

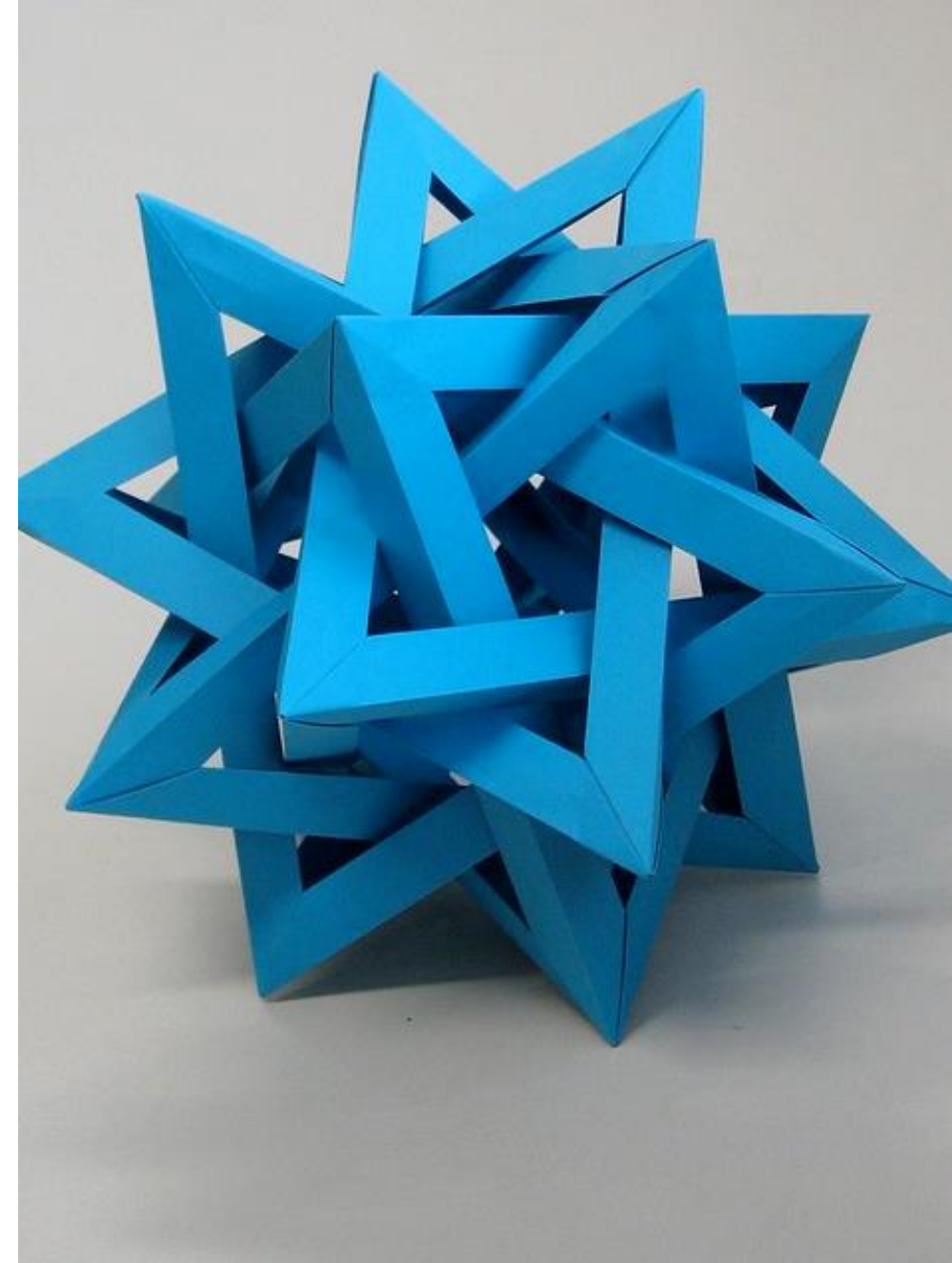


Unità P8: Strutture dati complesse

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



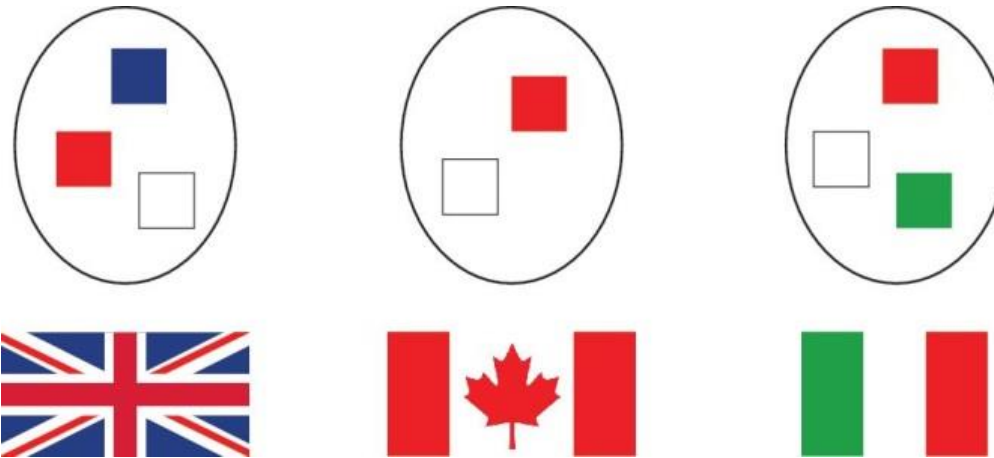
8.1

Insiemi

- Un insieme (set) è un tipo contenitore che memorizza una **raccolta di valori univoci**
- 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 insieme

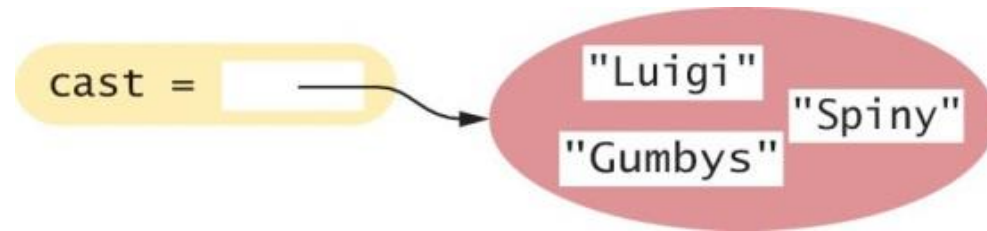
- Questi insiemi contengono tre colori: i colori che compongono le bandiere britannica, canadese ed italiana
- In ciascun insieme, l'ordine non è importante ed i colori non sono mai duplicati



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)
```

Valori ammessi negli insiemi

- Per ragioni tecniche legate all'implementazione del linguaggio, i valori di 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

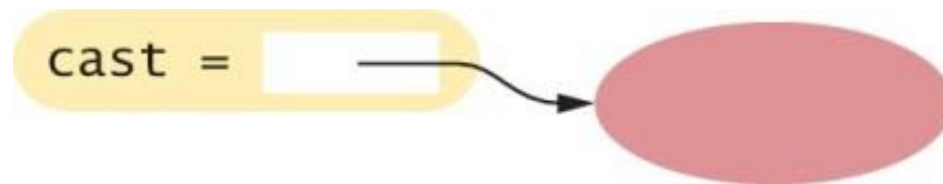
- Per approfondimenti vedere «hashable»:
<https://docs.python.org/3/glossary.html#term-hashable>

Creare un insieme vuoto

- Per ragioni storiche, 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:



```
numberOfMovieCharacters = 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

- Poiché gli insiemi sono **non ordinati**, **non** si può accedere agli elementi per **posizione** (come si farebbe con una lista)
- Si può usare un ciclo **for** ed iterare su ciascun elemento:

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

- Si noti che l'**ordine** in cui si gli elementi vengono visitati nel ciclo dipende da come essi sono memorizzati internamente – è imprevedibile

Accedere agli elementi di un insieme (2)

- 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)
```

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

Visualizzare un insieme in modo ordinato

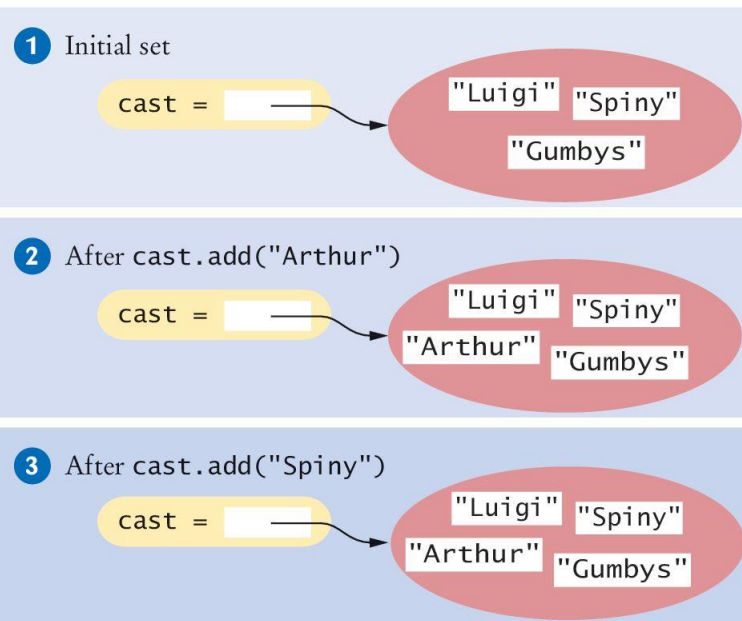
- Si usa la funzione `sorted()`, che restituisce una **lista (non un set)** degli elementi, elencati secondo un ordinamento definito (numeri crescenti/decrescenti, stringhe in ordine alfabetico, ...)
- Il ciclo seguente stampa il cast in ordine alfabetico crescente:

```
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
```



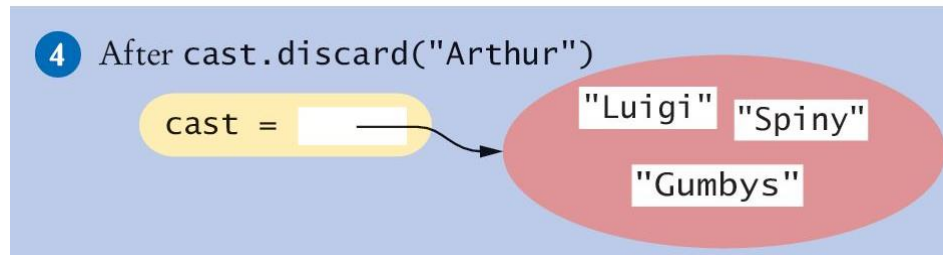
Arthur non è nel set, quindi viene aggiunto e la dimensione del set si incrementa di 1

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 se esiste
 - Solleva un'eccezione se l'elemento non fa parte del set:

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

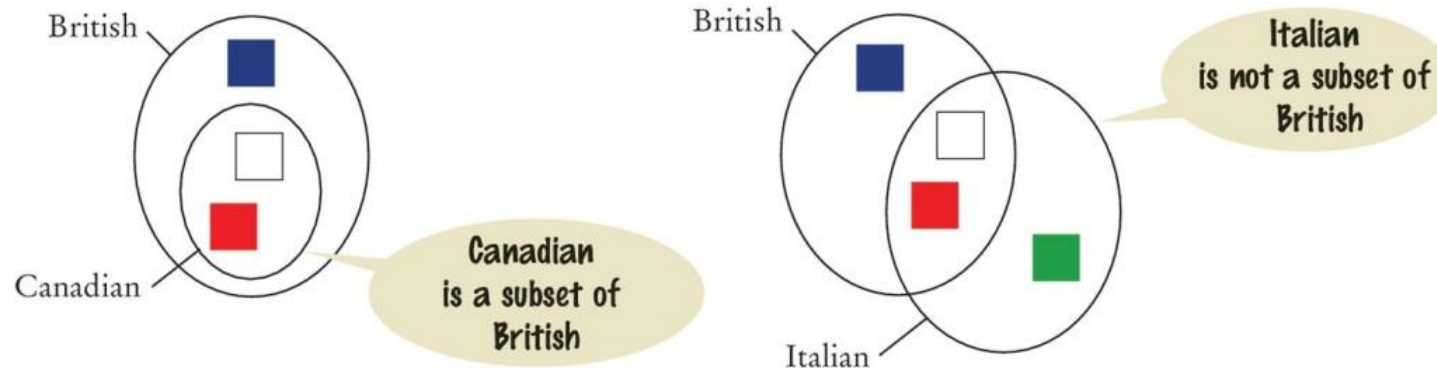

Cancellare elementi: `clear`

- 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 “==” and “!=”
- Due insiemi sono uguali se e solo se hanno esattamente gli stessi elementi

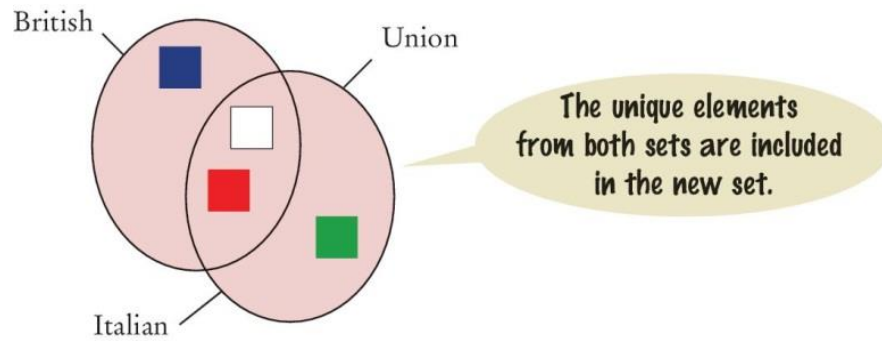
```
french = { "Red", "White", "Blue" }  
if british == french :  
    print("The British and French flags use the same colors.")
```

Unione di insiemi: `union`

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

```
# inEither: The set {"Blue", "Green", "White", "Red"}  
inEither = 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

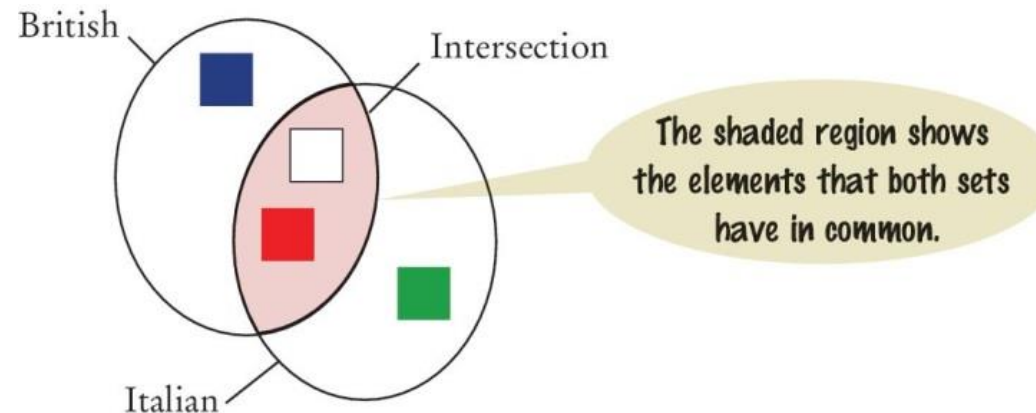


Si noti che il metodo `union()` restituisce un nuovo insieme. Infatti non modifica nessuno dei due insiemi coinvolti

Intersezione di insiemi : **intersection**

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

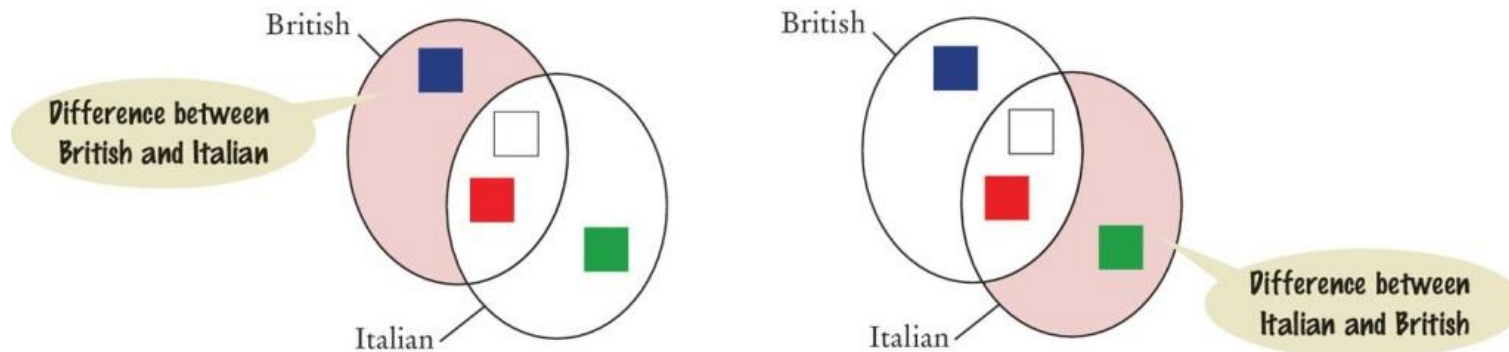
```
# inBoth: The set {"White", "Red"}  
inBoth = british.intersection(italian)
```



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 = {e₁, e₂, ..., e_n}</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>

Esempi

- Aprire il file: `set examples.py`

Esempio: Controllo ortografico

- Il programma `spellcheck.py` legge un file che contiene le parole correttamente scritte e le memorizza in un insieme
- Legge poi tutte le parole da un documento (in questo caso il libro Alice nel Paese delle Meraviglie) salvandole in un secondo insieme
- Infine stampa tutte le parole del documento che non si trovano tra le parole corrette

Esempio: spellcheck.py

```
1  ##
2  # This program checks which words in a file are not present in a list of
3  # correctly spelled words.
4  #
5
6  # Import the split function from the regular expression module.
7  from re import split
8
9  def main() :
10     # Read the word list and the document.
11     correctlySpelledWords = readWords("words")
12     documentWords = readWords("alice30.txt")
13
14     # Print all words that are in the document but not the word list.
15     misspellings = documentWords.difference(correctlySpelledWords)
16     for word in sorted(misspellings) :
17         print(word)
```

Esempio : Spellcheck.py

```
24 def readWords(filename) :
25     wordSet = set()
26     inputFile = open(filename, "r")
27
28     for line in inputFile :
29         line = line.strip()
30         # Use any character other than a-z or A-Z as word delimiters.
31         parts = split("[^a-zA-Z]+", line)
32         for word in parts :
33             if len(word) > 0 :
34                 wordSet.add(word.lower())
35
36     inputFile.close()
37     return wordSet
38
39 # Start the program.
40 main()
```

Esecuzione: Spellcheck.py

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

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:

```
itemSet.add(item)
```

- usano:

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

- Il programma in questo modo è molto più lento
 - Fino a 10 volte...!

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

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

```
for line in inputFile :
```

```
    theWords = line.split()
```

```
    for words in theWords :
```

```
        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`

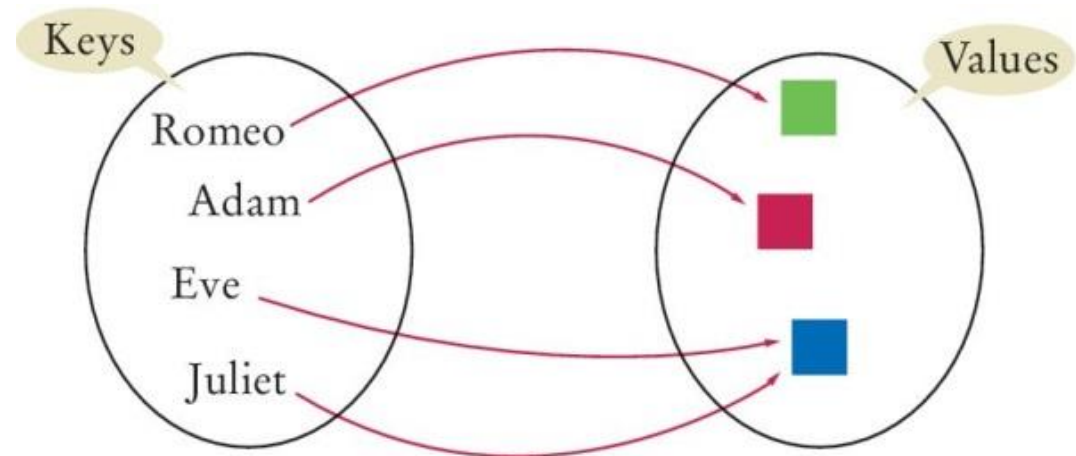
Dizionari



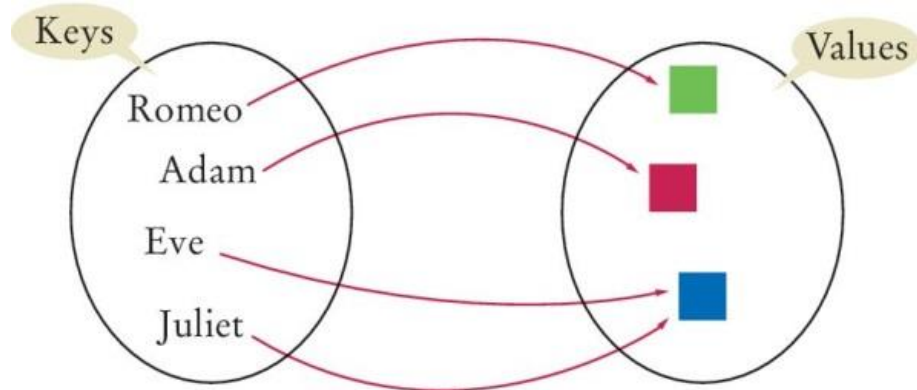
8.2

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 di 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" }
```

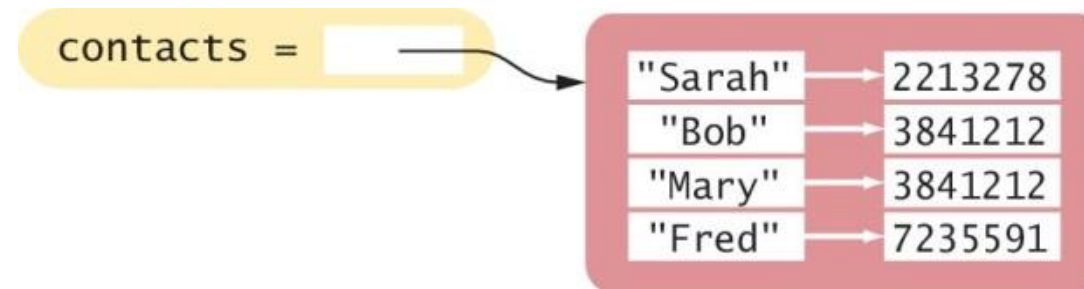
Una coppia di parentesi graffe vuota è un dizionario.

```
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, oppure il valore corrente di 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"
```

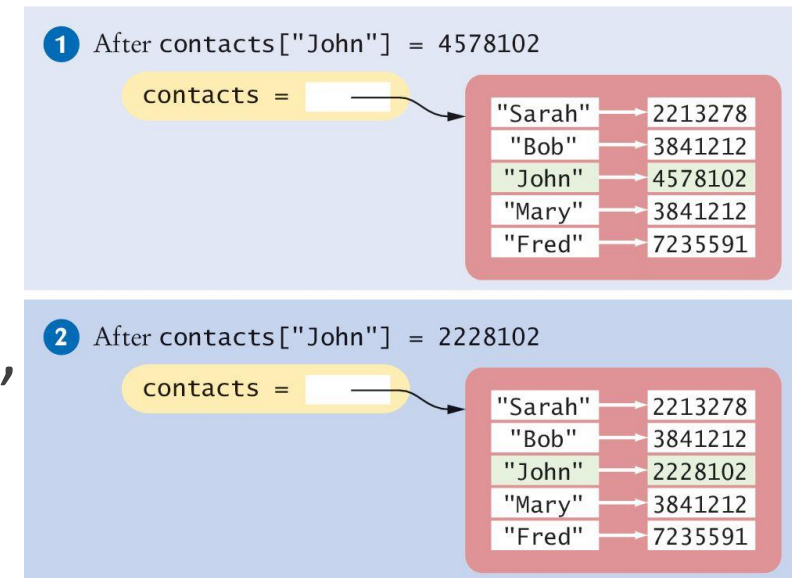
Aggiungere o modificare elementi

- 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 = {}
```

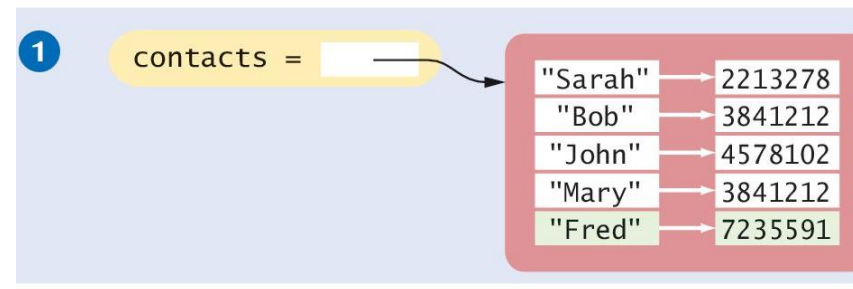
- e poi aggiungere gli elementi, quando li conosceremo

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

Cancellare elementi

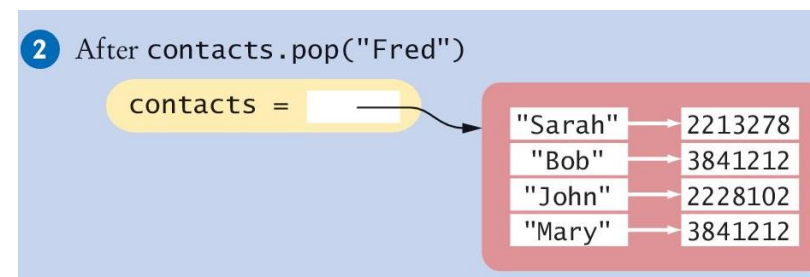
- Per eliminare un elemento da un dizionario, chiamare il metodo `pop()` con la chiave da eliminare come argomento:

```
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:

```
fredsNumber = 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:

Sarah

Bob

John

Mary

Fred

Notare che il dizionario memorizza gli elementi in un **ordine** che ne ottimizza l'efficienza; non è detto che coincida con l'ordine di **inserimento**

Attraversare un dizionario in ordine

- Per iterare sulle chiavi **in ordine**, 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("%-10s %d" % (key, contacts[key]))
```

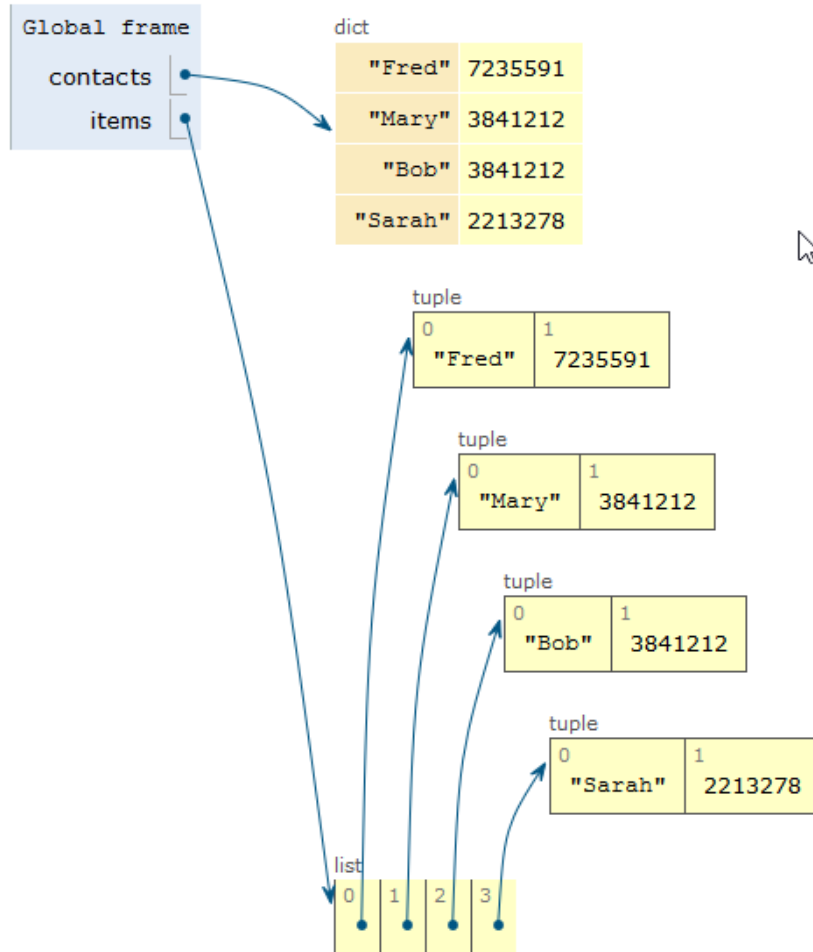

Iterare in modo più efficiente su un dizionario

- 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)
```

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 (key, value). 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 «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

```
def extractRecord(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 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

```
infile = open("populations.txt", "r")
record = extractRecord(infile)
while len(record) > 0 :
    print("%-20s %10d" % (record["country"],
                        record["population"]))
    record = extractRecord(infile)
```

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

Da file di record a lista di dizionari



```
FILENAME = '14BHDLZ_2021.csv'

file = open(FILENAME, 'r')

studenti = []
prima = True
for line in file:
    if not prima:
        record = line.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()
```

Da file di record a lista di dizionari

```
FILENAME = '14BHDLZ_2021.csv'
```

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

```
studenti = []
```

```
prima = True
```

```
for line in file:
```

```
    if not prima:
```

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

```
        studenti.append({
```

```
            'matricola':
```

```
            record[0],
```

```
            'cognome':
```

```
            record[1],
```


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

```
        })
```

```
    else:
```

```
        prima = False
```

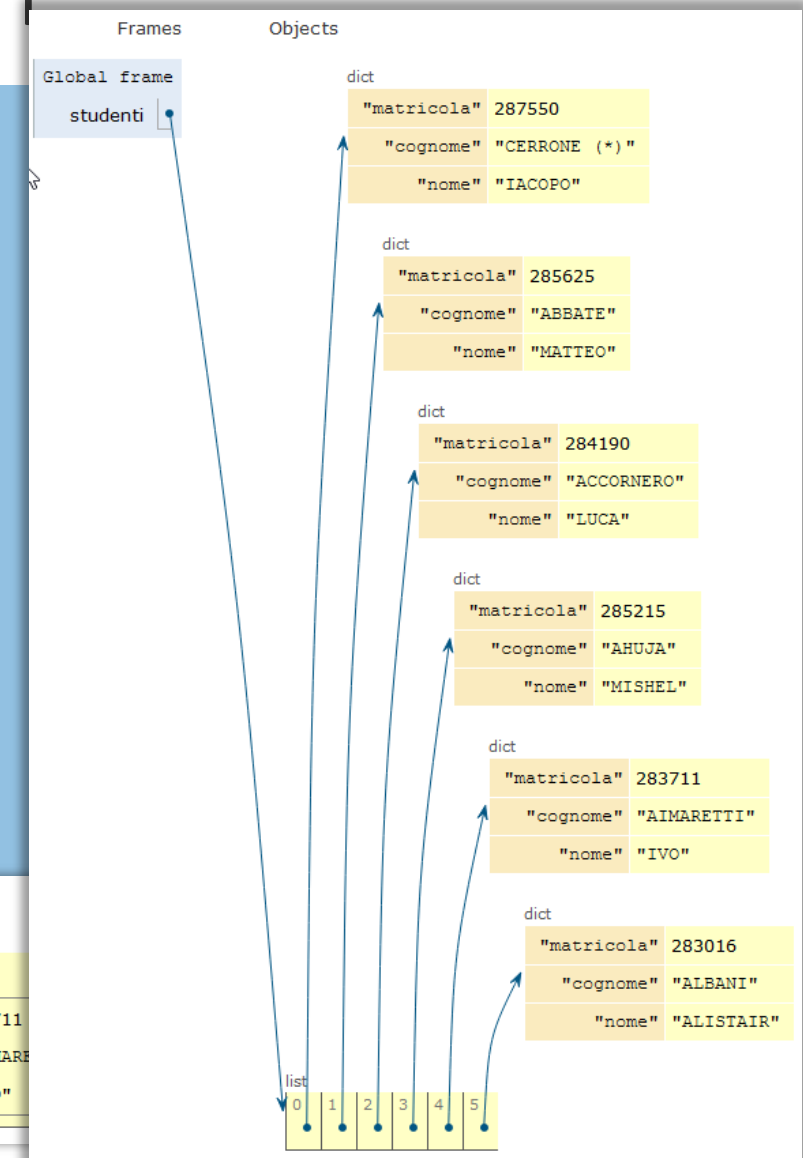
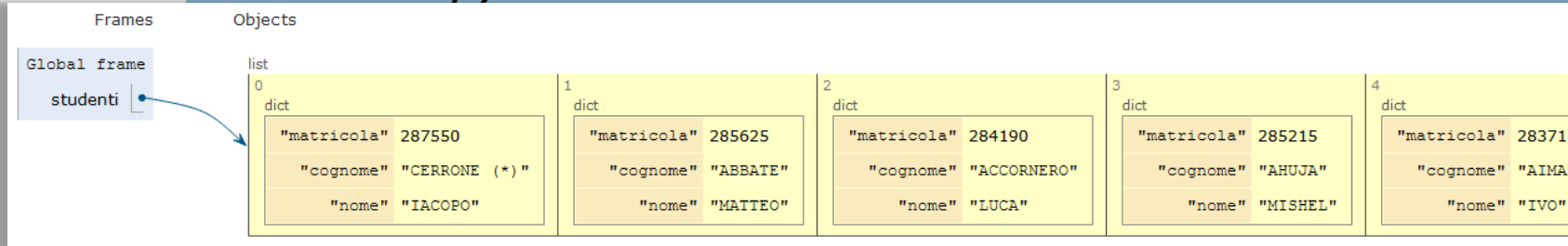
```
file.close()
```



```
> studenti = {list: 324} [{'matricola': 287550, 'cognome': 'CERRONE (*)', 'nome': 'IACOPO'}, {'matr... View
> 000 = {dict: 3} {'matricola': 287550, 'cognome': 'CERRONE (*)', 'nome': 'IACOPO'}
> 001 = {dict: 3} {'matricola': 285625, 'cognome': 'ABBATE', 'nome': 'MATTEO'}
  01 'matricola' = {int} 285625
  01 'cognome' = {str} 'ABBATE'
  01 'nome' = {str} 'MATTEO'
  01 __len__ = {int} 3
> 002 = {dict: 3} {'matricola': 284190, 'cognome': 'ACCORNERO', 'nome': 'LUCA'}
  01 'matricola' = {int} 284190
  01 'cognome' = {str} 'ACCORNERO'
  01 'nome' = {str} 'LUCA'
  01 __len__ = {int} 3
> 003 = {dict: 3} {'matricola': 285215, 'cognome': 'AHUJA', 'nome': 'MISHEL'}
  01 'matricola' = {int} 285215
  01 'cognome' = {str} 'AHUJA'
  01 'nome' = {str} 'MISHEL'
  01 __len__ = {int} 3
> 004 = {dict: 3} {'matricola': 283711, 'cognome': 'AIMARETTI', 'nome': 'IVO'}
> 005 = {dict: 3} {'matricola': 283016, 'cognome': 'ALBANI', 'nome': 'ALISTAIR'}
```

Da file di record a lista di dizionari

```
FILENAME = '14BHDLZ_2021.csv'  
  
file = open(FILENAME, 'r')  
  
studenti = []  
prima = True  
for line in file:  
    if not prima:  
        record = line.split(',')  
        studenti.append({  
            'matricola': int(record[0]),  
            'cognome': record[1],  
            'nome': record[2]  
        })
```



Ordinare una lista di dizionari (I)

- 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
- Bisogna fornire una “chiave di ordinamento”
 - `studenti.sort(key=...)`

Ordinare una lista di dizionari (II)

- L'ordinamento di una lista di dizionari richiede di specificare la “chiave di ordinamento”
`studenti.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’
- Nei casi più semplici, la chiave corrisponde ad **uno dei campi** del dizionario
 - In questo caso esiste la funzione `operator.itemgetter()` che implementa una chiave di ordinamento per selezionare il campo di interesse

Ordinare una lista di dizionari (III)



```
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])
```

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 complessi dei dati
 - Vedremo alcuni esempi che richiedono strutture dati più complesse
 - Si otterranno combinando opportunamente i tipi di strutture noti
 - Liste
 - Insiemi
 - Dizionari

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

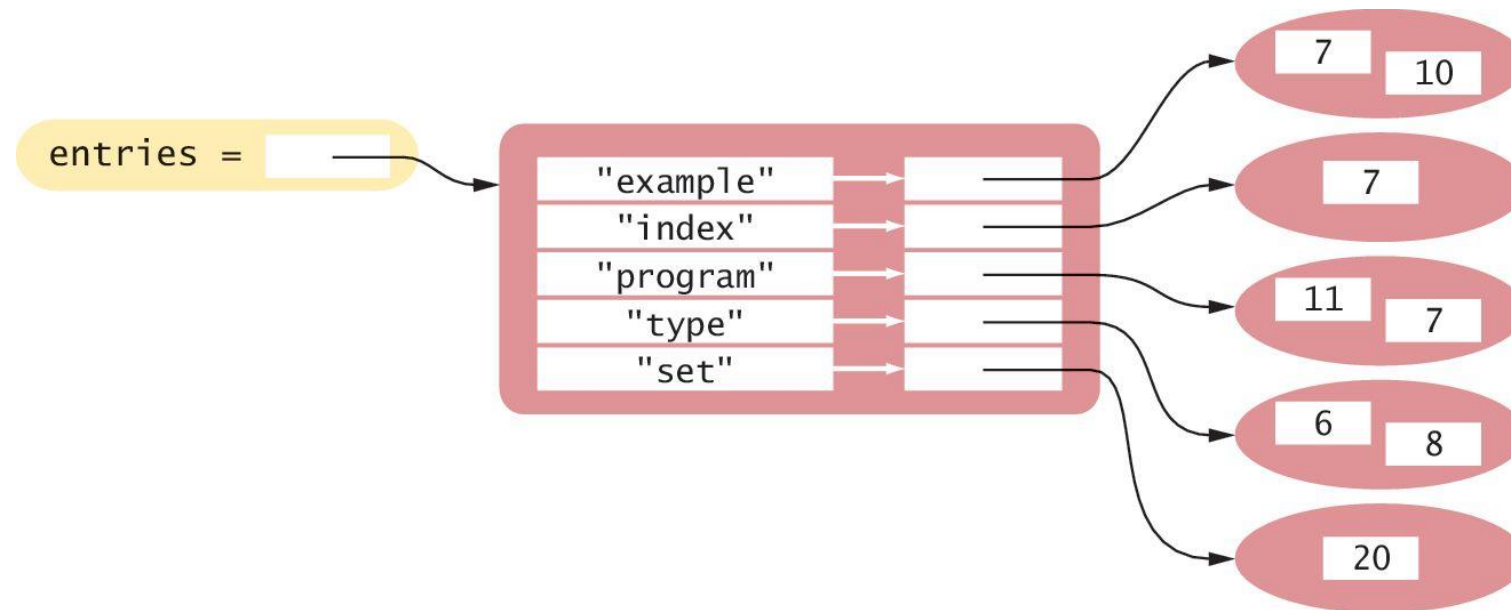
program: 7, 11

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)
```

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

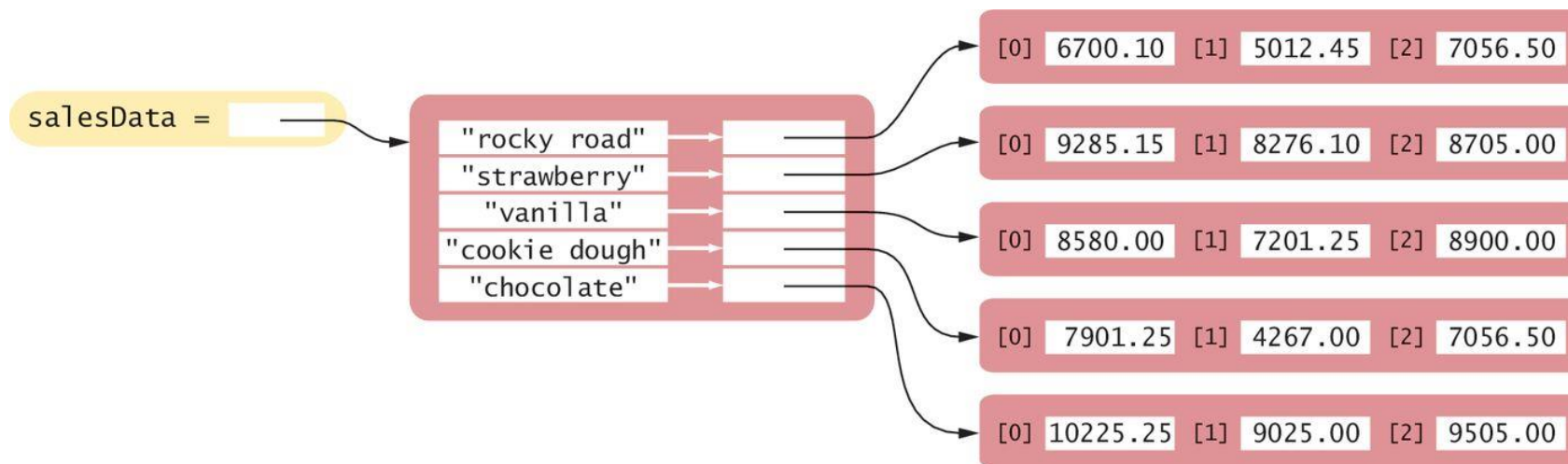
- Vogliamo elaborare i dati in modo da produrre un report simile a quello seguente:

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
```

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