Liste di liste

Abbiamo visto che, date due liste *l1*, *l2*, l'istruzione

```
l2 = l1
```

assegna a l2 lo stesso riferimento di l1 (l1 e l2 puntano alla stessa lista). Quindi modificando (la lista riferita da) l1, modifichiamo anche (la lista riferita da) l2, e viceversa. Se questo non è il comportamento desiderato, possiamo clonare l1 tramite l'operatore di slicing:

```
l2 = l1[:]
```

Ora l2 e l1 si riferiscono a due oggetti list diversi. Questo significa che **sostituendo** / **rimuovendo** un elemento di l2, l1 rimane invariata.

```
l1 = [1,1,1,[4,5,6]]
l2 = l1[:]
l2[2] = l[0] + l[1]
l2[3] = [7,8,9]
l1 # -> [1,1,1,[4,5,6]]
l2 # -> [1,1,2,[7,8,9]]
```

Questo accorgimento non è sufficiente se vogliamo anche **modificare** gli elementi di *l1*. Ciò avviene ad esempio quando si lavora con **liste di liste**

```
l1 = [[1,2,3],[4,5,6]]
l2 = l1[:]
l2[0][0] = 5
l1 # -> [[5,2,3],[4,5,6]]
l2 # -> [[5,2,3],[4,5,6]]
```

Affinchè due liste di questo tipo siano indipendenti, è necessario costruire *l2* clonando le sottoliste. Ad esempio sia

```
l1 = [[1,2],[-1,2]]
```

Una copia profonda di *l1* è

```
l2 = [l1[0][:],l1[1][:]]
```

Ora le modifiche su *l1* non si ripercuotono su *l2*, e viceversa.

Esercizio. Data la lista `l1 = [[7,8,9],[5,5,5],[1,2,3]]`, costruire e modificare *l2* a partire da *l1* in modo che

```
l1 # -> [[7,8,9],[5,5,5],[1,2,3]]
l2 # -> [[1,2,3],[7,8,9]]
```

In [1]:
```
l1 = [[7,8,9],[5,5,5],[1,2,3]]
l2 = l1[:]
del l2[1]
l2.reverse()
l1,l2
```

Out[1]:
```
([[7, 8, 9], [5, 5, 5], [1, 2, 3]], [[1, 2, 3], [7, 8, 9]])
```

Esercizio. Data la lista `l1 = [[9,8,9],[5,5,5],[1,2,-1,3]]`, costruire *l2* a partire da *l1* in modo che

```
l1 # -> [[9,8,9],[5,5,5],[1,2,-1,3]]
```

```
l2 # -> [[1,2,3],[7,8,9]]
```

In [2]:
```
l1 = [[9,8,9],[5,5,5],[1,2,-1,3]]
l2 = [l1[0][:],l1[2][:]]
l2.reverse()
del l2[0][-2]
l2[1][0] = 7
l1,l2
```

Out[2]:
```
([[9, 8, 9], [5, 5, 5], [1, 2, -1, 3]], [[1, 2, 3], [7, 8, 9]])
```

Esercizio. Data la matrice

```
A = [[1,2],[-1,2]]
```

Costruire l'inversa di A a partire da una copia profonda di A.

Nota: $A^{-1} = ((A_{11}, -A_{01}), (-A_{10}, A_{00}))$

In [4]:
```
A = [[1,2],[-1,2]]
B = [A[0][:],A[1][:]]
B[0][0] = A[1][1]
B[0][1] = -A[0][1]
B[1][0] = -A[1][0]
B[1][1] = A[0][0]
A,B
```

Out[4]:
```
([[1, 2], [-1, 2]], [[2, -2], [1, 1]])
```

Tuple e interpolazione di stringhe

In Python è possibile generare stringhe che contengono valori di variabili con un'apposita sintassi:

si segnalano le occorrenze di una variabile con marcatori del tipo "%_" e si elencano le variabili in coda alla stringa.

Ad esempio

```
year = 1986
album = "Master of Puppets"
artist = "Metallica"
"Album: %s - Artist: %s - (%d)"%(album,artist,year)
```

stampa

```
'Album: Master of Puppets - Artist: Metallica (1986)'
```

Alcuni marcatori utili

- %s - inserisce la conversione a str di una variabile
- %f - conversione a float di una variabile.
- %d - conversione a int di una variabile.

I marcatori di formattazione possono presentare dei parametri opzionali del tipo `%x.y_`, dove

- x è il minimo numero di caratteri
- y è il numero di caratteri/cifre significative

Esempi.

```
a = 7; "%4.3d"%(a) # stampa ' 007'
b = 1.4142; "%4.2d"%(b) # stampa '1.41'
```

Esercizio. Data la tupla

```
t = ([1,2,3], 3.1415, "arrivederci")
```

quali delle seguenti istruzioni producono errori? Perché?

```
a. t[-1] = "goodbye" # errore, le tuple sono immutabili
b. t[-1][0] = '!' # errore, le stringhe sono immutabili
c. t[0][0] = 5 # OK

d. "%s %s %s" % (t[0],t[1],t[2]) # Ok
e. "%s %d %8.4s" % t # Ok
f. "%s %.2f %s" % t # Ok
g. "%s %d %d" % t # Errore, impossibile convertire una string in int
h. "%d %d %s" % t # Errore, impossibile convertire una list in int
```

Metodi su str

- capitalize(), lower(), upper()

```
"waRnIng".upper() # -> "WARNING"
```

- find(), replace(), count()

```
'Python progr*mming is gre*t!'.count('*') # -> 2
'Python progr*mming is gre*t!'.replace('*','a') # -
> 'Python programming is great!'
```

Esercizio facoltativo. Operare su

```
s = "quaste non a' une fresA"
```

in modo da ottenere

```
s # -> "Questa non e' una frase"
```

Hint. Metodi da utilizzare: `lower`,`replace (3 volte)`,`capitalize`

Da str a list e viceversa

- conversione str -> list. La conversione di un oggetto str *s* a list produce una lista contenente i caratteri di *s*:

```
list("gli spazi contano")
# -> ['g','l','i',' ','s','p','a','z','i',' ','c','o','n','t','a','n','o']
```

Questo ci consente di lavorare sui singoli caratteri di un testo.

Se invece vogliamo operare sulle singole parole di un testo possiamo usare il metodo *split()*

- join(), split(). I metodi join e split permettono rispettivamente di passare da una lista a un testo e da un testo ad una lista.

```
"don't | mind | the | vertical bars".split('|')
# -> ["don't ", ' mind ', ' the ', ' vertical bars']
''.join(["don't ", ' mind ', ' the ', ' vertical bars'])
# -> "don't  mind  the  vertical bars"
'*'.join(['a','word','list'])
# -> 'a*word*list'
```

Metodi su list

- reverse(). Inverte l'ordine degli elementi di una lista

```
l1 = [3,2,1]
l1.reverse()
l1 # -> [1,2,3]
```

- append(). Aggiunge **un** elemento ad una lista

```
l1.append(4)
l1 # -> [1,2,3,4]
```

- extend(). Aggiunge ad una lista gli elementi di un'altra lista

```
l1.extend([5,6,7])
l1 # -> [1,2,3,4,5,6,7]
```

Esercizio facoltativo. Verificare il seguente palindromo (a meno di spazi)

```
s = "i topi non avevano nipoti"
```

Esercizio. dati

```
s = "long a what remember never I"
l = ['sentence','I made','up']
```

operare su l e s con l'ausilio dei metodi di cui sopra in modo da ottenere

```
s # -> "I | never | remember | what | a | long | sentence | I made | up"
```

Hint. Metodi utili: `split`,`reverse`,`extend`,`join`

In [12]:
```
s = "long a what remember never I"
l = ['sentence','I made','up']
l1 = s.split()
l1.reverse()
s = l1 + l
s = " | ".join(s)
s
```

Out[12]:
```
'I | never | remember | what | a | long | sentence | I made | up'
```

Operazioni sugli iterabili

- Appartenenza. Dato un oggetto iterabile (list/str/set/ecc.) *it*, è possibile verificare l'appartenenza di un oggetto *obj* con il costrutto

```
obj in it
```

Il costrutto restituisce True se *obj* si trova in *it*, False altrimenti.

- Data una lista *l*, capita spesso di voler effettuare la stessa operazione su ogni elemento di *l*.

  Python offre un potente costrutto per svolgere operazioni di questo tipo, denominato *List Comprehensions*.

  In generale, data una lista *l* lunga *N* e una funzione *f*, possiamo ottenere la lista `[f(l[0]),f(l[1]),...,f(l[N])]` dall'istruzione

```
[f(x) for x in l]
```

  Esempio. Supponiamo di voler calcolare il quadrato di una lista di numeri.

```
numbers = [2,3,5,7,11,13]
squares = [i**2 for i in numbers]
squares # -> [4, 9, 25, 49, 121, 169]
```

Esercizio. Data

```
s = "Una serie di sfortunati eventi"
```

- costruire una lista i cui elementi rappresentino il numero di lettere di ogni parola di *s* (hint: utilizzare la funzione *len*)
- costruire una lista i cui elementi indichino (con True/False) se le vocali `['a','e','i','o','u']` compaiono o meno in *s*
- costruire una lista i cui elementi rappresentino il numero di occorrenze delle vocali `['a','e','i','o','u']`

In [13]:
```
s = "Una serie di sfortunati eventi"
[len(word) for word in s.split()]
```

Out[13]:
```
[3, 5, 2, 10, 6]
```

In [14]:
```
vowels = ['a','e','i','o','u']
[v in s for v in vowels]
```

Out[14]:
```
[True, True, True, True, True]
```

In [16]:
```
[s.lower().count(v) for v in vowels]
```

Out[16]:
```
[2, 4, 4, 1, 2]
```

Esercizio. Data la lista di coordinate

```
l = [(1,5),(5,2),(3,9),(1,-3)]
```

- per ogni punto, calcolare la somma delle coordinate;
- calcolare il quadrato della distanza dei punti `l[i]` dal punto `P(1,1)`;
- calcolare le coordinate dei punti in un sistema di riferimento con origine `O'(3,-1)`

In [18]:
```
l = [(1,5),(5,2),(3,9),(1,-3)]
[sum(coord) for coord in l]
```

Out[18]:
```
[6, 7, 12, -2]
```

In [19]:
```
P = (1,1)
[(x-P[0])**2 + (y-P[1])**2 for x,y in l]
```

Out[19]:
```
[16, 17, 68, 16]
```

In [21]:
```
O = (3,-1)
[(x-O[0],y-O[1]) for x,y in l]
```

Out[21]:
```
[(-2, 6), (2, 3), (0, 10), (-2, -2)]
```

Esercizio. (Cifrario di Cesare) A lezione avete visto che è possibile applicare il cifrario di Cesare ad una singola parola *word* con l'istruzione

```
"".join([chr(ord('a') + (ord(ch) - ord('a') + key)%26) for ch in word])
```

Provate ad estendere il procedimento a un'intera frase innestando due descrittori di lista

Passo 1. Assegnare alla variabile *message* una frase di senso compiuto a piacere

Passo 2. Codificare il messaggio secondo lo schema del cifrario di Cesare con chiave `key = 3` ed assegnare il codice cifrato alla variabile `code`.

Passo 3. Verificare la validità della codifica decodificando `code`

Osservazione. I passi 2,3 sono identici a meno del valore di `key`

In [42]:
```
key = 3
messaggio = "Non tutti gli aforismi matematici hanno senso"
messaggio = messaggio.lower()
code = [[chr(ord('a') + (ord(ch) - ord('a') + key)%26) for ch in word] for
        word in messaggio.split()]
code
```

Out[42]:
```
[['q', 'r', 'q'],
 ['w', 'x', 'w', 'w', 'l'],
 ['j', 'o', 'l'],
 ['d', 'i', 'r', 'u', 'l', 'v', 'p', 'l'],
 ['p', 'd', 'w', 'h', 'p', 'd', 'w', 'l', 'f', 'l'],
 ['k', 'd', 'q', 'q', 'r'],
 ['v', 'h', 'q', 'v', 'r']]
```

In [36]:
```
code = ["".join(l) for l in code]
code
```

Out[36]:
```
['qrq', 'wxwwl', 'jol', 'dirulvpl', 'pdwhpdwlfl', 'kdqqr', 'vhqvr']
```

In [37]:
```
code = " ".join(code)
code # messaggio cifrato
# la decifratura segue lo stesso schema ma con key = - 3
```

Out[37]:
```
'qrq wxwwl jol dirulvpl pdwhpdwlfl kdqqr vhqvr'
```

La funzione zip

Date due liste della stessa lunghezza, `l1,l2`, la funzione zip() restituisce la trasposta della matrice di righe `l1,l2`

```
l1 = [1,2,3]
```

```
l2 = [3,4,5]
zip(l1,l2) # -> [(1, 3), (2, 4), (3, 5)]
```

La funzione zip torna utile unita all'impiego di list comprehension.

Supponiamo ad esempio di voler valutare una funzione $f : R^2 \to R$ in un insieme di punti di cui sono note ascisse e ordinate

```
x = (1,2,3,4,5) # ascisse di punti
y = (6,7,8,9,10) # ordinate di punti
points = zip(x,y)
[f(p[0],p[1]) for p in points] # oppure
[f(X,Y) for X,Y in points]
```

Esercizio. Dati

```
bases = (1,2,3,4,5)
exponents = (6,7,8,9,10)
```

calcolare `bases[i]**exponents[i]` per i = 0,...,4

In [39]:
```
bases = (1,2,3,4,5)
exponents = (6,7,8,9,10)
pairs = zip(bases,exponents)
pairs
```

Out[39]:
```
[(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
```

In [40]:
```
[x**y for x,y in pairs]
```

Out[40]:
```
[1, 128, 6561, 262144, 9765625]
```

Esercizi di riepilogo (facoltativi)

Esercizio. Determinare numericamente il minimo della funzione $f(x) = 3x^2 - 4x$

sul dominio `[0,1[`, valutando la funzione nei punti `[0,0.01,0.02,...,0.99]`

Hint. Per costruire il dominio usate la funzione *range()* e i descrittori di lista. Usate la funzione *min* per determinare il minimo

Esercizio.

- Si costruiscano le liste `name`, `surname`, `year_of_birth`, contenenti nome, cognome, ed anno di nascita dei vostri 5 attori preferiti.
- si utilizzi la funzione zip per raccogliere i dati relativi ad un attore in tuple
- si utilizzino i descrittori di lista e l'interpolazione su string per ottenere una lista del tipo

```
["Daniel Craig ha 44 anni", "Javier Bardem ha 43 anni",...]
```

Esercizio.

Passo 1. Costruire due stringhe s1,s2 con caratteri a piacere. Ad esempio

```
s1 = 'NS]qt@>tCrrzjllhg@WdLIO^OVxnCb`GeMDcluXiQdh]mSx@=kADiSrVRlSNupLvWqAfoqMtyCXDcGA'
s2 = 'iivtn^ot_fWXYkTNGjR_rt]]CsrjwsqOO_fzjCIszZW[qfjaz`xHcUHHxtDKlDHuUjBa]'
```

Passo 2. Determinare la lista dei caratteri (senza ripetizioni) che compaiono in entrambi gli oggetti e ordinarle in ordine alfabetico.

Esempio:

```
"hello" -> ['e', 'h', 'l', 'o']
```

Passo 3: determinare il numero di *parole* di lunghezza 6 a partire dall'alfabeto ricavato al passo 1

Hint. set(), sorted()