CAPITOLO 4

Le strutture di dati

GLI ARGOMENTI DI QUESTO CAPITOLO

- L'organizzazione dei dati in strutture
- Le liste
- Le matrici, le pile e le code con le liste
- Le liste e le stringhe
- Le liste con funzioni
- Le tuple
- Gli insiemi
- I dizionari
- I dizionari con contenitori e funzioni

1 L'organizzazione dei dati in strutture

In generale i **dati** sono le informazioni rappresentate in una forma trattabile con il computer.
In molti problemi si ha la necessità di aggregare molti dati di tipo semplice, per facilitarne la rappresentazione e rendere più veloce il loro ritrovamento. Una **struttura di dati** è un insieme di dati raggruppati e organizzati secondo uno schema ben definito.

Nel Capitolo 2 sono state introdotte le liste (vedi pag. 39). Come si è visto, si tratta di strutture nelle quali una sola variabile identifica un certo numero di dati, che sono singolarmente accessibili mediante un indice che ne specifica la posizione nella struttura. La possibilità di accedere e manipolare molti dati in modo compatto è di grande importanza nella progettazione e nello sviluppo di qualsiasi software. La scelta di rappresentare i dati di un problema da risolvere con una struttura piuttosto che con un'altra può influenzare notevolmente la complessità della soluzione.
In questo capitolo saranno presentate le strutture di dati del linguaggio Python, con le quali si possono implementare strutture composte molto potenti. Le strutture considerate sono le **liste**, le **tuple**, gli **insiemi**, i **dizionari**; inoltre saranno riviste anche le **stringhe** di caratteri.

Tali strutture sono oggetti istanziati da una specifica classe, per la quale sono definiti metodi e operazioni ammissibili. Le stringhe sono oggetti della classe **str**, le liste sono oggetti della classe **list**, le tuple sono oggetti della classe **tuple**, gli insiemi sono oggetti della classe **set** e infine i dizionari sono oggetti della classe **dict**.

- Una **stringa** consiste in un elenco di elementi dello stesso tipo (i caratteri che la compongono) delimitato da apici o doppi apici. Una stringa non è modificabile e si può accedere a ogni elemento tramite un indice che ne precisa la posizione nella stringa.
- **Liste** e **tuple** sono elenchi ordinati di elementi accessibili, come nel caso delle stringhe, mediante indici che ne identificano la posizione. Gli elementi che compongono liste e tuple possono essere dello stesso tipo o di tipo diverso ma, in ogni caso, il posizionamento di un elemento nell'elenco è significativo: la tupla (*1, 2, 3*) differisce dalla tupla (*1, 3, 2*). Liste e tuple ammettono la presenza di elementi duplicati. In pratica le liste differiscono dalle tuple solo per il fatto che le prime sono modificabili, mentre le seconde non lo sono: in una lista si possono aggiungere, togliere o modificare elementi, in una tupla no.
- Gli **insiemi** sono oggetti modificabili, composti da elementi di qualsiasi tipo purché differenti tra di loro: in un insieme non ci possono essere elementi duplicati. Un insieme è privo di ordinamento: gli insiemi {*1, 2, 3*} e {*2, 3, 1*} sono uguali. Non è possibile accedere ai singoli elementi di un insieme se non elencandoli o estraendoli dall'insieme.
- I **dizionari** sono collezioni non ordinate di coppie del tipo *chiave*: *valore*. Si accede a un elemento del dizionario tramite la chiave per conoscere il valore a essa associato: nel caso del dizionario dell'esempio seguente si può sapere che la stringa *'due'* è associata all'intero 2. Un dizionario è modificabile, non ci possono essere elementi duplicati ed è privo di ordinamento. Il dizionario {*'uno' : 1, 'due' : 2*} è quindi uguale al dizionario {*'due' : 2, 'uno' : 1*}.

ESEMPIO

```
stringa = "Una stringa è un elenco ordinato di caratteri"
lista = ['Agata', 'Claudia', 'Gianni', 'Giuseppe', 'Franco']
tupla = (1, 'Giuseppe', 'Bianchi', 73.5, 'Roma')
insieme = {'il', 'lo', 'la', 'i', 'gli', 'le'}
dizionario = {'uno' : 1, 'due' : 2, 'tre' : 3, 'quattro' : 4}
```

La seguente tabella mette a confronto le strutture di dati viste in precedenza.

Oggetto	Classe	Modificabile	Accesso agli elementi	Elementi ordinati	Elementi duplicati
Stringa	**str**	No	In base alla posizione	Sì	Sì
Lista	**list**	Sì	In base alla posizione	Sì	Sì
Tupla	**tuple**	No	In base alla posizione	Sì	Sì
Insieme	**set**	Sì	Elencazione, estrazione	No	No
Dizionario	**dict**	Sì	In base al valore della chiave	No	No

```
    for j in range(len(dati)):
        totale = totale + dati[j]
    print(30 * '-')
    print("Incasso settimanale:", totale)
    print(30 * '-')

# funzione principale
def main():
    incassi = leggi_incassi()
    calcola_totale(incassi)

# esecuzione del programma
main()
```

stampa una linea con 30 trattini (segni meno)

ESECUZIONE

```
Incasso di Lun: 215
Incasso di Mar: 385.5
Incasso di Mer: 450.27
Incasso di Gio: 395.85
Incasso di Ven: 455
Incasso di Sab: 753.84
Incasso di Dom: 420.75
------------------------------
Incasso settimanale: 3076.21
------------------------------
```

NOTA BENE

Una funzione può restituire una lista dopo averla creata e una lista può essere passata a una funzione. La memorizzazione dei dati letti nella lista *incassi* è utile per eventuali sviluppi del programma: per esempio per calcolare l'incasso medio giornaliero e lo scostamento dalla media nei diversi giorni della settimana. Analogamente la lista *settimana*, collocata al di fuori della funzione *leggi_incassi*, potrà essere usata da nuove funzioni.

La lista come array

In informatica si usa il termine **array** per indicare una struttura dati di tipo omogeneo. Le liste di Python composte da elementi dello stesso tipo implementano la struttura array. Nella terminologia della matematica si usa il termine **vettore** per indicare gli array a una dimensione, cioè gli array con un solo indice.

CODING

1 LA SOMMA DEGLI ELEMENTI DI UN ARRAY DI NUMERI

Acquisire da tastiera un array di interi con un numero di componenti scelto dall'utente e sommare le sue componenti

Dati di input: la dimensione dell'array e il valore delle sue componenti.
Dati di output: la somma delle componenti dell'array.

Occorre acquisire in input il numero e il valore delle componenti dell'array, calcolarne la somma e mostrare il risultato. Ognuna di queste operazioni è affidata a una differente funzione, con l'eccezione della visualizzazione del risultato che è eseguita dalla funzione *main()*.
Poiché si può assegnare un valore a un elemento di una lista solo se esso esiste, dopo avere acquisito il numero di componenti *n*, la funzione *main()* crea un array con *n* zeri.

Programma Python (*SommaArray.py*)

```
# SommaArray.py: creazione di un array di interi, caricamento dati,
#                calcolo della somma delle componenti

# numero massimo di elementi
MAX = 100

# lettura del numero di componenti: 1 <= dimensione <= MAX
def chiedi_dimensione(dim_max):
    num = int(input("Dimensione del vettore: "))
    while num < 1 or num > dim_max:
        print("Valore non ammesso")
        num = int(input("Dimensione del vettore: "))
    return num

# lettura degli elementi del vettore
def carica_vettore(v):
    for k in range(len(v)):
        v[k] = int(input("Elemento di posto " + str(k) + ": "))

# calcola somma delle componenti
def calcola_somma(v):
    somma = 0
    for k in range(len(v)):
        somma += v[k]
    return somma

# funzione principale
def main():

    n = chiedi_dimensione(MAX)
    # creazione di un vettore con n componenti uguali a 0
    vet = [0] * n

    print("Caricamento degli elementi del vettore")
    carica_vettore(vet)
    totale = calcola_somma(vet)
    print("\nSomma del vettore =", totale)

# esecuzione del programma
main()
```

lettura controllata del numero di componenti

str(dato) trasforma *dato* in stringa

2 Le liste

Un importante esempio di struttura di dati di Python è la *lista*. La **lista** è un insieme di elementi dello stesso tipo o di tipo diverso. Con una variabile si può indicare solo un dato, mentre con la *lista* si possono indicare tanti dati con un solo nome collettivo di variabile: l'identificatore della lista. Gli elementi si distinguono uno dall'altro attraverso l'indice che ne indica la posizione nella lista e che viene posto accanto all'identificatore della lista, tra parentesi quadre.

Come già accennato nel Capitolo 2 (vedi pag. 39), una **lista** si rappresenta elencando gli elementi che la compongono in sequenza, separati da una virgola. La sequenza è racchiusa in una coppia di parentesi quadre.

ESEMPIO

```
>>> anni = [2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028]
>>> frutta = ['mela', 'pera', 'arancia', 'uva']
>>> dati = [37, 52.5, 'camicia']
```

Le liste *anni* e *frutta* sono liste con dati tutti dello stesso tipo, mentre la lista *dati* contiene dati di tipo diverso.

L'accesso ai singoli elementi di una lista e la definizione di **sottoliste** avvengono con le stesse modalità con le quali si estraggono sottostringhe da una stringa. L'estrazione di una porzione di lista si indica con il termine *slicing* (vedi pag. 40).

ESEMPIO

Osservando che la lista *anni* elenca gli anni "olimpici" di questo secolo fino al 2028 compreso, le Olimpiadi che si sono tenute e si terranno a partire dal 2012 e i primi tre anni olimpici di questo secolo sono i seguenti.

```
>>> anni[3:]
[2012, 2016, 2020, 2024, 2028]
>>> anni[:3]
[2000, 2004, 2008]
```

Per accedere al primo e ultimo elemento di *frutta* si usano gli indici *[0]* e *[-1]* (oppure *[3]*). Un indice, per essere valido, deve essere compreso tra 0 e il numero di elementi della lista diminuito di 1. Un indice non valido produce il messaggio di errore **out of range**.

```
>>> frutta[0]
'mela'
>>> frutta[-1]
'uva'
>>> frutta[4]
IndexError: list index out of range
```

frutta ha 4 elementi:
frutta[0], . . , *frutta*[3]

Il numero di elementi che compongono una lista si ottiene con la funzione **len**(*lista*).

ESEMPIO

```
>>> len(anni)     # restituisce: 8
>>> len(frutta)   # restituisce: 4
>>> len(dati)     # restituisce: 3
```

numero di elementi di una lista

NOTA BENE

L'indice di una lista per essere valido deve soddisfare la disuguaglianza:

0 <= *indice* <= len(*lista*) - 1

Oppure, se si usano indici negativi:

-len(*lista*) <= *indice* <= -1

Per elencare gli elementi di una lista si usa un ciclo *for* dove l'indice viene fatto variare entro i limiti desiderati con la funzione *range*.

ESEMPIO

Per elencare tutti gli elementi di *frutta* dal primo all'ultimo e in ordine inverso si usa un ciclo *for*.

```
>>> for indice in range(0,len(frutta)):
        print(frutta[indice], end = ' ')
mela pera arancia uva
```

```
>>> for indice in range(-1,-len(frutta)-1,-1):
        print(frutta[indice], end = ' ')
uva arancia pera mela
```

Quando il valore dell'indice non è necessario nelle istruzioni entro il ciclo *for*, l'elenco degli elementi di una lista si può ottenere con una differente sintassi, semplice e allo stesso tempo autoesplicativa:

```
>>> for elemento in frutta:
        print(elemento, end = ' ')
mela pera arancia uva
```

In generale l'accesso agli elementi di una lista può avvenire in due modi:

for *indice* **in range**(**len**(*lista*)):
  comandi con *indice* e *lista*[*indice*]

for *elemento* **in** *lista*:
  comandi con *elemento*

indice in range(0,len(lista))
equivale a:
indice in range(len(lista))

Una lista può essere creata in diversi modi.
In particolare, la funzione **list()** trasforma in liste oggetti di diverse classi tra i quali le stringhe, i dati generati dalla funzione *range()* e, come si vedrà nei paragrafi successivi, tuple, insiemi e dizionari.

ESEMPIO

Gli esempi seguenti mostrano come creare una lista vuota, una lista con un certo numero di elementi identici, una lista ottenuta accodando una lista a un'altra e infine una lista ottenuta con la funzione *list()*.

```
>>> lista = []
>>> lis1 = 5 * [0]
>>> lis2 = [12, 35, 99, 52, 8]
>>> lista = lis1 + lis2
>>> lista
[0, 0, 0, 0, 0, 12, 35, 99, 52, 8]
>>> lis1 = list(range(1,5))
>>> lis1
[1, 2, 3, 4]
>>> vocali = 'aeiou'
>>> lisvocali = list(vocali)
>>> lisvocali
['a', 'e', 'i', 'o', 'u']
```

lista con elementi identici

accodamento di liste

NOTA BENE

La stringa *vocali* e la lista *lisvocali* contengono gli stessi elementi ai quali si accede allo stesso modo, ma *vocali*, a differenza di *lisvocali*, non può essere modificata.

A ogni elemento di una lista può essere assegnato un nuovo valore con l'istruzione di assegnazione:

lista[*pos*] = *val*

Per inserire nuovi elementi in una lista si usa il metodo **append**(*val*) per aggiungere in coda alla lista un elemento di valore *val* e il metodo **insert**(*pos*, *val*) per inserire il nuovo elemento nella posizione *pos*.
Per togliere elementi da una lista si usa il metodo **pop()** che estrae e restituisce l'ultimo elemento di una lista. Lo stesso metodo serve per estrarre l'elemento di posto *pos* scrivendo **pop**(*pos*).

ESEMPIO

```
>>> lista
[0, 0, 0, 0, 0, 12, 35, 99, 52, 8]
>>> lista[0] = 99
>>> lista
[99, 0, 0, 0, 0, 12, 35, 99, 52, 8]
>>> lista.pop()
8
>>> lista
[99, 0, 0, 0, 0, 12, 35, 99, 52]
>>> lista.pop(1)
0
>>> lista.pop(1)
0
>>> lista
[99, 0, 0, 12, 35, 99, 52]
>>> lista.append(125)
>>> lista
[99, 0, 0, 12, 35, 99, 52, 125]
>>> lista.insert(2, 200)
>>> lista
[99, 0, 200, 0, 12, 35, 99, 52, 125]
```

CODING

1 IL CALCOLO DELL'INCASSO

Calcolare l'incasso settimanale di un negozio dopo avere letto e memorizzato in una lista l'incasso giornaliero

- **Dati di input**: gli incassi giornalieri.
  Dati di output: la somma degli incassi giornalieri.
- Occorre acquisire in input gli incassi giornalieri, calcolare la loro somma e mostrare il risultato del calcolo. Il programma usa la funzione *leggi_incassi* che acquisisce i dati con gli incassi giornalieri e li inserisce in una lista. La lista con gli incassi giornalieri è passata alla funzione *calcola_totale* che somma gli elementi della lista e stampa il risultato.
- **Programma Python** (*IncassoSettimanale.py*)

```
# IncassoSettimanale.py: calcolo dell'incasso settimanale

# settimana è una variabile globale
settimana = ['Lun', 'Mar', 'Mer', 'Gio', 'Ven', 'Sab', 'Dom']

# acquisisce gli incassi per ogni giorno della settimana
def leggi_incassi():
    incassi = []
    for giorno in settimana:
        valore = float(input("Incasso di " + giorno + ": "))
        incassi.append(valore)
    return incassi

# calcola incasso settimanale e lo mostra
def calcola_totale(dati):
    totale = 0
```

```
    for j in range(len(dati)):
        totale = totale + dati[j]
    print(30 * '-')
    print("Incasso settimanale:", totale)
    print(30 * '-')

# funzione principale
def main():
    incassi = leggi_incassi()
    calcola_totale(incassi)

# esecuzione del programma
main()
```

stampa una linea con 30 trattini (segni meno)

ESECUZIONE

```
Incasso di Lun: 215
Incasso di Mar: 385.5
Incasso di Mer: 450.27
Incasso di Gio: 395.85
Incasso di Ven: 455
Incasso di Sab: 753.84
Incasso di Dom: 420.75
------------------------------
Incasso settimanale: 3076.21
------------------------------
```

NOTA BENE

Una funzione può restituire una lista dopo averla creata e una lista può essere passata a una funzione. La memorizzazione dei dati letti nella lista *incassi* è utile per eventuali sviluppi del programma: per esempio per calcolare l'incasso medio giornaliero e lo scostamento dalla media nei diversi giorni della settimana. Analogamente la lista *settimana*, collocata al di fuori della funzione *leggi_incassi*, potrà essere usata da nuove funzioni.

La lista come array

In informatica si usa il termine **array** per indicare una struttura dati di tipo omogeneo. Le liste di Python composte da elementi dello stesso tipo implementano la struttura array. Nella terminologia della matematica si usa il termine **vettore** per indicare gli array a una dimensione, cioè gli array con un solo indice.

CODING

1 LA SOMMA DEGLI ELEMENTI DI UN ARRAY DI NUMERI

Acquisire da tastiera un array di interi con un numero di componenti scelto dall'utente e sommare le sue componenti

Dati di input: la dimensione dell'array e il valore delle sue componenti.
Dati di output: la somma delle componenti dell'array.

Occorre acquisire in input il numero e il valore delle componenti dell'array, calcolarne la somma e mostrare il risultato. Ognuna di queste operazioni è affidata a una differente funzione, con l'eccezione della visualizzazione del risultato che è eseguita dalla funzione *main()*.
Poiché si può assegnare un valore a un elemento di una lista solo se esso esiste, dopo avere acquisito il numero di componenti *n*, la funzione *main()* crea un array con *n* zeri.

Programma Python (*SommaArray.py*)

```
# SommaArray.py: creazione di un array di interi, caricamento dati,
#                calcolo della somma delle componenti

# numero massimo di elementi
MAX = 100

# lettura del numero di componenti: 1 <= dimensione <= MAX
def chiedi_dimensione(dim_max):
    num = int(input("Dimensione del vettore: "))
    while num < 1 or num > dim_max:
        print("Valore non ammesso")
        num = int(input("Dimensione del vettore: "))
    return num

# lettura degli elementi del vettore
def carica_vettore(v):
    for k in range(len(v)):
        v[k] = int(input("Elemento di posto " + str(k) + ": "))

# calcola somma delle componenti
def calcola_somma(v):
    somma = 0
    for k in range(len(v)):
        somma += v[k]
    return somma

# funzione principale
def main():

    n = chiedi_dimensione(MAX)
    # creazione di un vettore con n componenti uguali a 0
    vet = [0] * n

    print("Caricamento degli elementi del vettore")
    carica_vettore(vet)
    totale = calcola_somma(vet)
    print("\nSomma del vettore =", totale)

# esecuzione del programma
main()
```

lettura controllata del numero di componenti

str(dato) trasforma *dato* in stringa

Il programma mostra il modo standard per scandire le componenti di un array. Sostituendo la funzione *calcola_somma* con funzioni opportune, si può usare la struttura generale di *SommaArray.py* per elaborazioni standard con gli array numerici quali: il calcolo del valore medio, del minimo e del massimo delle sue componenti.

NOTA BENE

Il linguaggio Python possiede una funzione predefinita **sum**(*nomelista*) che restituisce la somma di tutti gli elementi della lista fornita come argomento.

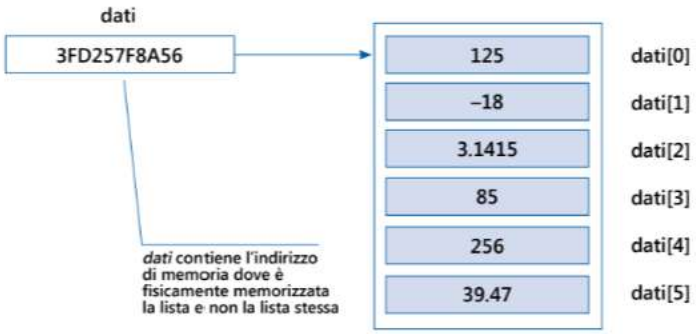
La gestione delle liste in memoria

Nel programma precedente si nota che la funzione *carica_vettore* riceve come parametro l'array *vet*, assegna alle sue componenti i valori acquisiti da tastiera e, mancando il comando *return*, non restituisce nulla a *main()*. Tuttavia, eseguendo il programma, si può controllare che tutto funziona correttamente e che *main()* "vede" le modifiche apportate all'array dalla funzione *carica_vettore*.

Per comprendere come questo sia possibile, bisogna conoscere le modalità con cui Python gestisce le liste. La seguente figura mostra, appunto, l'allocazione in memoria della lista *dati*.

dati = [125, -18, 3.1415, 85, 256, 39.47]

L'allocazione in memoria di una lista



Il contenuto di *dati* è *3FD257F8A56*, cioè l'**indirizzo di memoria** dove sono realmente memorizzati i contenuti della lista. Quando viene eseguito il comando *carica_vettore(dati)*, il valore inviato alla funzione *carica_vettore* è il valore *3FD257F8A56*.

La funzione chiamata agisce sulla lista collocata in memoria all'indirizzo *3FD257F8A56* e, in altre parole, sulla stessa lista del chiamante. Questo è il motivo per cui *main()* "vede" le modifiche apportate da *carica_vettore* alla lista passata come parametro.

NOTA BENE

In informatica si dice che il passaggio di valori tra una funzione chiamante e una funzione chiamata può avvenire **per valore** o **per indirizzo** e, nel caso di Python, si dice che "i numeri sono passati per valore e le liste sono passate per indirizzo". In realtà questo non accade: i dati sono sempre passati per valore. Però, poiché il valore di una variabile riferita a una lista è l'indirizzo della lista stessa, gli effetti concreti sono esattamente quelli di una chiamata per indirizzo.

Le precedenti considerazioni valgono anche per stringhe, tuple, insiemi e dizionari. L'assegnazione *lista1 = lista* avviene quindi assegnando a *lista1* l'indirizzo dove si trovano i valori di *lista*. Questo spiega cosa succede nel seguente esempio, dove si eseguono alcune assegnazioni che coinvolgono le liste *dati* e *valori*.

ESEMPIO

```
>>> dati = [125, -18, 3.1415, 85, 256, 39.47]
>>> valori = dati
>>> valori
[125, -18, 3.1415, 85, 256, 39.47]
>>> valori == dati
True
>>> valori is dati
True
>>> dati[0] = 'nuovo valore'
>>> valori
['nuovo valore', -18, 3.1415, 85, 256, 39.47]
>>> valori[3] = 300
>>> dati
['nuovo valore', -18, 3.1415, 300, 256, 39.47]
# crea una lista con gli stessi dati ma differente da valori
>>> dati = ['nuovo valore', -18, 3.1415, 300, 256, 39.47]
>>> valori == dati
True
>>> valori is dati
False
```

valori punta alla stessa lista di *dati*: *valori* è sinonimo di *dati*

== indica il confronto di contenuti; **is** indica il confronto di identità

ogni modifica su *dati* è una modifica su *valori* e viceversa

dati e *valori* sono due liste differenti

NOTA BENE

I due operatori di confronto **==** e **is** fanno cose diverse: il primo confronta due oggetti e ne valuta il contenuto, il secondo confronta l'identità degli oggetti per sapere se sono il medesimo oggetto. Questi operatori (e le rispettive negazioni **!=**, **is not**) si usano allo stesso modo anche con le altre strutture dati.

3 Le matrici, le pile e le code con le liste

Una lista può avere come elementi altre liste.

ESEMPIO

La lista *lis* è composta da tre elementi. *lis[0]* è una lista con tre interi, *lis[1]* è una lista con due stringhe, *lis[2]* è una lista con due valori numerici, dei quali il primo è un numero reale, il secondo complesso.

```
>>> lis = [[1,2,3], ['mele', 'pere'], [3.14, 5+3j]]
           lis[0]        lis[1]          lis[2]
```

Modificare il primo elemento della lista *lis[1]*, per sostituire *'mele'* con *'arance'*:

```
>>> lis[1][0] = ['arance']
```

La matrice

Di particolare interesse sono le liste che hanno per elementi liste con lo stesso numero di elementi, tutti dello stesso tipo. Si parla in tale caso di *array a due dimensioni* e, in matematica, di *matrici*.

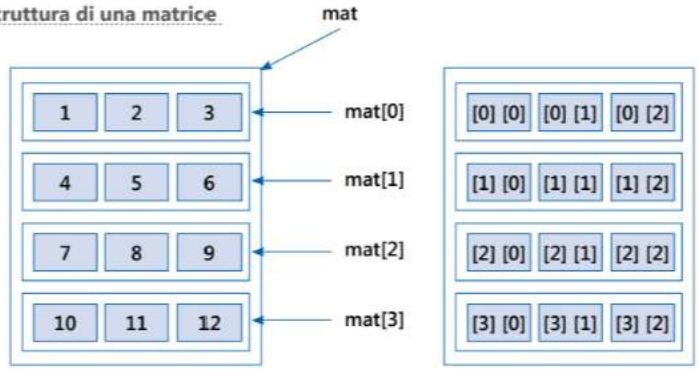
La **matrice** (o **array a due dimensioni**) è un insieme di elementi dello stesso tipo e in corrispondenza biunivoca con un insieme di coppie ordinate di numeri interi, che rappresentano rispettivamente il numero della riga e il numero della colonna della matrice.

La lista *mat*:

```
>>> mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

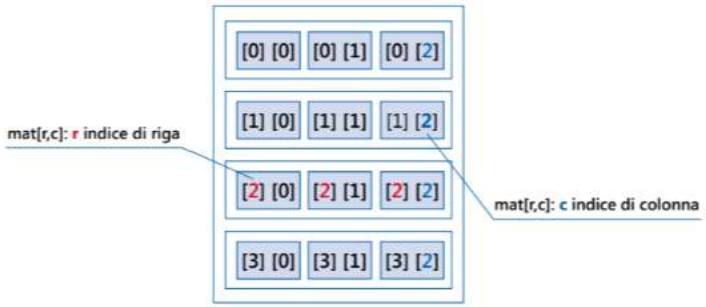
è un array a due dimensioni composto da quattro liste *mat[0]* , .. , *mat[3]* che possono essere disposte su più righe come appare nelle seguente figura, che mostra anche come usare i due indici per accedere ai diversi elementi della matrice.

La struttura di una matrice



Per accedere a un elemento della matrice *mat* si usano scritture con due indici, come in *mat[r][c]*, dove il primo indice è detto **indice di riga** e il secondo **indice di colonna**.

L'accesso agli elementi della matrice



ESEMPIO

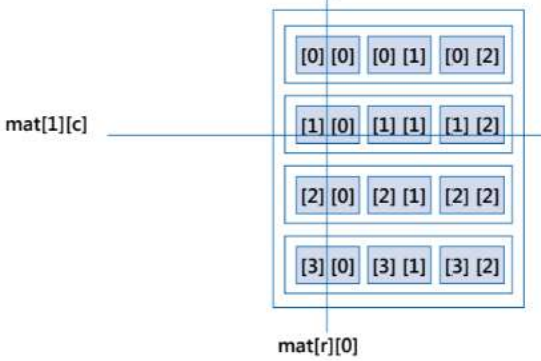
Per organizzare gli incassi nei reparti di un grande magazzino nei diversi giorni della settimana, la struttura dati più adatta è una matrice di dati di tipo numerico, con tante righe quanti sono i reparti e 7 colonne, in corrispondenza dei giorni della settimana. Volendo calcolare l'incasso settimanale di un certo reparto bisogna sommare i dati che stanno su una stessa riga, mentre per calcolare l'incasso del magazzino in un certo giorno bisogna sommare i dati che stanno sulla colonna corrispondente a quel giorno.

La seguente figura mostra come si procede per sommare i dati di una riga e di una colonna, nel caso della matrice *mat*.

La somma dei dati

```
tot_col = 0
for r in range(0,len(mat)):
    tot_col += mat[r][0]
```

scansione di una colonna: fissato l'indice di colonna, si fa variare l'indice di riga




```
tot_rig = 0
for c in range(0,len(mat[1])):
    tot_rig += mat[1][c]
```

scansione di una riga: fissato l'indice di riga, si fa variare l'indice di colonna

CODING

1 I TOTALI DI RIGA E COLONNA

Calcolare i totali di riga e di colonna degli elementi di una matrice

Dati di input: il numero di righe e di colonne e il valore degli elementi della matrice.
Dati di output: la somma delle righe e delle colonne.

Il problema va scomposto in due parti: la prima riguarda il caricamento dei dati in memoria, la seconda è relativa ai calcoli e alla scrittura dei risultati. La struttura dati definita è un array a due dimensioni.
Per il caricamento dei dati si considera ogni riga della matrice come un vettore e si procede come mostrato sopra. Questa operazione viene annidata in una ripetizione più esterna che permette di ripetere l'operazione di acquisizione dei dati per tutte le righe della matrice.
Nella seconda parte, relativa al calcolo dei totali di riga e di colonna, per ogni riga si azzera la variabile di accumulo, si totalizzano in questa variabile i valori della riga con una ripetizione, con il contatore che va da 0 al numero di colonne – 1, e alla fine si scrive il contenuto della variabile di totalizzazione.

```
# esecuzione di una lavorazione
def esegui():
    if len(lavoro) == 0:
        print("Coda lavorazioni vuota")
    else:
        codlav = lavoro.pop(0)
        print("Avvio lavorazione:", codlav)

# visualizzazione delle lavorazioni
def visualizza():
    print("Coda delle lavorazioni")

    for pos in range(len(lavoro)):
        codlav = lavoro[pos]
        print("Codice:", codlav)

# funzione principale
def main():
    proseguo = True
    while proseguo:
        presenta_menu()
        scelta = scegli_operazione()        # scelta da menu

        # esegue la scelta
        if scelta == 1: accoda()
        elif scelta == 2: esegui()
        elif scelta == 3: visualizza()
        else:
            print("Fine lavoro")            # scelta=4
            proseguo = False                # fine esecuzione

# esecuzione del programma
main()
```

preleva la lavorazione dalla cima della coda

Coda lavorazioni

4 Le liste e le stringhe

La funzione *list*

La funzione **list** costruisce una lista che ha per elementi i caratteri di una stringa.

ESEMPIO

```
>>> cognome = 'Bianchi'
>>> l_cognome = list(cognome)
>>> l_cognome
['B', 'i', 'a', 'n', 'c', 'h', 'i']
```

La stringa *cognome* differisce dalla lista *l_cognome* perché, a differenza della stringa che non è modificabile, alla lista di caratteri si può apportare ogni tipo di modifica inserendo o togliendo caratteri. Dopo avere modificato la composizione della lista, i caratteri che la compongono possono essere inseriti in una nuova stringa e, con un'assegnazione, è possibile modificare la stringa di partenza.

ESEMPIO

Data la lista di caratteri *lis*, il seguente frammento di programma costruisce una stringa con i caratteri della lista e li attribuisce alla stringa *nuova_stringa*.

```
for j in range(1, len(lis)):
    lis[0] += lis[j]
    nuova_stringa = lis[0]
```

l'operatore + qui indica il concatenamento e non la somma degli elementi

Si può usare questa tecnica per costruire una funzione che ordini i caratteri di una stringa. Prima di farlo occorre considerare il metodo **sort()** che ordina gli elementi di una lista.

ESEMPIO

```
>>> frutta = ['mela', 'pera', 'arancia', 'uva']
>>> frutta.sort()
>>> frutta
['arancia', 'mela', 'pera', 'uva']
```

CODING

1 ORDINARE E FONDERE STRINGHE

Creare un modulo con le funzioni ordina() *e* fondi()

- La prima funzione *ordina(stringa)* riceve una stringa e restituisce una copia ordinata della stringa.
  La seconda funzione *fondi(stringa1, stringa2)* riceve due stringhe e ne restituisce una con tutti i caratteri di *stringa1* e *stringa2* ordinati.
  La realizzazione di *fondi()* è molto semplice una volta realizzata *ordina()*, perché deve solo eseguire il comando: *ordina(stringa1 + stringa2)*.

- **Modulo** (*OrdinaStringhe.py*)

```
"""
Modulo: OrdinaStringhe.py
Autore: studente1
        Ordinamento e fusione di stringhe
"""

def ordina(stringa):
    """ restituisce la stringa ordinata """
    lis = list(stringa)
    lis.sort()
```

```
    for j in range(1, len(lis)):
        lis[0] += lis[j]
    return lis[0]

def fondi(stringa1, stringa2):
    """ restituisce la stringa ordinata con i caratteri
        di stringa1 e stringa2
    """
    stringa = ordina(stringa1 + stringa2)
    return stringa
```

Supponendo di aver salvato il modulo nella cartella *EserciziPython*, le funzioni del modulo possono essere collaudate dalla *Shell* dell'ambiente IDLE.

ESECUZIONE

```
>>> import sys
>>> sys.path.append('C:\\EserciziPython')
>>> from OrdinaStringhe import ordina, fondi
>>> stringa = "14358291"
>>> ordina(stringa)
'11234589'
>>> str1 = "limone"
>>> str2 = "arancia"
>>> fondi(str1, str2)
'aaaceiilmnnor'
```

NOTA BENE

La funzione *ordina()* che ordina una stringa non è un semplice esercizio, ma è una funzione di utilità generale, perché non esiste una funzione *sort()* che restituisca una stringa ordinata.

I metodi *split* e *strip*

Per analizzare il contenuto di una frase bisogna scomporla nelle parole che la compongono. Con Python si può usare il metodo **split()**, che genera una lista con le parole che compongono una frase. Si può scomporre la frase usando come separatore di *default* una sequenza di uno o più caratteri spazio, oppure scegliendo uno specifico separatore:

- *split()* usa come separatore una sequenza di uno o più spazi;
- *split(separatore)* usa come separatore la stringa *separatore*.

ESEMPIO

metodo **split()**

```
>>> # separatore: uno o più spazi
>>> frase = "Frase con elenco: 1, 2, 3, 4"
>>> frase.split()
['Frase', 'con', 'elenco:', '1,', '2,', '3,', '4']
```



```
>>> # separatore: la virgola seguita da uno spazio
>>> frase = "1, 2, 3, 4"
>>> frase.split(", ")
['1', '2', '3', '4']
>>> frase = "1,   2,  3, 4"
>>> frase.split(", ")
['1', '  2', ' 3', '4']

>>> # separatore: il carattere meno
>>> frase = "1 - 2 - 3 - 4"
>>> frase.split('-')
['1 ', ' 2 ', ' 3 ', ' 4']
```

L'esempio mostra che la separazione attuata da *split()* può generare parole spurie, cioè nelle quali sono presenti segni di interpunzione e/o caratteri spazio. Per eliminare questi caratteri indesiderati si usa il metodo **strip()**. Come nel caso di *split()*, anche *strip()* ha un comportamento di *default*, che si usa per togliere eventuali caratteri di spaziatura in testa e in coda a una stringa, oppure, usando un parametro, per eliminare i caratteri elencati nel parametro:

- *strip()* elimina uno o più caratteri di spaziatura (spazi, tabulazioni e *newline*) in testa e in coda;
- *strip(stringa)* elimina, in testa e in coda, i caratteri che compaiono in *stringa*.

ESEMPIO

metodo **strip()**

```
>>> # toglie spazi in testa e in coda
>>> "   parola : ".strip()
'parola :'
>>> # toglie due punti in testa e in coda
>>> "   parola:   ".strip(':')
'   parola:   '
>>> # toglie spazi e due punti in testa e in coda
>>> " : parola:   ".strip(': ')
'parola'
>>> # toglie i caratteri elencati
>>> elenco_car = "',;.:?! "
>>> " , : parola ? ".strip(elenco_car)
'parola'
```

I caratteri indesiderati possono essere eliminati in modo più selettivo con i metodi **lstrip()**, che sta per *left strip*, e **rstrip()**, abbreviazione di *right strip*, che agiscono, rispettivamente, solo all'inizio o solo alla fine della stringa alla quale sono applicati.

I metodi *split()* e *strip()* sono utili per scomporre una frase nelle parole che la compongono, rimuovendo gli spazi di separazione e i segni di punteggiatura, come mostrato nel *Coding* seguente.

CODING

1 L'ANALISI DI UNA FRASE

Analizzare una frase scomponendola nelle parole che la formano

L'analisi deve produrre il numero di parole di lunghezza maggiore di 2 caratteri e la lunghezza media delle parole considerate.
Per semplicità la frase da analizzare è assegnata a una variabile stringa direttamente nel codice del programma. Il programma scompone la frase elencando le parole che la compongono con il metodo *split()*. Dalle singole parole sono rimossi i segni di interpunzione con il metodo *strip()*.

Programma Python (*ContaParole.py*)

```
# ContaParole.py: analisi di una frase
#                 conta le parole con più di 2 caratteri di una frase
#                 usa i metodi stringa: split() e strip()

frase = "Utilizzando una stampante condivisa tra più postazioni, \
è frequente che si accodino stampe o lavori nelle rispettive code: \
i moduli software del sistema operativo che gestiscono queste \
attività del sistema di elaborazione seguono infatti il principio \
della coda, per cui se più utenti richiedono le stampe sulla stessa \
stampante, esse vengono eseguite nello stesso ordine con cui sono \
state richieste: si parla di coda di SPOOL."

print(80 * '-')
print(frase)
print(80 * '-')

punteggiatura = "',:;.!?"
lista_parole = frase.split()
num_totale = len(lista_parole)
num_par = 0
num_car = 0

for parola in lista_parole:
    parola = parola.strip(punteggiatura)
    if len(parola) > 2:
        num_par += 1
        num_car += len(parola)
        print("parola considerata:", parola)

media_parola = num_car // num_par

print(80 * '-')
print("Numero complessivo parole:", num_totale)
print("Numero parole considerate:", num_par)
print("Lunghezza media delle parole:", media_parola, "caratteri")
```

lista_parole: parole ottenute eliminando gli spazi ma non i segni di interpunzione

eliminazione dei segni di interpunzione dalle parole

sono considerate solo le parole di almeno 3 caratteri

Nella prova di esecuzione seguente è stata omessa la stampa della frase con l'elenco delle parole considerate.

ESECUZIONE

```
Numero complessivo parole: 66
Numero parole considerate: 55
Lunghezza media delle parole: 6 caratteri
```

Analisi di una frase

PER APPROFONDIRE

GLI INPUT MULTIPLI

Il metodo *split()* viene usato per la lettura contemporanea di più dati con un solo comando *input*. Per leggere i valori di più variabili, si usa il carattere che separa i dati come parametro del metodo *split()* per spezzare la stringa digitata nelle componenti da assegnare alle variabili.

ESEMPIO

Il seguente frammento di codice calcola il volume di un parallelepipedo, dopo aver acquisito il valore delle tre dimensioni che lo caratterizzano ed eseguito il loro prodotto:

```
a, b, c = input("Le tre dimensioni separate da una virgola: ").split(',')
volume = float(a) * float(b) * float(c)
print(volume)
```

Analogamente, per leggere le tre componenti di una data separate dal carattere – (segno meno), si usa la seguente istruzione:

```
>>> giorno, mese, anno = input("Data nel formato gg-mm-aaaa: ").split('-')
Data nel formato gg-mm-aaaa: 25-9-2020
>>> print(giorno, mese, anno)
25 9 2020
```

NOTA BENE

Le variabili ottenute con lo *split* della stringa di input (*giorno*, *mese*, *anno*, così come le tre dimensioni del parallelepipedo *a*, *b*, *c*) sono stringhe:

```
>>> type(giorno)
<class 'str'>
```

Le funzioni e i metodi per le stringhe

Per elaborare o analizzare le stringhe Python si possono usare anche altre funzioni, per esempio *ord()* e *chr()*, e altri metodi, per esempio *capitalize()*, *count()*, *replace()*, *index()*, *find()*.

Come già accennato nel Capitolo 3 (vedi pag. 92), le funzioni *ord()* e *chr()* sono l'una l'inverso dell'altra, perché **ord()** restituisce il numero intero che corrisponde alla rappresentazione di un carattere nella codifica adottata, mentre **chr()** restituisce il carattere corrispondente a un intero: *ord(chr(50))* vale 50 e *chr(ord('G'))* restituisce il carattere *'G'*.

ESEMPIO

```
>>> # ord - intero corrispondente a un carattere
>>> ord('A')
65
>>> ord('a')
97
>>> # chr - carattere corrispondente a un intero
>>> chr(66)
'B'
```

Il metodo **capitalize()** converte in maiuscolo il primo carattere di una stringa.

ESEMPIO

```
>>> 'presidente'.capitalize()
'Presidente'
```

Il metodo **count()** conta le occorrenze di una sottostringa.

ESEMPIO

```
>>> 'programmare con python'.count('r')
3
>>> 'programmare con python'.count('r', 4)
2
```

conta a partire dal carattere di posizione 4

Il metodo **replace()** sostituisce certe sequenze con altre.

ESEMPIO

```
>>> '0123ab----ab'.replace('ab', 'cd')
'0123cd----cd'
>>> '0123ab----ab'.replace('ab', 'cd', 1)
'0123cd----ab'
```

rimpiazza *'ab'* con *'cd'* una sola volta

I metodi **find()** e **index()** restituiscono la posizione di una sottostringa in una stringa. I due metodi differiscono per il fatto che, se la sottostringa non c'è, *find()* restituisce -1, mentre *index()* solleva un'eccezione.

ESEMPIO

```
>>> 'programmare'.find('r')
1
>>> 'programmare'.find('r', 3)
4
>>> 'programmare'.find('z')
-1
>>> 'programmare'.index('m')
6
>>> 'programmare'.index('z')
ValueError: substring not found
```

cerca la lettera *r* a partire dalla posizione 3

Le seguenti tabelle riassumono gli operatori, le funzioni e i metodi per le stringhe descritti in precedenza.

Operatori e funzioni	Risultato
st1 + st2	Restituisce una concatenazione di stringhe
st * n	Restituisce i caratteri di *st* ripetuti *n* volte
st[2:5]	Restituisce i caratteri di posto 2, 3, 4 di *st*
len(st)	Restituisce il numero dei caratteri in *st*
st1 in st	Controlla la presenza di *st1* in *st*: restituisce *True* o *False*
st1 not in st	Controlla l'assenza di *st1* in *st*: restituisce *True* o *False*
st1 == st2	Controlla se *st1* è uguale a *st2*: restituisce *True* o *False*
st1 is st2	Controlla se *st1* e *s2* sono la stessa stringa; restituisce *True* o *False*
ord(carattere)	Restituisce il valore decimale assegnato alla codifica di un carattere
chr(intero)	Restituisce il carattere corrispondente a un intero
str(argomento)	Restituisce una stringa con il contenuto dell'argomento (numero o lista)

Metodi	Risultato
st.lower()	Restituisce una copia di *st* in caratteri minuscoli
st.upper()	Restituisce una copia di *st* in caratteri maiuscoli
st.capitalize()	Restituisce una copia di *st* con iniziale maiuscola
st.count(st1)	Conta le occorrenze di *st1* in *st*
st.replace(st1, st2)	Restituisce una copia di *st* sostituendo *st1* in *st* con *st2*
st.split()	Restituisce una lista con le parole che compongono *st*
st.strip()	Restituisce una stringa senza caratteri di spaziatura (spazi, tabulazioni e *newline*) all'inizio e alla fine
st.index(st1)	Restituisce la posizione di *st1* in *st* (segnala errore se *st1* non c'è)
st.find(st1)	Restituisce la posizione di *st1* in *st* (-1 se *st1* non c'è)

5 Le liste con funzioni

Con Python si possono definire liste che hanno come elementi delle funzioni.

ESEMPIO

Dopo avere definito le funzioni *fun1()*, *fun2()*, *fun3()*, si può creare la lista *funzioni* che ha come componenti le funzioni stesse:

```
funzioni = [fun1, fun2, fun3]
```

Le tre funzioni dell'esempio possono essere invocate con il nome *funzioni* precisando la posizione che esse occupano nella lista. Il comando *funzioni[0]()* equivale a eseguire *fun1()* e, in modo simile, possono essere richiamate *fun2()* e *fun3()*. La lista *funzioni* si comporta come una "superfunzione" che ha effetti diversi a seconda del valore dell'indice. La lista di funzioni permette di attivare diverse operazioni in base a un menu di scelte.

ESEMPIO

Nel *Coding* per la gestione di una pila (vedi pag. 142) sono state eseguite diverse funzioni con le seguenti scelte:

- *Aggiungi un dato* per eseguire la funzione *aggiungi()*;
- *Estrai un dato* per eseguire la funzione *estrai()*;
- *Svuota la pila* per eseguire la funzione *svuota_pila()*;
- *Fine* per visualizzare il messaggio *"Fine operazioni"*.

Per usare una sola funzione che esegua le diverse operazioni, bisogna prima definire le tre funzioni *aggiungi()*, *estrai()* e *svuota_pila()*, come nel programma *pila.py*, e poi scrivere la seguente funzione *main()* che esegue le diverse scelte:

```
def main():
    operazioni = [aggiungi, estrai, svuota_pila]
    presenta_menu()
    scelta = scegli_operazione()
    while scelta < 4:
        operazioni[scelta-1]()
        presenta_menu()
        scelta = scegli_operazione()

    print("Fine operazioni")                # scelta=4
```

operazioni riassume le tre funzioni: *aggiungi*, *estrai* e *svuota_pila*

esecuzione parametrica di *operazioni*

La funzione *map*

La funzione **map(***f, collezione***)** esegue un *mapping* tra una funzione *f* e tutti gli elementi di una *collezione* di dati, che può essere una lista, una stringa, una tupla, un insieme o un dizionario.
La funzione restituisce un oggetto che, per essere utilizzato, deve concretizzarsi in una struttura di dati, quale una lista, con il comando:

```
list(map(f, collezione))
```


ESEMPIO

```
>>> lista = ['23', '12 ', '5',' 231']
>>> list(map(int,lista))
[23, 12, 5, 231]
>>> lista
['23', '12 ', '5',' 231']

>>> list(map(ord,'ABCDE'))
[65, 66, 67, 68, 69]
>>> list(map(chr,[65, 66, 67, 68, 69]))
['A', 'B', 'C', 'D', 'E']
```

la funzione *int()* applicata agli elementi di *lista*

la struttura *lista* iniziale rimane invariata

la funzione *ord()* applicata a una stringa e la funzione *chr()* applicata a una lista di numeri

L'uso di *map* per input multipli

Nell'approfondimento sugli input multipli (vedi pag. 151) sono state lette le tre dimensioni di un parallelepipedo usando il metodo *split()* per scomporre una stringa in tre parti e, in una successiva istruzione, le stringhe sono state convertite in numeri per calcolare il volume del parallelepipedo. La funzione *map()* permette di eseguire le due operazioni con una sola istruzione.
Infatti il comando:

```
input("Le tre dimensioni separate da una virgola: ").split(',')
```

produce una lista con tre stringhe alle quali bisogna applicare, separatamente, la funzione *float()* per trasformare i valori letti in numeri. La funzione *map()* serve appunto per applicare *float()* alla lista prodotta dal comando *input*.

ESEMPIO

```
>>> a,b,c = map(float, input("Tre valori separati dalla
virgola: ").split(','))
Tre valori separati dalla virgola: 4,5,9
>>> print(a, b, c)
4.0 5.0 9.0
```

L'uso di *map* per la lettura dei dati di una matrice

La funzione *map()* può essere usata per leggere i dati di una matrice, una riga alla volta, in modo indipendente dal numero di elementi presenti nella riga.

ESEMPIO

I dati sono letti nella stringa *riga*:

```
riga = input("Elenco di interi separati da spazi: ")
```

Se *riga* contiene la stringa *'12 13 -45 234 97'*, si estrae una lista di interi con il comando:

```
list(map(int, riga.split()))
```

La riga di interi dall'esempio precedente può essere aggiunta, con il metodo **append()**, in coda alle righe della matrice che si deve caricare. La lettura dei dati e l'inserimento nella matrice si eseguono con il seguente codice Python:

```
# creazione di una matrice vuota
mat = []

# caricamento dei dati
riga = input("Elenco interi separati da spazi (* = fine): ")

while riga != '*':
    dati = list(map(int, riga.split()))
    mat.append(dati)
    riga = input("Elenco interi separati da spazi (* = fine): ")

print(mat)
```

Il numero di elementi per riga non è prefissato e, inoltre, il numero di elementi letti può variare da riga a riga.

CODING

1 I CORSI ATTIVATI IN UN'UNIVERSITÀ

Costruire i codici identificativi dei corsi di un'università, formati dai primi cinque caratteri della descrizione, dalla cifra dell'anno universitario del corso (1, 2, 3) e dai primi tre caratteri del cognome del docente. I codici così formati sono registrati in una lista. Al termine il programma deve visualizzare l'elenco dei codici creati

I dati sui corsi sono inseriti da tastiera. Le informazioni sul singolo corso sono inserite in una stringa ottenuta dalla concatenazione di 40 caratteri per il nome del corso, 1 carattere per l'anno di corso e 20 caratteri per il nome del docente. L'elenco con i dati di tutti i corsi è quindi memorizzato in una lista di stringhe.
La funzione *genera()* definisce il codice identificativo di un esame. Essa è usata dalla funzione *map()* per generare la lista dei codici degli esami.

Programma Python (*CreaCodici.py*)

```
# CreaCodici.py: generazione dei codici degli esami universitari
#                richiede i dati da tastiera e li inserisce in lista

def carica_corsi():

    lista = []

    descrizione = input("Descrizione del corso (* = fine): ")

    while descrizione != '*':
        anno = input("Anno del corso (1-3): ")
        cognome = input("Cognome del docente : ")
        riga = impacca(descrizione, anno, cognome)
        lista.append(riga)
        descrizione = input("\nDescrizione del corso (* = fine): ")
    return lista

# impacca i dati in una stringa di 40 + 1 + 20 caratteri
def impacca (descrizione, anno, cognome):
    if len(descrizione) < 40:
        descrizione += (40 - len(descrizione)) * ' '
    if len(cognome) < 20:
        cognome += (20 - len(cognome)) * ' '
    return descrizione[0:40] + anno[0:1] + cognome[0:20]

# funzione usata da map per generare la lista dei codici
def genera(stringa):
    return stringa[0:5] + stringa[40:41] + stringa[41:44]

# stampa i codici dei corsi
def mostra_codici(lista_codici):
    for codice in lista_codici:
        print(codice)

# funzione principale
def main():
    dati_corsi = carica_corsi()
    codici = list(map(genera, dati_corsi))
    print()
    mostra_codici(codici)

# esecuzione del programma
main()
```

La funzione *filter*

La funzione **filter(***f, collezione***)**, come la funzione *map()*, applica una funzione agli elementi di una collezione di dati con lo scopo di selezionare gli elementi della collezione che soddisfano un predicato. La funzione restituisce un'altra collezione, composta dagli elementi della collezione iniziale per i quali il predicato è vero.

ESEMPIO

Si supponga di avere un elenco di dati di tipo numerico e di dover scartare quelli di valore inferiore al 75% del loro valore medio. Per usare *filter()* serve una funzione che selezioni i dati accettabili, ossia quelli di valore superiore o uguale al 75% della media.

```
>>> dati = [1, 5, 12, 2, 4, 8, 10]
>>> def valido(val):
        return val >= sum(dati) / len(dati) * 0.75
```

la funzione *sum()* restituisce la somma di tutti gli elementi della lista *dati*

La funzione *filter()* seleziona i dati validi in questo modo:

```
>>> dati_validi = list(filter(valido, dati))
>>> dati_validi
>>> [5, 12, 8, 10]
```

La funzione precedente di validazione dei dati, essendo composta dalla sola istruzione *return*, può essere opportunamente inserita in *filter* per mezzo di una funzione *lambda* (vedi pag. 90):

```
>>> list(filter(lambda val: val >= sum(dati) / len(dati) *
0.75, dati))
[5, 12, 8, 10]
```

Le seguenti tabelle elencano sinteticamente operazioni, funzioni e metodi delle liste. In particolare, il metodo **remove(*val*)** elimina dalla lista un elemento di valore *val* e il metodo **index(*val*)** restituisce la posizione dell'elemento di valore *val* nella lista. In entrambi i casi nella lista deve essere presente l'elemento di valore *val*, diversamente l'esecuzione del metodo causa un errore. Per evitarlo bisogna controllare preventivamente l'esistenza dell'elemento con l'operatore **in**: *val* in *lista*.

Comandi	Risultato
lis = [] oppure **lis = list()**	Crea una lista vuota
lis = [12, 3, 7, 25]	Crea una lista con 4 elementi di posto 0, 1, 2, 3
lis = list(range(1,15,3)	Crea la lista [1, 4, 7, 10, 13]
lis1 = lis	*lis1* è un sinonimo di *lis*; *lis1* e *lis* sono lo stesso oggetto
lis1 = list(lis)	*lis1* e *lis* sono uguali ma sono oggetti distinti
lis[k] = val	Assegna *val* all'elemento di posto *k* nella lista (con *k* valido)
val = lis[k]	Assegna a *val* il valore dell'elemento di posto *k* (con *k* valido)
list(stringa)	Restituisce una lista che ha per elementi i caratteri della stringa

Operazioni e funzioni	Risultato
lis1 + lis2	Contiene gli elementi di *lis2* accodati a quelli di *lis1*
lis * n	Restituisce la lista *lis* replicata n volte
lis[2:5]	Restituisce gli elementi di posto 2, 3, 4 di *lis*
len(lis)	Restituisce il numero degli elementi di *lis*
sum(lis)	Restituisce la somma degli elementi di *lis*
max(lis), min(lis)	Restituiscono il massimo e il minimo tra gli elementi di *lis*
ele in lis	Controlla l'esistenza di *ele* in *lis*; restituisce *True* o *False*
lis1 == lis2	*lis1* è uguale a *lis2*; restituisce *True* o *False*
lis1 is lis2	*lis1* e *lis2* sono lo stesso oggetto; restituisce *True* o *False*

Metodi	Risultato
lis.append(val)	Accoda un elemento di valore *val* alla lista
lis.insert(pos,val)	Inserisce un elemento di valore *val* nel posto *pos*
lis.extend(lis1)	Accoda *lis1* a *lis*; equivale a *lis =lis + lis1*
lis.pop()	Rimuove e restituisce l'ultimo elemento di *lis*
lis.pop(k)	Rimuove e restituisce l'elemento di posto *k* di *lis* (con *k* valido)
lis.remove(val)	Rimuove l'elemento che vale *val* (ci deve essere)
lis.index(val)	Restituisce la posizione dell'elemento che vale *val* (ci deve essere)
lis.sort()	Ordina gli elementi di *lis*

La funzione *reduce*

6 Le tuple

Le **tuple** di Python sono collezioni di dati simili alle liste; differiscono da queste perché le tuple sono **immutabili**, mentre le liste non lo sono. Le tuple sono spesso usate come strutture dati simili ai *record* di molti linguaggi di programmazione, ossia una collezione di pochi dati correlati, di tipo differente, trattati unitariamente.

Una **tupla** è un elenco di dati di tipo qualsiasi identificati in base alla posizione che essi occupano nella tupla. Gli elementi che la compongono sono indicati tra parentesi tonde, separati da una virgola. L'accesso agli elementi di una tupla avviene mediante un indice posizionale, con le stesse modalità con le quali si accede alle liste. Essendo *immutabile*, non è possibile modificare un elemento di una tupla.

ESEMPIO

```
>>> nascita = (25, 'marzo', 1995)
>>> giorni = ('Lun', 'Mar', 'Mer', 'Gio', 'Ven', 'Sab', 'Dom')
>>> colori = ('rosso', 'giallo', 'blu')
>>> nascita[1]
'marzo'
>>> giorni[2:]
('Mer', 'Gio', 'Ven', 'Sab', 'Dom')
>>> giorni[2:3] + nascita
('Mer', 25, 'marzo', 1995)
>>> nascita[1] = 3
TypeError: 'tuple' object does not support item assignment
```

le tuple non possono essere modificate

Le tuple sono adatte per rappresentare i dati che non cambiano mai, come accade per una data di nascita o per i giorni della settimana; la memorizzazione di questi dati in una tupla garantisce che essi non vengano modificati erroneamente. L'accesso ai dati di una tupla è più veloce rispetto alle liste, perché le tuple sono implementate come le liste, ma con alcune ottimizzazioni che tengono conto delle loro caratteristiche. Infine, come si vedrà meglio nei prossimi paragrafi, mentre le tuple possono essere elementi di un

insieme o chiavi di un dizionario, le liste non possono esserlo: se un elenco di dati deve essere usato come elemento di un insieme, bisogna usare le tuple.

Una tupla può essere creata in diversi modi.
In particolare, la funzione **tuple()** trasforma in tupla i dati generati dalla funzione *range()* oppure i dati di una lista.
Con le tuple si possono eseguire tutte le operazioni permesse con le liste, a eccezione di quelle che le modificano. Più esplicitamente: non è possibile assegnare valori agli elementi di una tupla e non si può usare nessuno dei metodi delle liste a eccezione del metodo *index*.

ESEMPIO

```
>>> tupla = ()
>>> tupla = tuple()

>>> tupla1 = 1,2,3,4
>>> tupla1
(1, 2, 3, 4)
>>> sum(tupla1)
10
>>> len(tupla1)
4
>>> min(tupla1)
1
>>> tupla2 = (5,6,7,8,)
>>> tupla2
(5, 6, 7, 8)
>>> tupla3 = tuple(range(9,14))
>>> tupla3
(9, 10, 11, 12, 13)
>>> tupla = tupla1 + tupla2
>>> tupla
(1, 2, 3, 4, 5, 6, 7, 8)
>>> 2 in tupla1
True
>>> 2 in tupla2
False
>>> tupla.index(2)
1

>>> tupla = (5)      # non è una tupla
>>> type(tupla)
<class 'int'>
>>> tupla = (5,)     # questa è una tupla
>>> type(tupla)
<class 'tuple'>

>>> lis_vocali = ['a', 'e', 'i', 'o', 'u']
>>> tup_vocali = tuple(lis_vocali)
>>> tup_vocali
('a', 'e', 'i', 'o', 'u')
```

le parentesi non sono obbligatorie

virgola finale ammessa

tuple(*x*) costruisce una tupla con i contenuti di *x*

tuple(*lista*) costruisce una tupla

NOTA BENE

La virgola dopo l'ultimo elemento è *obbligatoria* per le tuple con un solo elemento. Inoltre, per rappresentare una tupla, la virgola dopo l'ultimo elemento è *raccomandata* per evitare errori nell'accodare manualmente gli elementi di una tupla a un'altra con operazioni di copia/incolla.


```
>>> lista = list(tupla)
>>> lista
[1, 2, 3, 4, 5, 6, 7, 8]

>>> x, y, z = 12, -5, 4
>>> (x, y, z)
(12, -5, 4)

>>> tup_vocali.index('b')
ValueError: tuple.index(x): x not in tuple
```

list(*tupla*) costruisce una lista

assegnazione a una tupla

Il metodo *index* segnala un errore se l'elemento da ricercare non appartiene alla tupla. Le tuple, come le liste, non hanno un metodo *find*, analogo a quello delle stringhe, che restituisce -1 se l'elemento di cui si cerca la posizione non appartiene all'elenco. Si può supplire a questa mancanza costruendo una funzione *find* come quella del seguente esempio, che può essere usata sia con liste che con le tuple.

ESEMPIO

```
>>> def find(item, elenco):
        if item in elenco:
            return elenco.index(item)
        else:
            return -1

>>> find('b', tup_vocali)
-1
>>> find('e', tup_vocali)
1
>>> find('u', lis_vocali)
4
```

L'assegnazione *x, y = y, x* ha come effetto lo *scambio* di valori tra *x* e *y*.

ESEMPIO

```
>>> x, y = 25, 'Dicembre'
>>> x, y
(25, 'Dicembre')
>>> x, y = y, x
>>> x, y
('Dicembre', 25)
```

scambio di *x* con *y*

NOTA BENE

Gli effetti dell'istruzione *x,y = y,x* sono spiegati facilmente se l'istruzione viene analizzata e compresa per quello che è, cioè l'assegnazione tra tuple *(x,y)= (y,x)*. Come in ogni istruzione di assegnazione, la tupla *(y,x)*, a destra del segno di uguale, è l'espressione da valutare e assegnare alla tupla *(x,y)* a sinistra del segno di uguale. L'assegnazione va letta come: *(x,y)=('Dicembre', 25)*.

La tupla restituita da una funzione

Quando una funzione restituisce più valori restituisce, di fatto, una tupla. Questo accade nel caso di una funzione che restituisce una data acquisita da tastiera.

ESEMPIO

```
>>> def leggi_data():
        gg, mm, aa = input("Data (gg-mm-aaaa): ").split('-')
        return gg, mm, aa
```

La funzione *leggi_data* è usata per acquisire una data che viene memorizzata nella tupla *data*.

```
>>> data = leggi_data()
Data (gg-mm-aaaa): 18-10-2019
>>> data
('18', '10', '2019')
>>> type(data)
<class 'tuple'>
```

In alternativa i valori restituiti da *leggi_data* sono memorizzati nelle tre variabili: *giorno, mese, anno*.

```
>>> giorno, mese, anno = leggi_data()
Data (gg-mm-aaaa): 18-10-2019
>>> giorno, mese, anno
('18', '10', '2019')
```

Bisogna porre attenzione al fatto che *data* e la terna *giorno, mese, anno* non sono esattamente la stessa cosa: *data* è una tupla a tutti gli effetti, quindi *immutabile*, mentre la terna è un aggregato di variabili separatamente modificabili, come si può notare cercando di trasformare le stringhe che compongono la data in numeri interi:

```
>>> data[0] = int(data[0])
TypeError: 'tuple' object does not support item assignment
>>> giorno = int(giorno)
>>> giorno
18
>>> giorno, mese, anno
(18, '10', '2019')
```

NOTA BENE

La funzione *leggi_data* restituisce una tupla, anche se la stringa letta è stata scomposta nelle tre variabili *gg, mm, aa* con il metodo *split* che restituisce una lista.

```
gg, mm, aa = input("Data (gg-mm-aaaa): ").split('-')
```

Python ha eseguito un cambiamento di tipo **implicito**, restituendo un oggetto della classe *tuple* anziché una lista. Il comando effettivamente eseguito è:

```
gg, mm, aa = tuple(input("Data (gg-mm-aaaa): ").split('-'))
```


CODING

1 QUANTI GIORNI MANCANO?

Calcolare i giorni che separano due date

- Per ciascuna data inserita si calcola il giorno progressivo a partire dal primo gennaio 1900. La differenza dei giorni progressivi corrispondenti alle due date è il numero dei giorni che le separano.
  Per calcolare il giorno progressivo dell'anno a partire dal mese e dal giorno del mese viene definito un array di dodici elementi, ciascuno dei quali contiene il numero dei giorni trascorsi prima del mese che forma la data, mentre quest'ultimo è utilizzato come indice per accedere all'array.
  L'array descritto è un **array costante** ed è definito con una tupla nel seguente modo:

  TRASCORSI = (0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334)

- Per calcolare i giorni trascorsi fino all'inizio dell'anno indicato nella data, in prima approssimazione si moltiplicano i 365 giorni dell'anno per la differenza tra l'anno e 1900. Al risultato del prodotto vanno aggiunti tanti giorni quanti sono gli anni bisestili intercorsi prima dell'anno indicato. Poiché gli anni bisestili sono quelli che sono divisibili per quattro, il numero dei giorni da aggiungere è dato dalla divisione intera che ha come dividendo il numero degli anni –1 e come divisore il numero 4.
  Per l'anno della data va aggiunto un giorno solo se l'anno è bisestile e il mese è successivo a febbraio.
  Si rammenti che l'anno 1900 non è bisestile, mentre l'anno 2000 lo è (gli anni divisibili per 100 non sono bisestili, a meno che siano divisibili per 400). Questi vincoli sono rispettati nelle formule di calcolo.

- **Programma Python** (*DistanzaDate.py*)

```
# DistanzaDate.py: calcolo dei giorni che separano due date
#                  tupla con i giorni trascorsi
#                  dal 1 gennaio al primo di ogni mese

TRASCORSI = (0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334)

def leggi_data():
    gg, mm, aa = input("(gg-mm-aaaa): ").split('-')
    return int(gg), int(mm), int(aa)

# progressivo dal 1 gennaio 1900
# 1 gennaio 1900: progressivo = 1
def calcolo(giorno, mese, anno):
    anni = anno - 1900
    if anno == 1900:
        progr = TRASCORSI[mese-1] + giorno
    else:
        progr = anni * 365 + (anni - 1) // 4 + TRASCORSI[mese-1] + giorno
        if mese > 2 and (anno % 4) == 0:
            progr +=1
    return progr

# funzione principale
def main():
    # lettura delle due date
    print("Prima data", end = ' ')
    giorno1, mese1, anno1 = leggi_data()
    print("Seconda data", end = ' ')
    giorno2, mese2, anno2 = leggi_data()
```