

## Liste

### Descrizione

- Una lista è una **sequenza** di oggetti qualunque (anche di tipo diverso, anche altre liste)

```
>>> l = ["abc", 2, [1,2]]
```

- Essendo una sequenza, condivide le operazioni su sequenza viste per le stringhe

```
>>> len(l)
3
```

- La lista è un tipo **mutabile**: può essere allungata ed accorciata, e si possono modificare i suoi elementi

```
>>> l[2] = 10
>>> l
['abc', 2, 10]
```

### Operazioni su liste

#### Operatori

```
>>> l = [1,2,3]
>>> l + ["a","b"]
[1, 2, 3, 'a', 'b']
```

```
>>> l + [] # [] indica una lista vuota
[1, 2, 3]
```

```
>>> l * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> 3 in l
True
>>> [3,1] in l # vale solo per singoli elementi
False
```

### Operazioni su liste

#### Indicizzazione e sottolista

```
>>> l = [1,2,3,4,5,6]
>>> l[2]
3
>>> l[-1]
6
>>> l[:]
[1, 2, 3, 4, 5, 6]
>>> l[3:]
[4, 5, 6]
>>> l[:-2]
[1, 2, 3, 4]
```

## Operazioni su liste

### Matrici

- Una lista i cui elementi siano tutte liste implementa una *matrice*
- Si possono recuperare tramite uno o due indici righe o singoli elementi della matrice

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> matrix[2]
[7, 8, 9]
>>> matrix[2][0]
7
>>> matrix[2][0:2]
[7, 8]
```

### Modifica di liste

#### Modifica con assegnazione ad indici

- Essendo oggetti mutevoli, è possibile modificare il contenuto di una lista

```
>>> l = [1,2,3,4,5,6]
>>> p = l
>>> l[2] = "a"
>>> l
[1, 2, 'a', 4, 5, 6]
>>> p
[1, 2, 'a', 4, 5, 6]
```

- Poiché l'oggetto cambia, tutte le variabili che vi si riferiscono "vedono" il cambiamento

### Modifica di liste

#### Modifica con assegnazione a sottoliste

- E' possibile sostituire intere sottoliste di una certa lista

```
>>> l = [1,2,3,4,5,6]
>>> l[2:5] = ["a","b"]
>>> l
[1, 2, 'a', 'b', 6]
```

- Non è necessario sostituire la sottolista con una della stessa lunghezza, può essere più lunga o corta
- E' possibile eliminare intere sottoliste sostituendole con la lista vuota

```
>>> l[2:4] = []
>>> l
[1, 2, 6]
```

## Modifica di liste

### Modifica con assegnazione a sottoliste

- E' possibile sostituire sottoliste con sottoliste provenienti da pezzi diversi (anche in sovrapposizione) della lista stessa. In tal caso prima viene estratta la sottolista da inserire, poi sostituita a quella da cancellare

```
>>> l = [1,2,3,4,5,6]
>>> l[2:5] = l[4:6]
>>> l
[1, 2, 5, 6, 6]
```

## Modifica di liste

### Modifica tramite il comando del

- Il comando del permette di cancellare elementi o sottoliste specificando posizione o gamma di posizioni

```
>>> l = [1,2,3,4,5,6]
>>> del l[2]
>>> l
[1, 2, 4, 5, 6]
>>> del l[-1]
>>> l
[1, 2, 4, 5]
>>> del l[2:4]
>>> l
[1, 2]
>>> del l[:]
>>> l
[]
```

## Metodi di lista

### Metodi di modifica

- Come i tipi stringa, anche le liste hanno una serie di metodi da eseguire
- Essendo oggetti mutabili, molti di questi metodi modificano direttamente l'oggetto

**append** aggiunge un elemento in fondo alla lista

```
>>> l = [1,2,3,4,5,6]
>>> l.append(7)
>>> l
[1, 2, 3, 4, 5, 6, 7]
```

**extend** estende la lista attaccando in fondo tutti gli elementi di una lista

```
>>> l.extend([8,9])
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## Metodi di lista

### Metodi di modifica

**insert** Inserisce un elemento prima di una certa posizione

```
>>> l = [1,2,3,4,5,6]
>>> l.insert(3,3.5)
>>> l
[1, 2, 3, 3.5, 4, 5, 6]
```

**pop** Elimina un elemento (l'ultimo o quello a posizione specificata) dalla lista e lo restituisce in uscita

```
>>> l.pop()
6
>>> l
[1, 2, 3, 3.5, 4, 5]
>>> l.pop(3)
3.5
>>> l
[1, 2, 3, 4, 5]
```

## Metodi di lista

### Metodi di modifica

**sort** Ordina la lista (assume l'esistenza di una funzione di ordinamento tra elementi, si usa per liste di elementi dello stesso tipo)

```
>>> l = ["basso", "medio", "alto"]
>>> l.sort()
>>> l
['alto', 'basso', 'medio']
```

**reverse** Riordina la lista dall'ultimo al primo elemento

```
>>> l.reverse()
>>> l
['medio', 'basso', 'alto']
```

## Documentazione

### dir e help

- E' sempre possibile visualizzare l'elenco dei metodi di lista disponibili con la funzione `dir`

```
>>> dir([])
[....., 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

- E recuperare informazioni su uno specifico metodo con la funzione `help`

```
>>> help([].remove)
Help on built-in function remove:

remove(...)
    L.remove(value) -- remove first occurrence
                       of value
```

## List comprehensions

### Operazioni sugli elementi di una lista

- Il costrutto nominato *list comprehension* è un tipo di espressione caratteristica del Python che permette di creare una lista come risultato di un'operazione da fare sugli elementi di un'altra lista

```
>>> L = [0,1,2,3,4]
>>> [i+1 for i in L]
[1, 2, 3, 4, 5]
```

- Il costrutto è composto da:
  - Un oggetto di tipo sequenza su cui iterare (e.g. la lista L)
  - Una variabile (o più, vedremo) che raccolga di volta in volta gli elementi (e.g. i)
  - Una espressione che faccia qualche operazione che coinvolga l'elemento cui la variabile si riferisce (e.g. i+1)
  - Le parole riservate `for` ed `in`, e le parentesi quadre a delimitazione

## List comprehensions

### Operazioni sugli elementi di una lista

- Ad esempio può essere usato per estrarre una colonna da una matrice:

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> [ row[2] for row in matrix]
[3, 6, 9]
```

- L'operazione sugli elementi può anche essere una qualsiasi funzione o metodo compatibile con il loro tipo

```
>>> [ str.upper() for str in ["a","b","c"]]
['A', 'B', 'C']
```

- E' possibile selezionare gli elementi della lista da processare tramite una condizione (parola chiave `if`)

```
>>> [i for i in [1,3,4,5,6,2,8,9] if i % 2 != 0]
[1, 3, 5, 9] # solo numeri dispari
```

## List comprehensions

### Operazioni sugli elementi di una lista

- Il costrutto può essere usato con oggetti che siano *sequenze*, non solo con liste

```
>>> [ r for r in "AHHDCCEDEGGTA" if r in "HC"]
['H', 'H', 'C', 'C']
```

- Può ovviamente essere combinato con metodi o funzioni che processino liste

```
>>> "".join([ r for r in "AHHDCCEDEGGTA"
... if r in "HC"])
'HHCC'
```

- E' uno dei costrutti più utili del linguaggio, ne vedremo molti esempi

## Dizionari

### Descrizione

- Un dizionario è una collezione *non ordinata* di oggetti indirizzati tramite *chiavi* (invece che per posizione come nelle mappe)

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
```

- Un dizionario è quindi una **mappa** da chiavi a valori, e i suoi oggetti si recuperano specificandone la chiave corrispondente

```
>>> D["medio"]
1
```

- Essendo una collezione non ordinata, gli oggetti non manterranno l'ordine con cui sono stati inseriti, ma verranno riarrangiati in maniera casuale per velocizzare le operazioni di ricerca e recupero.

```
>>> D
{'alto': 2, 'basso': 0, 'medio': 1}
```

## Dizionari

### Creazione

- Un dizionario può essere creato specificandone l'insieme di coppie chiave:valore separate da virgole e delimitato da parentesi graffe

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
```

- Le chiavi di un dizionario non devono necessariamente essere stringhe, basta che siano oggetti *immutabili* (e.g. numeri o tuple). E' così possibile creare ad esempio vettori o array multidimensionali *sparsi* (con pochi indici con valore specificato, assumendo ad es. un valore di default per gli altri)

```
>>> D = {10 : 2.3, 50 : 1.5, 223 : -3}
>>> D = {(-1,3.2,4) : "rosso",
... (5,4.5,8) : "verde" ,
... (0,5,-4.4) : "blu"}
```

## Dizionari

### Ricerca

- Verifica della presenza di una certa chiave (operatore `in` o metodo `has_key`)

```
>>> "alto" in D
True
>>> "altissimo" in D
False
>>> D.has_key("alto") # uso del metodo has_key
True
```

- metodo `get`: recupero di un oggetto specificando chiave e valore di default da restituire se non presente (None se non specificato)

```
>>> D.get("alto")
2
>>> D.get("altissimo") # None non viene stampato
>>> D.get("altissimo", "alto")
'alto'
```

## Dizionari

### Metodi di esplorazione

**len** restituisce il numero di elementi (coppie chiave,valore) contenuti nel dizionario)

```
>>> D = {"a" : 0, "b" : 1, "c" : 2}
>>> len(D)
3
```

**keys** restituisce la lista delle chiavi contenute nel dizionario

```
>>> D.keys()
['a', 'c', 'b']
```

**values** restituisce la lista dei valori contenuti nel dizionario

```
>>> D.values()
[0, 2, 1]
```

**items** restituisce la lista delle coppie chiave,valore (come tuple) contenute nel dizionario

```
>>> D.items()
[('a', 0), ('c', 2), ('b', 1)]
```

## Modifica di dizionari

### Oggetti mutabili

- Il modo più semplice per recuperare un oggetto è accedendovi per chiave (come si farebbe in una lista con l'indice)

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
>>> D["alto"]
2
```

- Come le liste, i dizionari sono oggetti *mutabili*. E' quindi modificare un oggetto associato ad una certa chiave

```
>>> D["alto"] = 3
>>> D
{'alto': 3, 'basso': 0, 'medio': 1}
```

## Modifica di dizionari

### Oggetti mutabili

- A differenza delle liste se si assegna un valore ad una chiave non presente, la coppia chiave:valore viene aggiunta al dizionario:

```
>>> D["altissimo"] = 3
>>> D
{'alto': 3, 'basso': 0, 'altissimo': 3, 'medio': 1}
```

- In una lista, l'assegnazione di un valore ad un indice fuori dimensione genera invece un errore (si deve usare `append`)

```
>>> L = ["a", "b", "c"]
>>> L[3] = "d"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

## Modifica di dizionari

### Modifica tramite il comando `del`

- In modo analogo alle liste, il comando `del` permette di cancellare elementi (ma non sottoinsiemi di elementi) specificandone la chiave

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
>>> del D["medio"]
>>> D
{'alto': 2, 'basso': 0}
```

### Comando `del`

#### Rimozione di variabili

- Il comando `del` applicato ad una variabile la elimina, ma non cancella l'oggetto a cui si riferisce

```
>>> D = {"basso" : 0, "medio" : 1, "alto" : 2}
>>> D2 = D
>>> del D
>>> D
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'D' is not defined
>>> D2
{'alto': 2, 'basso': 0, 'medio': 1}
```

## Modifica di dizionari

### Metodi di modifica

**pop** elimina l'elemento specificato (tramite la chiave) e ne restituisce il valore

```
>>> D = {"basso" : 0, "medio" : 1,
... "alto" : 2}
>>> D.pop("basso")
0
>>> D
{'alto': 2, 'medio': 1}
```

**update** concatena al dizionario un dizionario fornito come argomento

```
>>> D = {"a" : 0, "b" : 1, "c" : 2}
>>> D2 = {"d" : 3, "e" : 4}
>>> D.update(D2)
>>> D
{'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3}
```

## I dizionari sono mappe NON sequenze

### Le operazioni su sequenza non funzionano

- Le operazioni che presuppongono un ordinamento degli elementi non funzionano nei dizionari:
- concatenazione o ripetizione

```
>>> D = {"a" : 0, "b" : 1, "c" : 2}
>>> D2 = {"d" : 3, "e" : 4}
>>> D + D2
>>> D + D2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
'dict' and 'dict'
```

- estrazione o modifica di sottodizionari

```
>>> D["a":"b"] = {}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type
```

## Documentazione

### dir e help

- Al solito, `dir` permette di visualizzare l'elenco dei metodi di dizionario disponibili

```
>>> dir({})
[....., 'clear', 'copy', 'fromkeys',
'get', 'has_key', 'items', 'iteritems',
'iterkeys', 'itervalues', 'keys', 'pop',
'popitem', 'setdefault', 'update', 'values']
```

- e informazioni su uno specifico metodo si possono recuperare con la funzione `help`

```
>>> help({}.fromkeys)
Help on built-in function fromkeys:

fromkeys(...)
    dict.fromkeys(S[,v]) -> New dict with keys from S
                        and values equal to v.
                        v defaults to None.
```

## Tuple

### Descrizione

- Una tupla è una collezione ordinata di oggetti (racchiusi da parentesi tonde)

```
>>> t = (1,-5,4)
```

- Come stringhe e liste, è un tipo *sequenza*, accessibile per indice e su cui funzionano le operazioni definite su sequenze

```
>>> t[2]
4
```

- Come le stringhe, è *immutabile* e non è possibile modificarla

```
>>> t[2] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
        assignment
```

## Operazioni su tuple

### Operatori

```
>>> t = (1, -5, 4)
>>> t + (1, 2)
(1, -5, 4, 1, 2)
>>> t + ()
(1, -5, 4)
>>> t * 2
(1, -5, 4, 1, -5, 4)
```

## Operazioni su tuple

### Tuple con un solo elemento

```
>>> t = (1, -5, 4)
>>> t + (1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple
        (not "int") to tuple
>>> t + (1,)
(1, -5, 4, 1)
```

### Nota

- Anche un'espressione può essere racchiusa tra parentesi
- Per distinguere una tupla con un solo elemento da un'espressione, va aggiunta una virgola dopo l'elemento (e.g. (1,))

## Operazioni su tuple

### Indicizzazione e sottotupla

```

>>> t = (1, 2, 3, 4, 5, 6)
>>> t[2]
3
>>> t[-1]
6
>>> t[:]
(1, 2, 3, 4, 5, 6)
>>> t[3:]
(4, 5, 6)
>>> t[:-2]
(1, 2, 3, 4)

```

## Immutabilità di tuple

### Oggetti immutabili

- Come le stringhe, le tuple sono oggetti immutabili

```

>>> t = (12, "abc", [1,2,3])
>>> t[2] = "f"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
                                assignment

```

- Tale immutabilità non si applica ai contenuti della tupla, per cui i suoi oggetti possono essere modificati *se mutabili*

```

>>> t[2].append(4)
>>> t
(12, 'abc', [1, 2, 3, 4])
>>> t[1][2] = "d"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
                                assignment

```

## Conversioni

### Conversioni tra tupla e lista

- Se si desidera modificare i contenuti di una tupla, è possibile:
  1. create una lista che abbia come elementi gli elementi della tupla (funzione `list(t)`)
  2. fare le necessarie modifiche sulla lista
  3. creare una tupla che abbia come elementi gli elementi della lista (funzione `tuple(l)`)
  4. eventualmente assegnare la nuova tupla alla variabile che si riferiva alla tupla originaria

## Conversioni

### Esempio di conversione

```
>>> t = ("a","b","c","d")
>>> l = list(t)
>>> l
['a', 'b', 'c', 'd']
>>> l.reverse()
>>> l
['d', 'c', 'b', 'a']
>>> t = tuple(l)
>>> t
('d', 'c', 'b', 'a')
```

### Nota

- La tupla originaria ("a", "b", "c", "d") NON è stata modificata (essendo un oggetto immutabile)
- Abbiamo semplicemente riassegnato la variabile `t` ad un'altra tupla

## Perché le tuple?

### Utilità delle tuple

- Le tuple sono semplicemente liste immutabili
- Non hanno quindi nessun metodo (ma possono essere manipolate per creare nuove tuple con gli operatori)
- La loro utilità è data dal fatto di essere immutabili: possono essere passate tra parti diverse di un programma essendo sicuri che non verranno modificate (e.g. in una certa posizione ci sarà sempre lo stesso oggetto)
- Possono essere usate dove c'è bisogno di oggetti immutabili, ad esempio come indici di dizionari

## Perché le tuple?

### Utilità delle tuple

- Le tuple sono spesso usate per semplificare le operazioni di assegnazione a più variabili

```
>>> (x, y, z) = (1.2, 3, 4.5)
>>> y
3
```

- Dove non ambiguo, e' possibile specificare una tupla anche senza parentesi, semplificando ulteriormente la notazione

```
>>> x, y, z = 1.2, 3, 4.5
>>> y, z
(3, 4.5)
```

## Perché le tuple?

### Utilità delle tuple

- Vari metodi e funzioni restituiscono tuple in uscita, ad esempio il metodo `items` dei dizionari

```
>>> D = {"a" : "alpha", "b" : "beta",  
... "g" : "gamma"}  
>>> (k,v) = D.items()[0]  
>>> k,v  
( 'a', 'alpha' )
```

- Il costrutto `list comprehension` può gestire tuple di variabili invece di variabili singole:

```
>>> [ "%s => %s" % (k,v) for (k,v) in  
... D.items() ]  
['a => alpha', 'b => beta', 'g => gamma']
```