Appunti di Python

Il tuo nome

13 maggio 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

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 è progettata per essere chiara e leggibile
- **Versatilità**: Adatto a diversi contesti (web, data science, automazione, ecc.)
- **Tipizzazione dinamica**: I tipi di dato sono determinati a runtime
- **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:

```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!")
```

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

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`

Tabella 2: Metodi essenziali per le liste (`list`)

Metodo	Descrizione	Esempio
`append(x)`	Aggiunge elemento alla fine	`[1, 2].append(3) → [1, 2, 3]`
`extend(iterable)`	Estende con elementi da iterabile	`[1].extend([2, 3]) → [1, 2, 3]`
`insert(i, x)`	Inserisce a indice specifico	`[1, 3].insert(1, 2) → [1, 2, 3]`
`remove(x)`	Rimuove prima occorrenza	`[1, 2, 2].remove(2) → [1, 2]`
`pop([i])`	Rimuove e restituisce elemento	`[1, 2, 3].pop(1) → 2`
`sort()`	Ordina la lista (in-place)	`[3, 1, 2].sort() → [1, 2, 3]`
`reverse()`	Inverte la lista (in-place)	`[1, 2, 3].reverse() → [3, 2, 1]`

Tabella 3: Metodi essenziali per i dizionari (`dict`)

Metodo	Descrizione	Esempio
`keys()`	Restituisce le chiavi	`{"a": 1}.keys()`
`values()`	Restituisce i valori	`{"a": 1}.values()`
`items()`	Restituisce coppie (chiave, valore)	`{"a": 1}.items()`
`get(key, default)`	Ottiene valore, con default	`d.get("k", 0)`
`update(other)`	Aggiorna con altro dizionario	`d.update({"b": 2})`
`pop(key)`	Rimuove e restituisce valore	`d.pop("a")`

Tabella 4: Operazioni principali per i tipi numerici (`int` e `float`)

Funzione	Descrizione	Esempio
`abs(x)`	Valore assoluto	`abs(-5) → 5`
`round(x, n)`	Arrotonda a n decimali	`round(3.14159, 2) → 3.14`
`max(a, b, ...)`	Massimo valore	`max(5, 10, 3) → 10`
`min(a, b, ...)`	Minimo valore	`min(5, 10, 3) → 3`
`sum(iterable)`	Somma degli elementi	`sum([1, 2, 3]) → 6`

3.1 Operatori

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

```
# 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

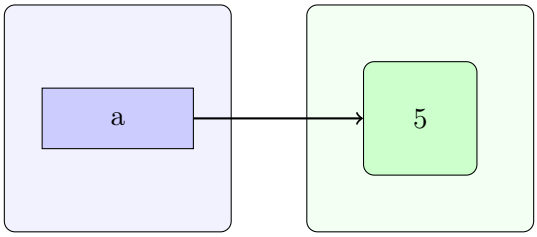
4.1 Il modello di riferimento in Python

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:

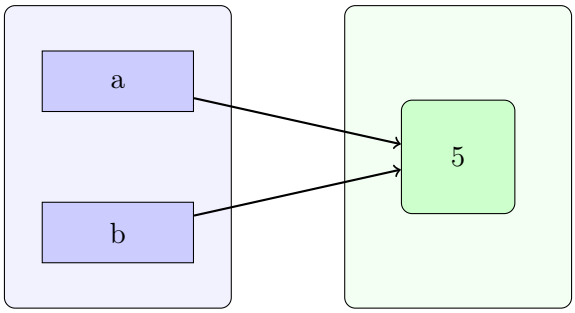


Python crea un oggetto intero con valore 5 in memoria e associa la variabile `a` a questo oggetto. La variabile `a` non "contiene" 5, ma "punta a" o "fa riferimento a" l'oggetto 5 in memoria.

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

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:

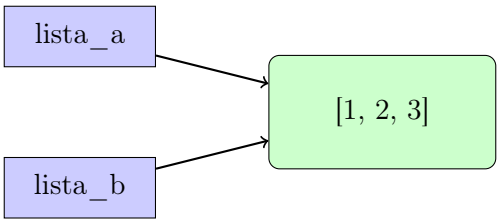
```
a = 5
b = a
a = 10  # Questo NON modifica l'oggetto originale, ma crea un nuovo riferimento
print(b)  # Output: 5 (b continua a riferirsi all'oggetto originale)
```

4.3.2 Comportamento con tipi mutabili

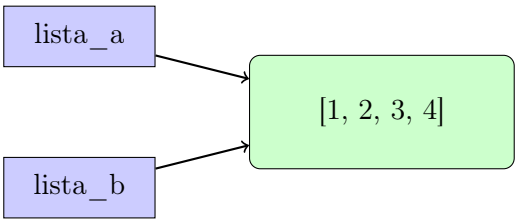
I tipi mutabili come le liste si comportano diversamente:

```
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)`:



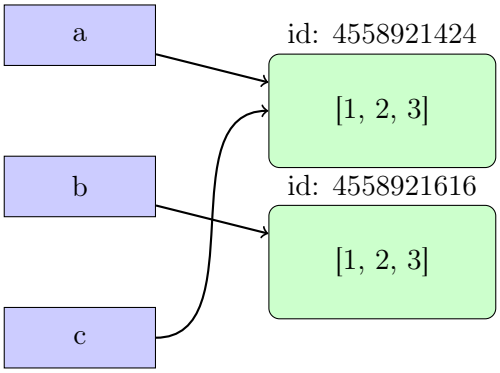
In questo caso, `lista_a.append(4)` modifica direttamente l'oggetto lista in memoria. Poiché sia `lista_a` che `lista_b` puntano allo stesso oggetto, entrambe "vedono" la modifica.

4.4 Verifica dell'identità degli oggetti

Python fornisce due operatori per confrontare le variabili:

- `==` 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)
```

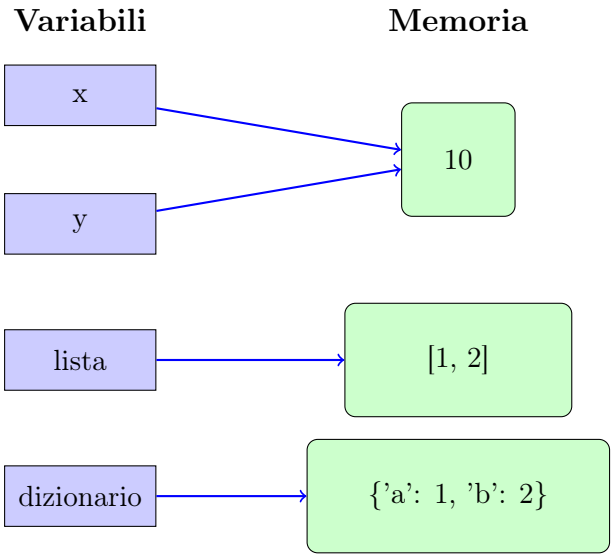


4.5 Esercizio pratico: tracciare le variabili

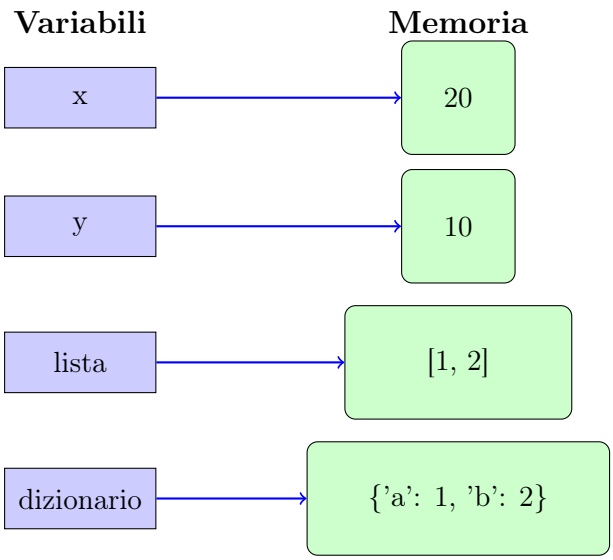
Esaminiamo un esercizio completo che mostra come tracciare le variabili in Python. Prevedere l'output di questo programma richiede di comprendere come Python gestisce i riferimenti in memoria.

```python
# 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

# 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}
```

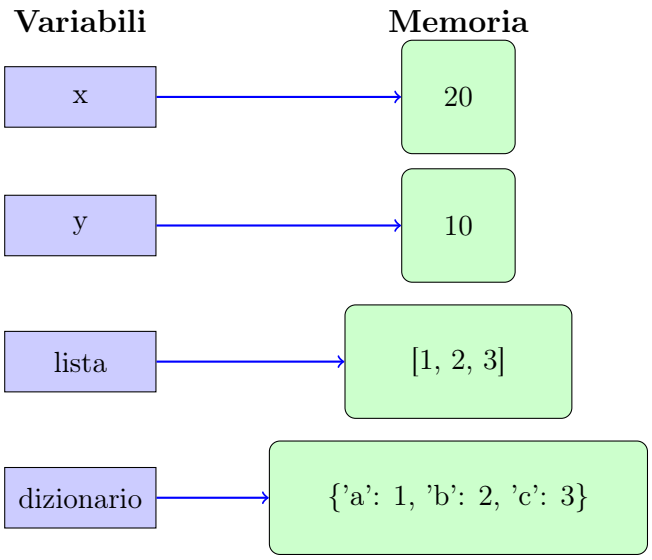
Visualizziamo lo stato della memoria in diversi momenti dell'esecuzione:
Stato iniziale:



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

```python
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

```python
# 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*

```python
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*

```python
# 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


```

```python
# 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*

```python
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*

```python
# 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 e mutabile**[1] 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"]`

[1] Tipi immutabili vs mutabili

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

"mela"	"pera"	"uva"	"banana"	"kiwi"
[0]	[1]	[2]	[3]	[4]

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

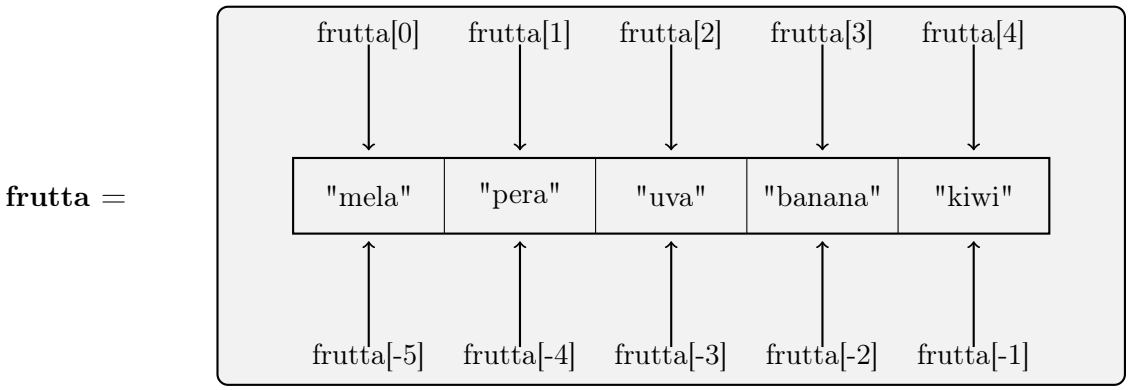
Gli indici negativi sono molto utili quando si vuole accedere agli elementi partendo dalla fine della lista, senza dover conoscere la lunghezza totale della lista. Ad esempio, `[-1]` si riferisce sempre all'ultimo elemento, indipendentemente da quanti elementi contiene la lista. Con l'unica differenza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

Indicizzazione Negativa (dall'ultimo elemento)

"mela"	"pera"	"uva"	"banana"	"kiwi"
[-5]	[-4]	[-3]	[-2]	[-1]

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.

5.1.2 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:

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

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.

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 pi'u elementi
numeri[1:4] = [20, 30, 40]
# Ora numeri 'e [1, 20, 30, 40, 5]

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

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:

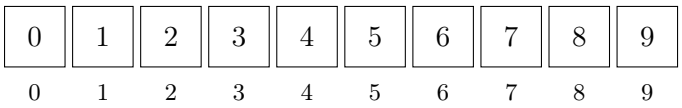
- `inizio` è l'indice da cui iniziare (incluso)
- `fine` è l'indice dove terminare (escluso)
- `passo` è il passo con cui avanzare (opzionale, default 1)

```
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
```

```python
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]`).

5.1.3 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.

Metodi per Aggiungere Elementi Python fornisce diversi metodi per aggiungere elementi a una lista:

```python
# 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
```

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]

# 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 []
```

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:

```
# 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
```

```
# 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
```

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

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

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:

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

# Contare le occorrenze di "verde"
occorrenze = colori.count("verde")    # Risultato: 2

# Trovare l'indice della prima occorrenza di "blu"
indice = colori.index("blu")          # Risultato: 2

# Trovare l'indice della seconda occorrenza di "verde"
indice_secondo_verde = colori.index("verde", 2)  # Risultato: 3
                                                 # Comincia a cercare dall'indice 2
```

Metodi per Ordinamento e Inversione Per ordinare o invertire gli elementi di una lista:

```python
numeri = [3, 1, 4, 1, 5, 9, 2]

# Metodo sort() - ordina la lista in ordine crescente (modifica la lista originale)
numeri.sort()
print(numeri)  # [1, 1, 2, 3, 4, 5, 9]

# Metodo sort() con parametro reverse - ordine decrescente
numeri.sort(reverse=True)
print(numeri)  # [9, 5, 4, 3, 2, 1, 1]

# Funzione sorted() - crea una nuova lista ordinata senza modificare l'originale
parole = ["zebra", "albero", "cane"]
parole_ordinate = sorted(parole)
print(parole)          # ["zebra", "albero", "cane"] (non modificata)
print(parole_ordinate) # ["albero", "cane", "zebra"]

# Metodo reverse() - inverte l'ordine degli elementi
numeri.reverse()
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, usa la funzione `sorted()` per l'ordinamento o lo slicing `[::-1]` per l'inversione.

```
# Metodo copy() - crea una copia superficiale della lista
originale = [1, 2, 3]
copia = originale.copy()
originale.append(4)
print(copia)  # [1, 2, 3] (la copia non 'e influenzata)

# Funzione list() - un altro modo per creare una copia
altra_copia = list(originale)
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]
```

5.2 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.

5.2.1 Variabili come Riferimenti

In Python, quando crei una variabile e le assegni un valore, non stai "mettendo" il valore dentro la variabile. Piuttosto, stai creando un riferimento all'oggetto che contiene quel valore. La variabile è più simile a un'etichetta che "punta" all'oggetto in memoria.

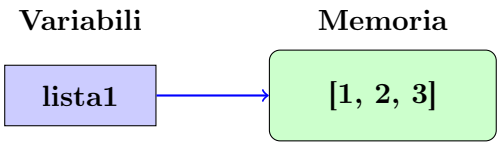
```
# Creazione di una lista
lista1 = [1, 2, 3]

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

5.2.2 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.

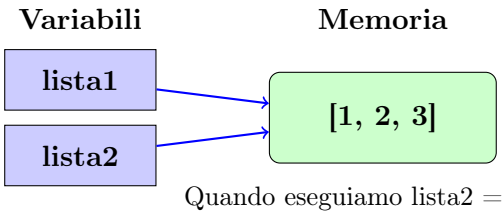
```
# Creazione di una lista
lista1 = [1, 2, 3]

# 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



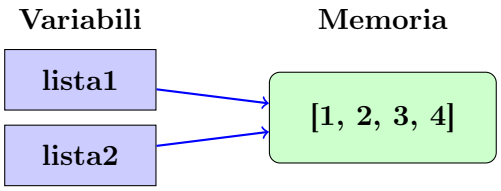
Inizialmente, solo lista1 punta all'oggetto lista in memoria

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.

5.2.3 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.

Osserviamo nel seguente esempio come si comportano le diverse tecniche di copia. Noterai che dopo aver modificato la lista originale con `append()`, le tre copie rimangono invariate, dimostrando che sono effettivamente indipendenti dall'originale:

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

# Tre modi per creare copie
copia1 = originale.copy()
copia2 = originale[:]  # Slicing completo
copia3 = list(originale)

# 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]
```

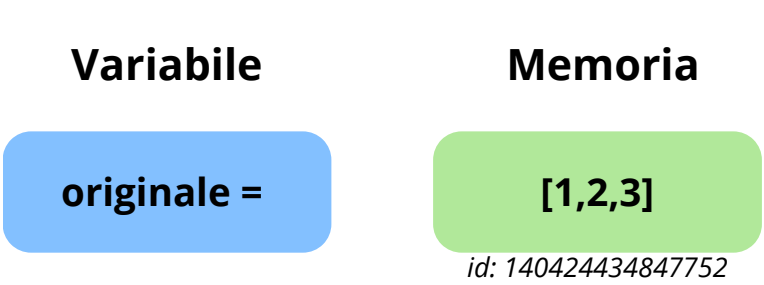
Metodi per la Creazione di Copie Indipendenti Per creare copie indipendenti di una lista, Python mette a disposizione tre metodi principali, tutti ugualmente efficaci per liste contenenti tipi di dati semplici. Vediamo questi metodi in azione e osserviamo come, dopo la modifica della lista originale,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

```python
# 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

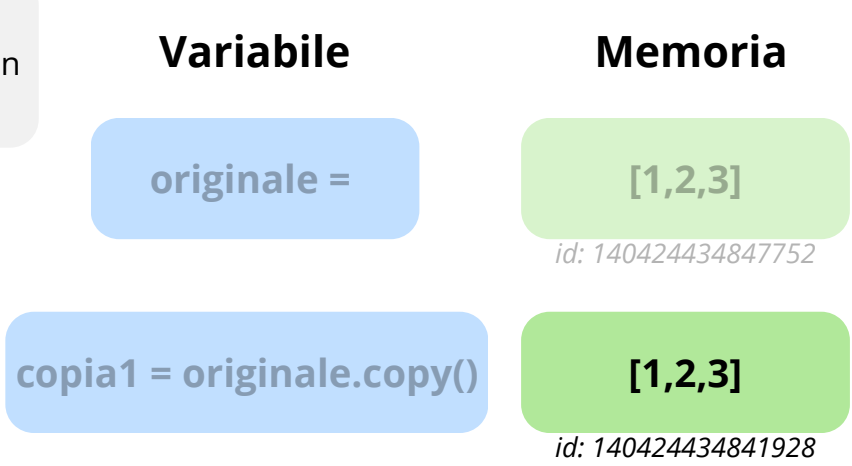


Una variabile in Python è un riferimento a un oggetto in memoria. Quando creiamo una lista, questa viene allocata in memoria e la variabile contiene solo un riferimento (*puntatore)* a quella posizione.

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()

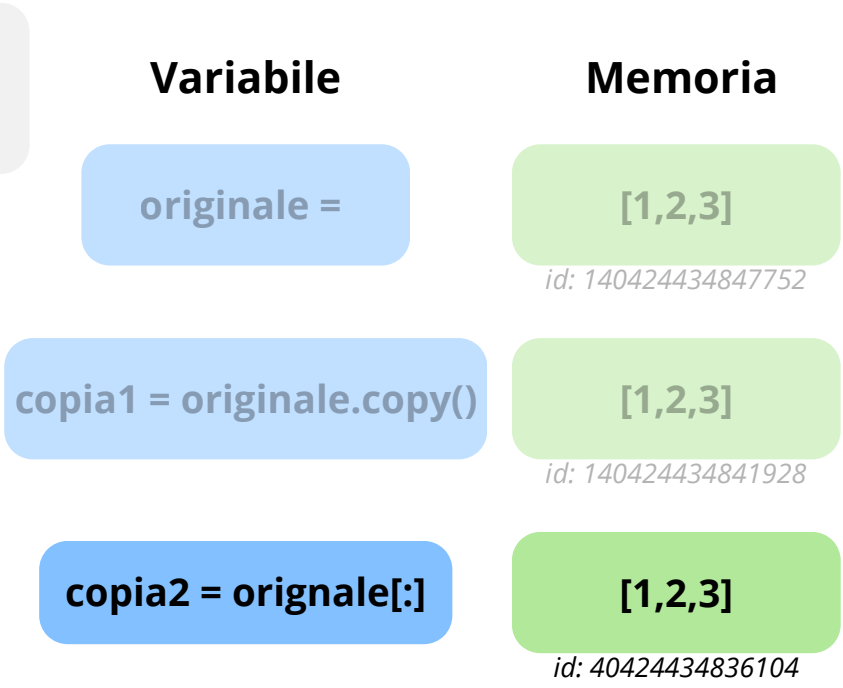
copia1 = originale.copy()
.copy() crea una nuova lista con gli stessi elementi originale



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

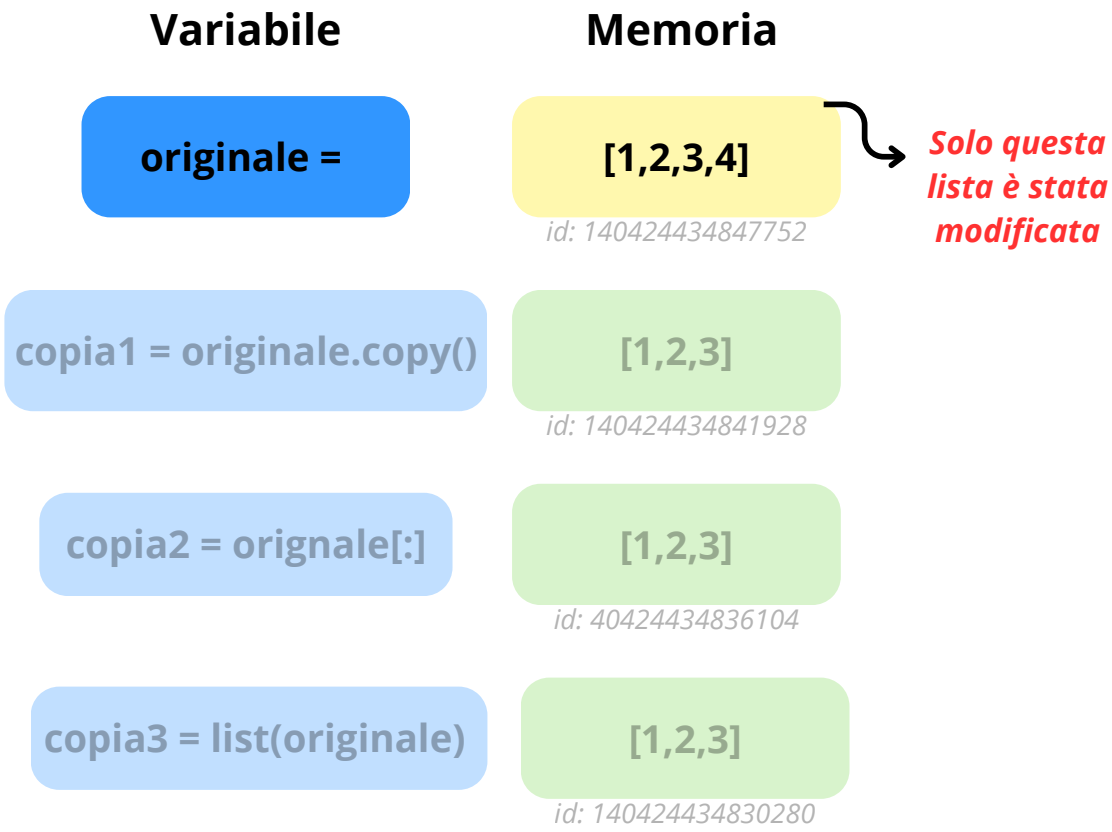
Confronto tra i metodi di copia:

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.



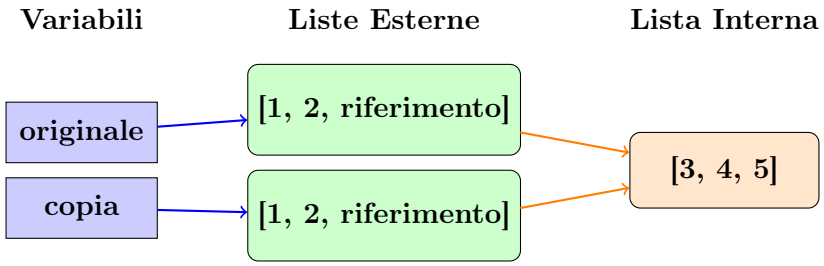
La modifica della lista originale con ***apppend()*** influisce solo su quella lista. Le altre copie rimangono invariate, dimostrando che tutte le tecniche di copia creano oggetti indipendenti della lista ***originale***

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.

```python
# 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, mantenendo i riferimenti agli stessi oggetti interni.

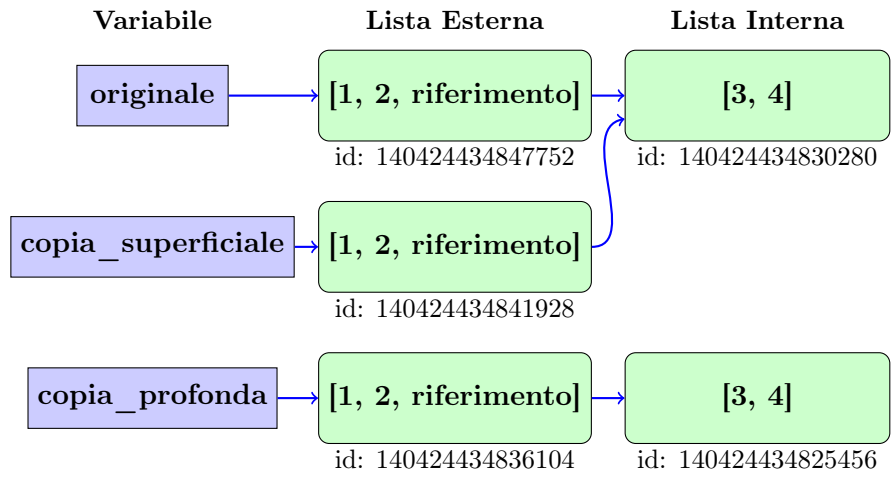


Figura 1: Confronto tra copia superficiale e copia profonda prima della modifica

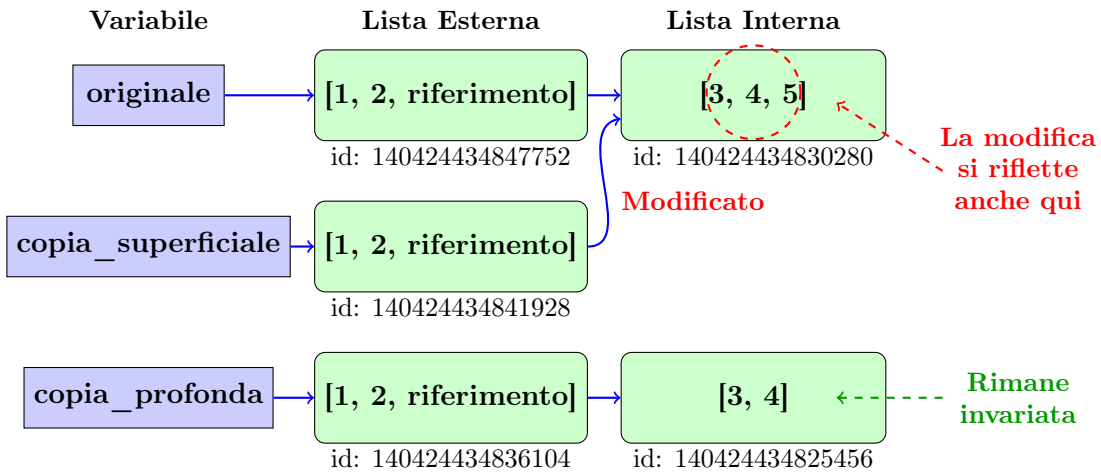
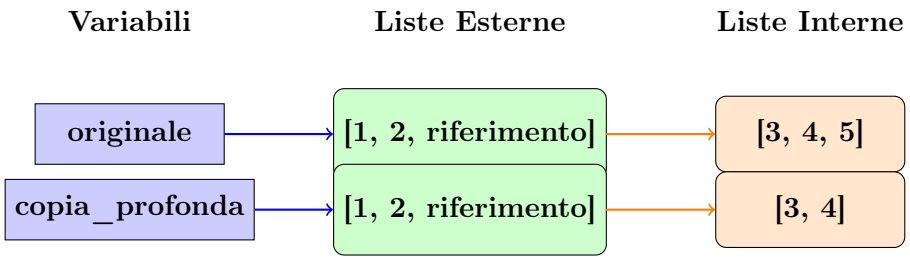


Figura 2: Effetto delle modifiche: differenza tra copia superficiale e profonda

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.

5.2.4 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)

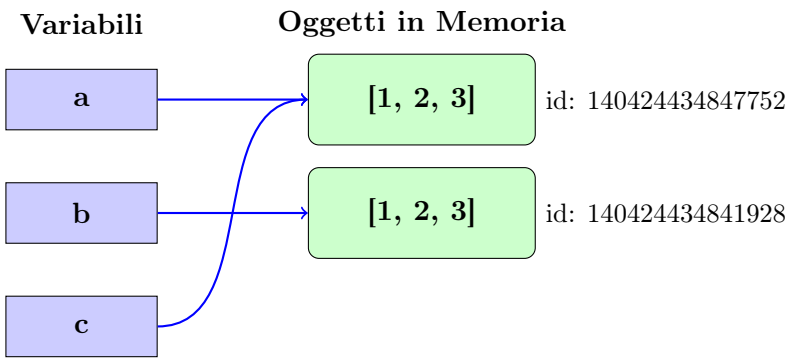
```
a = [1, 2, 3]
b = [1, 2, 3]  # Una nuova lista con gli stessi valori
c = a          # Riferimento alla stessa lista

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

5.2.5 Rappresentazione Visiva della Memoria

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

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

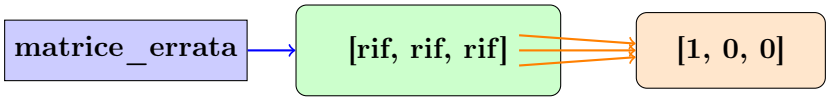


`a == b` è `True` (stesso contenuto)
`a is b` è `False` (oggetti diversi)
`a is c` è `True` (stesso oggetto)

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

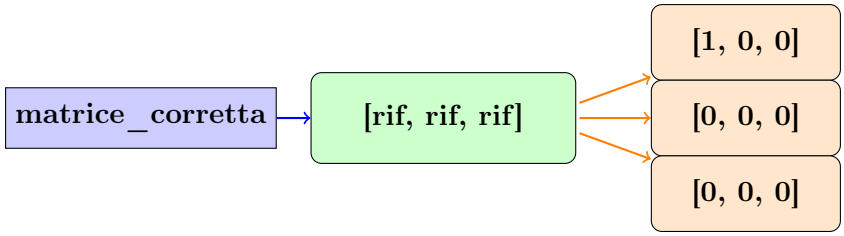
```
# Tentativo ERRATO di creare una matrice 3x3 di zeri
matrice_errata = [[0] * 3] * 3
print(matrice_errata)  # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

# 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

5.2.6 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