Python

In questa lezione:

- I tipi strutturati
- Liste
- Tuple
- Insiemi
- Dizionari
- Tabelle e Matrici
- File

I tipi strutturati

Nella programmazione è spesso necessario utilizzare combinazioni di dati da trattare in modo unitario. Si pensi alle coordinate di un punto nel piano: si tratta di una coppia valori reali. Oppure ai nomi degli studenti di una classe: è un insieme di stringhe.

Per questo sono state introdotto il concetto di "struttura dati" (struttura per la memorizzazione dei dati) che combina alcuni dati in un'entità unitaria, in modo tale da poterli considerare un oggetto unico.

Una stringa può essere considerata una struttura dati che rappresenta un testo sotto forma di sequenza di caratteri.

Il Python offre strutture dati molto avanzate come le liste, gli insiemi e i dizionari.

Liste

Una lista è una sequenza di elementi, ciascuno dei quali è associato ad un indice che ne specifica la posizione.

In Python si denota una lista (tipo list) con una sequenza di valori separati di virgole e delimitati da una coppia di parentesi quadre:

```
[1934, 1938, 1982, 2006]  # lista di numeri interi
["rosa", "raso", "orsa"]  # lista di stringhe
[ ]                       # lista vuota
```

Le liste possono essere memorizzate in variabili per potervi accedere in seguito:

```
frase = ['Era', 'una', 'notte', 'bulla', 'e', 'tempestosa']
voti = [18, 26, 30, 24, 27]
temperature = [16.0, 17.2, 18.1, 19.5, 20.7, 20.3, 19.4, 17.9, 15.6]
```


Liste

Accesso agli elementi

Per accedere ad un elemento di una lista si specifica l'indice dell'elemento facendo uso dell'operatore di indicizzazione [ ].
Gli indici sono valori interi che numerano gli elementi partendo da 0.

```
print (frase[0])    # stampa 'Era'
print (voti[2])     # stampa 30
print (frase[3])    # stampa 'bulla'
```

Al contrario delle stringhe, che sono immutabili, cioè non è possibile modificarne i caratteri, le liste sono *mutabili*. Si può sostituire un elemento con un altro:

```
frase[3] = 'buia'
voti[0] = 30
```

Liste

Scansione di una lista

Il numero degli elementi di una lista è fornito dalla funzione `len`.

```
esamiSostenuti = len(voti) # esamiSostenuti vale 5

print ( len(temperature) ) # stampa 9
```

Per fare una scansione della lista (cioè accedere a tutti gli elementi) si possono usare gli indici:

```
for i in range(9) :         # i assume tutti i valori tra 0 e 8
  print(i, temperature[i]) # stampa dell'indice e del valore associato

for i in range( len(temperature) ) : # stesso comportamento di sopra
  print(i, temperature[i])
```

oppure, se non interessa l'indice, con un ciclo sugli elementi:

```
for elemento in temperature : # elemento assume tutti i valori
  print( elemento )           # stampa ogni elemento
```

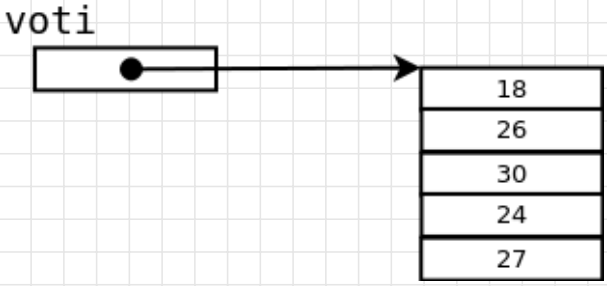

Liste

Riferimenti

Una variabile di tipo `list` come `voti` in realtà non memorizzana nessuno dei valori della lista, ma contiene un riferimento alla posizione in memoria della lista. In Python questo aspetto diventa rilevante quando si copiano i riferimenti alle liste.

```
# la variabile voti è un riferimento

voti = [18, 26, 30, 24, 27]
```



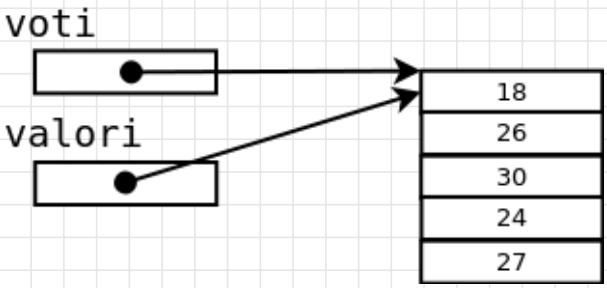
Liste

Riferimenti

In Python in generale non è necessario pensare che una variabile sia un riferimento ad una lista e non una lista in sé, ma questo aspetto diventa rilevante quando si copiano variabili di tipo `list`.

```
voti = [18, 26, 30, 24, 27]

valori = voti  # voti e valori sono la stessa lista!
```



Di fatto in caso di assegnazione vengono copiati i riferimenti (e non i valori presenti nella lista).

Liste

Metodi per liste

In Python le liste sono oggetti, e quindi, come le stringhe, hanno un comportamento definito da metodi. Vediamone alcuni per aggiungere elementi a fine lista, inserire elementi, cancellare elementi, restituire un indice e ordinare tutti gli elementi.

```
articoli = []              # lista vuota!

articoli.append("il")      # ["il"]
articoli.append("la")      # ["il", "la"]

articoli.insert(1, "lo") # ["il","lo", "la"] inserimento in posizione 1

articoli.pop(2)            # ["il","lo"] rimosso elemento in posizione 2

articoli.remove("il")      # ["lo"]  rimozione senza sapere la posizione

tutti = ["il","lo", "la", "i","gli", "le"]

a = tutti.index("gli")     # a vale 4

tutti.sort()               # ["gli", "i", "il", "la", "le", "lo"]
```

Liste

Operatori per liste

Le liste hanno gli operatori di concatenazione, ripetizione, sottolista (*slice*) e confronto come le stringhe. Esiste anche l'operatore booleano `in` che verifica se un elemento appartiene ad una lista

```
singolari = ["il","lo", "la"]
plurali =   ["i","gli", "le"]

tutti = singolari + plurali # tutti è ["il","lo", "la", "i","gli", "le"]

plurali_maschili = tutti[3:5] # plurali_maschili vale ["i","gli"]

molti_uno = [1] * 10  #[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

if "la" in singolari:
     print("la", "è un articolo singolare")
```

Gli operatori di confronto sono `==`, `!=`, `<=`, `>=`, `<`, `>`. Il confronto è lessicografico e gli elementi corrispondenti devono essere confrontabili.

Liste

Funzioni per liste

Oltre la funzione `len` esistono altre funzioni utili che operano su liste di numeri come `sum`, `min`, `max`:

```
valori = [22, 11, 34, 17, 52 , 26, 13]

print ( sum( valori ) )    # stampa 175
print ( min( valori ) )    # stampa 11
print ( max( valori ) )    # stampa 52
```

Si può usare `list` per creare una nuova lista da una stringa o da un'altra lista o sequenza di valori.

```
c = list("ciao")           # ["c", "i", "a", "o"]

copia = list( c )          # anche copia vale ["c", "i", "a", "o"]
                           # ma è un'altra lista

pari = list(range(0,10,2)) # pari vale [0, 2, 4, 6, 8]
```

Liste

Algoritmi per liste

Alcuni algoritmi sulle liste sono di particolare importanza: la ricerca lineare di un valore, la raccolta o il conteggio di particolari elementi, l'eliminazione di particolari elementi. Per capire il funzionamento di alcuni metodi o funzioni su liste, si propongono i seguenti esercizi (vedi libro di testo).
Date due liste non vuote `li` e `ls`, rispettivamente di interi e di stringhe, proporre gli algoritmi e quindi i programmi Python per i seguenti problemi:

- trovare il massimo elemento in `li` (senza usare la funzione `max`);
- trovare la stringa più lunga in `ls`;
- stampare tutti gli elementi pari di `li`;
- raccogliere in una lista tutti gli elementi minori di 10 di `li`;
- fornire una lista con le lunghezze delle stringhe in `ls`;
- cancellare tutte le stringhe più lunghe di 6 caratteri da `ls`;
- controllare se in `li` ci sono elementi ripetuti;
- fare il prodotto di tutti gli elementi in `li`.

Tuple

Il Python permette di rappresentare sequenze immutabili di elementi mediante il tipo di dato `tuple`. Una tupla è molto simile ad una lista, ma il suo contenuto non può essere modificato.
Una tupla si denota con una sequenza di valori separati da virgola tra una coppia di parentesi tonde (non necessarie se non c'è ambiguità sul tipo di dato). Tutti gli operatori e funzioni su liste che non modificano gli elementi possono essere usati per le tuple.

```
parole = ("Noli", "me", "tangere")

cifre = 1, 7, 2, 9

print( cifre )               # stampa (1, 7, 2, 9)

print( parole.index("me") ) # stampa 1

print( cifre[1:3] )          # stampa (7, 2)

print( sum(cifre) )          # stampa 19
```

Tuple

Usi delle tuple

Le tuple vengono usate per specificare i valori di formato di visualizzazione di stringhe:

```
a = 10
b = 99
print("I naturali tra %d e %d hanno due cifre" % (a, b) )
```

oppure per assegnare valori a variabili con un unico enunciato:

```
(numero, mese) = (4, "aprile") # numero vale 4 e mese vale "aprile"

numero, mese = 4, "aprile"     # idem
```

Vedremo che le tuple sono utili per le funzioni che hanno un numero variabile di parametri o che devono restituire più di un valore.

Insiemi

Un insieme è una raccolta di valori univoci, cioè senza duplicati. Gli elementi non sono memorizzati secondo un ordine e quindi non possono essere indicizzati.

In Python un insieme (`set`) si denota specificando i suoi elementi tra parentesi graffe. Si può usare la funzione `set` per creare l'insieme vuoto e per convertire altri tipi in un insieme.

```
semi = { "denari", "coppe", "spade", "bastoni" }

ingegneri = set()                 # insieme vuoto

a = [ "C++", "Java", "Python" ] # lista

linguaggi = set( a )              # insieme
```

Insiemi

Scansione, aggiunta e rimozione di elementi

Possiamo usare il ciclo `for` per accedere a tutti gli elementi dell'insieme.

```
Satelliti = {"Io", "Europa", "Ganimede", "Callisto"}

for luna in Satelliti :
   print( luna )
```

Il metodo `add` aggiunge un elemento. Per la rimozione usare `discard` (consigliato) o `remove`. Quest'ultimo solleva un'eccezione se l'elemento non è presente nell'insieme. Il metodo `clear` elimina tutti gli elementi.

```
linguaggi = {"C++", "Java", "Python", "Ruby", "Lisp"}

linguaggi.add("Pascal")

linguaggi.discard("Python")

linguaggi.remove("Prolog")      # Eccezione!

linguaggi.clear()               # insieme vuoto
```


Insiemi

Operazioni

Le operazioni di unione, intersezione e differenza tra insiemi si effettuano tramite i metodi `union`, `intersection`, `difference`.
Si può determinare se un insieme è sottinsieme di un altro tramite il metodo `issubset`.

```
pari = { 0, 2, 4, 6, 8 }

dispari = { 1, 3, 5, 7, 9 }

primi = { 2, 3, 5, 7}

cifre = pari.union( dispari )          # { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

primi_pari = pari.intersection( primi )          # { 2 }

dispari_non_primi = dispari.difference( primi ) # { 1, 9 }

print( primi.issubset(cifre) )  #stampa True
```

Insiemi

Funzioni ed operatori

La funzione `len` restituisce il numero degli elementi dell'insieme.
L'operatore `in` determina se un elemento è nell'insieme.
Il confronto tra due insieme si può fare con gli operatori `==` e `!=`.

```
print( len( primi ) )  # stampa 4

if 2 in primi :
   print ( "2 è primo" )

if {2} == primi_pari :
   print ( "2 è primo pari" )

if primi != dispari :
   print ( "i dispari non sono tutti primi" )
```


Insiemi

Algoritmi per gli insiemi

Utilizzando gli insiemi risolvere i seguenti esercizi

- Data una lista, eliminare gli elementi che appaiono due volte o più.
- Dati due insiemi calcolare la differenza simmetrica, cioè l'insieme degli elementi presenti solo in uno dei due insiemi
- Chiedendo all'utente un insieme di numeri naturali, creare l'insieme dei quadrati Q e quello dei cubi C dei numeri dati.
- Dati gli insiemi Q e C dell'esercizio precedente, questi sono disgiunti? scrivere un programma per verificare la risposta.
- Chiedere all'utente una frase e verificare se contiene tutte le lettere dell'alfabeto.
- Scrivere un progamma che costruisca gli insiemi X dei quadrati minori di 1000 e l'insieme Y dei cubi minori di 1000 e verifichi se esiste un naturale il cui predecessore è un quadrato e il cui successore è un cubo.

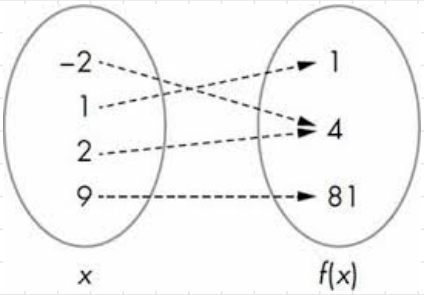
Dizionari

Premessa

Una funzione su un dominio discreto può essere vista come una serie di coppie e rappresentata in forma tabellare.
Il primo valore della coppia appartiene al dominio il secondo al codominio della funzione.

Temperature	
Chieti	21
L'Aquila	18
Pescara	22
Teramo	20

Quadrati	
-2	4
1	1
2	4
9	81



Dizionari

Premessa

Una lista può essere vista come una funzione tra il dominio degli indici e i valori degli elementi.
Per esempio:

```
"""
Lista di vocali
"""

vocali = ['a', 'e', 'i', 'o', 'u']
```

vocali	
0	'a'
1	'e'
2	'i'
3	'o'
4	'u'

Le funzioni Temperature e Quadrati degli esempi precedenti non possono essere rappresentati con liste perché il dominio non è quello degli indici.

Per superare questo limite, Python introduce i dizionari.

Dizionari

Un dizionario è un insieme di coppie il cui primo elemento si chiama chiave e il secondo valore. Le chiavi sono univoche.

In Python un dizionario (`dict`) è denotato da una sequenza di coppie chiave/valore tra parentesi graffe, dove ciascuna chiave è separata dal relativo valore dal carattere ":".
Esempio:

```
Temperature = { "Chieti":21, "L'Aquila":18, "Pescara":22, "Teramo":20 }

Quadrati = { -2: 4, 1: 1, 2: 4, 9: 81 }

Colori_preferiti = { "Felice": "verde",
                     "Tata": "giallo",
                   "Lucia": "verde",
                 "Antonio": "marrone" }

Vuoto = { }  # dizionario vuoto, non insieme vuoto!
```


Dizionari

Accesso agli elementi e copia

Per accedere agli elementi di un dizionario si può utilizzare l'operatore di indicizzazione `[ ]`.
Esempio:

```
print( "Gradi a Chieti:", Temperature["Chieti"] )

print( "Tata preferisce il", Colori_preferiti["Tata"] )
```

Si può copiare un dizionario usando la funzione dict.
Esempio:

```
Temperature = { "Chieti":21, "L'Aquila":18, "Pescara":22, "Teramo":20 }

Alias = Temperature       #  I riferimenti Alias e Temperature si
                          #  riferiscono allo stesso dizionario!

Copia = dict(Temperature) # Copia dei valori
```

Dizionari

Modificare e aggiungere elementi

Per modificare elementi si può utilizzare l'operatore di indicizzazione `[ ]`.
Esempio:

```
Colori_preferiti["Antonio"] = "blu"

Temperature["L'Aquila"] = 17
```

Se si usa l'operatore di indicizzazione come sopra, ma la chiave non esiste allora un nuovo elemento viene creato con quella chiave e con il valore indicato:
Esempio:

```
Colori_preferiti["Mario"] = "rosso"    # ora Colori_preferiti vale
                                       # { "Felice": "verde",
                                       #     "Tata": "giallo",
                                       #    "Lucia": "verde",
                                       #  "Antonio": "blu",
                                       #    "Mario": "rosso"}
```


Dizionari

Funzioni, operatori e metodi

Il numero di coppie per un dizionario è dato dalla funzione `len`. Due dizionari possono essere confrontati con gli operatori `==` e `!=`. L'operatore booleano `in` determina se una *chiave* è nel dizionario. Il metodo `pop` elimina una coppia data una chiave, mentre il metodo values restituisce tutti i valori.
Esempio:

```
print( len( Temperature ) ) # stampa 4

if Copia != Temperature :   # sono diversi perché cambiata
     print("diversi")       # la temperatura di L'Aquila

if 2 in Quadrati :          # stampa "presente"
     print("presente")

Temperature.pop("Chieti")
print( len( Temperature ) ) # stampa 3

print(list( Quadrati.values() ) ) # stampa [1, 4, 4, 81]
```

Dizionari

Un metodo particolare

Può accadere di non sapere se una chiave è già presente in un dizionario. Se si accedesse ad un dizionario tramite l'operatore di indicizzazione con una chiave non presente, l'interprete Python solleverebbe un'eccezione.

Si potrebbe utilizzare l'operatore `in` preventivamente per vedere se la chiave è presente, ma più semplicemente si può utilizzare il metodo `get` a cui si passa la chiave e un valore da restituire se la chiave non è presente. Esempio:

```
parole = ["casa", "dolce", "casa"]

conteggio = {} # dizionario con le occorrenze di ogni parola

for p in parole :
  conteggio[p] = conteggio.get(p,0) + 1

print(conteggio)   # stampa {"dolce": 1, "casa": 2}
```


Dizionari

Scansione di un dizionario

Per scandire gli elementi di un dizionario in base alle chiavi si può utilizzare il ciclo `for` .

Esempio:

```
for nome in Colori_preferiti :
   print (nome, "preferisce il", Colori_preferiti[nome] )
```

Se si vogliono scandire tutte le coppie si può usare il ciclo `for` insieme al metodo `items` che restituisce tutte le coppie come tuple.

```
for item in Colori_preferiti.items() :        # come sopra
   print (item[0], "preferisce il", item[1] )
```

Dizionari

Algoritmi per i dizionari

Utilizzando i dizionari risolvere i seguenti esercizi

- Chiedere all'utente una lista di nomi di persone e poi, mostrando un nome la volta, chiedere l'età relativa alla persona con quel nome. Creare un dizionario E con coppie nome/età.
- Dato il dizionario E dell'esercizio precedente, trovare l'età media, la persona con l'età minima e quella con l'età massima.
- Creare un dizionario D le cui chiavi sono i naturali fino a 100 e ogni valore è una lista contenente il quadrato, il cubo, e la quarta potenza della chiave.
- Stampare il dizionario D sotto forma di tabella in cui ogni riga contiene la chiave e i valori della lista associata.
- Dato il dizionario D, chiedere all'utente un numero n e dire se n è un quadrato, cubo o quarta potenza di un numero minore di 100.

Tabelle e Matrici

Una tabella (o matrice) è una disposizione di valori costituita da righe e colonne.
Useremo il termine matrice quando i valori sono numerici

Esempi: possibili configurazioni di 3 bit, tris e quadrato magico.

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

'X'	'O'	'X'
' '	'X'	'O'
'O'	' '	' '

7	12	1	14
2	13	8	11
16	3	10	5
9	6	15	4

Tabelle e Matrici

Il Python non ha un tipo di dato specifico per le tabelle, ma si può creare una struttura bidimensionale utilizzando liste di liste, cioè si può creare una lista di righe, che sono a loro volta liste di valori.

```
bits = [
    [ 0, 0, 0 ],
    [ 0, 0, 1 ],
    [ 0, 1, 0 ],
    [ 0, 1, 1 ],
    [ 1, 0, 0 ],
    [ 1, 0, 0 ],
    [ 1, 0, 0 ],
    [ 1, 0, 0 ]
 ]
```

```
tris = [
    ['X', 'O', 'X'],
    [' ', 'X', 'O'],
    ['O', ' ', ' ']
 ]

magico = [
    [ 7, 12,  1, 14],
    [ 2, 13,  8, 11],
    [16,  3, 10,  5],
    [ 9,  6, 15,  4]
 ]
```


Tabelle e Matrici

Creazione

Se però la tabella è molto grande si deve crearla a partire dalla tabella vuota inserendo una riga alla volta.

Esempio di una matrice 50×100 di valori pari a zero.

```
RIGHE = 50
COLONNE = 100

matrice = []

for i in range(RIGHE) :

    riga = [0] * COLONNE

    matrice.append(riga)
```

Tabelle e Matrici

Accesso agli elementi

Per accedere agli elementi di un tabella possiamo usare l'operatore di indicizzazione `[ ]`, ricordando che questa è una lista di liste.

Data la tabella `magico`, allora la prima riga sarà `magico[0]` e il primo valore della prima riga `magico[0][0]`.

```
magico = [
    [ 7, 12,  1, 14],
    [ 2, 13,  8, 11],
    [16,  3, 10,  5],
    [ 9,  6, 15,  4]
 ]

print(magico[0])        # stampa [7, 12, 1, 14]
print(magico[0][0])     # stampa 7
print(magico[0][2])     # stampa 1
print(magico[2])        # stampa [16, 3, 10, 5]
print(magico[2][1])     # stampa 3
```


Tabelle e Matrici

Scansione

Per scandire i valori di una tabella possiamo ricorrere a due cicli `for` annidati, come nell'esempio.

```
magico = [
    [ 7, 12,  1, 14],
    [ 2, 13,  8, 11],
    [16,  3, 10,  5],
    [ 9,  6, 15,  4]
 ]

for i in range( len(magico) ) :          # len(magico): numero di righe
  for j in range ( len(magico[i]) ) :    # len(magico[i]): num. di colonne
      print( magico[i][j], end = " " ) # stampa i valori di una riga
  print()                                # va a capo per la prossima riga
```

Matrici come array bidimensionali

Python è un linguaggio molto utilizzato in ambito tecnico e scientifico grazie a librerie come SciPy (algebra lineare, integrazioni, interpolazioni, FFT, processamento di segnali e immagini) e SymPy (calcolo simbolico). Alla base delle librerie per il calcolo numerico c'è NumPy che fornisce matrici n-dimensionali dette array.

Si rimanda lo studente allo studio di questa libreria (non in programma).

```
import numpy
from numpy.linalg import solve, inv

a = numpy.array([[1, 2, 3],
                 [6, 5, 4],
                 [8, 7, 9]])

t = a.transpose()          # t è la trasposta di a

c = a + b                  # somma di matrici

i= inv(a)                  # i è l'inversa di a

b = numpy.array([3, 2, 1]) # b è un vettore

d = solve(a, b)            # risolve l'equazione ax = b
```


Tabelle e Matrici

Algoritmi per Tabelle e Matrici

Utilizzando le tabelle risolvere i seguenti esercizi:

- Creare una tabella che rappresenti una scacchiera. Le caselle vuote sono rappresentate dal carattere spazio, mentre 'P', 'R', 'D', 'A', 'C' e 'T' rappresentano pedone, re, donna, alfiere, cavallo e torre per i bianchi. Usare le minuscole per i neri.
- Creare una matrice quadrata M 10×10 che rappresenti la tabellina pitagorica.
- Creare una matrice quadrata N 10×10 per cui alla riga i-esima e alla colonna j-esima sia memorizzato il valore `i - j`.
- Scrivere un programma che faccia la trasposta di N, calcoli la somma e il prodotto di N ed M.
- Scrivere un programma che dato un vettore di n elementi (n chiesto all'utente) costituito da un 1 seguito da tutti 0, costruisca una tabella T $n \times n$ che abbia come prima riga il vettore, prima colonna tutti 1 e come valore `T[i][j]` il valore `T[i-1][j-1]+ T[i-1][j]`, per `i`≥ 1 e `j`≥ 1.
- Difficile: riprodurre il gioco Life (vita) di Conway (e.g., https://bitstorm.org/gameoflife/).

File

Nell'elaborazione di dati reali è necessario leggere e scrivere file. Come abbiamo vsito, i file sono sequenze finite di byte che risiedono in memoria secondaria.

Nel caso in cui i byte codificano caratteri, parliamo di file di testo e quindi le operazioni di lettura e scrittura riguardano stringhe. Se i byte non codificano caratteri, parliamo di file binari e le operazioni di lettura e scrittura riguardano i byte stessi.

Benché il Python permette di trattare entrambi i tipi di file, ci limiteremo ai file di testo.
Si può fare accesso ai file in modalità lettura, se vogliamo leggere i dati, o in modalità scrittura se vogliamo scrivere dati.

File

Possiamo accedere ad un file tramite la funzione `open` usata normalmente con due parametri: il nome del file e la modalità di accesso. La funzione restituisce un riferimento ad oggetto di tipo file.

```
f1 = open("mio_file.txt","w")  # crea il file o lo sovrascrive se esiste

f1.write("Mattina\n\n")        # scrive una riga e ne lascia una vuota
f1.write("M'illumino\n")
f1.write("d'immenso.\n\n")
f1.write("    Giuseppe Ungaretti")

f1.close()                     # tutti i dati sono memorizzati nel file
```

Le possibili modalità di accesso ai file sono:

- `w` : per scrivere (o sovrascrivere se esiste) un file
- `r` : per leggere un file
- `a` : per aggiungere nuovi dati alla fine del file
- `r+` : per leggere e scrivere

Se il parametro di modalità viene omesso si assume la modalità `r`.

File

Scrittura

Come abbiamo visto, per scrivere in un file si usa il metodo `write`.

Al contrario della funzione `print`, che aggiunge il carattere di nuova riga automaticamente, quando si scrive su file bisogna esplicitamente inserire il carattere "`\n`".

Per questioni di efficienza, il metodo `write` scrive in un buffer della memoria centrale. Man mano che il buffer si riempie i dati vengono tasferiti su file.

Per assicurarsi che tutti i dati del buffer siano trasferiti nel file bisogna, al termine della scrittura, utilizzare il metodo `close`.

File

Lettura e scansione

Per leggere una riga di un file di testo si usa il metodo `readline`.
I seguenti programmi svolgono lo stesso compito di stampare il contenuto di un file di testo.

```
f1 = open("mio_file.txt","r") # apre il file in lettura

line = f1.readline()          # legge una riga
while line != "":             # la stringa vuota indica la fine del file
    print(line, end="")       # stampa la riga (line contiene già \n )
    line = f1.readline()

f1.close()                    # chiude il file
```

Per scandire tutte le righe di un file si può utilizzare il ciclo `for`:

```
f1 = open("mio_file.txt","r") # apre il file in lettura

for line in f1:               # per tutte le righe del file f1
    print(line, end="")       # stampa la riga

f1.close()                    # chiude il file
```

File

Lettura dell'intero file

Un file testo può essere visto come un'unica stringa.
Il metodo `read` utilizzato senza parametri legge l'intero file.

```
f1 = open("mio_file.txt","r") # apre il file in lettura

tutto = f1.read()             # legge tutto il file

print(tutto)                  # stampa tutto il file

f1.close()                    # chiude il file
```

Il metodo `read` può prendere come parametro il numero di caratteri da leggere. Se è 1 legge un singolo carattere:

```
f1 = open("mio_file.txt","r") # apre il file in lettura

char = f1.read(1)             # legge un carattere
while char != "":             # la stringa vuota indica la fine del file
    print(char, end="")       # stampa il carattere
    char = f1.read(1)

f1.close()                    # chiude il file
```


File

Lettura di righe: metodi utili del tipo stringa

Supponiamo di aver letto una riga di un file ed averla memorizzata nella stringa `s`. Come accedere alle singole parole?
Il metodo `split` restituisce la lista di parole: in realtà stringhe separate da spazi (o da un separatore passato per parametro).

```
s = " Il mattino ha l'oro in bocca. \n"

a = s.split()   #a vale ['Il', 'mattino', 'ha', "l'oro", 'in', 'bocca.']

b = s.split("'")  #b vale [' Il mattino ha l', 'oro in bocca. \n']
```

Per togliere i caratteri indesiderati all'inizio o alla fine della stringa (come i caratteri di punteggiatura) si può usare il metodo `strip` che usato senza parametri toglie i caratteri di spaziatura, altrimenti quelli indicati.

```
c = s.strip()         #c vale "Il mattino ha l'oro in bocca."

d = s.strip(" .\n")  #d vale "Il mattino ha l'oro in bocca"
```

Studiare le varianti `lsplit, rsplit, lstrip, rstrip` e l'uso dei parametri.

File

Algoritmi per File

Utilizzando i file di testo risolvere i seguenti esercizi:

- Realizzare una funzione che, ricevendo per parametro il nome di un file, restituisca una tripla con il numero di linee, il numero di parole e il numero dei caratteri presenti nel file.
- Realizzare una funzione che, ricevendo come parametri i nomi di un file di input ed uno di output, numeri le linee del file di input e le scriva nel file di output.
- Realizzare una funzione che conti le occorrenze di ogni parola presente in un file il cui nome è dato come parametro e le scriva su un altro file il cui nome è anch'esso dato come parametro.
- Un file contiene per ogni riga il nome di una persona e una cifra (numero intero) in euro rappresentante un versamento. Un nome di una persona può ripetersi più volte. Realizzare un programma che per ogni nome di persona stampi la somma dei versamenti di quella persona.