Python

Biagio Spada

May 2025

Sommario

Questi appunti raccolgono i concetti fondamentali del linguaggio Python, con esempi pratici tratti da esercizi svolti personalmente. Il documento è stato creato come strumento di ripasso e per facilitare l'apprendimento di Python.

Gli argomenti trattati spaziano dai concetti base (variabili, tipi di dati, strutture di controllo) a quelli più avanzati (classi, gestione dei file, moduli, algoritmi e strutture dati). Per ogni concetto vengono forniti esempi di codice commentati e spiegazioni dettagliate.

Indice

1 Introduzione a Python **5**
- 1.1 Caratteristiche Principali . . . 5
- 1.2 Installazione . . . 5
  - 1.2.1 Scelta della versione . . . 5
  - 1.2.2 Download e installazione . . . 5
  - 1.2.3 Verifica dell'installazione . . . 6
  - 1.2.4 Gestione dei pacchetti con pip . . . 6
  - 1.2.5 Ambienti virtuali . . . 6
  - 1.2.6 Opzioni alternative . . . 6
- 1.3 Il tuo primo programma Python . . . 6

2 Variabili e Tipi di Dati **8**
- 2.1 Dichiarazione e Assegnazione delle Variabili . . . 8
- 2.2 Regole e best practice per denominare le variabili . . . 8

3 Metodi principali per i tipi di dati Python **10**
- 3.1 Operatori . . . 12
  - 3.1.1 Operatori aritmetici . . . 12
  - 3.1.2 Operatori di confronto . . . 12
  - 3.1.3 Operatori logici . . . 12
- 3.2 Input e Output . . . 13

4 Approfondimento delle Variabili **14**
- 4.1 Il modello di riferimento in Python . . . 14
- 4.2 Assegnazione di variabili . . . 14
- 4.3 Tipi immutabili vs mutabili . . . 14
  - 4.3.1 Comportamento con tipi immutabili . . . 16
  - 4.3.2 Comportamento con tipi mutabili . . . 16
- 4.4 Verifica dell'identità degli oggetti . . . 17
- 4.5 Esercizio pratico: tracciare le variabili . . . 17
- 4.6 Esercizi Riassuntivi . . . 20
  - 4.6.1 Esercizio 1: Implementazione Input ed Output . . . 20
  - 4.6.2 Esercizio 2: Identificazione degli errori . . . 20
  - 4.6.3 Esercizio 3: Operazioni Matematiche . . . 20
  - 4.6.4 Esercizio 4: Operatori logici . . . 21
  - 4.6.5 Esercizio 5: MiniCalcolatrice . . . 21
  - 4.6.6 Esercizio 6: Mutabilità e Immutabilità . . . 22
- 4.7 Risoluzione Esercizi sui Fondamentali . . . 22
  - 4.7.1 Risoluzione Esercizio 1: *Esercizio 1: Implementazione Input ed Output* . . . 22
  - 4.7.2 Risoluzione Esercizio 2: *Esercizio 2: Identificazione degli errori* . . . 23
  - 4.7.3 Risoluzione Esercizio 3: *Esercizio 3: Operazioni Matematiche* . . . 23
  - 4.7.4 Risoluzione Esercizio 4: *Esercizio 4: Operatori logici* . . . 23
  - 4.7.5 Risoluzione Esercizio 5: *Esercizio 5: MiniCalcolatrice* . . . 24
  - 4.7.6 Risoluzione Esercizio 6: *Esercizio 6: Mutabilità e Immutabilità* . . . 24

5 Presentazione Liste, Tuple, Dizionari, Set **27**
5.1 Liste . . . 27
5.1.1 Definizione e Caratteristiche Principali . . . 27
5.2 L'Indicizzazione delle Liste in Python . . . 29
5.2.1 Un modello matematico per comprendere gli indici . . . 29
5.2.2 Vantaggi dell'indicizzazione bidirezionale . . . 30
5.2.3 Esempi pratici . . . 30
5.2.4 Il concetto di insieme ciclico chiuso . . . 30
5.2.5 Conclusione . . . 31
5.2.6 Sintassi di Base . . . 32
5.2.7 Accesso agli Elementi . . . 33
5.2.8 Modifica degli Elementi . . . 33
5.2.9 Slicing di Liste . . . 34
5.2.10 Operazioni e Metodi Base . . . 35
5.2.11 Metodi per Aggiungere Elementi . . . 35
5.2.12 Metodi per Rimuovere Elementi . . . 35
5.2.13 Gestione degli Errori . . . 36
5.2.14 Metodi per Ottenere Informazioni . . . 36
5.2.15 Esempi di index() e count() . . . 37
5.2.16 Metodi per Ordinamento e Inversione . . . 37
5.3 Comportamento in Memoria delle Liste . . . 39
5.3.1 Variabili come Riferimenti . . . 39
5.3.2 Assegnazione e Riferimenti Multipli . . . 39
5.3.3 Creazione di Copie Indipendenti . . . 40
5.3.4 Confronto tra Liste: '==' vs 'is' . . . 49
5.3.5 Rappresentazione Visiva della Memoria . . . 49
5.3.6 Implicazioni Pratiche . . . 51
5.4 Esercizi Riassuntivi . . . 52
5.4.1 Esercizio 1: Concetti Base e Indicizzazione . . . 52
5.4.2 Esercizio 2: Operazioni Base sulle Liste . . . 52
5.4.3 Esercizio 3: Slicing delle Liste . . . 52
5.4.4 Esercizio 4: Metodi delle Liste . . . 53
5.4.5 Esercizio 5: Comportamento in Memoria delle Liste . . . 53
5.4.6 Esercizio 6: Analisi di Codice e Risoluzione Problemi: . . . 54
5.5 Risoluzione Esercizi Sulle Liste . . . 54
5.5.1 Risoluzione Esercizio 1: Esercizio 1: Concetti Base e Indicizzazione . . . 54
5.5.2 Risoluzioe Esercizio 2: Esercizio 2: Operazioni Base sulle Liste . . . 55
5.5.3 Risoluzione Esercizio 3: Esercizio 3: Slicing delle Liste . . . 56
5.5.4 Risoluzione Esercizio 4: Esercizio 4: Metodi delle Liste . . . 56
5.5.5 Risoluzione Esercizio 5: Esercizio 5: Comportamento in Memoria delle Liste . . . 57
5.5.6 Risoluzione Esercizio 6: Esercizio 6: Analisi di Codice e Risoluzione Problemi: 58
5.6 Tuple . . . 60
5.6.1 Caratteristiche Principali delle Tuple . . . 60
5.6.2 Sintassi Base delle Tuple . . . 60
5.6.3 Operazioni comuni con le Tuple . . . 61
5.6.4 Acccesso agli elementi (indicizzazione): . . . 61
5.6.5 Slicing: . . . 61
5.6.6 Concatenazione delle Tuple . . . 62
5.6.7 Concatenazione vs altri metodi di combinazione . . . 63
5.6.8 Differenza tra Tuple e Liste: . . . 63
5.6.9 Altri metodi: . . . 65
5.7 Tuple e la Gestione della Memoria . . . 66

5.7.1 Allocazione della Memoria . . . 66
5.7.2 Struttura Interna . . . 66
5.7.3 Immutabilità e Ottimizzazioni di Memoria . . . 66
5.7.4 Tuple come Elementi Hashable . . . 67
5.7.5 Tuple di Tuple e strutture Annidate . . . 67
5.7.6 Garbage Collection . . . 67
5.7.7 Ciclo di Vita di una Tupla in Memoria . . . 67
5.7.8 Rappresentazione Grafica . . . 68
5.8 Esercizi Riassuntivi . . . 72
5.8.1 Esercizio 1: Creazione e sintassi Base delle Tuple . . . 72
5.8.2 Esercizio 2: Accesso agli Elementi (indicizzazione) . . . 72
5.8.3 Esercizio 3: Sclicing delle Tuple . . . 72
5.8.4 Esercizio 4: Metodi per le Tuple . . . 72
5.8.5 Esercizio 5: Concatenazione delle Tuple . . . 73
5.8.6 Esercizio 6: Immutabilità delle Tuple . . . 73
5.8.7 Esercizio 7: Tuple Annidate . . . 73
5.8.8 Esercizio 8: Operazioni di Confronto . . . 73
5.8.9 Esercizzio 9: Applicazione Pratica . . . 74
5.9 Risoluzione Esercizi sulle Tuple . . . 74
5.9.1 Risoluzione: Esercizio 1: Creazione e sintassi Base delle Tuple . . . 74
5.9.2 Risoluzione: Esercizio 2: Accesso agli Elementi (indicizzazione) . . . 74
5.9.3 Risoluzione: Esercizio 3: Sclicing delle Tuple . . . 75
5.9.4 Risoluzione: Esercizio 4: Metodi per le Tuple . . . 75
5.9.5 Risoluzione: Esercizio 5: Concatenazione delle Tuple . . . 76
5.9.6 Risoluzione: Esercizio 6: Immutabilità delle Tuple . . . 76
5.9.7 Risoluzione: Esercizio 7: Tuple Annidate . . . 77
5.9.8 Risoluzione: Esercizio 8: Operazioni di Confronto . . . 77
5.9.9 Risoluzione: Esercizzio 9: Applicazione Pratica . . . 78
5.10 Dizionari . . . 79
5.10.1 Metodi per dichiarare un Dizionario . . . 79
5.10.2 Chiavi - Valori . . . 80
5.10.3 Caratteristiche delle Chiavi . . . 80
5.10.4 Meccanismo delle Tabelle Hash . . . 80
5.10.5 Stabilità dell'Hash . . . 81
5.11 Set . . . 86
5.12 Caretteristiche Fondamentali . . . 86
5.12.1 Teoria degli insiemi . . . 86
5.12.2 Uguaglianza e Principio di Estensionalità . . . 86
5.12.3 Operazioni tra Insiemi . . . 86

1 Introduzione a Python

Python è un linguaggio di programmazione di alto livello, interpretato, orientato agli oggetti, con una sintassi semplice ed elegante. La sua popolarità è dovuta alla facilità di apprendimento e alla versatilità che lo rende adatto a molteplici ambiti.

1.1 Caratteristiche Principali

- **Leggibilità**: La sintassi progettata per essere chiara e leggibile
- **Versatilità**: Adatto a diversi contesti (web, data science, automazione, ecc.)
- **Gestione automatica della memoria**: Non è necessario allocare o deallocare manualmente la memoria

1.2 Installazione

Il primo passo da compiere per iniziare a programmare con Python è installarlo sul proprio sistema. Python è disponibile per tutti i principali sistemi operativi: Windows, macOS e Linux.

1.2.1 Scelta della versione

Attualmente esistono due versioni principali di Python: Python 2 e Python 3. Si consiglia vivamente di utilizzare Python 3, poiché Python 2 non è più supportato dal gennaio 2020 e molte librerie moderne funzionano esclusivamente con Python 3.

1.2.2 Download e installazione

- **Windows**:
  1. Visita il sito ufficiale Python (`https://www.python.org/downloads/`)
  2. Scarica l'ultima versione di Python 3 (ad esempio Python 3.11.x)
  3. Esegui il file di installazione scaricato
  4. **Importante:** Nella finestra di installazione, seleziona la casella `Add Python to PATH` prima di procedere
  5. Clicca su `Install Now` per un'installazione standard o `Customize installation` per opzioni avanzate
- **macOS**:
  1. Visita il sito ufficiale Python (`https://www.python.org/downloads/`)
  2. Scarica l'ultima versione di Python 3 per macOS
  3. Apri il file .pkg scaricato e segui le istruzioni di installazione
  4. In alternativa, se utilizzi Homebrew, puoi installare Python con il comando `brew install python3`
- **Linux**:
  1. Molte distribuzioni Linux includono già Python preinstallato. Puoi verificare la versione con `python3 -version` dal terminale
  2. Su Ubuntu/Debian: `sudo apt update && sudo apt install python3 python3-pip`
  3. Su Fedora: `sudo dnf install python3`
  4. Su Arch Linux: `sudo pacman -S python python-pip`

1.2.3 Verifica dell'installazione

Dopo l'installazione, è importante verificare che Python sia stato installato correttamente:

1. Apri il terminale (Command Prompt o PowerShell su Windows, Terminal su macOS/Linux)
2. Digita `python -version` o `python3 -version`
3. Dovresti vedere in output la versione di Python installata, ad esempio `Python 3.11.4`

1.2.4 Gestione dei pacchetti con pip

Python include un gestore di pacchetti chiamato pip, che permette di installare facilmente librerie aggiuntive. Per verificare che pip sia installato:

1. Dal terminale, esegui `pip -version` o `pip3 -version`
2. Dovresti vedere la versione di pip installata

Per installare una libreria, usa il comando:

```
pip install nome_libreria
```

1.2.5 Ambienti virtuali

Per i progetti più complessi, è consigliabile utilizzare ambienti virtuali per mantenere le dipendenze separate. Per creare un ambiente virtuale:

```
Creazione dell'ambiente virtuale
python -m venv mio_ambiente
Attivazione dell'ambiente virtuale
Su Windows
mio_ambiente\Scripts\activate
Su macOS/Linux
source mio_ambiente/bin/activate
```

1.2.6 Opzioni alternative

Se preferisci non installare Python direttamente sul tuo sistema, puoi considerare:

- **Anaconda**: Una distribuzione Python che include molte librerie scientifiche (`https://www.anaconda.com/`)
- **Piattaforme online**:
  - Google Colab: Per notebook Python eseguiti nel browser
  - Replit: Per sviluppo Python online
  - PythonAnywhere: Ambiente Python basato su cloud
  - Deepnote come Replit, ambiente di sviluppo online

1.3 Il tuo primo programma Python

Ecco un esempio classico per iniziare con Python che include sia l'output che l'input:

```
# Il mio primo programma Python
print("Hello, World!")  # Stampa un messaggio a schermo

# Richiesta di input all'utente
nome = input("Come ti chiami? ")
print(f"Ciao {nome}, benvenuto nel mondo di Python!")
```

Nota

Come eseguire il codice Python:

1. **Da terminale**: Salva il codice in un file con estensione .py (es. primo_programma.py) e eseguilo con il comando `python primo_programma.py`
2. **Da IDE**: Usa un ambiente di sviluppo come PyCharm, Visual Studio Code o IDLE (incluso nell'installazione di Python)
3. **Da notebook**: Jupyter Notebook permette di eseguire il codice in celle interattive
4. **Online**: Servizi come replit.com o Google Colab permettono di scrivere ed eseguire codice Python senza installazioni

2 Variabili e Tipi di Dati

Introduzione

In questa sezione esploreremo il cuore della programmazione Python: variabili e tipi di dati. Vedremo come Python gestisce diversi tipi di informazioni, come memorizzarle in variabili, e come manipolarle attraverso varie operazioni. Comprenderemo le regole per denominare le variabili, come Python interpreta i diversi tipi di dati, e come interagiscono tra loro.

In Python, le variabili sono contenitori per memorizzare valori di dati. A differenza di altri linguaggi di programmazione, Python non richiede di dichiarare esplicitamente il tipo di una variabile prima di usarla. Il tipo viene determinato automaticamente in base al valore assegnato.

2.1 Dichiarazione e Assegnazione delle Variabili

```
# Dichiarazione e assegnazione di variabili
nome = "Mario"      # Una stringa
eta = 30            # Un intero
altezza = 1.75      # Float
e_studente = False  # Booleano
```

2.2 Regole e best practice per denominare le variabili

In Python, i nomi delle variabili devono seguire regole precise e alcune convenzioni che rendono il codice più leggibile e manutenibile.

Regole sintattiche

- **Caratteri consentiti**: Lettere (a-z, A-Z), numeri (0-9) e underscore (_)
- **Primo carattere**: Deve essere una lettera o un underscore, mai un numero
- **Case sensitivity**: `nome`, `Nome` e `NOME` sono tre variabili diverse
- **Parole riservate**: Non è possibile usare parole chiave di Python (come `if`, `for`, `while`)

Convenzioni di stile (PEP 8)

- **snake_case**: Per variabili e funzioni (es. `nome_utente`)
- **SCREAMING_SNAKE_CASE**: Per costanti (es. `MAX_TENTATIVI`)
- **CamelCase**: Per classi (es. `PersonaUtente`)
- **Nomi significativi**: Evitare sigle come `x`, `y`, preferire nomi descrittivi
- **Underscore iniziale**: `_variabile` suggerisce uso interno o privato

```
# Esempi di nomi validi
nome = "Alice"
_eta = 25
nome_completo = "Alice Rossi"
variabile1 = 42
COSTANTE = 3.14159

```

```
# Esempi di nomi non validi
# 1variabile = 10    # Inizia con un numero (errore di sintassi)
# for = "ciclo"      # E una parola riservata (errore di sintassi)
# nome-utente = "Mario"  # Il trattino non e' consentito
```

3 Metodi principali per i tipi di dati Python

Cos'è un metodo in Python?

In Python, i **metodi** sono funzioni speciali associate a oggetti di specifici tipi di dati. A differenza delle funzioni standard che accettano dati come argomenti, i metodi sono "collegati" agli oggetti stessi attraverso la notazione punto (`oggetto.metodo()`).

Ogni tipo di dato in Python possiede metodi specifici progettati per operazioni comuni e naturali su quel tipo. Questa caratteristica è parte fondamentale del paradigma della programmazione orientata agli oggetti che Python supporta.

```
# Funzione standard
len("Python")  # 6

# Metodo
"Python".upper()  # "PYTHON"
```

Metodi e mutabilità

Un concetto cruciale per comprendere i metodi in Python è la **mutabilità**. I tipi di dati in Python si dividono in due categorie:

- **Tipi immutabili**: una volta creati, non possono essere modificati (stringhe, tuple, numeri)
- **Tipi mutabili**: possono essere modificati dopo la creazione (liste, dizionari, set)

Questa distinzione influenza profondamente il comportamento dei metodi:

Comportamento dei Metodi

Tipi immutabili	I metodi **restituiscono sempre** un nuovo oggetto, lasciando l'originale invariato
Tipi mutabili	I metodi spesso **modificano direttamente** l'oggetto (operazioni in-place)

Metodi delle stringhe

Le stringhe (`str`) sono sequenze immutabili di caratteri, quindi i loro metodi restituiscono sempre nuove stringhe. Questi metodi consentono di trasformare e analizzare testo in modi potenti e flessibili.

La potenza dei metodi per stringhe si apprezza particolarmente nell'elaborazione del testo, dove possiamo concatenare più operazioni in sequenza:

```
# Pulizia e parsing di dati testuali
testo_grezzo = "  python, java, C++  "
linguaggi = testo_grezzo.strip().lower().split(", ")
print(linguaggi)  # ['python', 'java', 'c++']
```

Principi guida per l'uso dei metodi

Quando lavori con i metodi in Python, tieni a mente questi principi:

1. **Verifica la mutabilità**: prima di usare un metodo, capisci se stai lavorando con un tipo mutabile o immutabile

2. **Controlla il valore restituito**: alcuni metodi restituiscono un nuovo oggetto, altri modificano l'oggetto esistente

3. **Leggi la documentazione**: quando sei in dubbio, consulta la documentazione ufficiale

Nota

La documentazione ufficiale di Python (`https://docs.python.org/3/library/stdtypes.html`) contiene l'elenco completo di tutti i metodi disponibili per ciascun tipo di dato, con spiegazioni dettagliate ed esempi.

Di seguito vengono riportate le tabelle con i principali metodi per la manipolazione delle stringhe in Python, si tratta di metodi preinstallati in Python:

Tabella 1: Metodi essenziali per le stringhe (`str`)

Metodo	Descrizione	Esempio
`upper()`	Converte in maiuscolo	`"python".upper() → "PYTHON"`
`lower()`	Converte in minuscolo	`"Python".lower() → "python"`
`strip()`	Rimuove spazi agli estremi	`" Python ".strip() → "Python"`
`replace(old, new)`	Sostituisce le occorrenze	`"Hello".replace("l", "x") → "Hexxo"`
`split(sep)`	Divide in una lista	`"a,b,c".split(",") → ["a", "b", "c"]`
`join(iterable)`	Unisce una lista in stringa	`"-".join(["a", "b"]) → "a-b"`
`find(sub)`	Trova posizione sottostringa	`"Python".find("th") → 2`

3.1 Operatori

In questa sottosezione andremo a visionare i principali Operatori presenti su Python, possiamo considerarli come "strumenti" che permettono di manipolare dati, eseguire calcoli e prendere decisioni all'interno del codice. Gli opreatori sono essenzialmente simboli speciale che indicano al computer di eseguire operazioni specifiche sui valori (o operandi) a cui sono applicati.

Alcuni di essi possono risultare alquanto ambigui per la loro essenza e semplicità ma rappresentano un valore chiave per la comprensione dell'azione che bisogna eseguire.

- **Operatori aritmetici**: permettono di eseguire calcoli matematici sui numeri
- **Operatori di confronto**: consentono di comparare valori e restituiscono risultati booleani (vero o falso)
- **Operatori logici**: permettono di combinare condizioni booleane e sono essenziali nelle strutture decisionali

Accanto agli operatori, le funzionalità di input e output rappresentano le "porte" attraverso cui un programma comunica con l'utente o con l'ambiente esterno. Attraverso queste funzioni, i programmi possono acquisire dati, presentare risultati e interagire con chi li utilizza.

Gli esempi che seguono illustrano la sintassi e l'uso pratico di ciascun tipo di operatore e delle funzioni di input/output di base in Python.

3.1.1 Operatori aritmetici

```
a = 10
b = 3

somma = a + b          # 13
differenza = a - b     # 7
prodotto = a * b       # 30
divisione = a / b      # 3.3333... (divisione che restituisce float)
div_intera = a // b    # 3 (divisione intera)
modulo = a % b         # 1 (resto della divisione)
potenza = a ** b       # 1000 (a elevato alla b)
```

3.1.2 Operatori di confronto

```
a == b    # Uguaglianza (False)
a != b    # Disuguaglianza (True)
a > b     # Maggiore (True)
a < b     # Minore (False)
a >= b    # Maggiore o uguale (True)
a <= b    # Minore o uguale (False)
```

3.1.3 Operatori logici

```
x = True
y = False

x and y    # False (AND logico)
x or y     # True (OR logico)
not x      # False (NOT logico)
```

3.2 Input e Output

```python
# Output sulla console
print("Ciao, mondo!")

# Input da tastiera
nome = input("Come ti chiami? ")
print(f"Ciao, {nome}!")

# Formattazione delle stringhe
eta = 25
messaggio = f"Ho {eta} anni."
print(messaggio)
```

4 Approfondimento delle Variabili

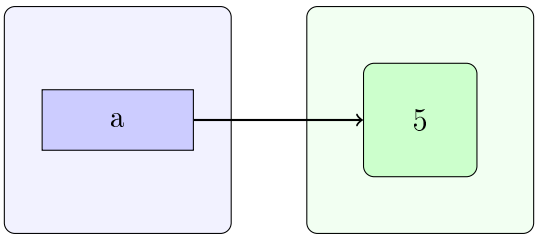
4.1 Il modello di riferimento in Python

In Python, le variabili funzionano in modo diverso rispetto ad altri linguaggi di programmazione. Anziché essere "contenitori" che memorizzano direttamente un valore, le variabili Python sono meglio descritte come "etichette" o "nomi" che fanno riferimento a oggetti in memoria.

Per comprendere veramente questo concetto, visualizziamo come Python gestisce le variabili in memoria.

```
# Creazione di una variabile
a = 5
```

Quando eseguiamo questa istruzione, ecco cosa succede:

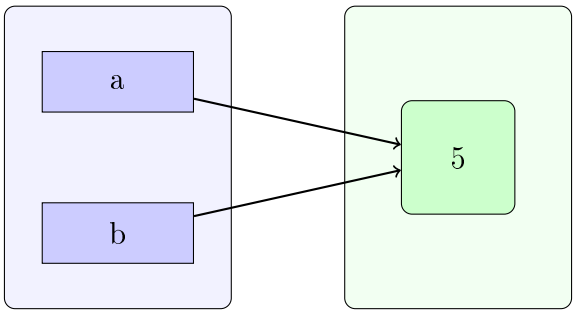


Python crea un oggetto intero con valore 5 in memoria e associa la variabile `a` a questo oggetto. La variabile `a` non "contiene" 5, ma "punta a" o "fa riferimento a" l'oggetto 5 in memoria.

4.2 Assegnazione di variabili

Vediamo cosa succede quando creiamo una seconda variabile e le assegniamo il valore della prima:

```
a = 5
b = a  # b ora fa riferimento allo stesso oggetto di a
```



Ora sia `a` che `b` fanno riferimento allo stesso oggetto in memoria. Non viene creata una copia del valore.

4.3 Tipi immutabili vs mutabili

In Python, la distinzione tra tipi di dati mutabili e immutabili rappresenta uno dei concetti fondamentali che influenzano profondamente il comportamento del codice. Questa distinzione, apparentemente semplice, ha implicazioni significative sulla manipolazione dati, sull'ottimizzazione della memoria.

La natura dell'immutabilità e della mutabilità

Quando parliamo di **immutabilità**, ci riferiamo a oggetti che, una volta creati, non possono essere modificati nel loro stato interno. Qualsiasi opeazione sembra modificare un oggetto immutabile in realtà crea un nuovo oggetto con i valori aggiornati. È come se questi oggetti portassero con sé un cartello invisibile che dice: "Guardami, ma non toccarmi". Al contrario, gli oggetti mutabili possono essere alterati dopo la loro creazione. Il loro stato interno può cambiare mentre l'identità dell'oggetto rimane la stessa. Possiamo pensare agli oggetti mutabili come a contenitori il cui contenuto può essere modificato, rimosso o sostituito senza dover sostituire l'intero contenitore.

La classificazione In Python

Python organizza i suoi tipi di dati nativi in queste due categorie:

- **Tipi Immutabili:**
  - **Numeri** (*int, float, complex*)
  - **Stringe** (*str*)
  - **Tuple** (*tuple*)
  - **Frozen sets** (*frozenset*)
  - **Booleani** (*bool*)
  - **None** (*NoneType*)
- **Tipi Mutabili:**
  - **Liste** (*list*)
  - **Dizionari** (*dict*)
  - **Set** (*set*)

Un modello concettuale

Per comprendere meglio questa distinzione, possiamo utilizzare un'analogia:

Gli oggetti **immutabili** sono come documenti stampati su carta: se desideri modificare una parola, devi creare un nuovo documento. Non puoi alterare l'originale senza lasciare tracce evidenti.

Gli oggetti **mutabili** sono invece come documenti digitali prima della stampa, alla quale puoi ancora aggiungere, eliminare o modificare i contenuti, mantenendo lo stesso file.

Differenza tra tipi immutabili e mutabili

In Python, la distinzione tra tipi immutabili e mutabili è fondamentale per comprendere il comportamento delle variabili:

- **Tipi immutabili** (non modificabili dopo la creazione):
  - `int`: Numeri interi
  - `float`: Numeri in virgola mobile
  - `bool`: Valori booleani (True/False)
  - `str`: Stringhe di testo
  - `tuple`: Sequenze immutabili di elementi
  - `frozenset`: Insiemi immutabili
- **Tipi mutabili** (modificabili dopo la creazione):
  - `list`: Liste di elementi
  - `dict`: Dizionari (coppie chiave-valore)
  - `set`: Insiemi di elementi unici

Questa distinzione influenza profondamente come le variabili si comportano quando vengono assegnate, passate a funzioni o modificate.

4.3.1 Comportamento con tipi immutabili

Quando lavoriamo con tipi immutabili, qualsiasi operazione che sembra modificare il valore in realtà crea un nuovo oggetto in memoria:

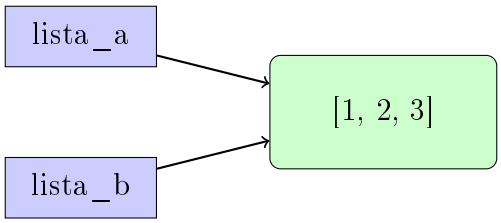
```
a = 5
b = a
a = 10  # Questo NON modifica l'oggetto originale, ma crea un nuovo riferimento
print(b)  # Output: 5 (b continua a riferirsi all'oggetto originale)
```

4.3.2 Comportamento con tipi mutabili

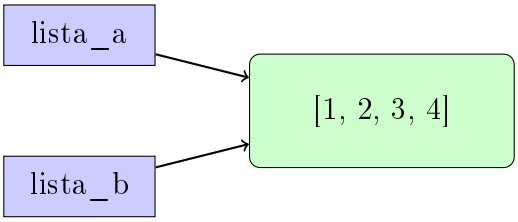
I tipi mutabili come le liste si comportano diversamente:

```
lista_a = [1, 2, 3]
lista_b = lista_a
lista_a.append(4)  # Questo modifica l'oggetto originale a cui puntano entrambe le variabili
```

Stato iniziale:



Dopo `lista_a.append(4)`:



In questo caso, `lista_a.append(4)` modifica direttamente l'oggetto lista in memoria. Poiché sia `lista_a` che `lista_b` puntano allo stesso oggetto, entrambe "vedono" la modifica.

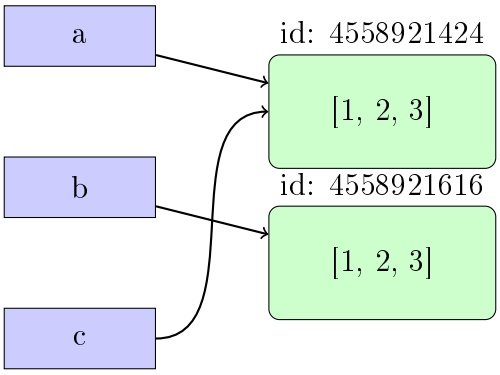
4.4 Verifica dell'identità degli oggetti

Python fornisce due operatori per confrontare le variabili:

consultare per ripasso la sezione degli Operatori logici: ***Operatori logici***

- `==` confronta i valori (uguaglianza)
- `is` confronta le identità (se le variabili puntano allo stesso oggetto)

```
a = [1, 2, 3]
b = [1, 2, 3]  # Una nuova lista con gli stessi valori
c = a          # Riferimento alla stessa lista

print(a == b)  # True (stesso valore)
print(a is b)  # False (oggetti diversi)
print(a is c)  # True (stesso oggetto)

# Possiamo verificare l'identita anche con id()
print(id(a))  # Un numero unico che identifica l'oggetto
print(id(b))  # Un numero diverso
print(id(c))  # Stesso numero di id(a)
```



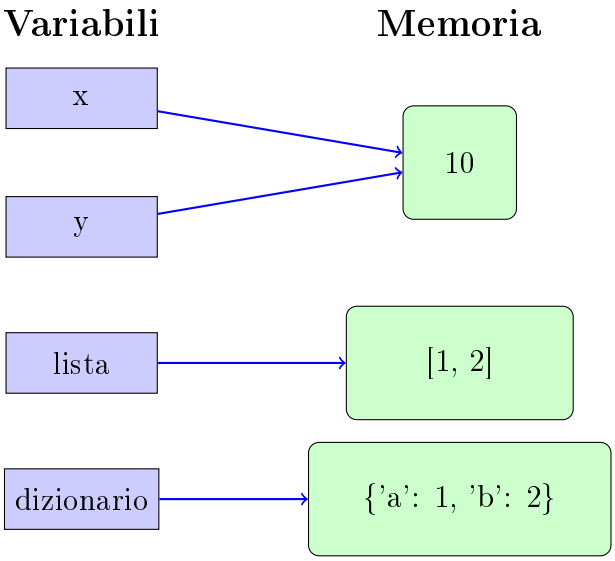
4.5 Esercizio pratico: tracciare le variabili

Esaminiamo un esercizio completo che mostra come tracciare le variabili in Python. Prevedere l'output di questo programma richiede di comprendere come Python gestisce i riferimenti in memoria.

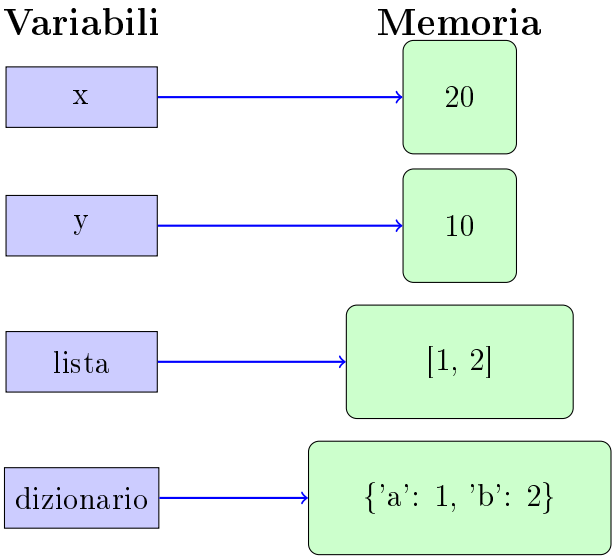
```
# Stato iniziale
x = 10
y = x
lista = [1, 2]
dizionario = {'a': 1, 'b': 2}

# Modifichiamo le variabili
x = 20
lista.append(3)
dizionario['c'] = 3
```

```python

# Verifichiamo lo stato finale
print(x)        # Output: 20
print(y)        # Output: 10
print(lista)    # Output: [1, 2, 3]
print(dizionario)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

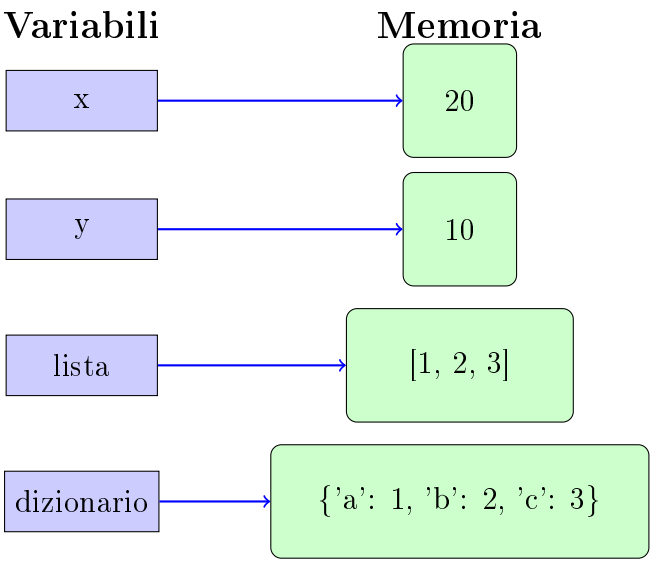
Visualizziamo lo stato della memoria in diversi momenti dell'esecuzione:
Stato iniziale:



Dopo x = 20:



Stato finale (dopo tutte le modifiche):



Nota come:

- `x` punta a un nuovo oggetto (20), mentre `y` continua a puntare all'oggetto originale (10)
- Gli oggetti mutabili (`lista` e `dizionario`) sono stati modificati in memoria, non è stato creato un nuovo oggetto
- Tutte le variabili che puntano a oggetti mutabili vedono le modifiche apportate

Questo esercizio illustra la differenza fondamentale tra l'assegnazione di nuovi valori (che crea nuovi riferimenti per tipi immutabili) e la modifica di oggetti mutabili (che altera gli oggetti esistenti in memoria).

4.6 Esercizi Riassuntivi

In questa sezione andremo a svolgere alcuni esercizi per familiarizzare con i concetti affrontati; a fine sezione si troveranno gli esercizi completi con le soluzioni spiegate passo passo.

4.6.1 Esercizio 1: Implementazione Input ed Output

Dall'esercizio che abbiamo affrontato nella prima sezione "Il tuo primo programma Python"
- implementare la possibilità di chiedere all'utente la sua età
- calcolare l'anno di nascita
- stampare il messaggio in modo che comprenda nome anno di nascita.

```
# Il mio primo programma Python
print("Hello, World!")  # Stampa un messaggio a schermo

# Richiesta di input all'utente
nome = input("Come ti chiami? ")
print(f"Ciao {nome}, benvenuto nel mondo di Python!")

# Aggiungi a questo script la possibilita di chiedere all'utente la sua eta

#Calcolare l'anno di nascita approssimativo (anno corrente - eta)

#Stampare un messaggio personalizzato che includa il nome e l'anno di nascita
```

4.6.2 Esercizio 2: Identificazione degli errori

In questo esercizio viene fornita una serie di variabili la quale denominazione presenta degli errori, identificare gli errori e fornire la denominazione corretta secondo le best practice discusse nella sezione: "Dichiarazione e Assegnazione delle Variabili".

```
1nome = "Paolo"
nome-cognome = "Mario Rossi"
class = "Principiante"

# Di seguito fornisci le tue correzioni
```

4.6.3 Esercizio 3: Operazioni Matematiche

In questo esercizio dobbiamo dichiarare due variabili numeriche e applicare tutte le possibili operazioni matematiche (addizione, sottrazione, moltiplicazione, divisione, divisione intera, modulo, potenza) e stampare i relativi risultati.
Le sezioni da ripassare per questo esercizio sono:
- Operatori aritmetici

```
# Dichiara le 2 variabili
numero1 = # immetti un tuo numero a piacere
numero2 = # immetti un tuo numero a piacere

    # Esegui tutte le operazioni matematiche  (addizione, sottrazione, moltiplicazione, divisione, divisione intera, modulo, potenza)

somma =
sottrazione =
divisione =
moltiplicazione =
divisione_intera =
modulo =
potenza =
```

4.6.4 Esercizio 4: Operatori logici

In questo esercizio dovremo capire come funzionano gli operatori logici, l'esercizio fornito ci offre un'espressione logica che dovremo descrivere manualmente per capire il suo significato ed il significato del suo risultato.
Bonus: prova differenti versioni dell'esercizio

```
a = 5
b = 10
c = 15
risultato = (a < b) and (b <= c) or not (a == c)
print(risultato)

# Spiega perche il risultato del print e' True
```

4.6.5 Esercizio 5: MiniCalcolatrice

In questo esercizio proveremo a creare una mini calcolatrice, chiederemo all'utente di inserire 2 numeri, dopodiché svolgeremo delle operazioni con i numeri chiesti all'utente.
Le sezioni da ripassare per questo esercizio sono le seguenti:
- Input e Output
- Operatori aritmetici

```
# Chiediamo all'utente il primo numero
primo_numero =
# Chiediamo il secondo numero
secondo_numero =

# Esegui le operazioni matematiche
somma =
divisione =
moltiplicazione =

# Stampa i risultati delle operazioni
```

Input e Tipi di Dati: Informazioni Importanti

- **Comportamento del metodo `input()`:**
  - Il metodo `input()` restituisce **sempre** una stringa di testo, anche quando l'utente digita numeri
  - Per una mini-calcolatrice, è necessario convertire questi input in numeri per eseguire calcoli matematici
  - Se non convertiamo il tipo, l'operazione `"5" + "3"` produrrà `"53"` (concatenazione di stringhe) invece di `8` (somma numerica)
- **Conversione dell'input in valori numerici:**
  - `int(input("Messaggio: "))` converte l'input in un numero intero (senza decimali)
  - `float(input("Messaggio: "))` converte l'input in un numero decimale (con punto)
  - Esempio pratico: `numero1 = float(input("Inserisci il primo numero: "))`
- **Suggerimento per la mini-calcolatrice:**
  - Per le operazioni matematiche, è consigliabile usare `float()` invece di `int()` per gestire correttamente i numeri decimali
  - Assicurati che l'utente inserisca effettivamente dei numeri e non del testo
  - Per la divisione, considera cosa succede se l'utente tenta di dividere per zero

4.6.6 Esercizio 6: Mutabilità e Immutabilità

In questo esercizio dovremo identificare quale e come le variabili mutano o restano immutate. Descrivi a mano quali variabili mutano e quali no ed il loro perché e come viene allocata la memoria. Le sezioni da ripassare per questo esercizio sono le seguenti:
- Tipi immutabili vs mutabili
- Approfondimento delle Variabili

```python
# Esempio 1
x = "hello"
y = x
x = "world"
print(y)  # Cosa verra stampato?

# Esempio 2
lista_a = [1, 2, 3]
lista_b = lista_a
lista_a[0] = 99
print(lista_b)  # Cosa verra stampato?
```

4.7 Risoluzione Esercizi sui Fondamentali

4.7.1 Risoluzione Esercizio 1: *Esercizio 1: Implementazione Input ed Output*

```python
# Il mio primo programma Python
print("Hello, World!")  # Stampa un messaggio a schermo

# Richiesta di input all'utente
nome = input("Come ti chiami? ")
print(f"Ciao {nome}, benvenuto nel mondo di Python!")
```

```

# Aggiungi a questo script la possibilita di chiedere all'utente la sua eta
eta_utente = int(input("Digita la tua eta:\n"))

#Calcolare l'anno di nascita approssimativo (anno corrente - eta)
anno_corrente = int(input("Digita l'anno corrente"))
anno_nascita = anno_corrente - eta_utente


#Stampare un messaggio personalizzato che includa il nome e l'anno di nascita
print(f"Ciao {nome}, sono nato/a il {anno_nascita}")
```

4.7.2 Risoluzione Esercizio 2: *Esercizio 2: Identificazione degli errori*

```
1nome = "Paolo"                    # Inizia con un numero
nome-cognome = "Mario Rossi"       # Usa il trattino
class = "Principiante"             # Usa una parola riservata

# Di seguito fornisci le tue correzioni
nome = "Paolo"
nome_cognome = "Mario Rossi"
classe = "Principiante"
```

4.7.3 Risoluzione Esercizio 3: *Esercizio 3: Operazioni Matematiche*

```
# Dichiara le 2 variabili
numero1 = 12
numero2 = 56

    # Esegui tutte le operazioni matematiche  (addizione, sottrazione, moltiplicazione, divisione,
↪ divisione intera, modulo, potenza)

somma = numero1 + numero2
sottrazione = numero1 - numero2
divisione = numero1 / numero2
moltiplicazione = numero1 * numero2
divisione_intera = numero1 // numero2
modulo = numero1 % numero2
potenza = numero1**numero2


# Stampiamo i risultati
print(somma, " somma")                              # Output 68
print(sottrazione, " sottrazione")                  # Output -44
print(divisione, " divisione")                      # Output 0.21428571428571427
print(moltiplicazione, " moltiplicazione")          # Output 672
print(divisione_intera, " divisione con intero")    # Output 0
print(modulo, " resto della divisione")             # Output 12
print(potenza, " potenza di numero1 alla numero2")  # Output 2.71 10^60
```

4.7.4 Risoluzione Esercizio 4: *Esercizio 4: Operatori logici*

```
a = 5
b = 10
c = 15
risultato = (a < b) and (b <= c) or not (a == c)
print(risultato)
```

```

# Spiega perche il risultato del print e' True

"""
Per comprendere perche il risultato dell'espressione e True, dobbiamo analizzarla passo per passo seguendo l'ordine di valutazione degli operatori in Python.
Prima valutiamo tutte le espressioni di confronto:
(a < b) significa (5 < 10), che e True
(b <= c) significa (10 <= 15), che e True
(a == c) significa (5 == 15), che e False
Poi applichiamo l'operatore not:
not (a == c) diventa not False, che e True
A questo punto la nostra espressione equivale a:
True and True or True
Ora applichiamo l'operatore and, che ha precedenza rispetto a or:
True and True risulta in True
Infine, applichiamo l'operatore or:
True or True risulta in True
In questo caso specifico, sia la parte a sinistra dell'operatore or ((a < b) and (b <= c)) sia la parte a destra (not (a == c)) sono entrambe True. Di conseguenza, l'intero risultato e True.
E interessante notare che anche se la parte sinistra fosse False, il risultato complessivo sarebbe comunque True perche la parte destra e True e l'operatore or restituisce True se almeno uno dei suoi operandi e True.

"""
```

4.7.5 Risoluzione Esercizio 5: *Esercizio 5: MiniCalcolatrice*

```
# Chiediamo all'utente il primo numero
primo_numero = int(input("Digita un numero intero\n"))
# Chiediamo il secondo numero
secondo_numero = int(input("Digita un secondo numero intero\n"))

# Esegui le operazioni matematiche
somma = primo_numero + secondo_numero
divisione = primo_numero / secondo_numero
moltiplicazione = primo_numero * secondo_numero

# Stampa i risultati delle operazioni
print(f"Somma: {somma}")                        # Output 16
print(f"Divisione: {divisione}")                # Output 3
print(f"Moltiplicazione: {moltiplicazione}")    # Output 48


# Avremmo pututo usare il modulo float al posto di int per avere numeri decimali
```

4.7.6 Risoluzione Esercizio 6: *Esercizio 6: Mutabilità e Immutabilità*

```

# Esempio 1
x = "hello"
y = x
x = "world"
print(y)  # Cosa verra stampato?

# Esempio 2
lista_a = [1, 2, 3]
lista_b = lista_a
lista_a[0] = 99
```

```
print(lista_b)  # Cosa verra stampato?

"""

Analisi passo per passo

Creazione della prima variabile:
x = "hello"
In questo passaggio, Python crea un oggetto stringa con valore "hello" in memoria e assegna alla variabile x un riferimento a quest'oggetto. Le stringhe in Python sono tipi immutabili, quindi una volta creato l'oggetto "hello", non potra essere modificato.
Creazione della seconda variabile:
y = x
Qui, Python non crea un nuovo oggetto stringa. Invece, fa puntare la variabile y allo stesso oggetto a cui punta x. Quindi ora sia x che y fanno riferimento allo stesso oggetto stringa "hello" in memoria.
Riassegnazione della prima variabile:
x = "world"
Poiche le stringhe sono immutabili, Python non puo modificare l'oggetto "hello" esistente. Invece, crea un nuovo oggetto stringa con valore "world" e fa puntare x a questo nuovo oggetto. Importante: la variabile y continua a puntare all'oggetto originale "hello".
Stampa del valore:
print(y)
Quando stampiamo y, otteniamo il valore dell'oggetto a cui y fa riferimento, che e ancora "hello".

Risultato
La risposta alla domanda "Cosa verra stampato?" e: hello
Spiegazione del comportamento
Questo esempio illustra perfettamente il comportamento dei tipi immutabili in Python:

Le stringhe sono tipi immutabili, quindi non possono essere modificate dopo la creazione.
Quando assegniamo un valore a una variabile (x = "hello"), creiamo un riferimento a un oggetto.
Quando copiamo una variabile in un'altra (y = x), entrambe le variabili puntano allo stesso oggetto.
Quando riassegniamo un nuovo valore a una variabile (x = "world"), non modifichiamo l'oggetto originale ma creiamo un nuovo oggetto e facciamo puntare la variabile al nuovo oggetto.
La variabile che non e stata riassegnata (y) continua a puntare all'oggetto originale.

Questo comportamento e diverso da quello dei tipi mutabili (come liste o dizionari), dove la modifica di un oggetto attraverso una variabile si rifletterebbe su tutte le variabili che puntano a quell'oggetto.




--------- Rappresentazione Grafica della memoria ---------


               Dopo x = "hello":
               Variabili       Memoria
               ---------       -------
                   x    ---->  "hello"


               Dopo y = x:
               Variabili       Memoria
               ---------       -------
                   x    ---->  "hello"
                   y    ----/


               Dopo x = "world":
```

```
                Variabili       Memoria
                ---------       -------
                    x    ---->  "world"
                    y    ---->  "hello


"""
```

5 Presentazione Liste, Tuple, Dizionari, Set

Questo capitolo esplorerà le quattro principali strutture dati incorporate nel linguaggio: **liste, tuple, dizionari, set**. Queste strutture rappresentano modi diversi di organizzare le informazioni, ognuna di esse con caratteristiche uniche e caso d'uso specifico.

Il modo di presentarle sarà uguale alle sezioni precedenti, quindi introduzione teorica sugli aspetti peculiari con relativi esempi, rappresentazione pratica dell'utilizzo con relativi approfondimenti sui metodi, esercizi per consolidamento della parte teorica, correlati con relative soluzioni.

Percorso di apprendimento: Strutture dati in Python

Questa è una introduzione alle strutture dati fondamentali. Nei capitoli successivi approfondiremo ciascuna struttura, esplorando algoritmi avanzati e pattern di programmazione che le utilizzano in contesti reali. →

5.1 Liste

5.1.1 Definizione e Caratteristiche Principali

Le liste sono una delle strutture dati più versatili e utilizzate in Python. Una lista è una sequenza ordinata di "**Tipi immutabili vs mutabili**" di elementi che può contenere valori di qualsiasi tipo.

Caratteristiche principali delle liste

- **Delimitate da parentesi quadre**: Le liste si creano racchiudendo gli elementi tra parentesi quadre [ ]
- **Ordinate**: Gli elementi mantengono l'ordine in cui sono stati inseriti
- **Mutabili**: È possibile modificare, aggiungere o rimuovere elementi dopo la creazione
- **Indicizzate**: Si può accedere agli elementi tramite indici numerici, partendo da 0
- **Eterogenee**: Possono contenere elementi di tipi diversi (numeri, stringhe, booleani, altre liste, ecc.)
- **Duplicati ammessi**: Possono contenere elementi ripetuti

Le liste sono estremamente utili per gestire collezioni di dati correlati, come una serie di numeri, un elenco di nomi, oppure qualsiasi altro raggruppamento di valori che devono mantenere un ordine specifico.

Esempio

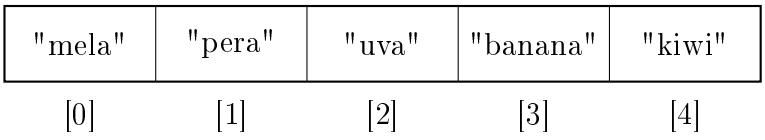
Una lista può rappresentare:

- Una lista della spesa: `["pane", "latte", "uova", "mele"]`
- Una sequenza di numeri: `[1, 2, 3, 4, 5]`
- Un insieme di valori di tipo diverso: `["Mario", 25, 1.75, True]`
- I giorni della settimana: `["Lunedì", "Martedì", "Mercoledì", "Giovedì", "Venerdì", "Sabato", "Domenica"]`

Nota

In molti altri linguaggi di programmazione, strutture simili alle liste di Python sono chiamate "array". Tuttavia, le liste Python sono più flessibili degli array tradizionali, poiché possono contenere elementi di tipi diversi e si ridimensionano automaticamente.

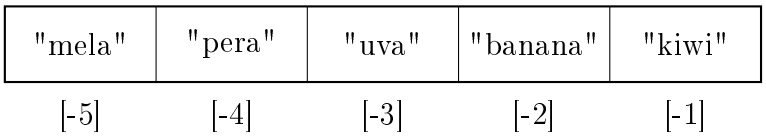
Esempio di Lista in Python



Python offre anche un secondo sistema di indicizzazione, utilizzando numeri negativi, che permette di accedere agli elementi partendo dalla fine della lista:

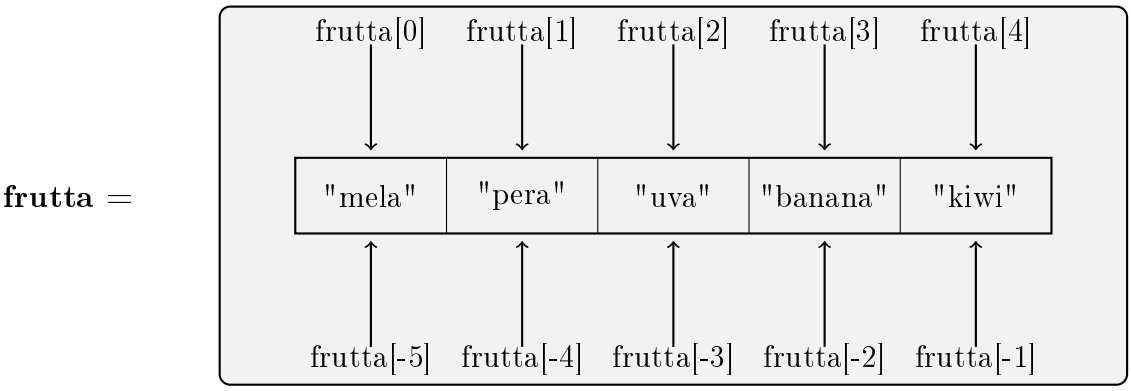
Gli indici negativi sono molto utili quando si vuole accedere agli elementi partendo dalla fine della lista, senza dover conoscere la lunghezza totale della lista. Ad esempio, `[-1]` si riferisce sempre all'ultimo elemento, indipendentemente da quanti elementi contiene la lista. Con l'unica differenza che contando al contrario non bisognerà arrivare a `[0]`. ma soltanto a `[-1]`.

Indicizzazione Negativa (dall'ultimo elemento)



Per comprendere meglio come funziona l'indicizzazione delle liste in Python, osserviamo una rappresentazione in stile Python Tutor che mostra visivamente la corrispondenza tra indici ed elementi:

Visualizzazione degli Indici in Python



Questa visualizzazione mostra:

- Come ogni elemento della lista può essere accessibile utilizzando sia un indice positivo sia un indice negativo
- `frutta[0]` e `frutta[-5]` si riferiscono entrambi al primo elemento ("mela")
- `frutta[4]` e `frutta[-1]` si riferiscono entrambi all'ultimo elemento ("kiwi")

Utilizzando la lista `frutta = ["mela", "pera", "uva", "banana", "kiwi"]`:

```
frutta[0]   # restituisce "mela" (primo elemento)
frutta[2]   # restituisce "uva" (terzo elemento)
frutta[-1]  # restituisce "kiwi" (ultimo elemento)
frutta[-2]  # restituisce "banana" (penultimo elemento)
```

A differenza di altre strutture dati che vedremo più avanti (come le tuple), le liste sono *mutabili*, il che significa che è possibile modificare il loro contenuto dopo la creazione. Questa caratteristica le rende ideali per situazioni in cui i dati devono cambiare nel tempo.

Attenzione

Gli indici delle liste in Python iniziano da 0, non da 1. Quindi, il primo elemento di una lista si trova all'indice 0, il secondo all'indice 1, e così via. Questo è un concetto fondamentale che si ritrova in molti linguaggi di programmazione.

5.2 L'Indicizzazione delle Liste in Python

5.2.1 Un modello matematico per comprendere gli indici

In Python, l'indicizzazione delle liste rappresenta uno dei concetti fondamentali che permette di accedere agli elementi di una collezione ordinata. Ciò che rende particolarmente elegante questo sistema è la possibilità di indicizzare in due direzioni: dalla sinistra verso destra (con indici positivi) e dalla destra verso sinistra (con indici negativi).

Il Modello dell'Anello Chiuso Possiamo immaginare una lista Python come un anello chiuso di elementi, simile a una struttura circolare. In matematica, quando abbiamo un insieme finito di elementi ordinati che può essere percorso ciclicamente, possiamo rappresentarlo come un insieme chiuso modulo il numero di elementi.

Per una lista di lunghezza n, abbiamo la seguente relazione fondamentale:

$$\boxed{\text{L'elemento all'indice } i \text{ è identico all'elemento all'indice } (i - n)}$$

In altre parole, se abbiamo una lista di 5 elementi, l'elemento all'indice 0 è lo stesso dell'elemento all'indice -5, l'elemento all'indice 1 è lo stesso dell'elemento all'indice -4, e così via.

Rappresentazione Visiva Consideriamo una lista `frutti = ["mela", "banana", "arancia", "kiwi", "uva"]`:

Indici positivi:	0	1	2	3	4
Elementi:	"mela"	"banana"	"arancia"	"kiwi"	"uva"
Indici negativi:	-5	-4	-3	-2	-1

Osserviamo che:

- L'indice 0 corrisponde a -5 $\quad (0 - 5 = -5)$
- L'indice 1 corrisponde a -4 $\quad (1 - 5 = -4)$
- L'indice 2 corrisponde a -3 $\quad (2 - 5 = -3)$
- L'indice 3 corrisponde a -2 $\quad (3 - 5 = -2)$
- L'indice 4 corrisponde a -1 $\quad (4 - 5 = -1)$

La Formula Matematica Per qualsiasi lista di lunghezza n, la relazione tra indici positivi e negativi segue questa formula:

$$\text{indice\_negativo} = \text{indice\_positivo} - n \tag{1}$$
$$\text{indice\_positivo} = \text{indice\_negativo} + n \tag{2}$$

Questa relazione algebrica è esattamente come operare in aritmetica modulare, dove i numeri "avvolgono" dopo aver raggiunto una certa dimensione.

5.2.2 Vantaggi dell'indicizzazione bidirezionale

L'indicizzazione negativa risolve un problema comune: accedere agli ultimi elementi di una lista senza conoscerne la lunghezza esatta.

Pensa a quante volte in programmazione abbiamo bisogno dell'ultimo elemento o degli ultimi elementi. Utilizzando indici negativi:

- `lista[-1]` → sempre l'ultimo elemento
- `lista[-2]` → sempre il penultimo elemento
- `lista[-3]` → sempre il terzultimo elemento

Senza questa funzionalità, dovremmo scrivere `lista[len(lista)-1]` per ottenere l'ultimo elemento, una sintassi molto più verbosa.

5.2.3 Esempi pratici

```
numeri = [10, 20, 30, 40, 50]

# Accesso con indici positivi
print(numeri[0])  # 10 (primo elemento)
print(numeri[2])  # 30 (terzo elemento)

# Accesso con indici negativi
print(numeri[-1])  # 50 (ultimo elemento)
print(numeri[-3])  # 30 (terzultimo elemento)

# Relazione matematica
for i in range(len(numeri)):
    print(f"numeri[{i}] = numeri[{i-len(numeri)}] = {numeri[i]}")
```

Questo codice dimostra che `numeri[i]` è sempre uguale a `numeri[i-len(numeri)]`.

5.2.4 Il concetto di insieme ciclico chiuso

Matematicamente, possiamo vedere l'indicizzazione delle liste come un'operazione su un insieme ciclico chiuso $\mathbb{Z}/n\mathbb{Z}$ (i numeri interi modulo n). Ogni elemento ha esattamente due identificatori: uno positivo e uno negativo.

Questo concetto di ciclicità è particolarmente potente perché ci permette di:

1. **Navigare bidirezionalmente**: possiamo pensare alla lista sia in avanti che all'indietro
2. **Esprimere relazioni posizionali**: possiamo esprimere la posizione di un elemento rispetto all'inizio (indici positivi) o rispetto alla fine (indici negativi)
3. **Sfruttare la modularità**: l'indicizzazione si comporta come un orologio circolare, dove dopo il 12 torniamo all'1

Nota

È importante notare che, nonostante questa apparente "ciclicità" nell'indicizzazione, Python non permette di usare indici arbitrariamente grandi per "avvolgere" la lista. Se proviamo ad accedere a `lista[len(lista)]` o `lista[-len(lista)-1]`, otterremo un errore `IndexError`. La ciclicità esiste solo nel modello concettuale, non nell'implementazione pratica dell'accesso agli elementi.

5.2.5 Conclusione

L'indicizzazione delle liste in Python può essere vista come un'applicazione dell'aritmetica modulare, dove l'elemento alla posizione i coincide con l'elemento alla posizione $i - n$. Questa doppia indicizzazione offre una flessibilità notevole, permettendo di riferirsi agli elementi sia dalla prospettiva dell'inizio della lista che dalla sua fine.

Questa caratteristica, apparentemente semplice, è un esempio della filosofia di Python: rendere le operazioni comuni immediate ed eleganti, riducendo la verbosità e aumentando la leggibilità del codice.

5.2.6 Sintassi di Base

Creazione di Liste In Python, esistono diversi modi per creare una lista. Ecco i metodi più comuni:

```
# Lista vuota
lista_vuota = []

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

# Lista di stringhe
nomi = ["Anna", "Bruno", "Carlo", "Daria"]

# Lista con elementi di tipi diversi
lista_mista = [1, "Python", 3.14, True]

# Lista nidificata (liste all'interno di liste)
matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Creazione di lista mediante la funzione list()
caratteri = list("Python")  # Crea la lista ['P', 'y', 't', 'h', 'o', 'n']
numeri_sequenza = list(range(1, 6))  # Crea la lista [1, 2, 3, 4, 5]
```

Nota

La funzione `list()` può convertire in lista qualsiasi oggetto che sia *iterabile* (ovvero che possa essere percorso elemento per elemento). Nel secondo esempio, `range(1, 6)` crea una sequenza di numeri da 1 a 5, che viene convertita in lista.

Best Practice per Denominare le Liste Una buona denominazione delle variabili è fondamentale per la leggibilità del codice. Per le liste, seguire queste linee guida può rendere il codice più comprensibile: come già affrontato per le variabili, breve ripasso;*Dichiarazione e Assegnazione delle Variabili*

Best practice per i nomi delle liste

- **Usa nomi al plurale**: Poiché le liste contengono tipicamente più elementi, è consigliabile usare nomi plurali (es. `studenti` anziché `studente`)
- **Scegli nomi descrittivi**: Il nome dovrebbe suggerire cosa contiene la lista
  - Buono: `prezzi_prodotti`, `nomi_studenti`, `temperature_giornaliere`
  - Da evitare: `l`, `lista1`, `dati`, `cose`
- **Segui le convenzioni Python**: Usa `snake_case` (parole in minuscolo separate da underscore)
- **Coerenza nei tipi**: Se la lista contiene elementi dello stesso tipo, considera di indicarlo nel nome (es. `numeri_pari`, `stringhe_input`)

Esempio

Confronto tra nomi poco chiari e nomi appropriati:

Nome sconsigliato	Nome consigliato	Contenuto
`l` o `lst`	`studenti`	Lista di nomi di studenti
`nums`	`voti_esame`	Lista di voti numerici
`x`	`coordinate_x`	Lista di valori delle coordinate x
`lista_dati`	`temperature_mensili`	Lista di temperature

5.2.7 Accesso agli Elementi

Per accedere a un elemento specifico di una lista, si utilizzano gli indici racchiusi tra parentesi quadre:

```
colori = ["rosso", "verde", "blu", "giallo", "viola"]

# Accesso al primo elemento (indice 0)
primo_colore = colori[0]  # "rosso"

# Accesso al terzo elemento (indice 2)
terzo_colore = colori[2]  # "blu"

# Accesso all'ultimo elemento (indice -1)
ultimo_colore = colori[-1]  # "viola"

# Accesso al penultimo elemento (indice -2)
penultimo_colore = colori[-2]  # "giallo"
```

Attenzione

Se si tenta di accedere a un indice che non esiste, Python genererà un errore `IndexError`. Ad esempio, `colori[10]` genererà un errore se la lista `colori` ha meno di 11 elementi.

5.2.8 Modifica degli Elementi

Essendo le liste mutabili, è possibile modificare i singoli elementi assegnando nuovi valori:

```
# Lista iniziale
frutta = ["mela", "banana", "pera", "arancia"]

# Modifica del secondo elemento
frutta[1] = "kiwi"
# Ora frutta e' ["mela", "kiwi", "pera", "arancia"]

# Modifica dell'ultimo elemento usando indice negativo
frutta[-1] = "limone"
# Ora frutta e' ["mela", "kiwi", "pera", "limone"]
```

È anche possibile modificare più elementi contemporaneamente utilizzando lo slicing:

```
numeri = [1, 2, 3, 4, 5]

# Sostituzione di piu' elementi
numeri[1:4] = [20, 30, 40]
# Ora numeri e' [1, 20, 30, 40, 5]

# Si puo' anche sostituire con un numero diverso di elementi
numeri[1:4] = [200, 300]
# Ora numeri e' [1, 200, 300, 5]
```

5.2.9 Slicing di Liste

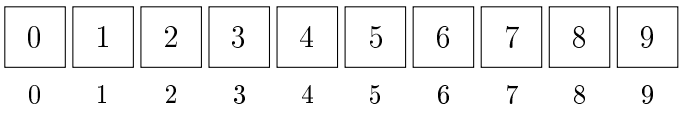
Lo slicing permette di estrarre una porzione di una lista, creando una nuova lista. La sintassi generale è `lista[inizio:fine:passo]`, dove:

- `inizio` è l'indice da cui iniziare (incluso)
- `fine` è l'indice dove terminare (escluso)
- `passo` è il passo con cui avanzare (opzionale, default 1)

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

# Elementi dal secondo al quarto (indici 1, 2, 3)
sottosequenza = numeri[1:4]  # [1, 2, 3]

# Primi tre elementi
inizio = numeri[:3]  # [0, 1, 2]

# Elementi dal quarto fino alla fine
fine = numeri[3:]  # [3, 4, 5, 6, 7, 8, 9]

# Ultimi tre elementi
ultimi = numeri[-3:]  # [7, 8, 9]

# Tutti gli elementi tranne gli ultimi due
senza_ultimi = numeri[:-2]  # [0, 1, 2, 3, 4, 5, 6, 7]

# Elementi con indici pari (passo 2)
pari = numeri[::2]  # [0, 2, 4, 6, 8]

# Elementi in ordine inverso
inverso = numeri[::-1]  # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Visualizzazione dello Slicing

Lista originale:



numeri[1:4] = 1 2 3

numeri[:3] = 0 1 2

numeri[-3:] = 7 8 9

numeri[::2] = 0 2 4 6 8

Nota

Lo slicing crea sempre una nuova lista, non modifica quella originale. Se vuoi modificare la lista originale, devi effettuare un'assegnazione esplicita (es. `lista = lista[1:4]`).

5.2.10 Operazioni e Metodi Base

Le liste in Python offrono numerosi metodi incorporati che permettono di manipolarle in modo efficiente. Questi metodi sono funzionalità già pronte che consentono di aggiungere, rimuovere, ordinare e modificare gli elementi di una lista senza dover scrivere codice complesso.

5.2.11 Metodi per Aggiungere Elementi

Python fornisce diversi metodi per aggiungere elementi a una lista:

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

# Metodo append() - aggiunge un elemento alla fine della lista
frutti.append("arancia")
# Ora frutti e' ["mela", "banana", "arancia"]

# Metodo insert() - inserisce un elemento in una posizione specifica
frutti.insert(1, "pera")  # Inserisce "pera" all'indice 1
# Ora frutti e' ["mela", "pera", "banana", "arancia"]

# Metodo extend() - aggiunge tutti gli elementi di un'altra lista
altri_frutti = ["kiwi", "ananas"]
frutti.extend(altri_frutti)
# Ora frutti e' ["mela", "pera", "banana", "arancia", "kiwi", "ananas"]
```

La differenza tra `append()` e `extend()` è importante:

- `append(x)` aggiunge `x` come un singolo elemento, anche se `x` è una lista
- `extend(lista)` aggiunge ogni singolo elemento di `lista` alla lista originale

Esempio:

```
lista1 = [1, 2, 3]
lista2 = [4, 5]

lista1.append(lista2)  # lista1 diventa [1, 2, 3, [4, 5]]
                       # La lista2 e' aggiunta come unico elemento

lista3 = [1, 2, 3]
lista3.extend(lista2)  # lista3 diventa [1, 2, 3, 4, 5]
                       # Gli elementi di lista2 sono aggiunti singolarmente
```

5.2.12 Metodi per Rimuovere Elementi

Per rimuovere elementi da una lista, Python offre vari metodi:

```
numeri = [10, 20, 30, 40, 50, 30]

# Metodo remove() - rimuove la prima occorrenza di un valore
numeri.remove(30)  # Rimuove il primo 30 trovato
# Ora numeri e' [10, 20, 40, 50, 30]

# Metodo pop() - rimuove e restituisce l'elemento all'indice specificato
elemento = numeri.pop(1)  # Rimuove e restituisce il secondo elemento (20)
print(elemento)  # 20
# Ora numeri e' [10, 40, 50, 30]

# Metodo pop() senza argomenti - rimuove e restituisce l'ultimo elemento
ultimo = numeri.pop()  # Rimuove e restituisce l'ultimo elemento (30)
print(ultimo)  # 30
# Ora numeri e' [10, 40, 50]
```

```python

# Istruzione del - rimuove l'elemento o la sezione specificata
del numeri[0]  # Rimuove il primo elemento
# Ora numeri e' [40, 50]

# Metodo clear() - svuota completamente la lista
numeri.clear()
# Ora numeri e' []
```

5.2.13 Gestione degli Errori

Quando si utilizzano i metodi di rimozione, è importante considerare alcuni possibili errori:

- `remove(x)` genera un errore `ValueError` se l'elemento `x` non è presente nella lista
- `pop(i)` e `del lista[i]` generano un errore `IndexError` se l'indice `i` è fuori range
- Una volta rimosso un elemento, gli indici degli elementi successivi si spostano per riflettere la nuova posizione

Ecco come appaiono questi errori quando si verificano:

```python
# Esempio di ValueError con remove()
numeri = [1, 2, 3]
numeri.remove(5)  # Provando a rimuovere un elemento che non esiste
```

Output del terminale

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

```python
# Esempio di IndexError con pop()
numeri = [1, 2, 3]
numeri.pop(10)  # Provando ad accedere a un indice che non esiste
```

Output del terminale

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

5.2.14 Metodi per Ottenere Informazioni

Per ottenere informazioni su una lista e i suoi elementi:

```python
colori = ["rosso", "verde", "blu", "verde", "giallo"]

# Funzione len() - restituisce il numero di elementi
lunghezza = len(colori)
print(lunghezza)  # 5

# Metodo count() - conta quante volte un elemento appare nella lista
occorrenze_verde = colori.count("verde")
print(occorrenze_verde)  # 2

# Metodo index() - trova l'indice della prima occorrenza di un elemento
indice_blu = colori.index("blu")
print(indice_blu)  # 2
```

```
14
15 # E' anche possibile specificare l'intervallo di ricerca per index()
16 indice_verde_secondo = colori.index("verde", 2)  # Cerca da indice 2 in poi
17 print(indice_verde_secondo)  # 3
```

5.2.15 Esempi di index() e count()

I metodi `index()` e `count()` sono utili per trovare e contare elementi in una lista. Vediamo come funzionano con un esempio visuale:

```
1 colori = ["rosso", "verde", "blu", "verde", "giallo"]
2
3 # Contare le occorrenze di "verde"
4 occorrenze = colori.count("verde")    # Risultato: 2
5
6 # Trovare l'indice della prima occorrenza di "blu"
7 indice = colori.index("blu")          # Risultato: 2
8
9 # Trovare l'indice della seconda occorrenza di "verde"
10 indice_secondo_verde = colori.index("verde", 2)  # Risultato: 3
11                                                  # Comincia a cercare dall'indice 2
```

5.2.16 Metodi per Ordinamento e Inversione

Per ordinare o invertire gli elementi di una lista:

```
1 numeri = [3, 1, 4, 1, 5, 9, 2]
2
3 # Metodo sort() - ordina la lista in ordine crescente (modifica la lista originale)
4 numeri.sort()
5 print(numeri)  # [1, 1, 2, 3, 4, 5, 9]
6
7 # Metodo sort() con parametro reverse - ordine decrescente
8 numeri.sort(reverse=True)
9 print(numeri)  # [9, 5, 4, 3, 2, 1, 1]
10
11 # Funzione sorted() - crea una nuova lista ordinata senza modificare l'originale
12 parole = ["zebra", "albero", "cane"]
13 parole_ordinate = sorted(parole)
14 print(parole)          # ["zebra", "albero", "cane"] (non modificata)
15 print(parole_ordinate) # ["albero", "cane", "zebra"]
16
17 # Metodo reverse() - inverte l'ordine degli elementi
18 numeri.reverse()
19 print(numeri)  # [1, 1, 2, 3, 4, 5, 9]
```

Nota

I metodi `sort()` e `reverse()` modificano direttamente la lista originale e non restituiscono una nuova lista. Se hai bisogno di mantenere la lista originale, creando una copia, usa la funzione `sorted()` per l'ordinamento o lo slicing `[::-1]` per l'inversione.

```
1 # Metodo copy() - crea una copia superficiale della lista
2 originale = [1, 2, 3]
3 copia = originale.copy()
4 originale.append(4)
5 print(copia)  # [1, 2, 3] (la copia non e' influenzata)
6
7 # Funzione list() - un altro modo per creare una copia
8 altra_copia = list(originale)
9 print(altra_copia)  # [1, 2, 3, 4]
```

Riassunto dei metodi principali delle liste

Metodo	Descrizione
`append(x)`	Aggiunge l'elemento `x` alla fine della lista
`insert(i, x)`	Inserisce l'elemento `x` alla posizione `i`
`extend(iterable)`	Aggiunge tutti gli elementi dell'iterabile alla fine della lista
`remove(x)`	Rimuove la prima occorrenza dell'elemento `x`
`pop([i])`	Rimuove e restituisce l'elemento alla posizione `i` (o l'ultimo se `i` non è specificato)
`clear()`	Rimuove tutti gli elementi dalla lista
`index(x[, start[, end]])`	Restituisce l'indice della prima occorrenza di `x` (opzionalmente cercando solo dalla posizione `start` alla `end`)
`count(x)`	Restituisce il numero di occorrenze di `x` nella lista
`sort([key=None, reverse=False])`	Ordina gli elementi della lista (in-place)
`reverse()`	Inverte l'ordine degli elementi nella lista (in-place)
`copy()`	Restituisce una copia superficiale della lista

Ecco un esempio completo che mostra l'uso di diversi metodi delle liste:

```
# Creazione di una lista di numeri
numeri = [5, 2, 8, 1, 9, 3]

# Manipolazione della lista
numeri.append(7)           # Aggiunge 7 alla fine: [5, 2, 8, 1, 9, 3, 7]
numeri.insert(0, 0)        # Inserisce 0 all'inizio: [0, 5, 2, 8, 1, 9, 3, 7]
numeri.remove(9)           # Rimuove il 9: [0, 5, 2, 8, 1, 3, 7]
elemento = numeri.pop(3)   # Rimuove e restituisce l'elemento all'indice 3 (8)
                           # numeri ora e' [0, 5, 2, 1, 3, 7]

# Informazioni sulla lista
lunghezza = len(numeri)    # 6
occorrenze = numeri.count(5)  # 1
posizione = numeri.index(3)   # 4

# Ordinamento e inversione
numeri.sort()              # [0, 1, 2, 3, 5, 7]
numeri.reverse()           # [7, 5, 3, 2, 1, 0]
```

5.3 Comportamento in Memoria delle Liste

Come già affrontato nel capitolo precedente:"Approfondimento delle Variabili", dove viene affrontata la questione della gestione della memoria per le variabili, anche in questo capitolo tratteremo la gestione della memoria con le liste.

Esploreremo infatti come Python gestisce le liste in memoria e come operare per ottimizzarle al meglio, e quali metodi usare a seconda dei casi d'uso. Esploreremo concetti come **Copia profonda** e **Copia superficiale**, **Liste annidate**, il tutto relegato a come Python gestisce la memoria in questi contesti.

5.3.1 Variabili come Riferimenti

Come già visto nella sezione dedicata alle "variabili" e ai riferimenti in memoria, anche per le liste il concetto di uso dei riferimenti in memoria è lo stesso, tranne per alcune eccezioni che vedremo in seguito.

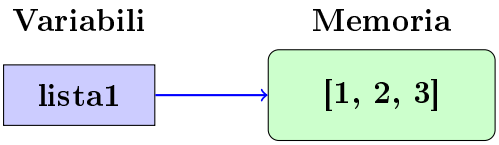
```
# Creazione di una lista
lista1 = [1, 2, 3]

# id() mostra l'identificatore univoco dell'oggetto in memoria
print(id(lista1))  # Ad esempio: 140424434847752
```

5.3.2 Assegnazione e Riferimenti Multipli

Quando assegni una lista a una seconda variabile, **non viene creata una nuova lista**, ma entrambe le variabili fanno riferimento alla stessa lista in memoria. Modificare la lista attraverso una variabile influenzerà anche l'altra.

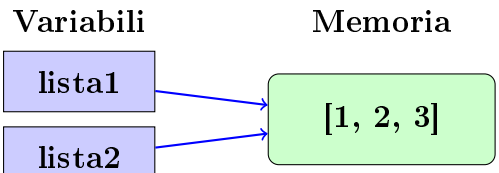
```
# Creazione di una lista
lista1 = [1, 2, 3]

# Assegnazione a una nuova variabile
lista2 = lista1  # Non crea una copia!

# Modifica attraverso lista1
lista1.append(4)

# Entrambe le liste sono state modificate
print(lista1)  # [1, 2, 3, 4]
print(lista2)  # [1, 2, 3, 4]

# Verifica che puntano allo stesso oggetto
print(id(lista1) == id(lista2))  # True
```

Fase 1: Creazione di lista1



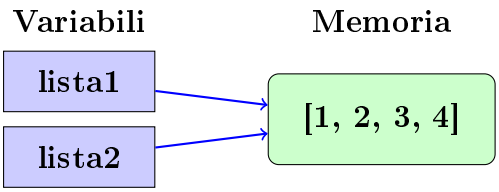
Inizialmente, solo lista1 punta all'oggetto lista in memoria

Fase 2: Assegnazione a lista2



Quando eseguiamo lista2 = lista1, non viene creata una nuova lista. Entrambe le variabili puntano allo stesso oggetto in memoria.

Fase 3: Dopo lista1.append(4)



Quando modifichiamo l'oggetto attraverso lista1, la modifica è visibile anche attraverso lista2 perché entrambe puntano allo stesso oggetto.

5.3.3 Creazione di Copie Indipendenti

Abbiamo appena visto come, in Python, l'assegnazione di una lista ad una nuova variabile non crei una copia, ma semplicemente un nuovo riferimento allo stesso oggetto. Questo comportamento, sebbene efficiente in termini di memoria, può portare a modifiche indesiderate quando entrambe le variabili vengono utilizzate in parti diverse del codice.

Per evitare questo problema, è spesso necessario creare **copie indipendenti** delle liste, in modo che le modifiche apportate a una non influenzino l'altra. Python offre diversi metodi per creare tali copie, ciascuno con caratteristiche specifiche.

In questa sezione, esploreremo:

- I metodi principali per creare copie superficiali di liste
- La differenza tra copia superficiale e copia profonda
- Quando e come utilizzare ciascun tipo di copia
- Gli errori comuni da evitare quando si lavora con le copie

Comprendere queste tecniche è fondamentale per scrivere codice robusto e prevedibile, specialmente in programmi di grandi dimensioni o quando si lavora con strutture dati complesse.

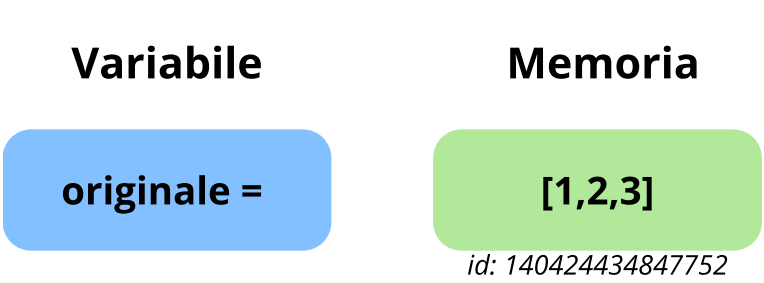
Metodi per la Creazione di Copie Indipendenti Per creare copie indipendenti di una lista, Python mette a disposizione tre metodi principali, tutti ugualmente efficaci per liste contenenti tipi di dati semplici. Vediamo questi metodi in azione e osserviamo come, dopo la modifica della lista originale, le copie mantengano i loro valori iniziali, dimostrando la loro indipendenza:

```
# Creazione di una lista
originale = [1, 2, 3]
# Tre modi per creare copie
copia1 = originale.copy()       # Metodo copy() disponibile da Python 3.3
copia2 = originale[:]           # Slicing completo (funziona in tutte le versioni)
copia3 = list(originale)        # Costruttore list()
# Modifica dell'originale
originale.append(4)
# Le copie non sono influenzate
print(originale)  # [1, 2, 3, 4]
print(copia1)     # [1, 2, 3]
print(copia2)     # [1, 2, 3]
print(copia3)     # [1, 2, 3]
```

Questi tre metodi creano una *copia superficiale* (shallow copy) della lista, sufficiente per la maggior parte degli usi con elementi semplici. È importante notare che, sebbene questi metodi producano lo stesso risultato in questo esempio, ciascuno ha peculiarità che potrebbero essere rilevanti in contesti più complessi, specialmente quando la lista contiene oggetti annidati.

Fase 1: Creazione della lista originale

originale = [1,2,3]
Crea una lista e assegna un riferimento alla variabile

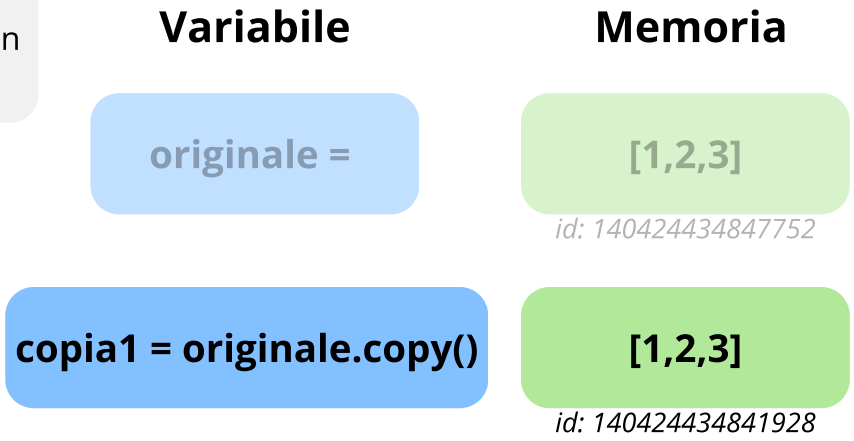


Una variabile in Python è un riferimento a un oggetto in memoria. Quando creiamo una lista, questa viene allocata in memoria e la variabile contiene solo un riferimento (*puntatore)* a quella posizione.

Nota: Le liste in Python sono oggetti mutabili. Il loro contenuto può essere modificato dopo la crezione.

Fase 2: Creazione di copia1 con metodo copy()

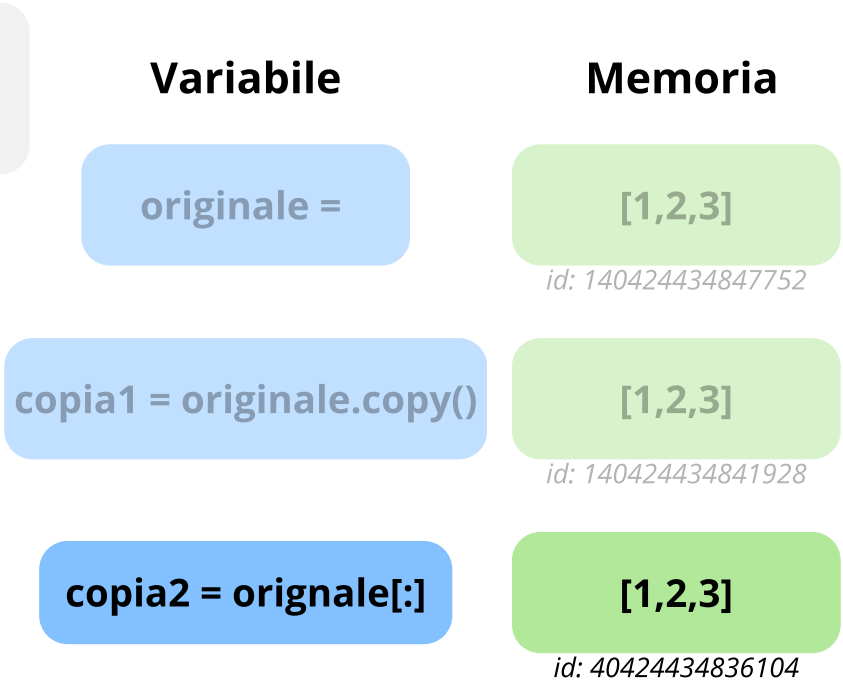
copia1 = originale.copy()
.copy() crea una nuova lista con gli stessi elementi originale



Il metodo *copy()* crea una nuova lista in memoria con gli stessi elementi dell'originale. Gli **ID** diversi confermano che *originale* e *copia1* sono oggetti distinti.

Fase 3: Creazione di copia2 con metodo slicing [:]

copia2 = originale[:]
Lo *slicing[:]* copia tutti gli elementi dall'indice 0 alla fine



Lo *slicing[:]* è un altro metodo per creare una copia della lista. Come *copy(),* crea un nuovo oggetto lista indipendente dall'originale con gli stessi valori.

Fase 4: Creazione di copia3 con costruttore list()

copia3 = list(originale)
list() accetta qualsiasi iterabile e crea una nuova lista con i suoi elementi

Il costruttore list() crea una copia indipendente della lista. A differenza di .copy() che è un metodo specifico delle liste, list() può trasformare qualsiasi oggetto iterabile (come tuple, set, stringhe, generatori) in una nuova lista

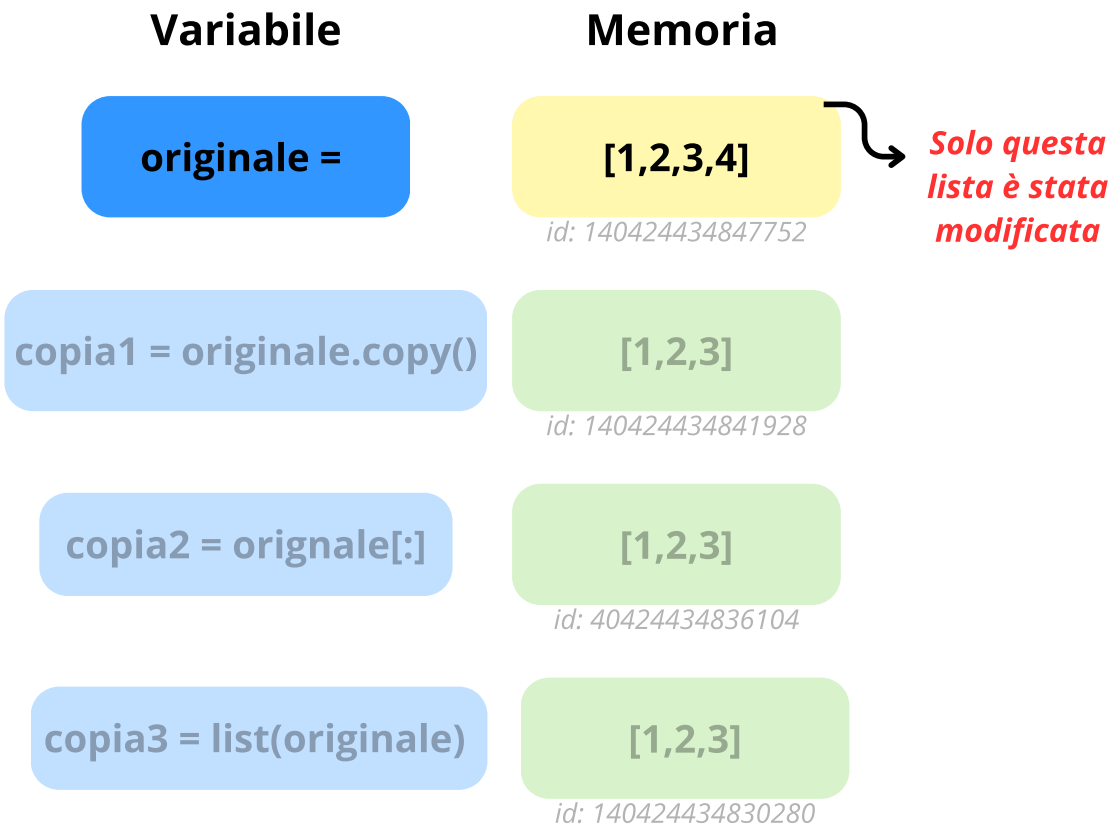
Confronto tra i metodi di copia:

Metodo	Sitnassi	Casi d'uso/Note
.copy()	*nomelista.copy()*	**Esplicito, solo Python 3+**
Slicing *[:]*	*nomelista[:]*	**Idiomatico, tutte le versioni**
list()	*list(nomelista)*	**Utile per convertire iterabili**

Nota importante: Tutti e tre i metodi creano copie superficiali (shallow copies). Questo è sufficiente per liste con elementi immutabili (numeri, stringhe, tuple), ma può causare problemi con elementi mutabili annidati (altre liste, dizionari).

Fase 5: Effetto delle modifiche sulla lista originale

originale.append(4)
Abbiamo modificato la lista originale aggiungendo l'elemento 4.



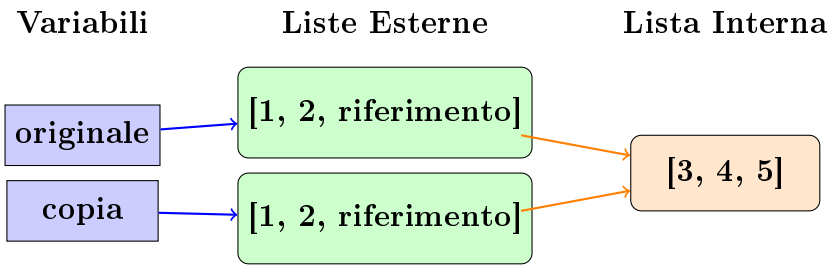
La modifica della lista originale con ***apppend()*** influisce solo su quella lista. Le altre copie rimangono invariate, dimostrando che tutte le tecniche di copia creano oggetti indipendenti della lista ***originale***

Il Problema delle Liste Annidate

I metodi di copia visti sopra creano "copie superficiali" (shallow copies). Se la lista contiene altre liste (o oggetti mutabili), i metodi di copia standard copiano solo i riferimenti a queste liste interne, non le liste stesse. In pratica ogni modifica apportata alla lista originale, tramite *shallow copies* influenzerà la sua copia.

Affrontiamo graficamente ciò che avviene con le liste annidate.

```
# Lista con una lista annidata
originale = [1, 2, [3, 4]]

# Creazione di una copia superficiale
copia = originale.copy()

# Modifica della lista interna nell'originale
originale[2].append(5)

# La modifica si riflette anche nella copia!
print(originale)  # [1, 2, [3, 4, 5]]
print(copia)      # [1, 2, [3, 4, 5]]
```



Entrambe le liste esterne contengono riferimenti alla stessa lista interna. La modifica della lista interna è visibile attraverso entrambi i riferimenti.

Il Problema della Copia profonda in Dettaglio Come abbiamo accennato sopra *"Metodi per la Creazione di Copie Indipendenti"*, sono metodi che creano una copia superficiale. Questo approccio presenta limitazioni significative quando si lavora con strutture di dati annidate:

Comprensione del Problema Le copie superficiali duplicano solo la struttura esterna, mantendendo i riferimenti agli stessi oggetti interni.

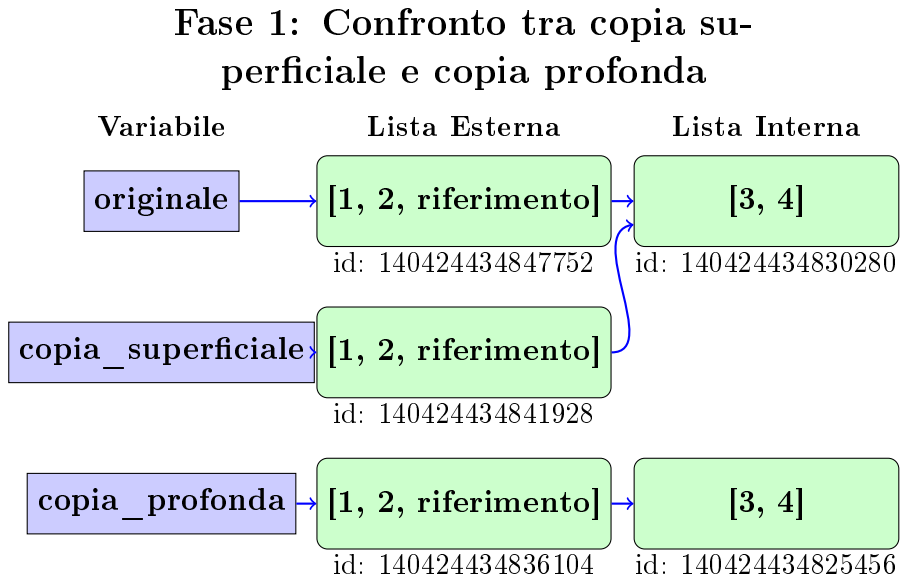


Figura 1: Confronto tra copia superficiale e copia profonda prima della modifica

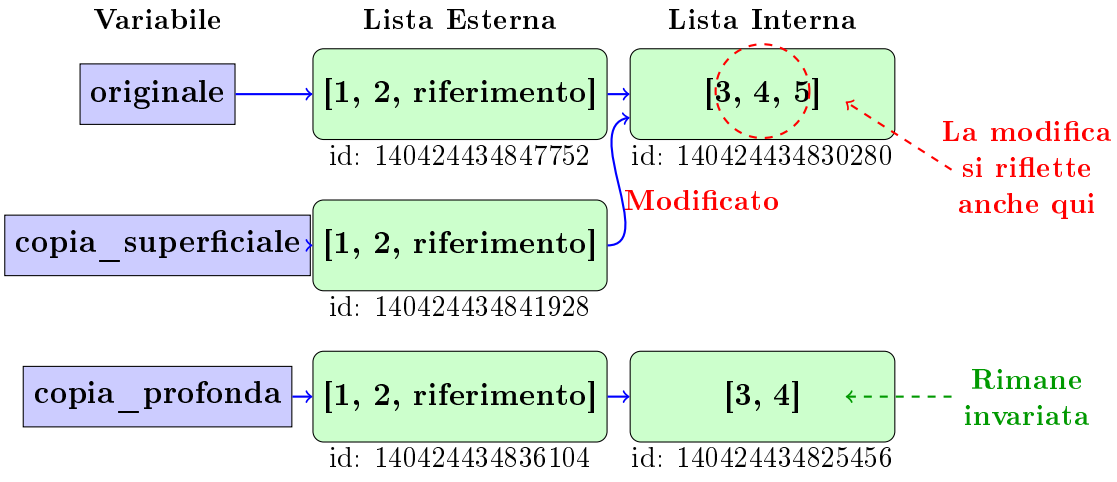
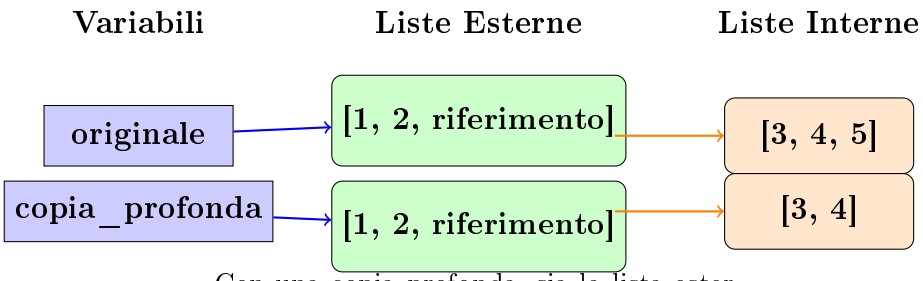


Figura 2: Effetto delle modifiche: differenza tra copia superficiale e profonda

Copie Profonde Per creare una "copia profonda" (deep copy) che copia anche gli oggetti annidati, si può utilizzare il modulo `copy`:

```
import copy

# Lista con una lista annidata
originale = [1, 2, [3, 4]]

# Creazione di una copia profonda
copia_profonda = copy.deepcopy(originale)

# Modifica della lista interna nell'originale
originale[2].append(5)

# La copia profonda non e' influenzata
print(originale)      # [1, 2, [3, 4, 5]]
print(copia_profonda) # [1, 2, [3, 4]]
```



Con una copia profonda, sia la lista esterna che quelle interne vengono copiate. Le modifiche alla lista interna dell'originale non influenzano la copia.

5.3.4 Confronto tra Liste: '==' vs 'is'

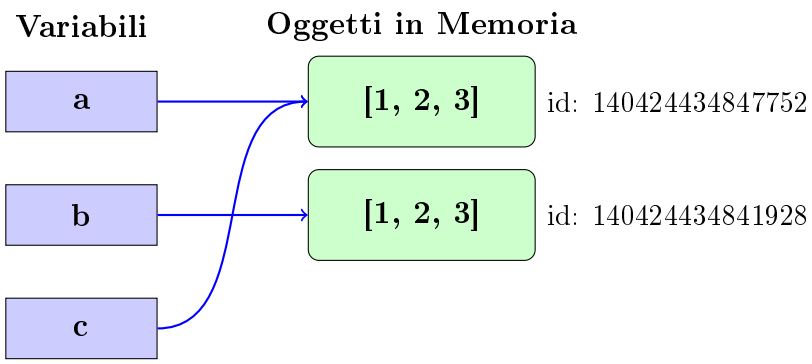
Python fornisce due modi per confrontare le liste:

- L'operatore == confronta il contenuto delle liste (uguaglianza di valore)
- L'operatore `is` confronta gli identificatori degli oggetti (uguaglianza di identità)

```
a = [1, 2, 3]
b = [1, 2, 3]  # Una nuova lista con gli stessi valori
c = a          # Riferimento alla stessa lista

print(a == b)  # True - stesso contenuto
print(a is b)  # False - oggetti diversi in memoria
print(a is c)  # True - stesso oggetto in memoria
```

5.3.5 Rappresentazione Visiva della Memoria

```
a = [1, 2, 3]
b = [1, 2, 3]  # Una nuova lista con gli stessi valori
c = a          # Riferimento alla stessa lista

print(a == b)  # True - stesso contenuto
print(a is b)  # False - oggetti diversi in memoria
print(a is c)  # True - stesso oggetto in memoria

# Visualizzazione degli id per conferma
print(id(a))  # Ad esempio: 140424434847752
print(id(b))  # Diverso da a
print(id(c))  # Stesso di a
```

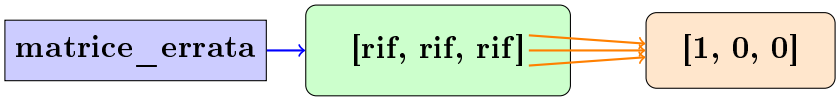


`a == b` è `True` (stesso contenuto)
`a is b` è `False` (oggetti diversi)
`a is c` è `True` (stesso oggetto)

Errore Comune: Liste Annidate Moltiplicate Un errore comune quando si lavora con liste annidate è la creazione di matrici usando la moltiplicazione di liste:

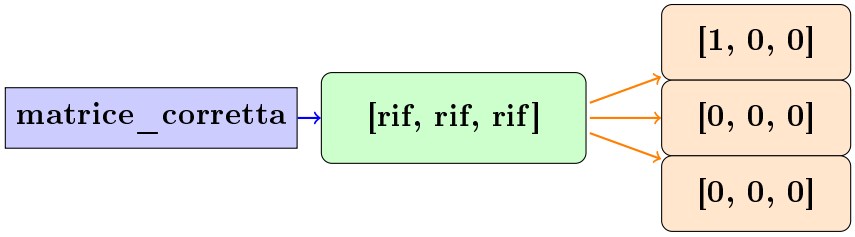
```
# Tentativo ERRATO di creare una matrice 3x3 di zeri
matrice_errata = [[0] * 3] * 3
print(matrice_errata)  # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

# Modifica di un solo elemento
matrice_errata[0][0] = 1
print(matrice_errata)  # [[1, 0, 0], [1, 0, 0], [1, 0, 0]]  Oops!
```

Cosa è successo? L'espressione '[[0] * 3] * 3' crea una lista contenente tre riferimenti alla stessa lista interna. Quando modifichi un elemento in una "riga", la modifica appare in tutte le righe perché puntano allo stesso oggetto.



Soluzione Corretta Il modo corretto per creare liste annidate indipendenti è utilizzare una comprensione di lista:

```
# Modo CORRETTO:
matrice_corretta = [[0 for _ in range(3)] for _ in range(3)]
matrice_corretta[0][0] = 1
print(matrice_corretta)  # [[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```



In questo caso, ogni riga è un oggetto lista separato e indipendente. Modificando un elemento in una riga, le altre righe non sono influenzate.

Implicazioni Pratiche Comprendere come funzionano i riferimenti in Python ha importanti implicazioni pratiche:

- **Modifiche indesiderate**: Modificare una lista attraverso un riferimento influenza tutti i riferimenti alla stessa lista

- **Copie superficiali non sufficienti**: Quando la lista contiene oggetti mutabili, una copia superficiale potrebbe non essere sufficiente
- **Confronto con `is`**: Utilizzare `is` quando si intende confrontare il contenuto può portare a risultati inaspettati

5.3.6 Implicazioni Pratiche

Comprendere come funzionano i riferimenti in Python ha importanti implicazioni pratiche:

Errori comuni con i riferimenti alle liste

- **Modifiche indesiderate**: Modificare una lista attraverso un riferimento influenza tutti i riferimenti alla stessa lista
- **Copie superficiali non sufficienti**: Quando la lista contiene oggetti mutabili, una copia superficiale potrebbe non essere sufficiente
- **Confronto con `is`**: Utilizzare `is` quando si intende confrontare il contenuto può portare a risultati inaspettati

5.4 Esercizi Riassuntivi

Come per la sezione sulle *variabili*, anche qui andiamo a svolgere alcuni esercizi per familiarizzare con i concetti affrontati, anche qui a fine sezione si troveranno gli esercizi svolti con le soluzioni spiegate passo passo.

5.4.1 Esercizio 1: Concetti Base e Indicizzazione

In questo esercizio andremo ad approfondire il concetto di indicizzazione, vedi capitolo L'Indicizzazione delle Liste in Python

Traccia gli Indici: Data la lista *colori = ["rosso", "verde", "blu", "giallo", "viola"]*

- Scrivi l'indice positivo e negativo per accedere a ciascun elemento
- Qual è il valore di colori[1]? E di colori[-2]
- Se volessi accedere a *"verde"* usando un indice negativo, quale sarebbe?

```
#Dichiariamo la lista colori
colori = ["rosso", "verde", "blu", "giallo", "viola"]

#Inserisci gli indici positivi e negativi di tutti gli elementi della lista

#Valore di colori[1]

#Valore di colori[-2]

#A quale indice corrisponde "verde" e quale sarebbe il suo corrispettivo indice negativo?
```

5.4.2 Esercizio 2: Operazioni Base sulle Liste

Modifica gli Elementi inizializziamo una lista punteggi.

```
punteggi = [85, 92, 78, 90, 88]
# - Sostituisci il terzo elemento con 95

# - Aggiungi 100 alla fine della lista

# - Inserisci 82 nella seconda posizione
```

Concatenazione e Ripetizione

```
# Scrivi il risultato di queste operazioni:

lista1 = [1, 2, 3]
lista2 = [4, 5]
risultato1 = lista1 + lista2
risultato2 = lista1 * 3
```

Per questo esercizio, ripassare le sezioni:

- Metodi per Aggiungere Elementi
- Metodi per Rimuovere Elementi

5.4.3 Esercizio 3: Slicing delle Liste

Estrazioni con Slicing

```
# Data una lista
lettere = ["a", "b", "c", "d", "e", "f", "g"]

# - Estrarre i primi due elementi

```

```
# - Estrai gli ultimi due elementi

# - Estrai gli elementi dallaa posizione 2 alla 5 (inclusa)

# - Estrai tutti gli elementi tranne il primo e l'ultimo
```

Slicing Avanzato

```
# Prevedi il risultato di questo codice:

sequenza = [10, 20, 30, 40, 50, 60, 70]
parte1 = sequenza[1:5]
parte2 = sequenza[::2]
parte3 = sequenza[5:1:-1]
```

Per questo esercizio, ripassare le sezioni:

- Slicing di Liste

5.4.4 Esercizio 4: Metodi delle Liste

Applicazione dei metodi base

```
# Data una lista
frutta = ["mela", "banana", "ciliegia"]

# - Usa il metodo appropriato per aggiungere "arancia" alla fine

# - Usa il metodo corretto per trovare la posizione di "banana"

# - Usa un metodo per contare quante volte appare "mela"

# - Usa un metodo per rimuovere "ciliegia"
```

Ordinamento e Inversione:

```
# Data una lista di numeri
numeri = [42, 8, 16, 23, 4, 15]

# - Ordina la lista in modo crescente

# - Inverti l'ordine degli elementi

# - Qual e' la differenza tra usare il metodo sort() e la funzione sorted()
```

Per questo esercizio ripassare le sezioni:

- Metodi per Ordinamento e Inversione
- Esempi di index() e count()
- Accesso agli Elementi
- Metodi per Ottenere Informazioni

5.4.5 Esercizio 5: Comportamento in Memoria delle Liste

Riferimenti e Copie

```
# Analizza e predici l'Output

a = [1, 2, 3]
b = a
b[0] = 10
print(a)
```

Creazione di Copie Indipendenti:

```
# Quali tra questi metodi creanno una copia indipendente?

lista2 = lista1

lista2 = lista1[:]

lista2 = list(lista1)

lista2 = lista1.copy()
```

Identità vs Uguaglianza:

```
# Predici il risultato di questi confronti:

lista1 = [1, 2, 3]
lista2 = [1, 2, 3]
lista3 = lista1

print(lista1 == lista2)
print(lista1 is lista2)
print(lista1 is lista3)
```

Per questo esercizio ripassare le sezioni:

- Comportamento in Memoria delle Liste

5.4.6 Esercizio 6: Analisi di Codice e Risoluzione Problemi:

Trova l'errore

```
# Indentifica e correggi l'errore in questo codice:

colori = ["rosso", "verde", "blu"]
colori[3] = "giallo"  # Aggiungi giallo alla lista
print(colori[4])      # Stampa il quinto elemento
```

Tracciamento della Memoria:

```
# Dissegna un diagramma di memoria che mostri lo stato delle varaibili dopo ogni istruzione

lista1 = [10, 20]
lista2 = lista1
lista1.append(30)
lista3 = lista1[:]
lista3[0] = 99
```

Test di Comprensione

```
# Cosa Stampa questo codice

a = [1, 2, 3]
b = [a, a]
b[0][1] = 10
print(b)
print(a)
```

Per questo esercizio ripassare le sezioni:

- Comportamento in Memoria delle Liste

5.5 Risoluzione Esercizi Sulle Liste

5.5.1 Risoluzione Esercizio 1: Esercizio 1: Concetti Base e Indicizzazione

```python
# Data la lista colori = ["rosso", "verde", "blu", "giallo", "viola"]

#Inserisci gli indici positivi e negativi di tutti gli elementi della lista
    #Per risolvere questo quesito potremmo prima vedere quanto e' lunga la lista
    print(len(colori))

    #In questo modo conosceremo quanti elementi sono presenti nella lista e potremo numerarli in entrambi i versi

    # Formula: indice_negativo = -lunghezza + indice_positivo

    # "rosso": indice positivo 0, indice negativo = -5 + 0 = -5
    # "verde": indice positivo 1, indice negativo = -5 + 1 = -4
    # "blu": indice positivo 2, indice negativo = -5 + 2 = -3
    # "giallo": indice positivo 3, indice negativo = -5 + 3 = -2
    # "viola": indice positivo 4, indice negativo = -5 + 4 = -1

#Valore di colori[1]
    #Dopo aver numerato gli elementi della lista ora possiamo ricavare i valori all'indice richiesto
#    - Verde
#Valore di colori[-2]
#    - Giallo
#A quale indice corrisponde "verde" e quale sarebbe il suo corrispettivo indice negativo?
#Ricaviamolo dalla numerazione svolta prima
#"verde"  - lunghezza 5 + indice positivo 1 = 1 -5 = -4
```

5.5.2 Risoluzioe Esercizio 2: Esercizio 2: Operazioni Base sulle Liste

```python
punteggi = [85, 92, 78, 90, 88]
# - Sostituisci il terzo elemento con 95

    # Prima di sostituire il numero troviamolo, stampiamo la lunghezza della lista cosi da facilitare la numerazione degli elementi.
    print(len(punteggi))
    # Ora numeramio gli elementi
    # b = 80 (30 + 50 = 80, il terzo elemento piu' l'ultimo)
    # a = 10 (il primo elemento)
    # c = 30 (il terzo elemento dalla fine)

# Ora possiamo sostituire il terzo elemento con 95
    punteggi[2] = 95

# - Aggiungi 100 alla fine della lista
    # Qui possiamo agire in diversi modi:

    #Concatenazione con una lista contenente il nuovo elemento:
        punteggi = punteggi + [100]

    #Operatore di aggiunta in-place:
        punteggi += [100]

    #Metodo extend con una lista singola:
        punteggi.extend([100])

    #Slice assignment alla fine della lista:
        punteggi[len(punteggi):] = [100]

    #Insert all'indice finale:
        punteggi.insert(len(punteggi), 100)

    # - Inserisci 82 nella seconda posizione
```

```
33
34
35 # === SECONDA PARTE ===
36
37 # Scrivi il risultato di queste operazioni:
38
39 lista1 = [1, 2, 3]
40 lista2 = [4, 5]
41 risultato1 = lista1 + lista2
42 risultato2 = lista1 * 3
43
44     # risultato1 = [1, 2, 3, 4, 5] (concatenazione delle due liste)
45     # risultato2 = [1, 2, 3, 1, 2, 3, 1, 2, 3] (ripetizione della prima lista 3 volte)
```

5.5.3 Risoluzione Esercizio 3: Esercizio 3: Slicing delle Liste

```
1  # Data una lista
2  lettere = ["a", "b", "c", "d", "e", "f", "g"]
3
4  # Primi tre elementi
5  primi_tre = lettere[:3]  # ["a", "b", "c"]
6
7  # Ultimi due elementi
8  ultimi_due = lettere[-2:]  # ["f", "g"]
9
10 # Elementi dalla posizione 2 alla 5 (inclusa)
11 dalla_2_alla_5 = lettere[2:6]  # ["c", "d", "e", "f"]
12
13 # Tutti tranne primo e ultimo
14 senza_primo_ultimo = lettere[1:-1]  # ["b", "c", "d", "e", "f"]
15
16 # === SECONDA PARTE ===
17
18 Prevedi il risultato di questo codice:
19
20 sequenza = [10, 20, 30, 40, 50, 60, 70]
21 parte1 = sequenza[1:5]
22 parte2 = sequenza[::2]
23 parte3 = sequenza[5:1:-1]
24
25 #parte1 = [20, 30, 40, 50] (elementi dall'indice 1 all'indice 4 incluso)
26 #parte2 = [10, 30, 50, 70] (ogni elemento a passo 2, partendo dall'inizio)
27 #parte3 = [60, 50, 40, 30] (dall'indice 5 all'indice 2 incluso, in ordine inverso)
```

5.5.4 Risoluzione Esercizio 4: Esercizio 4: Metodi delle Liste

```
1  # Data una lista
2  frutta = ["mela", "banana", "ciliegia"]
3
4  # - Usa il metodo appropriato per aggiungere "arancia" alla fine
5      frutta.append("arancia")
6      print(frutta)  # ["mela", "banana", "ciliegia", "arancia"]
7
8  # - Usa il metodo corretto per trovare la posizione di "banana"
9      posizione_banana = frutta.index("banana")  # 1
10
11 # - Usa un metodo per contare quante volte appare "mela"
12     occorrenze_mela = frutta.count("mela")  # 1
13
14 # - Usa un metodo per rimuovere "ciliegia"
```

```
    frutta.remove("ciliegia")
    print(frutta)  # ["mela", "banana", "arancia"]


# === SECONDA PARTE ===


# Data una lista di numeri
numeri = [42, 8, 16, 23, 4, 15]

# - Ordina la lista in modo crescente
    numeri.sort()
    print(numeri)  # [4, 8, 15, 16, 23, 42]

# - Inverti l'ordine degli elementi
    numeri.reverse()
    print(numeri)  # [42, 23, 16, 15, 8, 4]

# - Qual e' la differenza tra usare il metodo sort() e la funzione sorted()
    # Differenza tra sort() e sorted():
    # - sort() modifica la lista originale
    # - sorted() crea una nuova lista ordinata lasciando l'originale intatta

originale = [3, 1, 2]
nuova = sorted(originale)
print(originale)  # [3, 1, 2] rimane invariata
print(nuova)      # [1, 2, 3] nuova lista ordinata
```

5.5.5 Risoluzione Esercizio 5: Esercizio 5: Comportamento in Memoria delle Liste

```
# Analizza e predici l'Output

a = [1, 2, 3]
b = a
b[0] = 10
print(a)

    # Soluzione: L'output sar‘a [10, 2, 3]
    # Spiegazione: b e' un riferimento alla stessa lista di a. Modificando b, si  # modifica anche
↪ a perche' entrambe le variabili puntano allo stesso oggetto # in memoria.


# === SECONDA PARTTE ===


# Predici il risultato di questi confronti:

lista1 = [1, 2, 3]
lista2 = [1, 2, 3]
lista3 = lista1

print(lista1 == lista2)
print(lista1 is lista2)
print(lista1 is lista3)

# lista1 == lista2  -> True (confronta i contenui che sono uguali)
# lista1 is lista2 -> False (si riferiscono a oggetti diversi in memoria)
# lista1 is lista3  -> True (si riferiscono allo stesso oggetto in memoria)


lista1 == lista2
#True (confronta i contenuti, che sono uguali)
```

```
lista1 is lista2
#False (si riferiscono a oggetti diversi in memoria)
lista1 is lista3
#True (si riferiscono allo stesso oggetto in memoria)
```

5.5.6 Risoluzione Esercizio 6: Esercizio 6: Analisi di Codice e Risoluzione Problemi:

```
# Indentifica e correggi l'errore in questo codice:

colori = ["rosso", "verde", "blu"]
colori[3] = "giallo"  # Aggiungi giallo alla lista
print(colori[4])      # Stampa il quinto elemento

#Non si puo usare colori[3] = "giallo" perche' l'indice 3 e' fuori dal range (la #lista ha indici
↪ 0, 1, 2).
#Anche dopo aver aggiunto "giallo", colori[4] sarebbe fuori range.

# Correzione:
colori = ["rosso", "verde", "blu"]
colori.append("giallo")  # Corretto: aggiunge "giallo" alla fine
print(colori[3])         # Corretto: stampa il quarto elemento (giallo)



# === SECONDA PARTE ===

# Dissegna un diagramma di memoria che mostri lo stato delle varaibili dopo ogni istruzione


lista1 = [10, 20]
lista2 = lista1
lista1.append(30)
lista3 = lista1[:]
lista3[0] = 99


#lista1 = [10, 20]:

#lista1 -> [10, 20]


lista2 = lista1:

#lista1 -> [10, 20]
#lista2 -> [10, 20] (stesso oggetto di lista1)


lista1.append(30):

#lista1 -> [10, 20, 30]
#lista2 -> [10, 20, 30] (stesso oggetto di lista1)


lista3 = lista1[:]:

#lista1 -> [10, 20, 30]
#lista2 -> [10, 20, 30] (stesso oggetto di lista1)
#lista3 -> [10, 20, 30] (nuovo oggetto, copia di lista1)


lista3[0] = 99:

```

```
#lista1 -> [10, 20, 30]
#lista2 -> [10, 20, 30] (stesso oggetto di lista1)
#lista3 -> [99, 20, 30]


# === TERZA PARTE ===

# Cosa stampa questo codice
a = [1, 2, 3]
b = [a, a]
b[0][1] = 10
print(b)
print(a)


print(b): [[1, 10, 3], [1, 10, 3]]
print(a): [1, 10, 3]

#Spiegazione:

#a e' una lista [1, 2, 3]
#b e' una lista che contiene due riferimenti alla stessa lista a: [[1, 2, 3], [1, #2, 3]]
#b[0][1] = 10 modifica il secondo elemento della prima sottolista di b (che e' #a), #quindi ora a e' [1, 10, 3]
#Poiche' entrambi gli elementi di b sono riferimenti alla stessa lista a, la #modifica si riflette in entrambiprint(b): [[1, 10, 3], [1, 10, 3]]
#print(a): [1, 10, 3]
```

5.6 Tuple

In questo capitolo andremo ad esplorare cosa sono e come usare le *Tuple* in Python.

In Python una **tupla** è una collezione ordinata e **immutabile** di elementi. Questo significa che una volta creata una tupla, non è possibile modificarne il contenuto (aggiungere, rimuovere o cambiare elementi).

Le tuple sono simili alle liste, ma con questa fondamentale differenza di immutabilità. Vengono spesso utilizzate per rappresentare collezioni di elementi eterogenei che non dovrebbero cambiare, come ad esempio le coordinate di un punto (x,y) o i record di un database.

Anche in questo capitolo andremo ad affrontare i vari casi d'uso, trattandoli successivamente in ambiti più avanzati.

5.6.1 Caratteristiche Principali delle Tuple

Analizziamo le caratteristiche principali delle tuple, come già accennato sopra, le tuple sono oggetti immutabili che non possono essere modificate, ma condividono molte caratteristiche con le liste.

Caratteristiche principali delle Tuple

- **Delimitate da parentesi tonde**: Le tuple si creano racchiudendo gli elementi tra parentesi tonde ()
- **Ordinate**: Gli elementi mantengono l'ordine in cui sono stati inseriti
- **Immutabili**: Non è possibile modificare, aggiungere o rimuovere elementi dopo la creazione
- **Indicizzate**: Si può accedere agli elementi tramite indici numerici, partendo da 0, come nelle liste
- **Eterogenee**: Possono contenere elementi di tipi diversi (numeri, stringhe, booleani, altre liste, ecc.)
- **Duplicati ammessi**: Possono contenere elementi ripetuti

5.6.2 Sintassi Base delle Tuple

Come si inizializzano le tuple?

In Python esistono diversi modi per creare una tupla, Esaminiamo i più comuni.

```
# Utilizzando le parentesi tonde ():
    # questo e' il metodo piu' comune, gli elementi devono essere separati da virgole.
mia_tupla = (1,2,3,"a",True,2.3)
coordinate = (19.3,20.5)

# Senza le parentesi tonde () Usando il Tuple Packing, e' possibile creare una tupla omettendo le parentesi, assegnando una sequenza di valori separati da virgola, in questo modo Python interpreter‘a automaticamente come tuple.

packing_Tuple = 1,23,"hello"  # Equivalente a packing_Tuple = (1,23,"hello")

# Come per le liste anche per le tuple esiste il suo costruttore:  tuple(), ma ha una sintassi leggermente diversa.

tupla_da_lista = tuple([1,2,3])

# Quando eseguiamo il costruttore su una stringa trasforma la stringa in un oggetto iterabile, del tutto simile a cio' che avviene con le liste.
```

```
15
16 tupla_da_stringa = tuple("python") # Risultato: ('p', 'y', 't', 'h', 'o', 'n')
17
18
19 # Per presentare una tupla con un singolo elemento e' necessario includere una virgola finale dopo
   ↪ l'elemento. Altrimenti Python interpreter'a l'espressione come il tipo dell'elemento stesso
   ↪ racchiuso tra parentesi (ad esempio, un intero, una stringa)
20
21 tupla_singola = (5,)
22 non_tupla = (5)  # Questo per python e' un intero
```

5.6.3 Operazioni comuni con le Tuple

Anche se immutabili le tuple supportano diverse operazioni che non modificano la tupla originale, ma ne creano di nuove o ne estraggono informazioni.

5.6.4 Acccesso agli elementi (indicizzazione):

Come per le liste, l'indicizzazione per le tuple è identica; pertanto, si può ripassare il capitolo in merito: ***Capitolo Indicizzazione.***

```
1 tupla = (10, 20, 30, 40, 50)
2
3 # Accesso tramite indice positivo (da sinistra)
4 primo_elemento = tupla[0]  # 10
5
6 # Accesso tramite indice negativo (da destra)
7 ultimo_elemento = tupla[-1]  # 50
8
9 # Slicing: estrazione di sottotuple
10 sottotupla = tupla[1:4]  # (20, 30, 40)
```

5.6.5 Slicing:

Anche qui, come per le liste, è possibile effettuare lo slicing, anche per ottenere una sotto-tupla. Le modalità dello slicing sono le stesse; pertanto, è possibile fare riferimento al capitolo dedicato nelle liste: ***Capitolo dedicato allo Slicing.***

```
1 mia_tupla[inizio:fine:passo]
2
3 # Elementi dall'indice 2 all'indice 5 (escluso)
4 sotto_tupla1 = numeri[2:5]
5 print(sotto_tupla1)  # Output: (2, 3, 4)
6
7 # Elementi dall'inizio fino all'indice 4 (escluso)
8 sotto_tupla2 = numeri[:4]
9 print(sotto_tupla2)  # Output: (0, 1, 2, 3)
10
11 # Elementi dall'indice 6 fino alla fine
12 sotto_tupla3 = numeri[6:]
13 print(sotto_tupla3)  # Output: (6, 7, 8, 9)
14
15 # Tutti gli elementi (crea una copia)
16 sotto_tupla4 = numeri[:]
17 print(sotto_tupla4)  # Output: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
18
19
20 # === Esempi usando Indici Negativi ===
```

```
21
22 # Ultimi 3 elementi
23 sotto_tupla5 = numeri[-3:]
24 print(sotto_tupla5)  # Output: (7, 8, 9)
25
26 # Dal terzo all'ultimo fino al penultimo
27 sotto_tupla6 = numeri[-3:-1]
28 print(sotto_tupla6)  # Output: (7, 8)
29
30 # Dal quarto elemento fino al terz'ultimo
31 sotto_tupla7 = numeri[3:-3]
32 print(sotto_tupla7)  # Output: (3, 4, 5, 6)
33
34
35 # === Slicing con Passo, Step ===
36
37 # Ogni secondo elemento dall'inizio alla fine
38 sotto_tupla8 = numeri[::2]
39 print(sotto_tupla8)  # Output: (0, 2, 4, 6, 8)
40
41 # Ogni terzo elemento dall'indice 1 all'indice 8
42 sotto_tupla9 = numeri[1:8:3]
43 print(sotto_tupla9)  # Output: (1, 4, 7)
44
45 # Passo negativo: inversione della tupla
46 sotto_tupla10 = numeri[::-1]
47 print(sotto_tupla10)  # Output: (9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
48
49 # Elementi pari in ordine inverso
50 sotto_tupla11 = numeri[::-2]
51 print(sotto_tupla11)  # Output: (9, 7, 5, 3, 1)
```

5.6.6 Concatenazione delle Tuple

In Python è possibile effettuare la concatenazione delle tuple, verosimilmente a ciò che avviene con le liste.

```
1     # Concatenazione di tuple
2 tupla1 = (1, 2, 3)
3 tupla2 = (4, 5, 6)
4 tupla_combinata = tupla1 + tupla2
5
6 print(tupla_combinata)  # Output: (1, 2, 3, 4, 5, 6)
```

Aspetti Teorici: Principi Fondamentali per la *Concatenazione* delle Tuple.

- **Creazione di una nuova tupla:**

  A differenza di alcuni metodi delle liste (es: *extend()*), l'operatore + con le tuple crea **sempre** una nuova tupla in memoria, lasciando immutate le tuple ***originali.***

- **Ordine preservato:**

  Gli elementi nella nuova tupla appaiono nello stesso ordine delle tuple originali, con gli elementi della prima tupla seguiti dagli elementi della seconda.

- **Immutabilità preservata:**

  La concatenazione rispetta il principio di immutabilità delle tuple.

- **Unpacking:**

```python
tupla = (10, 20, 30)
a, b, c = tupla  # a=10, b=20, c=30

# Extended unpacking (Python 3.x)
prima, *resto = (1, 2, 3, 4)  # prima=1, resto=[2, 3, 4]

```

L'unpacking delle tuple è una potente funzionalità di Python che consente di assegnare i valori di una tupla a più variabili in un'unica riga. Questa tecnica rende il codice più leggibile ed efficiente. In altre parole si tratta di un processo in cui estraiamo i valori da una tupla e li assegniamo alle singole variabili in un unico passaggio.

```python
a = (1, 2)
b = (3, 4)
c = a + b

print(a)  # Output: (1, 2) - La tupla originale e' immutata
print(b)  # Output: (3, 4) - La tupla originale e' immutata
print(c)  # Output: (1, 2, 3, 4) - Nuova tupla
```

5.6.7 Concatenazione vs altri metodi di combinazione

A differenza delle liste, a causa del principio di immutabilità, le tuple non dispongono degli stessi metodi di combinazione degli elementi, presenti nelle liste. Ciò porta ad avere opzioni più limitate.

```python
# Con le liste, potremmo fare:
lista1 = [1, 2]
lista2 = [3, 4]
lista1.extend(lista2)  # Modifica lista1 in-place
print(lista1)  # Output: [1, 2, 3, 4]

# Con le tuple, dobbiamo usare concatenazione:
tupla1 = (1, 2)
tupla2 = (3, 4)
# Non esiste tupla1.extend(tupla2) perche' le tuple sono immutabili
tupla3 = tupla1 + tupla2  # Crea una nuova tupla
print(tupla3)  # Output: (1, 2, 3, 4)
```

5.6.8 Differenza tra Tuple e Liste:

La scelta tra tuple e liste per la concatenazione dipende dalle esigenze specifiche:

- Usa le tuple quando l'immutabilità è importante e la combinazione avviene raramente.

- Usa le liste quando la mutabilità è accettabile o desiderata, specialmente per operazioni frequenti o complesse.

Caratteristica	Tuple	Liste
Sintassi di dichiarazione	`tupla = (1, 2, 3)`	`lista = [1, 2, 3]`
Mutabilità	Immutabili (non modificabili dopo la creazione)	Mutabili (modificabili in qualsiasi momento)
Concatenazione	`tupla3 = tupla1 + tupla2` Crea sempre una nuova tupla	`lista3 = lista1 + lista2` o `lista1.extend(lista2)`
Metodi disponibili	Solo due metodi: `count()` e `index()`	Numerosi metodi: `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `sort()`, ecc.
Efficienza in memoria	Leggermente più efficiente per l'archiviazione di dati statici	Richiede memoria extra per supportare le modifiche
Performance per modifiche	Inefficiente (crea sempre nuovi oggetti)	Efficiente per modifiche frequenti
Uso come chiavi di dizionario	Possono essere usate come chiavi di dizionario	Non possono essere usate come chiavi di dizionario
Casi d'uso tipici	Record immutabili, coordinate, valori di ritorno multipli da funzioni	Collezioni dinamiche, sequenze che richiedono modifiche frequenti

Tabella 2: Confronto tra Tuple e Liste in Python

5.6.9 Altri metodi:

In questa breve trattazione faremo un riepilogo dei metodi presenti per le tuple, aggiungendone qualcuno per mero scopo didattico; alcuni faranno riferimento ad argomenti già affrontati, nella presentazione delle liste, quindi verranno indicizzati ai capitoli affrontati.

Metodo	Sintassi Esempio	Descrizione	Note/Confronto con Liste
`count(x)`	`mia_tupla.count(5)`	Conta quante volte appare l'elemento, argomento del metodo.	**Identico** al metodo *count()* delle liste
`index(x)`	`mia_tupla("a")`	Restituisce l'indice della prima occorrenza dell'elemento x.	**Identico** al metodo `index(x)` delle liste
`index(x,start)`	`mia_tupla.index("a",2)`	Restituisce l'indice della prima occorrenza di x a partire dall'indice *start*.	**Identico** alla variante con *start* del metodo *index()* delle liste.
`index(x,start, end`	`mia_tupla.index("a",2,5)`	Restituisce l'indice di x tra *start* (incluso) e *end* (escluso).	**Identico** alla variante con *start* ed *end* delle liste.

Tabella 3: Metodi per le Tuple

Come si può notare dalla tabella e come accennato ad inizio capitolo, le Tuple dispongono di un numero limitato di metodi rispetto alle liste. Questo è una diretta conseguenza della loro **immutabilità.** I metodi disponibili, *count()* e *index()*, sono focalizzati sull'ispezione e la ricerca di elementi e funzionano in modo **identico** ai loro corrispettivi nelle liste. (vedi Sezione Metodi Count e Index per liste Pag.37

5.7 Tuple e la Gestione della Memoria

Come per gli argomenti trattati nei capitolo precedenti, affrontiamo anche per le tuple la questione dell'allocazione della memoria. Vediamo come Python gestisce le tuple in memoria e quali vantaggi offrono rispetto ad altre strutture dati.

5.7.1 Allocazione della Memoria

Quando si crea una tupla in Python, il sistema alloca un blocco contiguo di memoria. A differenza delle liste, che vengono sovra-allocate per consentire future espansioni, le tuple ricevono esattamente lo spazio necessario per contenere i loro elementi. Questo perché Python sa che le tuple non cambieranno mai dimensione dopo la creazione, ci rifacciamo sempre al concetto di *immutabilità*

```
import sys

tupla = (1, 2, 3, 4, 5)
lista = [1, 2, 3, 4, 5]

print(f"Dimensione della tupla: {sys.getsizeof(tupla)} byte")
print(f"Dimensione della lista: {sys.getsizeof(lista)} byte")
```

Eseguendo questo codice, noterete che la lista occupa più spazio della tupla, anche se contiene esattamente gli stessi elementi. Questo è dovuto alla *"over-allocation"* che abbiamo accennato prima.

5.7.2 Struttura Interna

Internamente, una tupla in Python è rappresentata come un singolo oggetto con:

1. Un header[1] che contiene:
   - Un contatore di riferimenti (per il garbage collector)
   - Un tipo dell'oggetto (che identifica l'oggetto come tupla)
   - La dimensione della tupla (numero di elementi)
2. Un array di ***puntatori*** agli oggetti contenuti nella tupla

   Questa struttura semplice contribuisce all'efficienza delle tuple. Poiché non è necessario supportare inserimenti o eliminazioni, la struttura può essere ottimizzata specificatamente per un accesso rapido agli elementi.

5.7.3 Immutabilità e Ottimizzazioni di Memoria

La questione dell'immutabilità consente a Python di implementare diverse ottimizzazioni:

1. **Intering di Piccole Tuple**

   Python può *"internare"* (riutilizzare) piccole tuple. Quando crei una tupla con elementi semplici come numeri interi piccoli, Python potrebbe riutilizzare un oggetto tupla esistente anziché crearne uno nuovo:

   ```
   a = (1, 2, 3)
   b = (1, 2, 3)
   print(a is b)  # In alcuni casi, puo' restituire True
   ```

2. **Condivisione di Sottotuple**

   Durante le operazioni di slicing, le tuple possono condividere la memoria in modo più efficiente. Quando estrai una sottotupla, Python non deve necessariamente copiare tutti gli elementi:

[1] L'header in una tupla in Python è la sezione iniziale della sua rappresentazione in memoria, contenente metadati cruciali per la gestione dell'oggetto.

```
originale = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
slice = originale[2:7]  # Crea (2, 3, 4, 5, 6)
```

Anche se *"slice"* è un nuovo oggetto tupla, i suoi elementi non sono copie ma riferimenti agli stessi oggetti nell'originale. Questo è sicuro grazie all'immutabilità delle tuple.

5.7.4 Tuple come Elementi Hashable

Anche se in questa sezione troveremo elementi che non sono stati trattati nei capitoli precedenti, non potevamo trascurare questa capacità.

Un vantaggio significativo delle tuple in termini di memoria è che, essendo immutabili, sono ***hashable.*** Questo significa che possono essere usate come chiavi nei dizionari e come elementi nei set:

```
# Le tuple possono essere usate come chiavi nei dizionari
coordinate_valori = {
    (0, 0): "origine",
    (1, 0): "unit‘a X",
    (0, 1): "unit‘a Y"
}

# Le liste non possono essere usate come chiavi
# Questo genera un TypeError: unhashable type: 'list'
# coordinate_errate = {[0, 0]: "origine"}
```

Questa capacità è possibile perché il valore hash di una tupla non cambierà mai durante la sua esistenza, garantendo così l'integrità delle strutture dati che si basano sui valori hash.

5.7.5 Tuple di Tuple e strutture Annidate

Come per le liste anche per le tuple è possibile annidarle. Quando si creano strutture annidate con le tuple, si ottengono ulteriori benefici in termini di memoria:

```
matrice_tupla = ((1,2,3),(4,5,6),(7,8,9))
matrice_lista = [[1,2,3], [4,5,6],[7,8,9]]
```

La *matrice_ tupla* è completamente immutabile e ha un'impronta di memoria più piccola, come abbiamo visto nell'introduzione all'allocazione della memoria, inoltre, poiché è immutabile, Python può applicare altre ottimizzazioni interne.

5.7.6 Garbage Collection

Le tuple beneficiano anche di un comportamento più prevedibile con il garbage collector di Python. Quando un oggetto tuple non ha più riferimenti, tutti i suoi elementi possono essere marcati per la pulizia contemporaneamente (assumendo che non siano referenziati altrove). Non c'è rischio che la struttura dati cambi durante la scansione del garbage collector, che è un vantaggio sottile ma importante per sistemi con requisiti di memoria stringenti.

5.7.7 Ciclo di Vita di una Tupla in Memoria

Vediamo cil ciclo di vita di una tupla dal punto di vista della memoria:

1. **Creazione:** Python alloca un blocco di memoria della dimensione esatta necessaria.
2. **Utilizzo:** La struttura rimane invariata durante tutta la vita dell'oggetto.
3. **Distruzione:** Quando non ci sono più riferimenti alla tupla, il garbage collector libera lo spazio.

Questo ciclo è più semplice e prevedibile rispetto a quello delle liste, che possono cambiare dimensione e struttura più volte durante la loro esistenza.

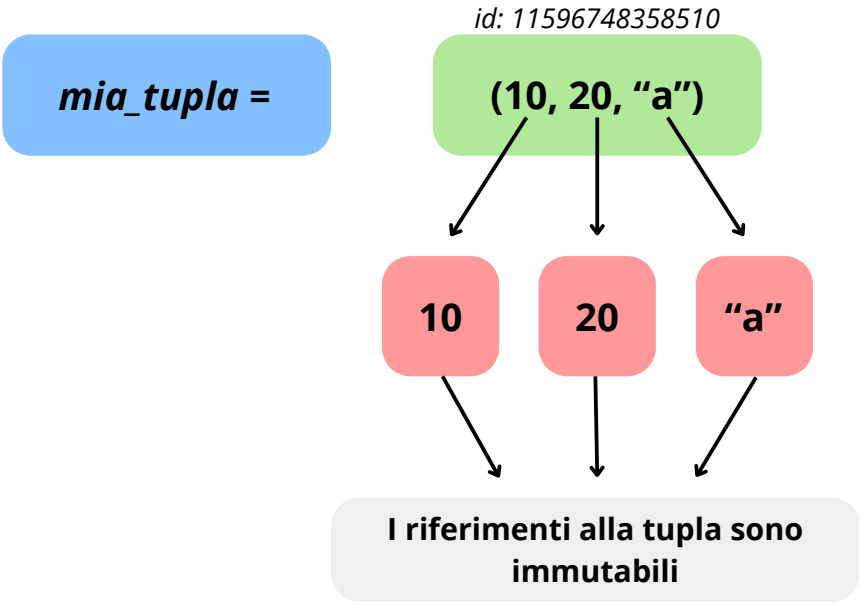
5.7.8 Rappresentazione Grafica

In questa parte tratteremo gli argomenti affrontati nei capitoli precedenti in maniera grafica, in modo tale da chiarire ulteriormente le Tuple e come esse vengono gestite da Python.

Gestione della memoria per le Tuple

Inizializzazione della Tupla

mia_tupla = (10,20,"a")

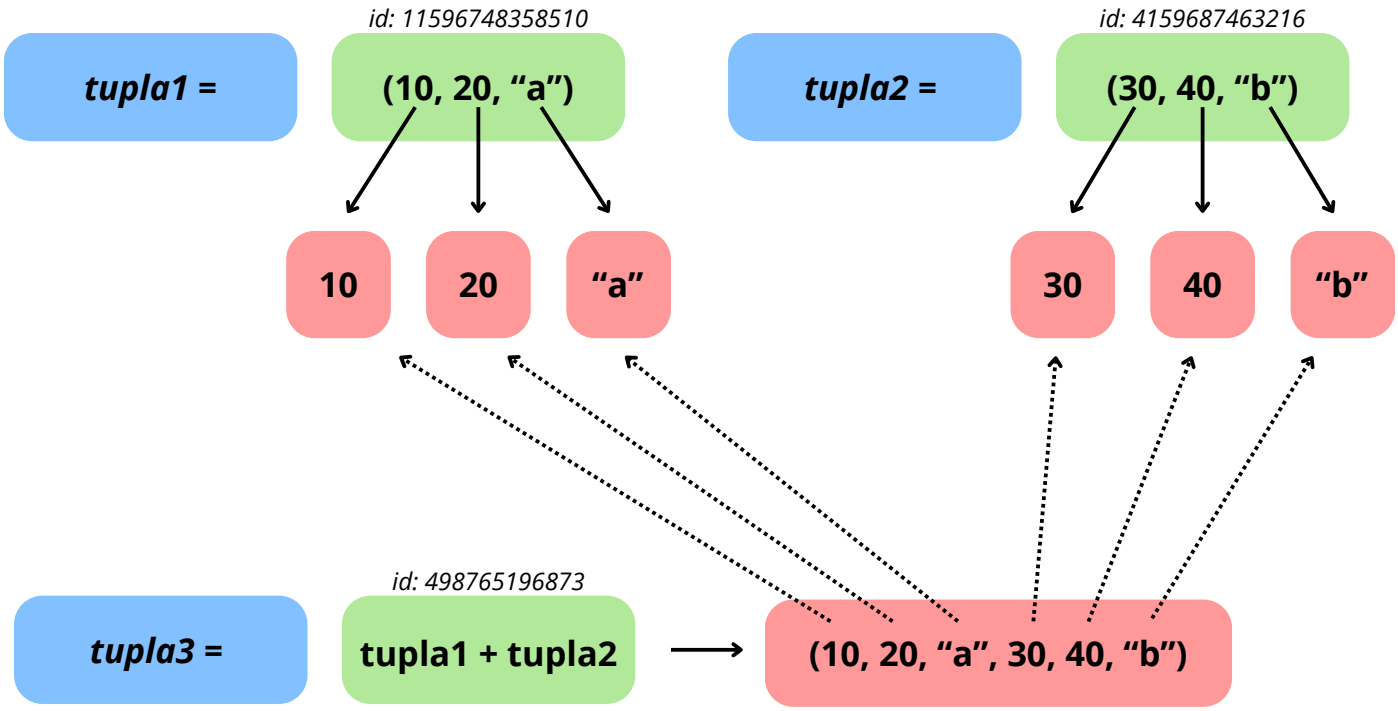


Quando creiamo una tupla con elementi di tipo diverso:

- Python alloca memoria per l'oggetto tupla stesso
- Ogni elemento viene conservato separatamente in memoria
- La tupla contiene riferimenti agli oggetti, non gli oggetti stessi
- I tipi diversi (interi, stringhe) sonno gestiti in modo uniforme

Concatenazione di Tuple

```
tupla1 = (10,20,"a")
tupla2 = (30,40,"b")
tupla_concatenata = tupla1 + tupla2
```

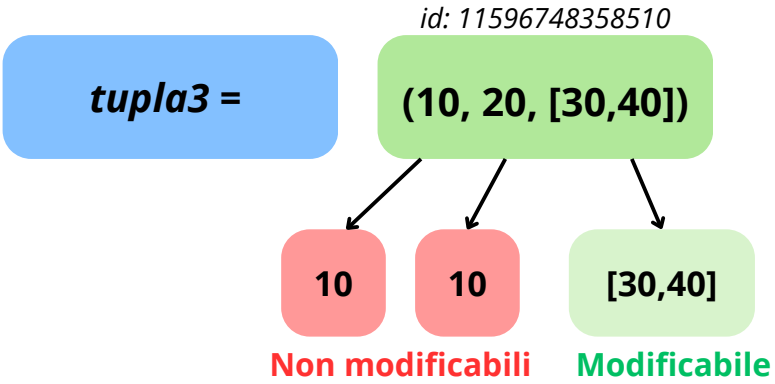


Nella concatenazione di Tuple:

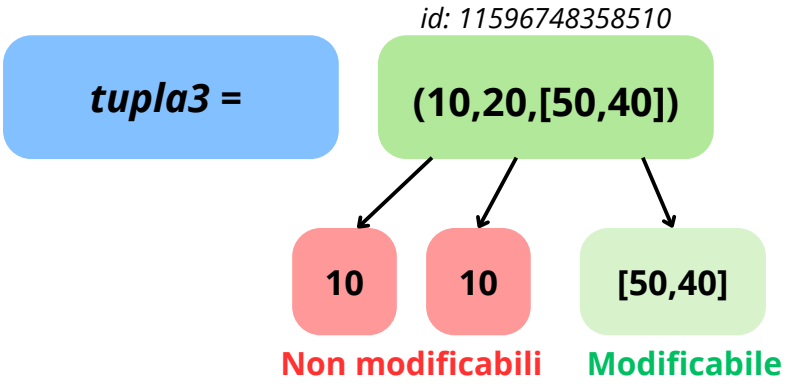
- Viene creata una nuova tupla in memoria
- Le tuple originali non vengono modificate
- I riferimenti agli oggetti vengono copiati nella nuova tupla
- Gli oggetti stessi non vengono duplicati in memoria

Concatenazione di Tuple

```
tupla_mista = (10,20,[30,40])

#Possiamo modificare la lista, all'interno della tupla
tupla_mista[2][0] = 50
```



Dopo la modifica



Nelle Tuple annidate con liste

- La tupla contiene un riferimento all'oggetto
- La tupla stessa rimane immutabile (non può cambiare riferimenti)
- Il contenuto della lista può essere modificato, poiché le liste sono mutabili
- In memoria, viene modificato l'oggetto lista, non la tupla

5.8 Esercizi Riassuntivi

Come di consueto affrontiamo una serie di esercizi riassuntivi per familiarizzare con i concetti affrontati, alcuni esercizi possono sembrare banali, ma hanno lo scopo di far pratica con la sintassi, evitando gli errori comuni, altri sono strutturati in modo da generare errori in modo tale da familiarizzare con le diciture degli errori e capire cosa riportano.

5.8.1 Esercizio 1: Creazione e sintassi Base delle Tuple

```
# Crea le seguenti tuple:
# Una tupla vuota
# Una tupla contenente solamente il numero 42
# Una tupla con i numeri da 1 a 5
# Una tupla mista con un numero, una stringa, un booleano
# Una tupla di stringhe
```

Per questo esercizio, ripassare le sezioni:

- Sintassi base Delle tuple. pag.60

5.8.2 Esercizio 2: Accesso agli Elementi (indicizzazione)

```
# Data la tupla
animali = ("cane", "gatto", "pesce", "uccello", "tartaruga")

# Stampare il primo elemento della tupla
# Stampare l'ultimo elemento tramite indicizzazione negativa
# Stampare il terzo elemento
# Stampare il penultimo elemento
# Cosa succede se provi ad accedere all'indice 10? Spiega l'errore
```

Per questo esercizio, ripassare le sezioni:

- Indicizzazione delle Tuple. pag.61

5.8.3 Esercizio 3: Sclicing delle Tuple

```
# Data la tupla
numeri = (0,1,2,3,4,5,6,7,8,9)

# Estrai i primi 3 elementi
# Estrai gli ultimi 4 elementi
# Estrai gli elementi dal 2 al 6 (inclusi)
# Estrai tutti gli elementi con indice pari
# Estrai tutti gli elementi in ordine inverso
# Estrai ogni secondo elemento a partire dal primo

```

Per questo esercizio, ripassare la sezione:

- Slicing per le Tuple. pag61

5.8.4 Esercizio 4: Metodi per le Tuple

```
# Data la tupla
frutti = ("mela", "banana", "mela", "arancia", "mela", "kiwi", "banana")

# Conta quante volte appare "mela" nella tupla
# Trova l'indice della prima occorrenza di "banana"
# Conta quante volte appare "pera"
```

```
# Trova la lunghezza deella tupla
# Verifica se "arancia" e' presente nella tupla usando l'operatore "in"

```

Per questo esercizio, ripassare la sezione: Tabella riassuntiva dei metodi per le Tuple. pag65

5.8.5 Esercizio 5: Concatenazione delle Tuple

```
# Date le tuple:
tuple1 = (1,2,3)
tuple2 = (4,5,6)

# Concatena le due tuple usando l'operatore + e salva il risultato in tuple3
# Moltiplica tuple1 per 3 usando l'operatore * e stampa il risultato
# Crea una nuova tupla concatenando (0,) con tuple1
# Crea una nuova tupla concatenando tuple2 con (7,)
```

Per questo esercizio, ripassare la sezione:

- Concatenazione delle Tuple. pag62

5.8.6 Esercizio 6: Immutabilità delle Tuple

```
# Crea una tupla: coordinate = (10,20)
# Prova a modificare il primo elemento scrivendo coordinate[0] = 15
# Spiega l'errore che ottieni
# Dimostra come modificare una tupla, creandone una nuova
# Confronta questo comportamento con quello delle liste

```

Per questo esercizio, ripassare la sezione:

- Caratteristiche delle Tuple. pag60

5.8.7 Esercizio 7: Tuple Annidate

```
# Data la tupla
coordinate = ((0,0),(1,1),(2,4),(3,9))

# Accedi alla seconda coppia di coordinate
# Accedi al valore "y" della terza coppia di coordinate
# Accedi al valore "x" della prima coppia di coordinate
# Conta quante coppie di coordinate ci sono

```

Per questo esercizio, ripassare la sezione: Concatenazione delle Tuple. pag62

5.8.8 Esercizio 8: Operazioni di Confronto

```
# Date le tuple
a = (1, 2, 3)
b = (1, 2, 3)
c = (3, 2, 1)

# Verificare se a == b
# Verificare se a == c
# Verificare se a is b
# Spiegare la differenza tra == e is
```

Per questo esercizio, ripassare la sezione: Operatore Confronto. pag49

5.8.9 Esercizio 9: Applicazione Pratica

```
# Inizializzazione di un sistema per memorizzare informazioni dei studenti

# Crea una tupla per rappresentare uno studente con: nome, eta', voto
# Crea una tupla contenente 3 studenti diversi
# Accedi alle informazioni del secondo studente
# Trova un modo per modificare alcuni studendi che potrebbero avere voti in sospeso
```

Per questo esercizio, ripassare le varie metodologie applicate alle tuple.

- Concatenazione. pag62
- Caratteristiche delle Tuple. pag60
- Tabella dei metodi per Tuple. pag65

5.9 Risoluzione Esercizi sulle Tuple

5.9.1 Risoluzione: Esercizio 1: Creazione e sintassi Base delle Tuple

```
# Crea le seguenti tuple:
# Una tupla vuota
# Una tupla contenente solamente il numero 42
# Una tupla con i numeri da 1 a 5
# Una tupla mista con un numero, una stringa, un booleano
# Una tupla di stringhe




tupla_vuota = ()
print(type(tupla_vuota))

tupla_num = (42,)
print(type(tupla_num),tupla_num)

tuple_serie = (1,2,3,4,5)
print(type(tuple_serie),tuple_serie)

tupla_range = tuple(range(1,5+1))
print(tupla_range, type(tupla_range))

tupla_mista = ("a",23,True)
print(type(tupla_mista),tupla_mista)

stringa_tupla = tuple("python")
print(type(stringa_tupla), stringa_tupla)

tupla_stringhe = ("rosso", "verde", "blu")
print(type(tupla_stringhe), tupla_stringhe)
```

5.9.2 Risoluzione: Esercizio 2: Accesso agli Elementi (indicizzazione)

```
#Data la tupla
animali = ("cane", "gatto", "pesce", "uccello", "tartaruga")
# Stampare il primo elemento della tupla
# Stampare l'ultimo elemento tramite indicizzazione negativa
# Stampare il terzo elemento
# Stampare il penultimo elemento
# Cosa succede se provi ad accedere all'indice 10? Spiega l'errore
```

```
8
9
10 print("primo elemento: ", animali[0])
11 print("ultimo elemento con indice negativo: ",animali[-1])
12 print("terzo elemento: ",animali[2])
13 print("penultimo elemento: ", animali[-2])
14 print("indice errato: ", animali[10])
```

5.9.3 Risoluzione: Esercizio 3: Sclicing delle Tuple

```
1 #Data la tupla
2 numeri = (0,1,2,3,4,5,6,7,8,9)
3 # Estrai i primi 3 elementi
4 # Estrai gli ultimi 4 elementi
5 # Estrai gli elementi dal 2 al 6 (inclusi)
6 # Estrai tutti gli elementi con indice pari
7 # Estrai tutti gli elementi in ordine inverso
8 # Estrai ogni secondo elemento a partire dal primo
9
10
11 #Primi 3
12 print(numeri[0:3])
13
14 #Ultimi 4
15 print(numeri[-4:])
16
17 #Dal 2 al 6
18 print(numeri[2:-3])
19
20 #Indice pari
21 print(numeri[::2])
22
23 #Ordine inverso
24 print(numeri[::-1])
25
26 #Ogni secondo elemento
27 print(numeri[0::2])
```

5.9.4 Risoluzione: Esercizio 4: Metodi per le Tuple

```
1 # Data la tupla
2 frutti = ("mela", "banana", "mela", "arancia", "mela", "kiwi", "banana")
3 # Conta quante volte appare "mela" nella tupla
4 # Trova l'indice della prima occorrenza di "banana"
5 # Conta quante volte appare "pera"
6
7 # Trova la lunghezza deella tupla
8 # Verifica se "arancia" e' presente nella tupla usando l'operatore "in"
9
10 #Contare quante volte appare mela
11 print(frutti.count("mela"))
12
13 #Trovare l'indice di banana
14 print(frutti.index("banana"))
15
16 #Contare quante volte appare pera
17 print(frutti.count("pera"))
18
19 #Trovare la lunghezza della tupla
20 print(len(frutti))
```

```
21
22 #Cercare se c'e' arancia in frutti
23 print("arancia" in frutti)
```

5.9.5 Risoluzione: Esercizio 5: Concatenazione delle Tuple

```
1 # Date le tuple:
2 tuple1 = (1,2,3)
3 tuple2 = (4,5,6)
4 # Concatena le due tuple usando l'operatore + e salva il risultato in tuple3
5 # Moltiplica tuple1 per 3 usando l'operatore * e stampa il risultato
6 # Crea una nuova tupla concatenando (0,) con tuple1
7 # Crea una nuova tupla concatenando tuple2 con (7,)
8
9
10 tuple3 = tuple1 + tuple2
11 print(tuple3)
12
13 print(tuple1 * 3)
14
15 tupla0 = (0,) + tuple1
16 print(tupla0)
17
18 tupla4 =  tuple2 + (7,)
19 print(tupla4)
```

5.9.6 Risoluzione: Esercizio 6: Immutabilità delle Tuple

```
1 # Crea una tupla: coordinate = (10,20)
2 # Prova a modificare il primo elemento scrivendo coordinate[0] = 15
3 # Spiega l'errore che ottieni
4 # Dimostra come modificare una tupla, creandone una nuova
5 # Confronta questo comportamento con quello delle liste
6
7
8 # 1. Crea una tupla
9 coordinate = (10, 20)
10 print("Tupla originale:", coordinate)
11
12 # 2. Prova a modificare il primo elemento (questo causer'a un errore!)
13 try:
14     coordinate[0] = 15
15 except TypeError as e:
16     print("ERRORE:", e)
17
18 # 3. Spiegazione dell'errore:
19 print("\nSpiegazione:")
20 print("L'errore 'tuple object does not support item assignment' significa che")
21 print("le tuple sono IMMUTABILI - non possono essere modificate dopo la creazione.")
22
23 # 4. Come 'modificare' una tupla creandone una nuova
24 print("\n4. 'Modifica' creando una nuova tupla:")
25 nuove_coordinate = (15, coordinate[1])
26 print("Coordinate originali:", coordinate)  # (10, 20) - invariata!
27 print("Nuove coordinate:", nuove_coordinate)  # (15, 20)
28
29 # 5. Confronto con le liste
30 print("\n5. Confronto con le liste:")
31 lista_coordinate = [10, 20]
32 print("Lista originale:", lista_coordinate)
```

```
33
34 lista_coordinate[0] = 15  # Questo funziona!
35 print("Lista modificata:", lista_coordinate)
36
37 print("\nRIASSUNTO:")
38 print("- TUPLE: immutabili, non si possono modificare")
39 print("- LISTE: mutabili, si possono modificare")
```

5.9.7 Risoluzione: Esercizio 7: Tuple Annidate

```
1  # Data la tupla
2  coordinate = ((0,0),(1,1),(2,4),(3,9))
3
4  print("Tupla coordinate:", coordinate)
5  print("Ogni elemento e' una coppia (x, y)")
6  print()
7
8  # 1. Accedi alla seconda coppia di coordinate
9  print("1. Seconda coppia di coordinate:")
10 seconda_coppia = coordinate[1]
11 print("coordinate[1] =", seconda_coppia)
12
13 # 2. Accedi al valore "y" della terza coppia (il valore 4)
14 print("2. Valore 'y' della terza coppia:")
15 y_terza = coordinate[2][1]
16 print("coordinate[2][1] =", y_terza)
17
18 # 3. Accedi al valore "x" della prima coppia
19 print("3. Valore 'x' della prima coppia:")
20 x_prima = coordinate[0][0]
21 print("coordinate[0][0] =", x_prima)
22
23 # 4. Conta quante coppie di coordinate ci sono
24 print("4. Numero di coppie:")
25 numero_coppie = len(coordinate)
26 print("len(coordinate) =", numero_coppie)
```

5.9.8 Risoluzione: Esercizio 8: Operazioni di Confronto

```
1  # Date le tuple
2  a = (1, 2, 3)
3  b = (1, 2, 3)
4  c = (3, 2, 1)
5
6  # Verificare se a == b
7  # Verificare se a == c
8  # Verificare se a is b
9  # Spiegare la differenza tra == e is
10
11
12 #Verifichiamo se a == b
13 print(a == b)
14
15 #Verifichiamo se a == c
16 print(a == c)
17
18 #Verifichiamo se a is b
19 print(a is b)
20
21 """
```

```
Spiegazione dettagliata:

== (equality): Confronta il contenuto/valori

a == b -> True (stessi valori: 1,2,3)
a == c -> False (valori diversi: 1,2,3 vs 3,2,1)


is (identity): Confronta l'identit‘a in memoria

a is b -> Solitamente False (oggetti separati)
Solo se puntano allo stesso oggetto in memoria restituisce True



Regola pratica:

Usa == per confrontare contenuti
Usa is per verificare se sono lo stesso oggetto

"""
```

5.9.9 Risoluzione: Esercizzio 9: Applicazione Pratica

```
# Inizializzazione di un sistema per memorizzare informazioni dei studenti

# 1. Crea una tupla per rappresentare uno studente con: nome, et‘a, voto
# 2. Crea una tupla contenente 3 studenti diversi
studenti = (("Mario", 19, 27), ("Luca", 18, 29), ("Giulia", 19, 30))

print("Sistema studenti:", studenti)


# 3. Accedi alle informazioni del secondo studente
print("Secondo studente:", studenti[1])


# 4. Trova un modo per modificare alcuni studenti che potrebbero avere voti in sospeso
print("=== MODIFICA VOTI IN SOSPESO ===")


#Dato che la tupla puo' contenere anche delle liste lo studente con il voto in sospeso lo mettiamo dentro una lista, in modo tale da poterlo modificare
studenti = (["Mario", 19, 27], ("Luca", 18, 29), ("Giulia", 19, 30))

#In questo modo ora possiamo accedere allo studente e modificare il voto
    #Osserviamo che lo studente [0] risulta essere una lista
print(type(studenti[0]))

studenti[0][2]= 30

print(studenti[0])

#Ovviamente creare una struttura dove alcuni elementi sono liste e altri tuple, senza logiche evidenti, potrebbe confondere chi legge il codice, che deve controllare ogni volta che tipo type() e' ogni elemento per capire o meno se e' modificabile.
```

5.10 Dizionari

Fino ad ora abbiamo imparato a organizzare i dati usando Liste e Tuple. Ma Python ci offre un terzo strumento ancora più potente: **i Dizionari.**

Pensiamo per esempio alla rubrica del telefono. Quando vogliamo chiamare, per esempio Marco, non scorriamo i contatti dall'inizio fino ad arrivare a Marco, giusto? Digitiamo semplicemente "Marco" nella barra di ricerca e ci viene mostrato instantaneamente il numero di Marco.

Cerchiamo di creare una rubrica con le conoscenze apprese fino ad ora, immaginiamo di organizzare una serie di contatti per la nostra rubrica. Abbiamo avuto a che fare con Liste e Tuple, ed ognuna di essa presenta delle peculiarità diverse ma che possono tornare utile per il nostro bisogno. Optiamo per creare la rubrica tramite Tuple concatenate con le Liste, in modo tale da ovviare al problema dell'Immutabilità dei dati, in questo modo possiamo modificare i nomi e i numeri di telefono.

```
rubrica = (
["Marco", 33298194],
["Lucia", 3982129391, 998908123],
["Andrea", 3289174982])
```

Sembrerebbe una soluzione ottimale, ma presenta un piccolo problema, la ricerca del contatto risulterebbe complicata, dato che dovremmo conoscere l'indice di riferimento del contatto per poterlo selezionare immediatamente, altrimenti dovremmo cercare tra tutti i contatti fino a quando non troviamo il contatto desiderato.

Bene i **Dizionari** ci aiutano a risolvere questo problema tramite il sistema **chiave-valore:** ogni **valore** (numero di telefono) è collegato direttamente alla sua **chiave** (nome del contatto) identificativa. Proprio ciò che avviene nella rubrica del telefono. Affronteremo meglio questo sistema nei capitoli seguenti in modo tale da capire limitazioni e utilità che ne concernano.

5.10.1 Metodi per dichiarare un Dizionario

Per dichiarare un **dizionario** in Python esistono vari metodi, esattamente per come abbiamo visto con le Metodi dichiarativi per Liste

Si dichiarano tramite parentesi graffe e come le liste possono contenere elementi di tipo diversi: *Booleani, stringhe, interi, float.* Come per le liste esiste il metodo built-in di Python per dichiarare un dizionario: dict()

Metodi Dichiarativi per Dizionari:

```
#Dichiarazione Dizionario tramite parentesi {}
dizionario = {
    "chiave1": valore1,
    "chiave2": valore2,
    "chiave3": valore3
}

#Dichiarazione tramite metodo dict()
dizionario_Dict = dict(chiave=valore1, chiave2=valore2, chiave3=valore3)
```

Per quanto riguarda il metodo ***dict()***, bisogna approfondire alcuni aspetti, in quanto questo metodo ha alcune restrizioni, rispetto alla dichiarazione canonica.

```
"""
Con {} puoi usare chiavi di qualsiasi tipo (anche numeri o tuple), mentre con dict() le chiavi
↪ devono essere stringhe valide come nomi di variabili (quindi niente spazi, accenti, ecc.).

Se vuoi creare un dizionario da una lista di coppie chiave-valore, dict() puo' essere comodo:
"""

coppie = [("nome", "Luca"), ("eta'", 30), ("citta'", "Milano")]
```

```
persona = dict(coppie)
```

5.10.2 Chiavi - Valori

Questa capacità di associare un valore a una determinata chiave fa sì che si possa costruire una mappa associativa, dove ogni elemento è costituito da una coppia chiave - valore, permettendo così un accesso rapido e efficiente ai dati, grazie alla peculiarità delle caratteristiche delle chiavi.

5.10.3 Caratteristiche delle Chiavi

Come già accennato, la rapidità e l'efficienza che offrono i ***dizionari*** sono date principalmente dalle caratteristiche che determinano le chiavi.

```
#Esempio Dizionario
dizionario = {
    "chiave1":valore1,
    "chiave2":valore2,
    "chiave3":valore3
}
```

Le chiavi hanno delle restrizioni specifiche che derivano dall'implementazione basata su **tabelle hash** sottostanti. Comprendere questi aspetti teorici aiuta a utilizzare i dizionari in modo più efficace e consapevole.

5.10.4 Meccanismo delle Tabelle Hash

Analizziamo attentamente la struttura delle tabelle Hash. I dizionari in Python implementano una strutta dati tramite tabelle Hash, che garantisce accesso rapido agli elementi. Il processo funziona in questa maniera.

- **Calcolo dell'Hash:** Quandi si inserisce una coppia chiave-valore, Python calcola un valore hash della chiave
- **Mappatura all'Indice:** Il valore Hash viene convertito in un indice dell'array interno
- **Memorizzazione:** Il valore viene memorizzato a quell'Indice
- **Recupero:** Per accedere al valore, si ricalcola l'hash della chiave e si accede direttamente all'indice

```
# Esempio concettuale del processo interno
# 1. Inserimento
chiave =  "nome"
valore = "Mario"
hash_valore = hash(chiave) # Es. -674930604
indice = hash_valore % dimensione_tabella # Es. -674930604 % 8 = 4

# 2. Accesso
chiave_cercata = "nome"
hash_cercato = hash(chiave_cercata)  # Stesso hash: -674930604
indice_calcolato = hash_cercato % dimensione_tabella  # Stesso indice: 4
# return tabella[4][1]  # "Mario"

print(f"Hash di 'nome': {hash('nome')}")
print(f"Hash di 'cognome': {hash('cognome')}")
print(f"Hash di 42: {hash(42)}")
```

5.10.5 Stabilità dell'Hash

Il valore hash deve rimanere **costante** durante tutta la vita dell'oggetto, altrimenti il dizionario non riuscirebbe più a trovare le chiavi inserite precedentemente.

```
# Dimostrazione della stabilit‘a dell’hash
chiave_stabile = "test"
hash_iniziale = hash(chiave_stabile)

# Simulazione di accessi multipli
for i in range(3):
    hash_corrente = hash(chiave_stabile)
    print(f"Accesso {i+1}: hash = {hash_corrente}")
    assert hash_corrente == hash_iniziale, "Hash instabile!"

print("Hash stabile per tutta la durata del programma")
```

Caratteristiche principali delle Chiavi

- **Hashabilità**: Devono avere un valore hash costante.
- **Immutabilità**: Non possono essere modificate dopo la creazione.
- **Unicità**: Ogni chiave appare una sola volta nel dizionario.
- **Confrontabilità**: Devono supportare l'operatore == di uguaglianza.
- **Tipologia Libera**: Possono essere di qualsiasi tipo purché hashable.

Vediamo, tramite esempi pratici, a cosa fanno riferimento queste caratteristiche.

Hashabilità

```
# Le chiavi devono essere oggetti hashable
# Devono implementare il metodo __hash__()
# Il valore hash deve rimanere costante durante la vita dell’oggetto

print(f"Hash di ’ciao’: {hash(’ciao’)}")    # Output -5659573120285894662
print(f"Hash di 42: {hash(42)}")            # Output 42
print(f"Hash di (1,2,3): {hash((1,2,3))}")  # Output 529344067295497451
```

Immutabilità

```
# Le chiavi devono essere immutabili dopo la crezione
# Non possono essere modificate in-place
# Garantiscono che il valore hash rimanga stabile

dizionario = {}
chiave_stringa = "prodotto"
dizionario[chiave_stringa] = "Laptop"
# la chiave stringa non puo’ essere modificata in-place
print(dizionario)      # Output {’prodotto’: ’Laptop’}
```

Unicità

```
# Ogni chiave puo’ apparire una sola volta nel dizionario
# Se si inserisce una chiave esistente , il valore precedente viene sovrascritto
# L’unicita’ e’ verificata tramite l’operatore ==

```

```
prodotti = {}

# Prima inserzione
prodotti["laptop"] = {"prezzo":800, "quantita":4}
print(f"Dopo prima inserzione: {prodotti}")

# Output Dopo prima inserzione: {'laptop': {'prezzo': 800, 'quantita': 4}}

# Seconda inserzione con stessa chiave - sovrascrive!
prodotti["laptop"] = {"prezzo": 750, "quantit‘a": 3}
print(f"Dopo sovrascrittura: {prodotti}")
# Output Dopo sovrascrittura: {'laptop': {'prezzo': 750, 'quantit‘a': 3}}


# Verifica che abbiamo sempre una sola chiave
print(f"Numero di chiavi: {len(prodotti)}")
# Output Numero di chiavi: 1

# Anche con tipi numerici
numeri = {}
numeri[1] = "uno"
numeri[1.0] = "uno punto zero"
# Output 1 == 1.0 in Python!

print(f"Chiavi numeriche: {numeri}")
# Output {1: 'uno punto zero'}
print(f"1 == 1.0? {1 == 1.0}")
# Output True
```

Confrontabilità

```
# Le chiavi devono implementare il metodo __eq__()
# Permette di determinare se due chiavi sono identiche
# Usato per risolvere eventuali collisioni di hash

# Esempio 1: Numeri equivalenti
inventario = {}

# Inserimento con chiave intera
inventario[1] = "Un articolo"
print(f"Dopo inserimento con 1: {inventario}")

# Inserimento con chiave float equivalente
inventario[1.0] = "Un articolo (aggiornato)"  # 1 == 1.0!
print(f"Dopo inserimento con 1.0: {inventario}")
print(f"Numero di chiavi: {len(inventario)}")  # Solo 1 chiave!

# Verifica dell'uguaglianza
print(f"1 == 1.0? {1 == 1.0}")  # True
print(f"hash(1) == hash(1.0)? {hash(1) == hash(1.0)}")  # True

# Esempio 2: Tuple con contenuto identico
coordinate = {}

# Due tuple create separatamente ma con contenuto uguale
punto_a = (3, 4)
punto_b = (3, 4)

coordinate[punto_a] = "Prima posizione"
print(f"Dopo prima inserzione: {coordinate}")

coordinate[punto_b] = "Seconda posizione"  # Sovrascrive!
print(f"Dopo seconda inserzione: {coordinate}")
```

```
33
34 # Verifica dell'uguaglianza
35 print(f"punto_a == punto_b? {punto_a == punto_b}")  # True
36 print(f"punto_a is punto_b? {punto_a is punto_b}")  # Puo' essere True o False
37 print(f"hash(punto_a) == hash(punto_b)? {hash(punto_a) == hash(punto_b)}")  # True
38
39 # Esempio 3: Stringhe con contenuto identico
40 database = {}
41
42 # Stringhe create in modi diversi ma con contenuto uguale
43 chiave1 = "utente_001"
44 chiave2 = "utente" + "_" + "001"
45 chiave3 = f"utente_{1:03d}"
46
47 database[chiave1] = "Mario Rossi"
48 database[chiave2] = "Luigi Verdi"  # Sovrascrive Mario Rossi
49 database[chiave3] = "Anna Bianchi"  # Sovrascrive Luigi Verdi
50
51 print(f"Database finale: {database}")
52 print(f"Numero di utenti: {len(database)}")  # Solo 1!
53
54 # Verifica delle uguaglianze
55 print(f"chiave1 == chiave2? {chiave1 == chiave2}")  # True
56 print(f"chiave2 == chiave3? {chiave2 == chiave3}")  # True
57 print(f"Tutte le chiavi: '{chiave1}', '{chiave2}', '{chiave3}'")
58
59 # Esempio 4: Caso interessante con True/False e 1/0
60 configurazione = {}
61
62 # Bool e int equivalenti
63 configurazione[True] = "Modalit`a attiva"
64 configurazione[1] = "Valore uno"  # Sovrascrive True!
65
66 configurazione[False] = "Modalit`a disattiva"
67 configurazione[0] = "Valore zero"  # Sovrascrive False!
68
69 print(f"Configurazione: {configurazione}")
70 print(f"True == 1? {True == 1}")    # True
71 print(f"False == 0? {False == 0}")  # True
```

Tipologia libera

```
1 # Non esiste restrizione sul tipo di oggetto (basta che sia hashable)
2 # Possono coesistere chiavi di tipi diversi nello stesso dizionario
3 # Raccomandato mantenere coerenza per leggibilita'
4
5 # Dizionario con chiavi di tipi diversi - tutti nello stesso dizionario!
6 collezione_mista = {
7     # Stringa come chiave
8     "nome": "Mario Rossi",
9
10    # Numero intero come chiave
11    42: "La risposta universale",
12
13    # Numero decimale come chiave
14    3.14: "Pi greco approssimato",
15
16    # Tupla come chiave
17    (1, 2, 3): "Coordinate tridimensionali",
18
19    # Valore booleano come chiave
20    True: "Stato attivo",
21
```

```
    # None come chiave
    None: "Valore indefinito"
}

# Accesso alle diverse chiavi
print("Esempi di accesso con chiavi di tipi diversi:")
print(f"Nome: {collezione_mista['nome']}")
print(f"Numero: {collezione_mista[42]}")
print(f"Pi greco: {collezione_mista[3.14]}")
print(f"Coordinate: {collezione_mista[(1, 2, 3)]}")
print(f"Stato: {collezione_mista[True]}")
print(f"Indefinito: {collezione_mista[None]}")

# Esempio pratico: sistema di configurazione
configurazione_sistema = {
    # Chiavi stringa per impostazioni testuali
    "host": "localhost",
    "database": "app_principale",

    # Chiavi numeriche per valori numerici
    8080: "Porta del server",
    443: "Porta HTTPS",

    # Chiavi booleane per flag
    True: "Modalit`a di produzione attiva",
    False: "Modalit`a di debug disattiva",

    # Tupla per configurazioni composite
    ("cache", "redis"): "127.0.0.1:6379"
}

print("\nConfigurazione sistema:")
print(f"Host: {configurazione_sistema['host']}")
print(f"Porta HTTP: {configurazione_sistema[8080]}")
print(f"Produzione: {configurazione_sistema[True]}")
print(f"Cache Redis: {configurazione_sistema[('cache', 'redis')]}")

# Attenzione: comportamenti speciali con tipi misti
problemi_comuni = {}

# True e 1 sono considerati la stessa chiave!
problemi_comuni[True] = "Valore booleano"
problemi_comuni[1] = "Valore intero"  # Sovrascrive True!
print(f"\nProblema con True/1: {problemi_comuni}")  # Solo una chiave

# False e 0 sono considerati la stessa chiave!
problemi_comuni[False] = "Valore booleano falso"
problemi_comuni[0] = "Valore zero"  # Sovrascrive False!
print(f"Problema con False/0: {problemi_comuni}")  # Solo due chiavi totali

# Esempio di buona pratica: mantenere coerenza nei tipi
# BENE - chiavi tutte stringhe
anagrafica_coerente = {
    "nome": "Luigi",
    "cognome": "Bianchi",
    "et`a": "25",  # Anche l'et`a come stringa per coerenza
    "citt`a": "Roma"
}

# MENO LEGGIBILE - tipi misti senza motivo
anagrafica_confusa = {
    "nome": "Luigi",
    42: "Bianchi",  # Perche' 42 per il cognome?
```

```
    (1, 2): "25",   # Perche' tupla per l'et'a?
    True: "Roma"    # Perche' True per la citt'a?
}

print(f"\nAnagrafica coerente: {anagrafica_coerente}")
print(f"Anagrafica confusa: {anagrafica_confusa}")

# Verifica della dimensione del dizionario
print(f"\nNumero di elementi in collezione_mista: {len(collezione_mista)}")
print(f"Numero di elementi in configurazione_sistema: {len(configurazione_sistema)}")
```

5.11 Set

In questo capitolo ci troviamo ad affrontare l'implementazione dei ***set,*** la loro struttura rimarca la struttura dati matematica degli insiemi e con essa anche le operazioni annesse.

Come già consetudine anche in questo capitolo affronteremo la parte teorica, esploreremo i vari casi d'uso correlati di esempi pratici.

5.12 Caretteristiche Fondamentali

Come già accennato ad inizio capitolo i ***set*** si rifanno alla teoria degli insiemi matematici, quindi prima di esplorare i ***set*** in Python è bene conoscere la teoria degli insiemi.

5.12.1 Teoria degli insiemi

In matematica un insieme rappresenta una collezione di oggetti, *non ordinati, distinti*, ogni oggetto, (in matematichese denominato elemento) può appartenere o non all'insieme, senza possibilità di duplicazione o ordinamento intrinseco.

Una definizione formale stabilisce che due insiemi sono uguali se e solo se contengono esattamente gli stessi elementi, indipendentemente dall'ordine in cui questi sono disposti, stessa considerazione avviene in Python, dove l'identità della collezione dipende esclusivamente dal contenuto e non dalla sequenza di disposizione.

5.12.2 Uguaglianza e Principio di Estensionalità

Il principio di estensionalità stabilisce che due insiemi sono uguali se e solo se contengono esattamente gli stessi elementi. Questo principio implica che l'ordine di enumerazione degli elementi è irrilevante per determinare l'identità dell'insieme, così come il numero di volte in cui un elemento viene menzionato.

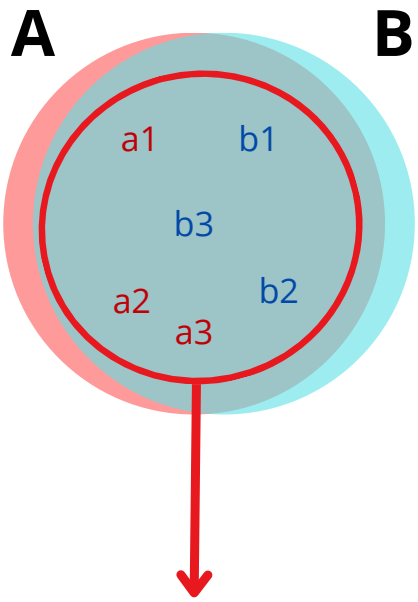
Gli insiemi 1, 2, 3, 3, 1, 2, e 1, 2, 3, 1 rappresentano la medesima entità matematica. Questa proprietà fondamentale si traduce direttamente nel comportamento dei set Python, dove l'operatore di uguaglianza verifica la coincidenza degli elementi indipendentemente dall'ordine di inserimento o dalla presenza di tentativi di inserimento duplicati.

5.12.3 Operazioni tra Insiemi

Iniziamo ad affrontare le operazioni possibili tra insiemi, dato che sono equivalenti con le operazioni tra ***set*** in Python.

Unione tra Insiemi L'unione di due insiemi A e B in matematica equivale all'insieme contenente tutti gli elementi che appartengono ad **A** e a **B**

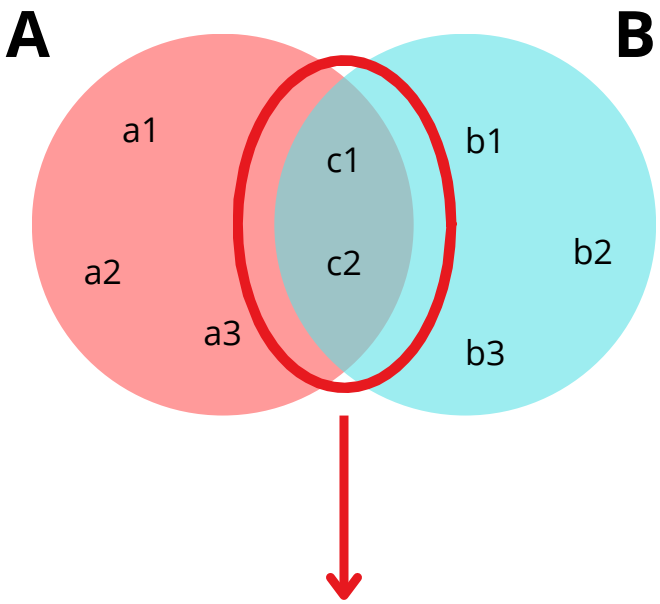
Unione tra Insiemi



L’insieme generato è l’unione tra l’insieme A e l’insieme B, comprendendone tutti gli elementi di entrambi.

Intersezione tra Insiemi L'intersezione tra insiemi, invece, comprende solo gli elementi comuni tra gli insiemi.

Intersezione tra Insiemi



L’ insieme generato racchiude gli elementi in comune tra l’insieme A e l’insieme B.

Differenza tra Insiemi L'operazione di differenza tra insiemi, A - B, non è altro che l'insieme contenente tutti gli elementi che appartengono ad A ma non a B

Un'altra operazione in comune tra i ***set*** e gli insiemi matematici è quella della differenza simmetrica tra insiemi, come è facile intuire dal nome dell'operazione, l'insieme risultante sarà dato soltanto dagli elementi non comuni tra gli insiemi esaminati.