri di pagina, perché non c'è alcuna informazione aggiuntiva associata a tali numeri

# Indice analitico: buildindex.py

```
5 def main() :
6     # Create an empty dictionary.
7     indexEntries = {}
8
9     # Extract the data from the text file.
10    infile = open("indexdata.txt", "r")
11    fields = extractRecord(infile)
12    while len(fields) > 0 :
13        addWord(indexEntries, fields[1], fields[0])
14        fields = extractRecord(infile)
15
16    infile.close()
17
18    # Print the index listing.
19    printIndex(indexEntries)
```

 buildindex.py

# Indice analitico: buildindex.py

```
26 def extractRecord(infile) :
27     line = infile.readline()
28     if line != "" :
29         fields = line.split(":")
30         page = int(fields[0])
31         term = fields[1].rstrip()
32         return [page, term]
33     else :
34         return []
```

# Indice analitico: buildindex.py

```
41 def addWord(entries, term, page) :
42     # If the term is already in the dictionary, add the page to the set.
43     if term in entries :
44         pageSet = entries[term]
45         pageSet.add(page)
46
47     # Otherwise, create a new set that contains the page and add an entry.
48     else :
49         pageSet = set([page])
50         entries[term] = pageSet
```

# Indice analitico: buildindex.py

```
56     for key in sorted(entries) :
57         print(key, end=" ")
58         pageSet = entries[key]
59         first = True
60         for page in sorted(pageSet) :
61             if first :
62                 print(page, end="")
63                 first = False
64             else :
65                 print(", ", page, end="")
66
67     print()
```

# Esempio: vendite gelati

- Un altro uso frequente dei dizionari in Python è la memorizzazione di una **raccolta di liste** in cui ciascuna lista sia associata ad un **nome unico** (o chiave unica)
- Per esempio, si consideri il problema di estrarre dati da un file di testo che rappresenta le vendite annue di differenti gusti di gelato, in diversi negozi di una catena di gelaterie. Ciascuna riga rappresenta un gusto, ciascuna colonna rappresenta un negozio.
- vanilla:8580.0:7201.25:8900.0  
chocolate:10225.25:9025.0:9505.0  
rocky road:6700.1:5012.45:6011.0  
strawberry:9285.15:8276.1:8705.0  
cookie dough:7901.25:4267.0:7056.5

# Esempio: vendite gelati

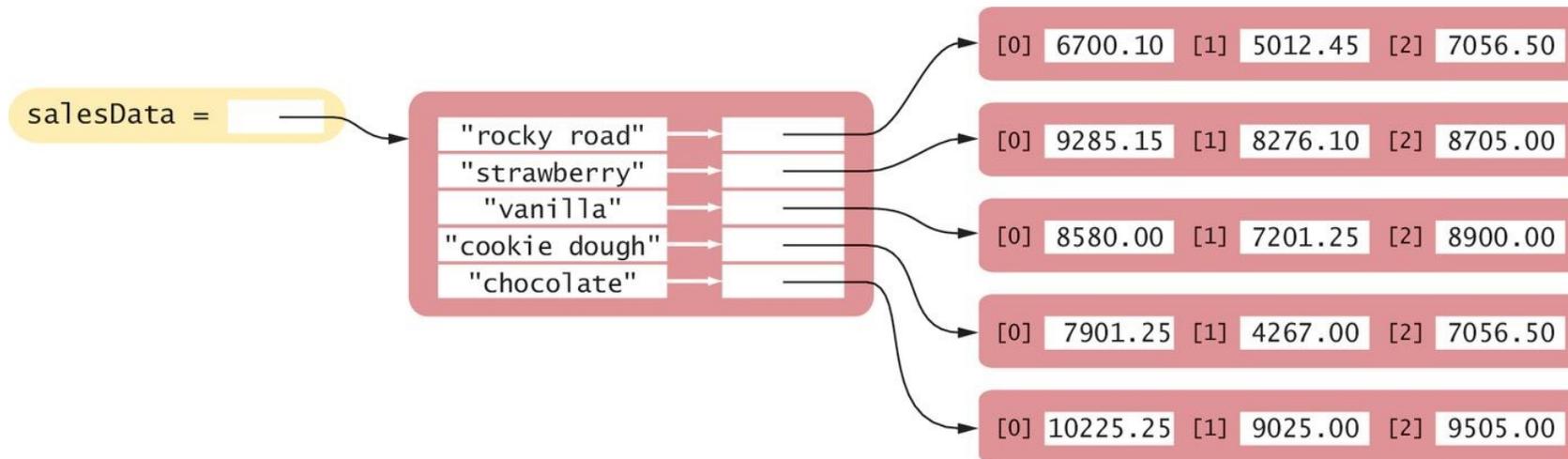
- Vogliamo elaborare i dati in modo da produrre un report simile a quello seguente:

chocolate	10225.25	9025.00	9505.00	28755.25
cookie dough	7901.25	4267.00	7056.50	19224.75
rocky road	6700.10	5012.45	6011.00	17723.55
strawberry	9285.15	8276.10	8705.00	26266.25
vanilla	8580.00	7201.25	8900.00	24681.25
	42691.75	33781.80	40177.50	

- Usare semplicemente una lista non è la soluzione migliore:
  - Gli elementi consistono sia di stringhe che di numeri in virgola mobile, ed occorre ordinarli per nome del gusto

# Esempio: vendite gelati

- Possiamo creare una struttura in cui ciascuna riga della tabella corrisponda ad un elemento del dizionario
- Il nome del gusto di gelato è la chiave utilizzata per identificare una riga specifica
- Il valore associato a ciascuna chiave è una lista che contiene le vendite, negozio per negozio, di quel gusto



# Vendite gelati: icecreamsales.py

```
6 def main() :  
7     salesData = readData("icecream.txt")  
8     printReport(salesData)
```

# Vendite gelati: icecreamsales.py

```
14 def readData(filename) :
15     # Create an empty dictionary.
16     salesData = {}
17
18     infile = open(filename, "r")
19
20     # Read each record from the file.
21     for line in infile :
22         fields = line.split(":")
23         flavor = fields[0]
24         salesData[flavor] = buildList(fields)
25
26     infile.close()
27     return salesData
```

 icecreamsales.py

# Vendite gelati: icecreamsales.py

```
33 def buildList(fields) :  
34     storeSales = []  
35     for i in range(1, len(fields)) :  
36         sales = float(fields[i])  
37         storeSales.append(sales)  
38  
39     return storeSales
```

# Vendite gelati: icecreamsales.py

```
44 def printReport(salesData) :
45     # Find the number of stores as the length of the longest store sales list.
46     numStores = 0
47     for storeSales in salesData.values() :
48         if len(storeSales) > numStores :
49             numStores = len(storeSales)
50
51     # Create a list of store totals.
52     storeTotals = [0.0] * numStores
53
54     # Print the flavor sales.
55     for flavor in sorted(salesData) :
56         print("%-15s" % flavor, end="")
57
```

# Vendite gelati: icecreamsales.py

```
58     flavorTotal = 0.0
59     storeSales = salesData[flavor]
60     for i in range(len(storeSales)) :
61         sales = storeSales[i]
62         flavorTotal = flavorTotal + sales
63         storeTotals[i] = storeTotals[i] + sales
64         print("%10.2f" % sales, end="")
65
66     print("%15.2f" % flavorTotal)
67
68     # Print the store totals.
69     print("%15s" % " ", end="")
70     for i in range(numStores) :
71         print("%10.2f" % storeTotals[i], end="")
72     print()
```

# Moduli



8.3

---

DIVIDERE IL PROGRAMMA IN PIÙ PARTI

# Moduli

- Quando creiamo programmi piccoli, possiamo scrivere tutto il codice in un unico file sorgente
- Quando i programmi iniziano a diventare più grandi, o quando si lavora in gruppo, la situazione cambia
- Vorremmo riuscire a strutturare il codice, dividendolo in file sorgenti separati («moduli»)

# Motivazioni per l'uso di moduli

- Programmi grandi possono contenere **centinaia di funzioni** che diventano **difficili da gestire** e debuggare se fossero tutte in un unico file sorgente
  - Distribuire le funzioni **su più file sorgenti** e raggruppare **insieme le funzioni correlate** permette di facilitare il test ed il debug delle varie funzioni
- Un secondo motivo diventa evidente **quando si lavora con altri programmatori** in un gruppo di lavoro
  - Sarebbe molto difficile per più programmatori modificare contemporaneamente un singolo file sorgente
  - Il sorgente viene diviso in modo che ciascun programmatore sia responsabile di un insieme ben definito di file

# Tipica divisione in moduli

- I programmi Python di grandi dimensioni tipicamente consistono di un **modulo principale** (**driver module**) ed uno o più moduli supplementari
- Il **modulo principale** contiene la funzione **main()** oppure la prima istruzione eseguibile (nel caso non si usi una funzione main)
- I moduli supplementari contengono **funzioni di supporto**, costanti, variabili, ...

# Esempio di divisione in moduli

- Obiettivo: Dividere in moduli l'esercizio sui gusti di gelato (dizionario di liste)
- Il modulo `tabulardata.py` contiene funzioni per **leggere** i dati dal **file** e **stampare** un dizionario di liste aggiungendo i totali di riga e di colonna
- Il modulo `salesreport.py` è il modulo **driver** (o **principale**) che contiene la funzione **main**
- Dividendo il programma in due moduli, le funzioni del modulo `tabulardata.py` potranno essere riusate in altri programmi che debbano elaborare elenchi di numeri associati ad un nome

# Usare il codice di altri moduli

- Per chiamare una funzione o usare una costante definita in un modulo, occorre innanzitutto importare il modulo, nello stesso modo in cui si importerebbe un modulo della libreria standard

```
from tabulardata import readData, printReport
```

- Se un modulo definisce molte funzioni, può essere più comoda la forma (rende accessibili tutti i nomi definiti nel modulo):

```
import tabulardata
```

- In questo caso, bisogna anteporre il nome del modulo al nome delle funzioni:

```
tabulardata.printReport(salesData)
```

# Sommario

---

# Insiemi

- Un **insieme** memorizza una raccolta di valori unici
- Un insieme viene creato usando un'espressione costante di tipo insieme `{...}` o attraverso la funzione `set()`
- L'operatore `in` verifica se un elemento sia membro di un insieme
- Si possono aggiungere nuovi elementi con il metodo `add()`
- Si possono eliminare elementi dall'insieme con il metodo `discard()`
- Il metodo `issubset()` verifica se un insieme sia sotto-insieme di un altro

# Insiemi

- Il metodo `union()` produce un nuovo insieme che contiene gli elementi presenti in almeno uno degli insiemi di partenza
- Il metodo `intersection()` produce un nuovo insieme che contiene gli elementi contenuti in entrambi gli insiemi di partenza
- Il metodo `difference()` produce un nuovo insieme che contiene gli elementi contenuti in nel primo insieme ma non nel secondo
- L'implementazione degli insiemi organizza gli elementi in modo che la loro ricerca sia molto rapida

# Dizionari

- Un **dizionario** ricorda delle **associazioni** tra **chiavi** e **valori**
- L'operatore **[ ]** accede al valore associato ad una chiave
  - Usando il metodo **get()** si può definire un valore di default da ritornare se la chiave non viene trovata
- L'operatore **in** può verificare se una chiave appartenga ad un dizionario
- Si possono aggiungere nuovi elementi, o aggiornare elementi esistenti, usando l'operatore **[ ]**
- Il metodo **pop()** rimuove un elemento dal dizionario

# Strutture dati complesse

- Le strutture dati più complesse aiutano ad organizzare meglio le informazioni, per poterle elaborare più facilmente
- Il codice sorgente, nei programmi complessi, è distribuito su più file sorgenti diversi (moduli)