Liste, tuple, dizionari

1. Liste
2. Tuple
3. Dizionari

Liste

In Python, le **liste** sono una delle strutture dati più versatili e utilizzate. Permettono di memorizzare una collezione ordinata di elementi, che possono essere di tipi diversi (interi, float, stringhe, altre liste, ecc.). Le liste sono **mutabili**, il che significa che è possibile modificarle dopo la loro creazione (aggiungere, rimuovere o cambiare elementi).

Come creare una lista:
Le liste vengono create racchiudendo gli elementi tra parentesi quadre [], separati da virgole.

```
# Lista di numeri interi
numeri = [1, 2, 3, 4, 5]

# Lista di stringhe
nomi = ["Alice", "Bob", "Charlie"]

# Lista mista di diversi tipi di dati
mista = [1, "ciao", 3.14, True]

# Lista vuota
vuota = []
```

Accesso agli elementi:
Gli elementi di una lista sono accessibili tramite il loro **indice**, che inizia da 0 per il primo elemento.

```
nomi = ["Alice", "Bob", "Charlie"]
print(nomi[0])  # Output: Alice
print(nomi[1])  # Output: Bob
print(nomi[2])  # Output: Charlie

# È possibile usare anche indici negativi per accedere agli elementi
dalla fine
print(nomi[-1]) # Output: Charlie (l'ultimo elemento)
print(nomi[-2]) # Output: Bob (il penultimo elemento)
```

Slicing delle liste:
Lo slicing permette di estrarre una sottosequenza di elementi da una lista. La sintassi è lista[inizio:fine:passo].

```
numeri = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numeri[2:5])   # Output: [2, 3, 4] (elementi dall'indice 2 incluso
all'indice 5 escluso)
print(numeri[:4])    # Output: [0, 1, 2, 3] (elementi dall'inizio fino
all'indice 4 escluso)
print(numeri[6:])    # Output: [6, 7, 8, 9] (elementi dall'indice 6 fino
alla fine)
print(numeri[::2])   # Output: [0, 2, 4, 6, 8] (tutti gli elementi con
passo 2)
print(numeri[::-1])  # Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] (lista
invertita)
```

Metodi delle liste:
Python offre diversi metodi built-in per manipolare le liste. Ecco alcuni dei più comuni:

- **append(elemento)**: Aggiunge un elemento alla fine della lista.
```
frutta = ["mela", "banana"]
frutta.append("arancia")
print(frutta)  # Output: ['mela', 'banana', 'arancia']
```

- **extend(iterable)**: Aggiunge tutti gli elementi di un iterable (come un'altra lista) alla fine della lista.
```
lista1 = [1, 2]
lista2 = [3, 4, 5]
lista1.extend(lista2)
print(lista1)  # Output: [1, 2, 3, 4, 5]
```

- **insert(indice, elemento)**: Inserisce un elemento in una posizione specifica indicata dall'indice.
```
colori = ["rosso", "giallo"]
colori.insert(1, "verde")
print(colori)  # Output: ['rosso', 'verde', 'giallo']
```

- **remove(elemento)**: Rimuove la prima occorrenza dell'elemento specificato dalla lista. Se l'elemento non è presente, solleva un errore ValueError.
```
animali = ["cane", "gatto", "topo", "gatto"]
animali.remove("gatto")
print(animali)  # Output: ['cane', 'topo', 'gatto']
```

- **pop(indice)**: Rimuove e restituisce l'elemento presente all'indice specificato. Se l'indice non è fornito, rimuove e restituisce l'ultimo elemento della lista.
```
numeri = [10, 20, 30, 40]
elemento_rimosso = numeri.pop(1)
print(elemento_rimosso)  # Output: 20
print(numeri)            # Output: [10, 30, 40]

ultimo_elemento = numeri.pop()
print(ultimo_elemento)  # Output: 40
print(numeri)           # Output: [10, 30]
```

- **index(elemento, inizio, fine)**: Restituisce l'indice della prima occorrenza dell'elemento nella lista (o nella sottosequenza specificata da inizio e fine). Se l'elemento non è trovato, solleva un errore ValueError. Gli argomenti inizio e fine sono opzionali.
```
lettere = ["a", "b", "c", "b", "d"]
indice_b = lettere.index("b")
print(indice_b)  # Output: 1

indice_b_dopo_uno = lettere.index("b", 2)
print(indice_b_dopo_uno) # Output: 3
```

- **count(elemento)**: Restituisce il numero di volte in cui l'elemento specificato compare nella lista.
```
colori = ["rosso", "verde", "blu", "verde", "rosso"]
conteggio_rosso = colori.count("rosso")
print(conteggio_rosso)  # Output: 2
```

- **sort(key=None, reverse=False)**: Ordina gli elementi della lista sul posto (modifica la lista originale).
    - key specifica una funzione da chiamare su ciascun elemento della lista prima di effettuare i confronti.

- reverse=True ordina la lista in ordine decrescente.

```
numeri = [3, 1, 4, 1, 5, 9, 2, 6]
numeri.sort()
print(numeri)  # Output: [1, 1, 2, 3, 4, 5, 6, 9]

nomi = ["Charlie", "Alice", "Bob"]
nomi.sort()
print(nomi)    # Output: ['Alice', 'Bob', 'Charlie']

numeri.sort(reverse=True)
print(numeri)  # Output: [9, 6, 5, 4, 3, 2, 1, 1]

def lunghezza_stringa(s):
    return len(s)

parole = ["mela", "banana", "kiwi", "fragola"]
parole.sort(key=lunghezza_stringa)
print(parole)  # Output: ['mela', 'kiwi', 'banana', 'fragola']
```

- **reverse()**: Inverte l'ordine degli elementi della lista sul posto.

```
alfabeto = ["a", "b", "c", "d"]
alfabeto.reverse()
print(alfabeto)  # Output: ['d', 'c', 'b', 'a']
```

- **clear()**: Rimuove tutti gli elementi dalla lista, lasciandola vuota.

```
dati = [1, 2, 3]
dati.clear()
print(dati)  # Output: []
```

- **copy()**: Restituisce una copia superficiale della lista. Le modifiche alla copia non influenzano la lista originale e viceversa (a meno che gli elementi contenuti non siano oggetti mutabili).

```
originale = [10, 20, 30]
copia = originale.copy()
copia.append(40)
print(originale)  # Output: [10, 20, 30]
print(copia)      # Output: [10, 20, 30, 40]
```

Esempi pratici:

1. **Calcolare la media di una lista di numeri:**

```
numeri = [10, 20, 30, 40, 50]
somma = sum(numeri)
media = somma / len(numeri)
print(f"La media è: {media}")  # Output: La media è: 30.0
```

2. **Filtrare una lista di stringhe per lunghezza:**

```
parole = ["casa", "albero", "libro", "sole", "tavolo"]
parole_lunghe = [parola for parola in parole if len(parola) > 4]
print(parole_lunghe)  # Output: ['albero', 'libro', 'tavolo']
```

3. **Trovare il massimo e il minimo in una lista:**

```
valori = [5, 2, 8, 1, 9, 3]
massimo = max(valori)
minimo = min(valori)
print(f"Il valore massimo è: {massimo}")  # Output: Il valore
massimo è: 9
print(f"Il valore minimo è: {minimo}")  # Output: Il valore minimo
```

```
è: 1
```

Le tre modalità per leggere e modificare gli elementi di una lista

Le liste sono uno strumento fondamentale in Python per gestire collezioni di dati in modo efficiente e flessibile. La comprensione dei metodi associati permette di manipolare e analizzare questi dati in svariati modi.

```
print("***** primo metodo *****")
print(len(nomi))
for i in range(len(nomi)):
   print("stampo indice e valore :", i, nomi[i])

print("***** secondo metodo *****")
i = 0
for numero in numeri:
   print("stampo indice e valore : ", i, numero)
   i +=1

print("******* terzo metodo ******")
for indice,valore in enumerate(mista):
   print("stampo indice e valore : ", indice, valore)
```

In Python, la funzione enumerate(lista) produce un **oggetto enumerato**.
Questo oggetto enumerato è un **iteratore** che genera una sequenza di **tuple**. Ogni tupla contiene due elementi:

1. L'**indice** dell'elemento nella lista (a partire da 0 per impostazione predefinita).
2. L'**elemento** stesso proveniente dalla lista.

In sostanza, enumerate() aggiunge un contatore a un iterabile (come una lista) e restituisce un oggetto che può essere utilizzato per scorrere sia l'indice che l'elemento corrispondente ad ogni passo dell'iterazione.

Esempio:

```
mia_lista = ['a', 'b', 'c']

oggetto_enumerato = enumerate(mia_lista)
print(oggetto_enumerato)  # Output: <enumerate object at 0x...>

# Per visualizzare il contenuto, possiamo convertirlo in una lista di
tuple
lista_di_tuple = list(oggetto_enumerato)
print(lista_di_tuple)  # Output: [(0, 'a'), (1, 'b'), (2, 'c')]

# È comune usare enumerate direttamente in un ciclo for:
for indice, elemento in enumerate(mia_lista):
    print(f"L'elemento all'indice {indice} è: {elemento}")

# Output:
# L'elemento all'indice 0 è: a
# L'elemento all'indice 1 è: b
# L'elemento all'indice 2 è: c
```

Tuple

Le **tuple** in Python sono una struttura dati sequenziale che viene utilizzata per memorizzare una collezione ordinata di elementi. Sono molto simili alle liste, ma con una differenza fondamentale: **le tuple sono immutabili**. Questo significa che una volta creata una tupla, non puoi modificarne gli elementi (non puoi aggiungere, rimuovere o cambiare gli elementi esistenti).
Ecco i punti chiave da sapere sulle tuple:

- **Sequenziali e Ordinate:** Gli elementi all'interno di una tupla mantengono un ordine specifico, proprio come in una lista. L'ordine in cui gli elementi vengono definiti è l'ordine in cui vengono memorizzati e a cui si può accedere tramite l'indice.
- **Immutabili:** Questa è la caratteristica distintiva principale. L'immutabilità rende le tuple adatte a rappresentare dati che non dovrebbero essere modificati durante l'esecuzione di un programma.
- **Possono contenere elementi di tipi diversi:** Proprio come le liste, le tuple possono contenere elementi di tipi di dati diversi (interi, float, stringhe, altre tuple, liste, ecc.).
- **Definizione:** Le tuple vengono definite utilizzando le **parentesi tonde ()**. Gli elementi all'interno della tupla sono separati da virgole.
  ```
  mia_tupla = (1, 2, "ciao", 3.14)
  ```

- **Tupla con un solo elemento:** Per creare una tupla con un singolo elemento, è necessario includere una virgola dopo l'elemento. Senza la virgola, Python interpreterebbe l'espressione come il tipo dell'elemento stesso.
  ```
  tupla_singolo_elemento = (5,)
  non_una_tupla = (5)  # Questo è solo un intero
  ```

- **Creazione senza parentesi (in alcuni contesti):** In alcuni contesti, le parentesi possono essere omesse quando si definisce una tupla, specialmente nell'assegnazione multipla. Tuttavia, è generalmente considerata una buona pratica includerle per chiarezza.
  ```
  x, y = 10, 20  # x e y diventano una tupla (10, 20) implicitamente
  ```

- **Accesso agli elementi:** Gli elementi di una tupla possono essere acceduti utilizzando l'indice (a partire da 0 per il primo elemento).
  ```
  mia_tupla = (10, 20, 30)
  primo_elemento = mia_tupla[0]  # primo_elemento sarà 10
  secondo_elemento = mia_tupla[1] # secondo_elemento sarà 20
  ```

- **Slicing:** È possibile estrarre sottosequenze di elementi da una tupla utilizzando lo slicing, proprio come con le liste e le stringhe.
  ```
  mia_tupla = (1, 2, 3, 4, 5)
  sotto_tupla = mia_tupla[1:4]  # sotto_tupla sarà (2, 3, 4)
  ```

Quando usare le tuple?
Le tuple sono utili in diverse situazioni:

- **Rappresentare dati immutabili:** Quando hai bisogno di una collezione di elementi che non dovrebbero essere cambiate (ad esempio, le coordinate di un punto, i giorni della settimana, i mesi dell'anno).
- **Passare dati in modo sicuro:** L'immutabilità garantisce che i dati passati come tuple a una funzione non vengano accidentalmente modificati all'interno della funzione.
- **Usare come chiavi in un dizionario:** Poiché le tuple sono immutabili, possono essere utilizzate come chiavi in un dizionario Python (le liste non possono essere usate come chiavi perché sono mutabili).
- **Restituire più valori da una funzione:** Una funzione può restituire più valori impacchettandoli in una tupla.
- **Miglioramento delle prestazioni (in alcuni casi):** Le tuple possono essere leggermente più efficienti in termini di memoria e velocità di accesso rispetto alle liste, poiché la loro dimensione è fissa.

In sintesi, le tuple sono una potente e utile struttura dati in Python per rappresentare sequenze immutabili di elementi. La loro immutabilità le rende adatte a specifici casi d'uso dove la protezione dei dati e la coerenza sono importanti.

Le differenza principale tra liste e tuple in Python:

Lista (list)

- **Mutabile:** Una volta creata, puoi modificare gli elementi di una lista. Puoi aggiungere, rimuovere o cambiare gli elementi.
- **Dichiarazione:** Le liste sono definite utilizzando le parentesi quadre []. Gli elementi sono separati da virgole.
- **Esempio:**

```
mia_lista = [1, 2, "ciao", 3.14]
mia_lista[0] = 10  # Modifica il primo elemento
mia_lista.append("mondo")  # Aggiunge un nuovo elemento alla fine
del mia_lista[2]  # Rimuove l'elemento all'indice 2
print(mia_lista)  # Output: [10, 2, 3.14, 'mondo']
```

- **Uso comune:** Le liste sono usate quando hai bisogno di una sequenza di elementi che potrebbe cambiare nel tempo. Sono adatte per collezioni di elementi dello stesso tipo o di tipi diversi.

Tupla (tuple)

- **Immutabile:** Una volta creata, non puoi modificare gli elementi di una tupla. Non puoi aggiungere, rimuovere o cambiare gli elementi.
- **Dichiarazione:** Le tuple sono definite utilizzando le parentesi tonde (). Gli elementi sono separati da virgole. Anche se le parentesi non sono strettamente necessarie in alcuni casi (ad esempio, quando si assegnano più valori a variabili contemporaneamente), è buona pratica includerle per chiarezza.
- **Esempio:**

```
mia_tupla = (1, 2, "ciao", 3.14)
# mia_tupla[0] = 10  # Questo genererebbe un errore (TypeError:
'tuple' object does not support item assignment)
# mia_tupla.append("mondo")  # Questo genererebbe un errore ('tuple'
object has no attribute 'append')
print(mia_tupla)  # Output: (1, 2, 'ciao', 3.14)
```

- **Uso comune:** Le tuple sono usate per rappresentare collezioni di elementi che non dovrebbero essere cambiate. Sono spesso usate per:
  - Rappresentare record (ad esempio, le coordinate di un punto: (x, y)).
  - Restituire più valori da una funzione.
  - Essere usate come chiavi in un dizionario (le liste non possono essere usate come chiavi perché sono mutabili).
  - Garantire l'integrità dei dati, poiché gli elementi non possono essere modificati accidentalmente.

In sintesi:

Caratteristica	Lista (list)	Tupla (tuple)
Mutabilità	Mutabile (gli elementi possono essere modificati)	Immutabile (gli elementi non possono essere modificati)
Sintassi	[]	()
Uso principale	Collezioni di elementi che possono cambiare	Collezioni di elementi fissi, record, chiavi di dizionari
Prestazioni (in generale)	Leggermente più lenta per l'iterazione e l'accesso agli elementi rispetto alle tuple (a causa della gestione della mutabilità)	Leggermente più veloce per l'iterazione e l'accesso agli elementi
Dimensione in memoria (in generale)	Solitamente occupa più memoria rispetto a una tupla con gli stessi elementi	Solitamente occupa meno memoria rispetto a una lista con gli stessi elementi

Dizionari

In Python, un **dizionario** è una struttura dati che memorizza coppie di **chiave-valore**. Pensa a un dizionario cartaceo: cerchi una parola (la chiave) e trovi la sua definizione (il valore). Nei dizionari Python, le chiavi devono essere uniche e immutabili (stringhe, numeri o tuple), mentre i valori possono essere di qualsiasi tipo.

Caratteristiche principali dei dizionari:

- **Non ordinati:** A partire da Python 3.7, i dizionari mantengono l'ordine di inserimento degli elementi. Nelle versioni precedenti, non era garantito alcun ordine specifico.
- **Mutabili:** Puoi aggiungere, rimuovere e modificare elementi in un dizionario dopo la sua creazione.
- **Chiavi uniche:** Ogni chiave all'interno di un dizionario deve essere unica. Se provi ad aggiungere una chiave già esistente, il vecchio valore associato verrà sovrascritto.
- **Accesso tramite chiave:** Accedi ai valori utilizzando le loro chiavi, in modo simile a come accedi agli elementi di una lista tramite il loro indice.

Come creare un dizionario:

Puoi creare un dizionario in diversi modi:

1. **Utilizzando le parentesi graffe {}:**
```
mio_dizionario = {"nome": "Alice", "età": 30, "città": "Roma"}
print(mio_dizionario)
```

2. **Utilizzando la funzione dict():**
```
altro_dizionario = dict(nome="Bob", età=25, professione="Ingegnere")
print(altro_dizionario)

# Puoi anche passare una lista di tuple (chiave, valore)
terzo_dizionario = dict([("colore", "blu"), ("taglia", "M")])
print(terzo_dizionario)
```

3. **Creando un dizionario vuoto e aggiungendo elementi successivamente:**
```
dizionario_vuoto = {}
dizionario_vuoto["paese"] = "Italia"
dizionario_vuoto["lingua"] = "Italiano"
print(dizionario_vuoto)
```

Metodi comuni dei dizionari:

Ecco alcuni dei metodi più utilizzati per manipolare i dizionari in Python:

- **get(chiave, valore_predefinito):** Restituisce il valore associato alla chiave. Se la chiave non esiste, restituisce valore_predefinito (se specificato) o None. È un modo più sicuro per accedere ai valori rispetto all'indicizzazione diretta (mio_dizionario["chiave"]), che genererebbe un errore KeyError se la chiave non esiste.
```
persona = {"nome": "Carlo", "età": 35}
nome = persona.get("nome")
print(f"Nome: {nome}")  # Output: Nome: Carlo

citta = persona.get("città")
print(f"Città: {citta}")  # Output: Città: None

paese = persona.get("paese", "Sconosciuto")
print(f"Paese: {paese}")  # Output: Paese: Sconosciuto
```

- **keys():** Restituisce una vista (simile a un set) contenente tutte le chiavi del dizionario.
```
info = {"nome": "Elena", "lavoro": "Artista", "anni": 28}
chiavi = info.keys()
print(chiavi)  # Output: dict_keys(['nome', 'lavoro', 'anni'])

# Puoi iterare sulle chiavi
for chiave in info.keys():
```

```
    print(chiave)
```

- **values():** Restituisce una vista contenente tutti i valori del dizionario.
```
valori = info.values()
print(valori)  # Output: dict_values(['Elena', 'Artista', 28])

# Puoi iterare sui valori
for valore in info.values():
    print(valore)
```

- **items():** Restituisce una vista contenente tutte le coppie (chiave, valore) come tuple.
```
elementi = info.items()
print(elementi)  # Output: dict_items([('nome', 'Elena'), ('lavoro',
'Artista'), ('anni', 28)])

# Puoi iterare sulle coppie chiave-valore
for chiave, valore in info.items():
    print(f"{chiave}: {valore}")
```

- **update(altro_dizionario):** Aggiorna il dizionario con le coppie chiave-valore di altro_dizionario. Se una chiave esiste già, il suo valore viene sovrascritto; altrimenti, vengono aggiunte nuove coppie.
```
d1 = {"a": 1, "b": 2}
d2 = {"b": 3, "c": 4}
d1.update(d2)
print(d1)  # Output: {'a': 1, 'b': 3, 'c': 4}
```

- **pop(chiave, valore_predefinito):** Rimuove la chiave specificata dal dizionario e ne restituisce il valore. Se la chiave non esiste, solleva un KeyError a meno che non venga fornito un valore_predefinito, nel qual caso restituisce quel valore.
```
frutta = {"mela": 2, "banana": 5, "arancia": 3}
quantita_banana = frutta.pop("banana")
print(f"Quantità di banana rimossa: {quantita_banana}")  # Output:
Quantità di banana rimossa: 5
print(frutta)  # Output: {'mela': 2, 'arancia': 3}

quantita_uva = frutta.pop("uva", 0)
print(f"Quantità di uva (non presente): {quantita_uva}")  # Output:
Quantità di uva (non presente): 0
```

- **popitem():** Rimuove e restituisce l'ultima coppia (chiave, valore) inserita nel dizionario (in Python 3.7+). Nelle versioni precedenti, rimuoveva una coppia arbitraria.
```
colori = {"rosso": "#FF0000", "verde": "#00FF00", "blu": "#0000FF"}
ultimo_elemento = colori.popitem()
print(f"Ultimo elemento rimosso: {ultimo_elemento}")
print(colori)
```

- **clear():** Rimuove tutti gli elementi dal dizionario, rendendolo vuoto.
```
numeri = {1: "uno", 2: "due", 3: "tre"}
numeri.clear()
print(numeri)  # Output: {}
```

- **copy():** Restituisce una copia superficiale del dizionario. Le modifiche alla copia non influenzeranno il dizionario originale e viceversa per gli oggetti immutabili. Tuttavia, se i valori sono oggetti mutabili (come le liste), le modifiche a questi oggetti si rifletteranno in

entrambe le copie.

```
originale = {"x": [1, 2], "y": 3}
copia = originale.copy()
copia["y"] = 4
copia["x"].append(3)  # Modifica l'oggetto mutabile (la lista)

print(f"Originale: {originale}")  # Output: Originale: {'x': [1, 2,
3], 'y': 3}
print(f"Copia: {copia}")      # Output: Copia: {'x': [1, 2, 3], 'y':
4}
```