Collezioni in Python

Python

In Python, le strutture dati sono modi per organizzare e immagazzinare dati in modo che possano essere utilizzati in modo efficiente, possiamo immaginarle come collezioni di elementi basilari che però puntano tutti a un determinato punto in memoria in modo da poterli ritrovare in maniera più semplice ma SOPRATUTTO UNIVOCA.

Python offre diverse strutture dati integrate che sono altamente flessibili e supportano una varietà di operazioni. Esploriamo le principali:

Liste

- Le liste sono collezioni ordinate e modificabili che consentono elementi duplicati. Sono tra le strutture dati più versatili e comunemente usate in Python. Le liste possono contenere elementi di diversi tipi e supportano operazioni come l'aggiunta, la rimozione e la modifica degli elementi.

Tuple

- Le tuple sono simili alle liste, ma immutabili, il che significa che non possono essere modificate dopo la loro creazione. Questo le rende più veloci delle liste per alcune operazioni e ideali per conservare dati che non dovrebbero cambiare, come le costanti di un programma.

Insiemi (Sets)

- Gli insiemi sono collezioni non ordinate di elementi unici. Sono utili per operazioni come la verifica dell'appartenenza, l'eliminazione dei duplicati da una sequenza e il calcolo delle operazioni matematiche tra insiemi, come l'unione, l'intersezione, la differenza e la differenza simmetrica.

Dizionari

- I dizionari sono collezioni non ordinate di coppie chiave-valore. A differenza degli altri tipi di dati che utilizzano indici per accedere agli elementi, i dizionari utilizzano le chiavi. Questo li rende incredibilmente efficienti per la ricerca e l'aggiornamento dei valori. I dizionari sono ottimi per rappresentare relazioni chiave-valore, dati strutturati e JSON-like.

Collezioni Specializzate

- Python fornisce anche il modulo collections, che offre strutture dati aggiuntive come defaultdict, OrderedDict, Counter, e deque. Queste strutture dati specializzate forniscono alternative con funzionalità aggiuntive rispetto ai tipi di dati integrati, come l'ordine di inserimento dei dizionari o le code a doppio terminale.

Array

- Sebbene non siano parte delle strutture dati Python native (richiedono l'importazione del modulo array), gli array possono essere usati per memorizzare dati in un modo simile alle liste, ma con la restrizione che tutti gli elementi devono essere dello stesso tipo. Questo può migliorare l'efficienza dello spazio e la velocità per grandi quantità di dati di tipo numerico.

Conclusioni

La scelta della struttura dati giusta dipende dal tipo di operazioni che devi eseguire sui tuoi dati. Per esempio:

- Liste e Tuple: per collezioni ordinate di elementi.
- Insiemi: per operazioni rapide di verifica dell'appartenenza e per eliminare duplicati.
- Dizionari: per associazioni chiave-valore con accesso rapido.
- Collezioni Specializzate: per esigenze di dati specifiche che vanno oltre le funzionalità delle strutture dati standard.

Python rende l'utilizzo di queste strutture dati particolarmente semplice e intuitivo, consentendoti di concentrarti sulla logica del tuo programma piuttosto che sui dettagli dell'implementazione delle strutture dati.

Liste

Le liste sono utilizzate per immagazzinare una lista ordinata di elementi. Sono modificabili, il che significa che puoi cambiare i loro contenuti.

```
# Creazione di una lista
frutti = ["mela", "banana", "ciliegia"]

# Aggiungere un elemento
frutti.append("arancia")

# Rimuovere un elemento
frutti.remove("banana")

# Accedere a un elemento
print(frutti[0])  # Stampa "mela"

# Iterare su una lista
for frutto in frutti:
    print(frutto)
```

Tuple

Le tuple sono simili alle liste, ma sono immutabili. Sono utilizzate per memorizzare dati che non devono essere modificati.

```
# Creazione di una tupla
coordinate = (10.0, 20.0)

# Accedere a un elemento
print(coordinate[0])  # Stampa 10.0

# Tentativo di modifica (questo causerà un errore)
# coordinate[0] = 15.0
```

Insiemi (Sets)

Gli insiemi sono collezioni non ordinate di elementi unici. Sono utili per testare l'appartenenza e rimuovere i duplicati.

```
# Creazione di un insieme
numeri = {1, 2, 3, 4, 5}

# Aggiungere un elemento
numeri.add(6)

# Rimuovere un elemento
numeri.remove(1)

# Verificare se un elemento è presente
print(2 in numeri)  # Stampa True

# Operazioni tra insiemi
altri_numeri = {4, 5, 6, 7}
print(numeri.intersection(altri_numeri))  # Elementi comuni
```

I Dizionari

I dizionari memorizzano coppie di chiavi e valori. Sono utili per memorizzare e recuperare dati in modo efficiente tramite chiavi.

```
# Creazione di un dizionario
studente = {
    "nome": "Mario",
    "età": 21,
    "corso": "Informatica"
}

# Accedere a un valore tramite chiave
print(studente["nome"])  # Stampa "Mario"

# Aggiungere o modificare un elemento
studente["anno"] = 3  # Aggiunge la chiave "anno" con il valore 3

# Rimuovere una coppia chiave-valore
del studente["età"]

# Iterare su un dizionario
for chiave, valore in studente.items():
    print(f"{chiave}: {valore}")
```

I dizionari in Python sono strutture dati che memorizzano elementi come coppie chiave-valore, consentendo l'accesso veloce ai valori attraverso le loro chiavi.

Sono mutabili, dinamici e possono contenere oggetti di qualsiasi tipo come valori.

Vediamo le funzioni e i metodi principali associati ai dizionari in Python, con esempi di codice per ciascuno.

Creazione di un Dizionario

Dizionario Vuoto

1. `mio_dizionario = {}`

Dizionario con Elementi

1. `mio_dizionario = {'nome': 'Mario', 'età': 30}`

Utilizzo del Costruttore dict()

1. `mio_dizionario = dict(nome='Mario', età=30)`

Accesso agli Elementi

Accesso tramite Chiave

```
1. print(mio_dizionario['nome'])  # Mario
```

Metodo get()

```
1. print(mio_dizionario.get('nome'))  # Mario
```

Modifica di un Dizionario

Aggiungere o Modificare Elementi

1. mio_dizionario['indirizzo'] = 'Via Roma 10'

Rimuovere Elementi

pop()

1. mio_dizionario.pop('età')

popitem() (Rimuove l'ultimo elemento inserito in Python 3.7+)

1. mio_dizionario.popitem()

del

1. del mio_dizionario['nome']

Svuotare il Dizionario

1. mio_dizionario.clear()

Esplorazione del Dizionario

Chiavi

1. chiavi = mio_dizionario.keys()

Valori

1. valori = mio_dizionario.values()

Coppie Chiave-Valore

1. elementi = mio_dizionario.items()

Controllo di Esistenza

In un Dizionario

1. 'nome' in mio_dizionario # True o False

Iterazione

Iterazione su Chiavi

```
for chiave in mio_dizionario:
    print(chiave)
```

Iterazione su Chiavi e Valori

```
for chiave, valore in mio_dizionario.items():
    print(chiave, valore)
```

Copia di Dizionari

Copia Superficiale

1. mio_dizionario_copia = mio_dizionario.copy()

Copia Profonda

1. import copy
2. mio_dizionario_copia_profonda = copy.deepcopy(mio_dizionario)

Dizionari e Comprehension

Comprehension dei Dizionari

1. quadrati = {x: x**2 for x in range(5)}

Metodi Avanzati

update() (Per unire due dizionari)

1. **altro_dizionario = {'città': 'Roma'}**
2. **mio_dizionario.update(altro_dizionario)**

setdefault() (Ottiene il valore di una chiave, e se non esiste, imposta un valore predefinito)

1. **mio_dizionario.setdefault('città', 'Milano')**

I dizionari in Python sono incredibilmente versatili e utili in molteplici scenari di programmazione, grazie alla loro struttura basata su coppie chiave-valore che permette un accesso veloce e efficiente ai dati. Ecco alcuni contesti in cui i dizionari risultano particolarmente utili:

1. Rappresentazione di Dati Strutturati
I dizionari sono ideali per rappresentare dati strutturati, come informazioni relative a un utente, configurazioni di un'applicazione, o metadati su un file. Per esempio, possono essere usati per memorizzare i dettagli di un libro in un catalogo o le specifiche di un prodotto in un inventario.

2. Mappature e Associazioni
Quando c'è bisogno di mappare una chiave a un valore specifico, come associare un nome utente alla sua email o un identificativo prodotto alla sua descrizione, i dizionari sono la soluzione ideale. Questo permette di recuperare velocemente il valore associato a una chiave.

3. Conteggio e Raggruppamento Elementi

I dizionari sono estremamente utili per conteggiare occorrenze o per raggruppare elementi. Ad esempio, possono essere usati per contare le parole in un testo o per organizzare gli studenti in base al corso di studi. L'utilizzo del metodo get() o di defaultdict da collections facilita questi compiti.

4. Caching e Memorizzazione Risultati

I dizionari possono essere utilizzati come cache per memorizzare i risultati di operazioni costose in termini di tempo. Salvando il risultato di una funzione dato un certo input nel dizionario, si può evitare di eseguire nuovamente l'operazione se l'input si ripresenta, accedendo direttamente al risultato memorizzato.

5. Implementazione di Grafi e Altre Strutture Dati

In alcune situazioni, i dizionari possono essere usati per rappresentare grafi, alberi o altre strutture dati complesse. Ad esempio, le chiavi possono rappresentare i nodi, mentre i valori, che possono essere liste o altri dizionari, rappresentano le connessioni o le relazioni tra i nodi.

Come Utilizzarli Efficacemente

- Scegliere Chiavi Appropriate: Le chiavi dovrebbero essere immutabili (stringhe, numeri o tuple contenenti solo tipi immutabili) e univocamente identificative per i valori che mappano.
- Accesso Sicuro ai Valori: Usare il metodo get() per accedere ai valori può prevenire eccezioni se la chiave non è presente. In alternativa, try-except può gestire casi in cui una chiave non esiste.
- Iterazione ed Esplorazione: Utilizzare metodi come .items(), .keys(), e .values() per iterare efficacemente su chiavi, valori, o entrambi. Questo è particolarmente utile per l'analisi dei dati o per la trasformazione dei contenuti di un dizionario.
- Utilizzo di Comprehension: I dizionari supportano la comprehension, che può essere utilizzata per creare o trasformare dizionari in modo conciso ed efficace.
- Mantenimento dell'Ordine: A partire da Python 3.7, i dizionari mantengono l'ordine di inserimento degli elementi, il che può essere utile per alcune applicazioni che dipendono dall'ordine dei dati.


Buon Davante a tutti