INDICE



# le LISTE (list, mutable, iterable)

Una lista e' un insieme ordinato di elementi che in generale sono eterogenei. Il termine 'ordinato' sta a dire che c'e' un primo oggetto, un secondo oggetto etc. etc. e non riflette necessariamente l'ordinamento degli elementi nella lista secondo un criterio.

In [27]:
```
lista_interi = [1,5,-2] # questa e' una lista di interi
lista_interi
```

Out[27]:
```
[1, 5, -2]
```

In [28]:
```
lista_varie = [1,"ciao",True] #questa e' una lista che contiene elementi di diverso tipo
lista_varie
```

Out[28]:
```
[1, 'ciao', True]
```

Le liste possono contenere davvero qualsiasi tipo di oggetto:

In [29]:
```
import math
l=[math , math.sin ,type(25.)]
#una lista che contiene un modulo (l[0]), una funzione (l[1]) ed un tipo di dati (l[2])
l
```

Out[29]:
```
[<module 'math' (built-in)>, <function math.sin>, builtins.float]
```

# Creare le liste

## Per creare una lista vuota:

In [30]:
```
#per creare una lista vuota
l1 = []
l2 = list()
print(l1,l2)
```

```
[] []
```

## la funzione list

Oltre a poter creare una lista vuota (vedi sopra) ...

### copia di una stringa con list

In [31]:
```
#per creare una lista e riempirla
l1 = [1 , 'ciao' , 10.6]
l2 = list(l1) #creo una copia della lista passata come argomento
print(l1,l2)
```

```
[1, 'ciao', 10.6] [1, 'ciao', 10.6]
```

più in generale, la funzione list restituisce una lista a partire da altri tipi di sequenze:

### lista dei caratteri di una stringa con list

In [32]:
```
l=list('stringa')
print(l)
```

```
['s', 't', 'r', 'i', 'n', 'g', 'a']
```

# I range e le liste di interi

I range sono oggetti diversi dalle liste che sono usati direttamente quando vogliamo 'iterare' su sequenze di interi (solo un accenno l'iterazione ed il ciclo for li approfondiremo in seguito).

In [33]:
```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [34]:
```
#non e' una lista!!!
type(range(10))
```

Out[34]:
```
builtins.range
```

Inoltre, sono usati molto spesso insieme alla funzione list per generare liste di interi. Vediamo come.

### creare una lista di interi da 0 a n, n escluso

In [35]:
```
n=10
list(range(n))
```

Out[35]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### creare una lista di interi da ni a nf, ni incluso, nf escluso

In [36]:
```
ni=3
nf=10
list(range(ni,nf))
```

Out[36]:
```
[3, 4, 5, 6, 7, 8, 9]
```

### creare una lista di interi da ni a nf a passi di ns, ni incluso, nf escluso

In [37]:
```
ni=3
nf=12
ns=3
list(range(ni,nf,ns))
```

Out[37]:
```
[3, 6, 9]
```

## Esempi di funzioni che operano su liste

Ci sono diverse funzioni che operano su liste, ad esempio la funzione (len) determina il numero di elementi di una lista. Queste funzioni richiedono che si indichi come argomento la lista di cui si vogliono contare gli elementi.

### len

In [38]:
```
len(l)
```

Out[38]:
```
7
```

### sorted

La funzione sorted ordina gli elementi di una lista (in senso decrescente se si imposta a True il parametro bool reversed).

```
In [39]: sorted([2,5,9,1,34])
```

```
Out[39]: [1, 2, 5, 9, 34]
```

parametro reversed

```
In [120]: sorted([2,5,9,1,34],reverse=True)
```

```
Out[120]: [34, 9, 5, 2, 1]
```

Le funzioni che operano su liste il piu' delle volte restituiscono una copia elaborata della lista di partenza:

```
In [121]: l=[6,5,8,3,2,1]
          m=sorted(l)
          print(l)
          print(m)
```

```
[6, 5, 8, 3, 2, 1]
[1, 2, 3, 5, 6, 8]
```

parametro key

è possibile specificare un criterio di ordinamento diverso da quello standard (operatore < applicato agli elementi della lista) specificando una funzione in modo che l'ordinamento avvenga in base al valore restituito dalla funzione per cisacuno degli elementi della lista

data una lista di stringhe

```
In [138]: l=['aa','Assfsfsffssw','asdasf','SDDFhoiumn,mi','DGGuuyDSD','dSSDSD']
```

ordino la lista per lunghezza

```
In [141]: sorted(l,key=len)
```

```
Out[141]: ['aa', 'asdasf', 'dSSDSD', 'DGGuuyDSD', 'Assfsfsffssw', 'SDDFhoiumn,mi']
```

ho usato come key la funzione predefinita len

oppure la ordino in base al contenuto del secondo carattere:

```
In [142]: def mykey(s):
              #restituisco il secondo carattere
              return s[1]

          sorted(l,key=mykey)
```

```
Out[142]: ['SDDFhoiumn,mi', 'DGGuuyDSD', 'dSSDSD', 'aa', 'Assfsfsffssw', 'asdasf']
```

se voglio trascurare le maiuscole e le minuscole:

```
In [144]: def mykey(s):
              #restituisco il secondo carattere
              return s[1].lower()

          sorted(l,key=mykey)
```

```
Out[144]: ['aa', 'SDDFhoiumn,mi', 'DGGuuyDSD', 'Assfsfsffssw', 'asdasf', 'dSSDSD']
```

vediamo un altro esempio di uso di sorted + key con i numeri

vediamo un esempio: voglio ordinare una lista contenente degli angoli in gradi

```
In [ ]: #lista degli angoli tra 0 e 180
        l = list(range(0,190,10))
```

```
print(l)
```

ordinandoli secondo il valore del valore assoluto del coseno

definisco una funzione che definisce il criterio da usare per ordinare la lista

In [122]:
```
def myorderingfunc(i):
    # restituisce il valore assoluto del coseno
    # per un angolo i specificato in gradi
    return abs(cos(2*pi*i/180))
```

ordino la lista con sorted:

In [119]:
```
#lista ordinata per valore del coseno
print(sorted(l,key=myorderingfunc))
```

```
[90, 100, 80, 110, 70, 120, 60, 130, 50, 140, 40, 30, 150, 160, 20, 10, 170, 0, 180]
```

altre funzioni Built-in analoghe ...

### reversed

In [114]:
```
l=[1,23,5,8,9.099,4]
lr=reversed(l)
print(l)
print(lr)
```

```
[1, 23, 5, 8, 9.099, 4]
<list_reverseiterator object at 0x02DE0B90>
```

ooops

In [123]:
```
print(list(lr))
```

```
[4, 9.099, 8, 5, 23, 1]
```

### sum

somma gli elementi della lista:

In [127]:
```
l = list(range(1,10))
sum(l)
```

Out[127]: 45

con un valore iniziale diverso da 0 :

In [136]:
```
__builtin__.sum(l,5)
```

Out[136]: 50

## Esempi di funzioni membro delle liste

Ci sono anche funzioni intrinsecamente associate ad una specifica lista.
Si parla di funzioni membro dell'oggetto lista e costituiscono un primo elemento di programmazione ad oggetti.
Queste funzioni si invocano con il punto tra il nome della lista (prima) ed il nome della funzione (dopo).

### modifiche in-place

A differenza delle funzioni che agiscono sulle liste, le funzioni membro in genere effettuano le eventuali modifiche in-place.

Vediamo ad esempio la funzione membro `sort`

In [42]:
```
l=[6,5,8,3,2,1]
print('prima: ',l)
l.sort()
print('dopo',l)
```

```
prima:  [6, 5, 8, 3, 2, 1]
dopo [1, 2, 3, 5, 6, 8]
```

## contare elementi con count

Ad esempio la funzione `count` conta il numero di occorrenze nella lista dell'elemento indicati tra parentesi

In [43]:
```
l
```

Out[43]:
```
[1, 2, 3, 5, 6, 8]
```

In [44]:
```
l.count(1)
```

Out[44]:
```
1
```

## aggiungere elementi con append

la funzione `append` aggiunge l'elemento indicato tra parentesi in fondo alla lista:

In [45]:
```
l.append(1)
l.append(1)
l.append(1)
```

In [46]:
```
print(l)
```

```
[1, 2, 3, 5, 6, 8, 1, 1, 1]
```

In [47]:
```
l.count(1)
```

Out[47]:
```
4
```

## ordinare con sort e reverse

Oltre a sort c'è anche una `reverse`:

In [48]:
```
#per ordinare la lista oltre alla funzione builtin sorted
#possiamo usare il metodo sort o il metodo reverse della classe lista
l=[1,5,2,1,4,6,4,5]
l.sort()
print(l)
l.reverse()
print(l)
```

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

## trovare con index

In [49]:
```
#determinazione dell'indice di un elemento
l.index(2)
```

Out[49]: 5

In [50]:
```
#se ci sono più elementi con lo stesso valore
#index restituisce l'indice del primo che trova
l.index(5)
```

Out[50]: 1

## Indicizzazione degli elementi di una lista

Gli elementi contenuti nella lista sono `indicizzabili` e l'indice parte da zero
Accedo al primo elemento con l'indice 0

In [51]:
```
l = [1,5,-2]
l[0]
```

Out[51]: 1

Se cerco di accedere ad un elemento inesistente Python mi segnala un `IndexError`:

In [52]:
```
l[5]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-52-437134752604> in <module>()
----> 1 l[5]

IndexError: list index out of range
```

Accedo all'ultimo elemento della lista con l'indice -1

In [53]:
```
print(l)
l[-1]
```

```
[1, 5, -2]
```

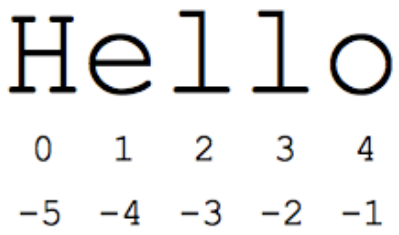
Out[53]: -2

Accedo al penultimo elemento della lista con l'indice -2

In [54]:
```
l[-2]
```

Out[54]: 5

Quanto si e' detto a proposito dell'indicizzazione degli elementi di una lista e' riassunto nella seguente figura che fa riferimento alla lista che contiene i caratteri della parola 'hello'




In [55]:
```
l=list('Hello')
l[-5]
```

Out[55]: 'H'

# Lo slicing

Consideriamo una lista che contiene i numeri interi da 10 a 200 presi a passi di 10.
per creare una lista simile (ed evitare di digitare tutti gli interi della lista) si utilizza la funzione range

In [56]:
```
l=list(range(10,200,10))
#NB:range(start, stop, step) dove stop NON viene incluso nella lista!!!!
print(l)
```

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190]
```

posso estrarre una sotto-lista, ad esempio che va dal primo elemento (indice 0) al secondo (1)

In [57]:
```
#[i-start : i-stop]
#con l'elemento con indice i-stop non incluso
l[0:2]
```

Out[57]:
```
[10, 20]
```

In [58]:
```
#dal primo (0) al sesto (5) (compreso)
l[0:6]
```

Out[58]:
```
[10, 20, 30, 40, 50, 60]
```

In [59]:
```
#dal 7mo (6) fino alla fine
l[6:]
```

Out[59]:
```
[70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190]
```

In [60]:
```
#dal 7mo (6) al penultimo
l[6:-1]
```

Out[60]:
```
[70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180]
```

In [61]:
```
#Gli ultimi sei elementi
#se non specifico il secondo elemento si intende 'FINO ALLA FINE'
l[-6:]
```

Out[61]:
```
[140, 150, 160, 170, 180, 190]
```

In [62]:
```
#Tutti gli elementi tranne gli ultimi sei
#se non specifico il primo elemento dello slice si intende 'DALL'INIZIO'
l[:-6]
```

Out[62]:
```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130]
```

In [63]:
```
#occhio a non invertire l'ordine degli indici
#altrimenti si ottiene una lista vuota
l[6:1]
```

Out[63]:
```
[]
```

In [64]:
```
#negli slice posso specificare anche un 'passo':
#dal primo al decimo elemento a passi di due
l[0:10:2]
```

Out[64]:
```
[10, 30, 50, 70, 90]
```

## slice e copia

NB: con la notazione l[:] ottengo una COPIA della lista originaria

In [65]:
```
l2 = l[:]
print(l2 == l)
print(l2 is l)
```

```
True
False
```

## Le liste sono 'mutable'

Le liste sono oggetti mutable (modificabili) i cui elementi possono cioe' essere modificati 'in-place' (senza cioe' che cambi l'id dell'oggetto):

In [66]:
```
print('l=',l)
print('id(l)=',id(l))
```

```
l= [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190]
id(l)= 48021632
```

In [67]:
```
#modifico il primo elemento
l[0]=1000000
```

In [68]:
```
print('l=',l)
print('id(l) =',id(l))
```

```
l= [1000000, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190]
id(l) = 48021632
```

Ho modificato la lista ma l'id e' rimasto lo stesso.

vediamo cosa succedeva con un oggetto di tipo NON-mutable (un intero ad esempio):

In [150]:
```
a=10000000000000222334
print('id(a)=',id(a))
a=1000222334
print('id(a)=',id(a))
a=10000000000000222334 ## lo stesso valore di 4 righe sopra
print('id(a)=',id(a))
```

```
id(a)= 48559144
id(a)= 48530128
id(a)= 48559168
```

## Onere di accesso alle liste

Il fatto che le liste possano contenere oggetti di diverso tipo rende oneroso l'accesso ai loro elementi (complessita' o(n) per l'accesso all'n-esimo elemento)

## Inserimento di elementi da una lista

In [69]:
```
l=list(range(10))
print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### append

In [70]:
```
#inserimento di un elemento in fondo alla lista con append
l.append('aa')
```

```
print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'aa']
```

### extend

In [71]:
```
#estensione di una lista con extend
l.extend(['bb','cc'])
print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']
```

### insert

In [72]:
```
#inserimento di un elemento in un generico punto della lista
l.insert(3,'nuovo')
print(l)
```

```
[0, 1, 2, 'nuovo', 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']
```

### inserimento con slices

In [73]:
```
#inserimento con slice
l[4:4]=['x','x','x','x']
print(l)
```

```
[0, 1, 2, 'nuovo', 'x', 'x', 'x', 'x', 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']
```

### sostituzione con slices

In [74]:
```
#sostituzione con slice
l[4:8]=['z','z2','z','z2']
print(l)
```

```
[0, 1, 2, 'nuovo', 'z', 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']
```

## Eliminazione di elementi da una lista

### remove

In [75]:
```
#rimozione con remove
print(l)
#la prima occorenza di un elemento passato per valore
l.remove('nuovo')
print(l)
l.remove('z')
print(l)
```

```
[0, 1, 2, 'nuovo', 'z', 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']
[0, 1, 2, 'z', 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']
[0, 1, 2, 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']
```

In [76]:
```
#se l'elemento non c'e' ...
l.remove('nuovo')
```

```
---------------------------------------------------------------------------
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-76-e1451a296661> in <module>()
      1 #se l'elemento non c'e' ...
----> 2 l.remove('nuovo')

ValueError: list.remove(x): x not in list
```

### pop

In [77]:
```
#rimozione con pop dell'ultimo elemnto della lista (pop() senza argomenti)
print(l)
print("Ho rimosso l'elemento:",l.pop())
print(l)
print("Ho rimosso l'elemento:",l.pop())
print(l)
```

```
[0, 1, 2, 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb', 'cc']
Ho rimosso l'elemento: cc
[0, 1, 2, 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa', 'bb']
Ho rimosso l'elemento: bb
[0, 1, 2, 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa']
```

In [78]:
```
#rimozione con pop dell'elemento con indice i
print(l)
print("Ho rimosso l'elemento:",l.pop(2))
print(l)
```

```
[0, 1, 2, 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa']
Ho rimosso l'elemento: 2
[0, 1, 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa']
```

NB: pop restituisce l'elemento rimosso dalla lista

In [79]:
```
#rimozione con slice
print(l)
l[2:6]=[] # cancello dal terzo al sesto elemento
print(l)
```

```
[0, 1, 'z2', 'z', 'z2', 3, 4, 5, 6, 7, 8, 9, 'aa']
[0, 1, 4, 5, 6, 7, 8, 9, 'aa']
```

### del

Posso anche usare la funzione builtin `del` che acccetta anche slice:

In [80]:
```
l=list(range(10))
print(l)
#cancello un elemento
del(l[8])
print(l)
#cancello più elementi anche non contigui
del(l[1:6:2])
print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 9]
[0, 2, 4, 6, 7, 9]
```

# PILLOLE DI SINTASSI: la proposizione for e l'iterazione su liste

Introduciamo un altro elemento di sintassi che qui applicheremo specificatamente alle liste.

# esempio: data una lista creare una nuova lista con il quadrato degli elementi della lista di partenza

## in c++

Se in un linguaggio come il C++ vogliamo eseguire ciclicamente un operazione sugli elementi di un vettore, normalmente si usa un ciclo for basato sugli indici del vettore stesso

vector.cpp

(Global Scope)

```cpp
#include <vector>
using namespace std;

int main(int argc, char* argv[])
{
    //due vettori di interi
    vector<int> v,vsq;

    //il vettore v contiene gli interi da 10 a 90 a passi di 10
    for (unsigned int i=0;i<10;i++)//NB: ciclo sugli indici
        v.push_back(10*i);

    //il vettore vsq contiene i quadrati degli elementi di v
    for (unsigned int i=0;i<v.size();i++)//NB: ciclo sugli indici
        vsq.push_back(v[i]*v[i]);

    return 0;
}
```

## In Python

### Iterazione sugli indici

In Python posso 'iterare' sugli indici (come in c):

In [81]:
```
#creo la lista di partenza
v = list(range(10,100,10))
print(v)
```

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

In [82]:
```
#creo la lista vuota che conterra' il risultato dell'iterazione
vsq = []

#ciclo sugli indici (a la C++):
for i in range(len(v)):
    #inserisco il quadrato nella lista dei risultati
    vsq.append(v[i]**2)

print(vsq)
```

```
[100, 400, 900, 1600, 2500, 3600, 4900, 6400, 8100]
```

### Iterazione sugli elementi

Ma posso iterare direttamente sugli elementi di v:

In [83]:
```
vsq=[]
#ciclo direttamente sugli elementi di v
for val in v:
    vsq.append(val**2)
print(vsq)
```

```
[100, 400, 900, 1600, 2500, 3600, 4900, 6400, 8100]
```

## Sintassi del ciclo for

Per iterare sugli elementi della lista abbiamo utilizzato il `ciclo for` che in python ha la seguente sintassi:

for in : ... istuzioni che fanno riferimento all' ... istuzioni che fanno riferimento all' #la fine del ciclo e' determinata dall'indentazione del codice!!!!!!

Nell'utilizzo interattivo di Python la prima pressione di invio dopo un istruzione all'interno di un ciclo permette di inserire una nuova istruzione all'interno del ciclo stesso.
Se premo una seconda volta invio si esce dal ciclo e si eseguono le istruzioni.

La sequenza sulla quale itero puo' essere un lista ma anche altre tipi di sequenze che vedremo in seguito (stringhe, touple ...)

In [84]:
```
for s in 'stringa':
    print(3*s)
```

```
sss
ttt
rrr
iii
nnn
ggg
aaa
```

Gli oggetti sui quali posso iterare con un ciclo for sono detti appunto `iterabili`

## Altre tecniche di iterazione

### enumerate

La funzione enumerate restituisce sia l'indice sia l'elemento della sequenza sulla quale si sta iterando:

In [85]:
```
ln=['Mario','Luigi','Anna','Paolo']
for indice,nome in enumerate(ln):
    print(indice,nome)
```

```
0 Mario
1 Luigi
2 Anna
3 Paolo
```

In [86]:
```
#vediamo cosa restituisce la funzione enumerate:
list(enumerate(ln))
```

Out[86]:
```
[(0, 'Mario'), (1, 'Luigi'), (2, 'Anna'), (3, 'Paolo')]
```

### zip

Per iterare nello stesso ciclo su `due o più sequenze` contemporaneamente, queste possono essere `accoppiate` con la funzione zip.

In [87]:
```
domande      = ['Cibo', 'Colore', 'Sport']
prime_scelte = ['bistecca', 'rosso', 'bocce']
seconde_scelte = ['lasagne', 'verde', 'parapendio']
for d, ps, ss in zip(domande, prime_scelte, seconde_scelte):
```

```
    print(d+" preferito:"+ps+" e poi "+ss)
```

```
Cibo preferito:bistecca e poi lasagne
Colore preferito:rosso e poi verde
Sport preferito:bocce e poi parapendio
```

Se le sequenze hanno lunghezze diverse l'iterazione si ferma all'ultimo elemento della sequenza più corta.

# List comprehension

La List comprehensions e' un modo conciso per creare nuove liste.

Uso tipico:

- Creare nuove liste nelle quali ciascun elemento e' il risultato di una o più operazioni effettuate sugli elementi di un'altra sequenza o iterabile.
- Creare una sottosequenza degli elementi di una sequenza che soddisfano specifiche condizioni.

Sostanzialmente si tratta di integrare un (o piu') ciclo for all'interno di una definizione di lista.

Come spesso accade un esempio vale più di molte parole ....

In [88]:
```
#riprendiamo il caso dei quadrati visto per il ciclo for
vsq=[]                      #inizializzo
for val in range(10,100,10): #itero
    vsq.append(val**2)       #aggiungo
print(vsq)
```

```
[100, 400, 900, 1600, 2500, 3600, 4900, 6400, 8100]
```

In [89]:
```
vsq = [v**2 for v in range(10,100,10)] #tutto in una botta!
print(vsq)
```

```
[100, 400, 900, 1600, 2500, 3600, 4900, 6400, 8100]
```

Sintassi:
La list comprehension consiste in parentesi quadre che contengono un'espressione seguita da una proposizione for seguita da zero o più proposizioni for o if. Il risultato sara' una nuova lista risultante dalla valutazione della espressione nel contesto delle proposizioni for ed if che la seguono.

## Applicazione di funzioni a liste con list comprehension

Un tipico utilizzo della list comprehension e' quello di applicare una o più funzioni a tutti gli elementi di una lista:

In [90]:
```
l  = [-1,-20,40,-90 , 80, -50]
print(l)
#valore assoluto
l1 = [abs(x) for x in l]
print(l1)
```

```
[-1, -20, 40, -90, 80, -50]
[1, 20, 40, 90, 80, 50]
```

## come sopra + ordinamento

In [91]:
```
#ordinamento + valore assoluto
l2 = [abs(x) for x in sorted(l)]
print(l2)
```

```
[90, 50, 20, 1, 40, 80]
```

# List comprehension nidificate

La list comprehension puo' essere nidificata (una list comprehension al posto della espressione)

In [107]:
```
#quale e' il contenuto di l2 ?
l2 = [ [i*j for i in range(1,5)] for j in range(10,60,10)]
```

Out[107]:
```
[[10, 20, 30, 40],
 [20, 40, 60, 80],
 [30, 60, 90, 120],
 [40, 80, 120, 160],
 [50, 100, 150, 200]]
```

spezziamo in due cicli for distinti:

In [110]:
```
#creo una lista vuota
l2=[]
for j in range(10,60,10):
    #creo una sotto lista vuota e la aggiungo in fondo a l2
    l2.append([])
    for i in range(1,5):
        #appendo i*j in fondo all'ultima lista aggiunta alla lista l2
        l2[-1].append(i*j)
print(l2)
```

```
[[10, 20, 30, 40], [20, 40, 60, 80], [30, 60, 90, 120], [40, 80, 120, 160], [50, 100, 150, 200]]
```

.....1 riga di codice contro 5 .....

# List comprehension condizionale (con if)

In [93]:
```
#aggiungiamo una proposizione if ed estraiamo solo i valori tra 1000 e 5000
vsq = [v**2 for v in range(10,100,10) if (v**2 > 1000.0) and (v**2 < 5000.0) ]
print(vsq)
```

```
[1600, 2500, 3600, 4900]
```

# List comprehension ed operatore ternario

Per usare una proposizione if con clausola else in una list comprehension, e' necessario usare l'`operatore ternario` che e' un costrutto if che si riconduce ad una singola espressione:

if else

esempio di utilizzo operatore ternario:

In [99]:
```
cond = 0
print('Vero') if cond else print('Falso')
```

```
Falso
```

uso di operatore ternario in una list comprehension

In [94]:
```
[(v**2 if v<5 else v**3) for v in range(1,10) ]
```

Out[94]:
```
[1, 4, 9, 16, 125, 216, 343, 512, 729]
```