

Appunti di Python

Il tuo nome

24 aprile 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	4
1.1	Caratteristiche principali	4
1.2	Installazione	4
1.2.1	Scelta della versione	4
1.2.2	Download e installazione	4
1.2.3	Verifica dell'installazione	5
1.2.4	Gestione dei pacchetti con pip	5
1.2.5	Ambienti virtuali	5
1.2.6	Opzioni alternative	6
1.3	Il tuo primo programma Python	6
2	Variabili e Tipi di Dati	7
2.1	Dichiarazione e Assegnazione delle Variabili	7
2.2	Regole e best practice per denominare le variabili	7
3	Metodi principali per i tipi di dati Python	9
3.1	Operatori	11
3.1.1	Operatori aritmetici	11
3.1.2	Operatori di confronto	11
3.1.3	Operatori logici	11
3.2	Input e Output	11
4	Approfondimento delle Variabili	12
4.1	Il modello di riferimento in Python	12
4.2	Assegnazione di variabili	12
4.3	Tipi immutabili vs mutabili	13
4.3.1	Comportamento con tipi immutabili	13
4.3.2	Comportamento con tipi mutabili	13
4.4	Verifica dell'identità degli oggetti	14
4.5	Esercizio pratico: tracciare le variabili	15
4.6	Esercizi Riassuntivi	18
4.6.1	Esercizio 1: Implementazione Input ed Output	18
4.6.2	Esercizio 2: Identificazione degli errori	18
4.6.3	Esercizio 3: Operazioni Matematiche	18
4.6.4	Esercizio 4: Operatori logici	19
4.6.5	Esercizio 5: MiniCalcolatrice	19
4.6.6	Esercizio 6: Mutabilità e Immutabilità	20
4.7	Risoluzione Esercizi sui Fondamentali	21
4.7.1	Risoluzione Esercizio 1: <i>Esercizio 1: Implementazione Input ed Output</i>	21

4.7.2	Risoluzione Esercizio 2: <i>Esercizio 2: Identificazione degli errori</i> . .	21
4.7.3	Risoluzione Esercizio 3: <i>Esercizio 3: Operazioni Matematiche</i> . . .	21
4.7.4	Risoluzione Esercizio 4: <i>Esercizio 4: Operatori logici</i>	22
4.7.5	Risoluzione Esercizio 5: <i>Esercizio 5: MiniCalcolatrice</i>	23
4.7.6	Risoluzione Esercizio 6: <i>Esercizio 6: Mutabilità e Immutabilità</i> . .	23
5	Presentazione Liste, Tuple, Dizionari, Set	26
5.1	Liste	26
5.1.1	Definizione e Caratteristiche Principali	26
5.1.2	Sintassi di Base	30
5.1.3	Operazioni e Metodi Base	33
5.2	Comportamento in Memoria delle Liste	38
5.2.1	Variabili come Riferimenti	38
5.2.2	Assegnazione e Riferimenti Multipli	38
5.2.3	Creazione di Copie Indipendenti	40
5.2.4	Confronto tra Liste: '==' vs 'is'	48
5.2.5	Rappresentazione Visiva della Memoria	48
5.2.6	Implicazioni Pratiche	50

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:

```
1 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:

```
1 Creazione dell'ambiente virtuale
2 python -m venv mio_ambiente
3 Attivazione dell'ambiente virtuale
4 Su Windows
5 mio_ambiente\Scripts\activate
6 Su macOS/Linux
7 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:

```
1 # Il mio primo programma Python
2 print("Hello, World!") # Stampa un messaggio a schermo
3
4 # Richiesta di input all'utente
5 nome = input("Come ti chiami? ")
6 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 esegilo 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. Questa conoscenza è fondamentale per qualsiasi programmazione efficace in Python.

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

```
1 # Dichiarazione e assegnazione di variabili
2 nome = "Mario"      # Una stringa
3 eta = 30             # Un intero
4 altezza = 1.75       # Float
5 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

```
1 # Esempi di nomi validi
2 nome = "Alice"
3 _eta = 25
4 nome_completo = "Alice Rossi"
5 variabile1 = 42
6 COSTANTE = 3.14159
7
8 # Esempi di nomi non validi
9 # lvariabile = 10      # Inizia con un numero (errore di sintassi)
10 # for = "ciclo"       # E' una parola riservata (errore di sintassi)
11 # 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
<code>upper()</code>	Converte in maiuscolo	<code>"python".upper()</code> → <code>"PYTHON"</code>
<code>lower()</code>	Converte in minuscolo	<code>"Python".lower()</code> → <code>"python"</code>
<code>strip()</code>	Rimuove spazi agli estremi	<code>" Python ".strip()</code> → <code>"Python"</code>
<code>replace(old, new)</code>	Sostituisce le occorrenze	<code>"Hello".replace("l", "x")</code> → <code>"Hexxo"</code>
<code>split(sep)</code>	Divide in una lista	<code>"a,b,c".split(",")</code> → <code>["a", "b", "c"]</code>
<code>join(iterable)</code>	Unisce una lista in stringa	<code>"-".join(["a", "b"])</code> → <code>"a-b"</code>
<code>find(sub)</code>	Trova posizione sottostringa	<code>"Python".find("th")</code> → <code>2</code>

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

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

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

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

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

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

3.1 Operatori

3.1.1 Operatori aritmetici

```
1 a = 10
2 b = 3
3
4 somma = a + b      # 13
5 differenza = a - b  # 7
6 prodotto = a * b    # 30
7 divisione = a / b   # 3.3333... (divisione che restituisce float)
8 div_intera = a // b # 3 (divisione intera)
9 modulo = a % b      # 1 (resto della divisione)
10 potenza = a ** b   # 1000 (a elevato alla b)
```

3.1.2 Operatori di confronto

```
1 a == b # Uguaglianza (False)
2 a != b # Disuguaglianza (True)
3 a > b  # Maggiore (True)
4 a < b  # Minore (False)
5 a >= b # Maggiore o uguale (True)
6 a <= b # Minore o uguale (False)
```

3.1.3 Operatori logici

```
1 x = True
2 y = False
3
4 x and y # False (AND logico)
5 x or y  # True (OR logico)
6 not x   # False (NOT logico)
```

3.2 Input e Output

```
1 # Output sulla console
2 print("Ciao, mondo!")
3
4 # Input da tastiera
5 nome = input("Come ti chiami? ")
6 print(f"Ciao, {nome}!")
7
8 # Formattazione delle stringhe
9 eta = 25
10 messaggio = f"Ho {eta} anni."
11 print(messaggio)
```

4 Approfondimento delle Variabili

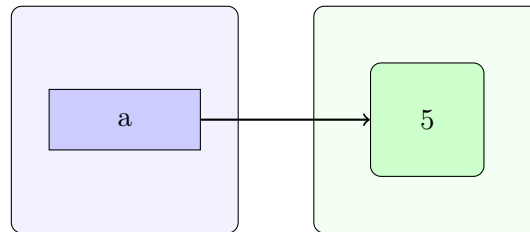
4.1 Il modello di riferimento in Python

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.

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

Quando eseguiamo questa istruzione, ecco cosa succede:



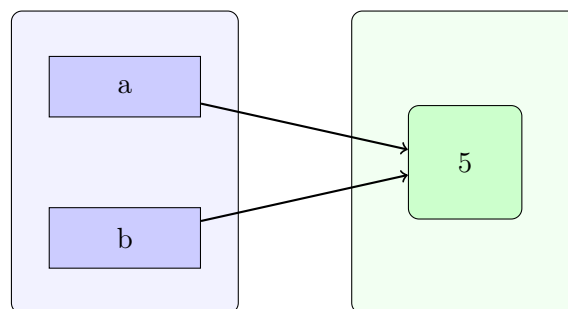
Python crea un oggetto intero con valore 5 in memoria e associa la variabile **a** a questo oggetto.

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:

```
1 a = 5
2 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:

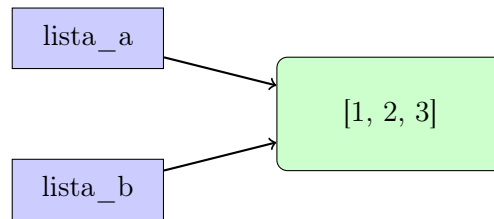
```
1 a = 5
2 b = a
3 a = 10 # Questo NON modifica l'oggetto originale, ma crea un nuovo riferimento
4 print(b) # Output: 5 (b continua a riferirsi all'oggetto originale)
```

4.3.2 Comportamento con tipi mutabili

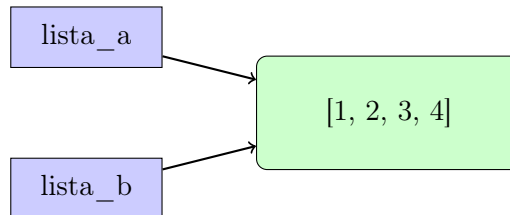
I tipi mutabili come le liste si comportano diversamente:

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

Stato iniziale:



Dopo `lista_a.append(4)`:



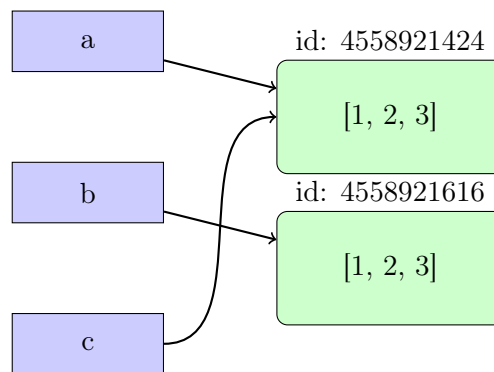
In questo caso, `lista_a.append(4)` modifica direttamente l'oggetto lista in memoria. Poiché sia `lista_a` che `lista_b` puntano allo stesso oggetto, entrambe "vedono" la modifica.

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)

```
1 a = [1, 2, 3]
2 b = [1, 2, 3] # Una nuova lista con gli stessi valori
3 c = a        # Riferimento alla stessa lista
4
5 print(a == b) # True (stesso valore)
6 print(a is b) # False (oggetti diversi)
7 print(a is c) # True (stesso oggetto)
8
9 # Possiamo verificare l'identità anche con id()
10 print(id(a)) # Un numero unico che identifica l'oggetto
11 print(id(b)) # Un numero diverso
12 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.

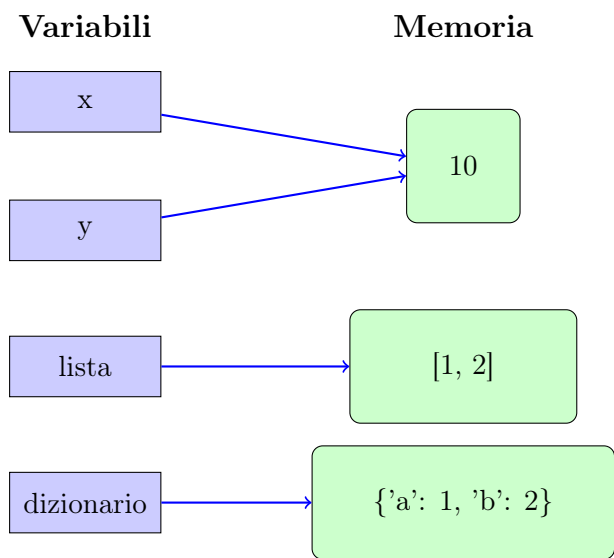
```

1 # Stato iniziale
2 x = 10
3 y = x
4 lista = [1, 2]
5 dizionario = {'a': 1, 'b': 2}
6
7 # Modifichiamo le variabili
8 x = 20
9 lista.append(3)
10 dizionario['c'] = 3
11
12 # Verifichiamo lo stato finale
13 print(x)      # Output: 20
14 print(y)      # Output: 10
15 print(lista)   # Output: [1, 2, 3]
16 print(dizionario) # Output: {'a': 1, 'b': 2, 'c': 3}

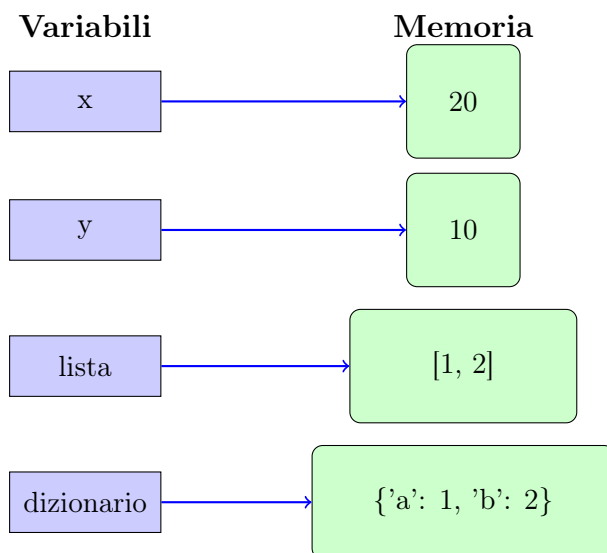
```

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

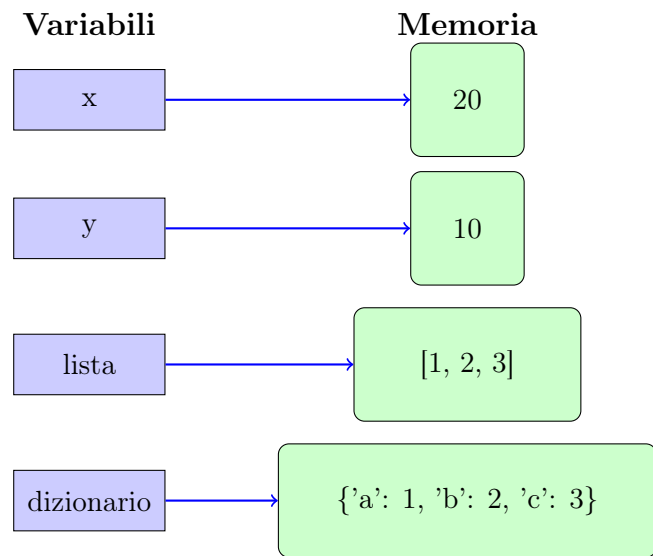
Stato iniziale:



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.

```
1 # Il mio primo programma Python
2 print("Hello, World!") # Stampa un messaggio a schermo
3
4 # Richiesta di input all'utente
5 nome = input("Come ti chiami? ")
6 print(f"Ciao {nome}, benvenuto nel mondo di Python!")
7
8 # Aggiungi a questo script la possibilità di chiedere all'utente la sua età
9
10 #Calcolare l'anno di nascita approssimativo (anno corrente - età)
11
12 #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".

```
1 inome = "Paolo"
2 nome-cognome = "Mario Rossi"
3 class = "Principiante"
4
5 # 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

```
1 # Dichiarare le 2 variabili
2 numero1 = # immetti un tuo numero a piacere
3 numero2 = # immetti un tuo numero a piacere
```

```

4
5     # Esegui tutte le operazioni matematiche (addizione, sottrazione, moltiplicazione,
↪ divisione, divisione intera, modulo, potenza)
6
7 somma =
8 sottrazione =
9 divisione =
10 moltiplicazione =
11 divisione_intera =
12 modulo =
13 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*

```

1 a = 5
2 b = 10
3 c = 15
4 risultato = (a < b) and (b <= c) or not (a == c)
5 print(risultato)
6
7 # 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

```

1 # Chiediamo all'utente il primo numero
2 primo_numero =
3 # Chiediamo il secondo numero
4 secondo_numero =
5
6 # Esegui le operazioni matematiche
7 somma =
8 divisione =
9 moltiplicazione =
10
11 # 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

```
1 # Esempio 1
2 x = "hello"
3 y = x
4 x = "world"
5 print(y) # Cosa verra stampato?
6
```

```

7 # Esempio 2
8 lista_a = [1, 2, 3]
9 lista_b = lista_a
10 lista_a[0] = 99
11 print(lista_b) # Cosa verra stampato?

```

4.7 Risoluzione Esercizi sui Fondamentali

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

```

1 # Il mio primo programma Python
2 print("Hello, World!") # Stampa un messaggio a schermo
3
4 # Richiesta di input all'utente
5 nome = input("Come ti chiami? ")
6 print(f"Ciao {nome}, benvenuto nel mondo di Python!")
7
8 # Aggiungi a questo script la possibilita di chiedere all'utente la sua eta
9 eta_utente = int(input("Digita la tua eta:\n"))
10
11 #Calcolare l'anno di nascita approssimativo (anno corrente - eta)
12 anno_corrente = int(input("Digita l'anno corrente"))
13 anno_nascita = anno_corrente - eta_utente
14
15
16 #Stampare un messaggio personalizzato che includa il nome e l'anno di nascita
17 print(f"Ciao {nome}, sono nato/a il {anno_nascita}")

```

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

```

1 lname = "Paolo" # Inizia con un numero
2 nome-cognome = "Mario Rossi" # Usa il trattino
3 class = "Principiante" # Usa una parola riservata
4
5 # Di seguito fornisci le tue correzioni
6 nome = "Paolo"
7 nome_cognome = "Mario Rossi"
8 classe = "Principiante"

```

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

```

1 # Dichiarare le 2 variabili
2 numero1 = 12
3 numero2 = 56
4
5 # Esegui tutte le operazioni matematiche (addizione, sottrazione, moltiplicazione,
↪ divisione, divisione intera, modulo, potenza)

```

```

6
7 somma = numero1 + numero2
8 sottrazione = numero1 - numero2
9 divisione = numero1 / numero2
10 moltiplicazione = numero1 * numero2
11 divisione_intera = numero1 // numero2
12 modulo = numero1 % numero2
13 potenza = numero1**numero2
14
15
16 # Stampiamo i risultati
17 print(somma, " somma") # Output 68
18 print(sottrazione, " sottrazione") # Output -44
19 print(divisione, " divisione") # Output 0.21428571428571427
20 print(moltiplicazione, " moltiplicazione") # Output 672
21 print(divisione_intera, " divisione con intero") # Output 0
22 print(modulo, " resto della divisione") # Output 12
23 print(potenza, " potenza di numero1 alla numero2") # Output 2.71 10^60

```

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

```

1 a = 5
2 b = 10
3 c = 15
4 risultato = (a < b) and (b <= c) or not (a == c)
5 print(risultato)
6
7 # Spiega perche il risultato del print e' True
8
9 """
10 Per comprendere perche il risultato dell'espressione e True, dobbiamo analizzarla passo
  ↳ per passo seguendo l'ordine di valutazione degli operatori in Python.
11 Prima valutiamo tutte le espressioni di confronto:
12 (a < b) significa (5 < 10), che e True
13 (b <= c) significa (10 <= 15), che e True
14 (a == c) significa (5 == 15), che e False
15 Poi applichiamo l'operatore not:
16 not (a == c) diventa not False, che e True
17 A questo punto la nostra espressione equivale a:
18 True and True or True
19 Ora applichiamo l'operatore and, che ha precedenza rispetto a or:
20 True and True risulta in True
21 Infine, applichiamo l'operatore or:
22 True or True risulta in True
23 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.
24 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.
25

```

26 """

4.7.5 Risoluzione Esercizio 5: *Esercizio 5: MiniCalcolatrice*

```
1 # Chiediamo all'utente il primo numero
2 primo_numero = int(input("Digita un numero intero\n"))
3 # Chiediamo il secondo numero
4 secondo_numero = int(input("Digita un secondo numero intero\n"))
5
6 # Esegui le operazioni matematiche
7 somma = primo_numero + secondo_numero
8 divisione = primo_numero / secondo_numero
9 moltiplicazione = primo_numero * secondo_numero
10
11 # Stampa i risultati delle operazioni
12 print(f"Somma: {somma}")           # Output 16
13 print(f"Divisione: {divisione}")   # Output 3
14 print(f"Moltiplicazione: {moltiplicazione}") # Output 48
15
16
17 # Avremmo potuto usare il modulo float al posto di int per avere numeri decimali
```

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

```
1
2 # Esempio 1
3 x = "hello"
4 y = x
5 x = "world"
6 print(y) # Cosa verrà stampato?
7
8 # Esempio 2
9 lista_a = [1, 2, 3]
10 lista_b = lista_a
11 lista_a[0] = 99
12 print(lista_b) # Cosa verrà stampato?
13
14 """
15
16 Analisi passo per passo
17
18 Creazione della prima variabile:
19 x = "hello"
20 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 potrà essere
   ↳ modificato.
21 Creazione della seconda variabile:
22 y = x
```

```

23 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.
24 Riassegnazione della prima variabile:
25 x = "world"
26 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".
27 Stampa del valore:
28 print(y)
29 Quando stampiamo y, otteniamo il valore dell'oggetto a cui y fa riferimento, che e ancora
    ↳ "hello".
30
31 Risultato
32 La risposta alla domanda "Cosa verra stampato?" e: hello
33 Spiegazione del comportamento
34 Questo esempio illustra perfettamente il comportamento dei tipi immutabili in Python:
35
36 Le stringhe sono tipi immutabili, quindi non possono essere modificate dopo la creazione.
37 Quando assegniamo un valore a una variabile (x = "hello"), creiamo un riferimento a un
    ↳ oggetto.
38 Quando copiamo una variabile in un'altra (y = x), entrambe le variabili puntano allo
    ↳ stesso oggetto.
39 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.
40 La variabile che non e stata riassegnata (y) continua a puntare all'oggetto originale.
41
42 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.
43
44
45
46
47
48 ----- Rappresentazione Grafica della memoria -----
49
50
51      Dopo x = "hello":
52      Variabili      Memoria
53      -----      -
54      x      ---->  "hello"
55
56
57      Dopo y = x:
58      Variabili      Memoria
59      -----      -
60      x      ---->  "hello"
61      y      ----/
62
63

```



```
64     Dopo x = "world":
65     Variabili      Memoria
66     -----
67         x      ---->  "world"
68         y      ---->  "hello"
69
70
71
72     """
```

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

¹Tipi immutabili vs mutabili

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

"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:

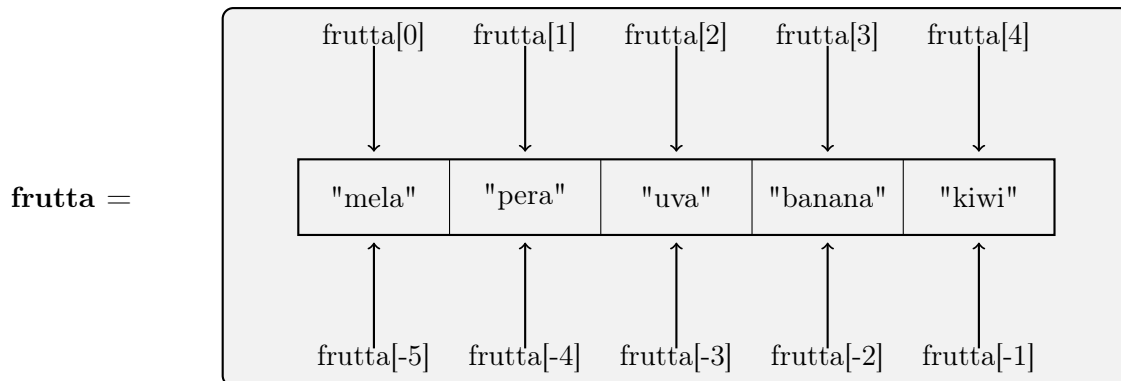
Gli indici negativi sono molto utili quando si vuole accedere agli elementi partendo dalla fine della lista, senza dover conoscere la lunghezza totale della lista. Ad esempio, [-1] si riferisce sempre all'ultimo elemento, indipendentemente da quanti elementi contiene la lista. Con l'unica differenza 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"]`:

```
1 frutta[0] # restituisce "mela" (primo elemento)
2 frutta[2] # restituisce "uva" (terzo elemento)
3 frutta[-1] # restituisce "kiwi" (ultimo elemento)
4 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:

```
1 # Lista vuota
2 lista_vuota = []
3
4 # Lista di numeri
5 numeri = [1, 2, 3, 4, 5]
6
7 # Lista di stringhe
8 nomi = ["Anna", "Bruno", "Carlo", "Daria"]
9
10 # Lista con elementi di tipi diversi
11 lista_mista = [1, "Python", 3.14, True]
12
13 # Lista nidificata (liste all'interno di liste)
14 matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
15
16 # Creazione di lista mediante la funzione list()
17 caratteri = list("Python") # Crea la lista ['P', 'y', 't', 'h', 'o', 'n']
18 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
<code>l</code> o <code>lst</code>	<code>studenti</code>	Lista di nomi di studenti
<code>nums</code>	<code>voti_esame</code>	Lista di voti numerici
<code>x</code>	<code>coordinate_x</code>	Lista di valori delle coordinate x
<code>lista_dati</code>	<code>temperature_mensili</code>	Lista di temperature

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

```
1 colori = ["rosso", "verde", "blu", "giallo", "viola"]
2
3 # Accesso al primo elemento (indice 0)
4 primo_colore = colori[0] # "rosso"
5
6 # Accesso al terzo elemento (indice 2)
7 terzo_colore = colori[2] # "blu"
8
9 # Accesso all'ultimo elemento (indice -1)
10 ultimo_colore = colori[-1] # "viola"
11
12 # Accesso al penultimo elemento (indice -2)
13 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:

```
1 # Lista iniziale
2 frutta = ["mela", "banana", "pera", "arancia"]
3
4 # Modifica del secondo elemento
5 frutta[1] = "kiwi"
6 # Ora frutta è ["mela", "kiwi", "pera", "arancia"]
7
8 # Modifica dell'ultimo elemento usando indice negativo
9 frutta[-1] = "limone"
10 # Ora frutta è ["mela", "kiwi", "pera", "limone"]
```

È anche possibile modificare più elementi contemporaneamente utilizzando lo slicing:

```
1 numeri = [1, 2, 3, 4, 5]
2
3 # Sostituzione di piú elementi
4 numeri[1:4] = [20, 30, 40]
5 # Ora numeri è [1, 20, 30, 40, 5]
6
7 # Si pu' anche sostituire con un numero diverso di elementi
8 numeri[1:4] = [200, 300]
9 # Ora numeri è [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)

```
1 numeri = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2
3 # Elementi dal secondo al quarto (indici 1, 2, 3)
4 sottosequenza = numeri[1:4] # [1, 2, 3]
5
6 # Primi tre elementi
7 inizio = numeri[:3] # [0, 1, 2]
8
9 # Elementi dal quarto fino alla fine
10 fine = numeri[3:] # [3, 4, 5, 6, 7, 8, 9]
```



```

11
12 # Ultimi tre elementi
13 ultimi = numeri[-3:] # [7, 8, 9]
14
15 # Tutti gli elementi tranne gli ultimi due
16 senza_ultimi = numeri[:-2] # [0, 1, 2, 3, 4, 5, 6, 7]
17
18 # Elementi con indici pari (passo 2)
19 pari = numeri[::2] # [0, 2, 4, 6, 8]
20
21 # Elementi in ordine inverso
22 inverso = numeri[::-1] # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

```

Visualizzazione dello Slicing

Lista originale:

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

`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:

```
1 # Creazione di una lista di base
2 frutti = ["mela", "banana"]
3
4 # Metodo append() - aggiunge un elemento alla fine della lista
5 frutti.append("arancia")
6 # Ora frutti 'e ["mela", "banana", "arancia"]
7
8 # Metodo insert() - inserisce un elemento in una posizione specifica
9 frutti.insert(1, "pera") # Inserisce "pera" all'indice 1
10 # Ora frutti 'e ["mela", "pera", "banana", "arancia"]
11
12 # Metodo extend() - aggiunge tutti gli elementi di un'altra lista
13 altri_frutti = ["kiwi", "ananas"]
14 frutti.extend(altri_frutti)
15 # 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:

```
1 lista1 = [1, 2, 3]
2 lista2 = [4, 5]
3
4 lista1.append(lista2) # lista1 diventa [1, 2, 3, [4, 5]]
5                       # La lista2 'e aggiunta come unico elemento
6
7 lista3 = [1, 2, 3]
8 lista3.extend(lista2) # lista3 diventa [1, 2, 3, 4, 5]
9                       # Gli elementi di lista2 sono aggiunti singolarmente
```

Metodi per Rimuovere Elementi Per rimuovere elementi da una lista, Python offre vari metodi:

```
1 numeri = [10, 20, 30, 40, 50, 30]
2
3 # Metodo remove() - rimuove la prima occorrenza di un valore
4 numeri.remove(30) # Rimuove il primo 30 trovato
5 # Ora numeri 'e [10, 20, 40, 50, 30]
6
7 # Metodo pop() - rimuove e restituisce l'elemento all'indice specificato
8 elemento = numeri.pop(1) # Rimuove e restituisce il secondo elemento (20)
9 print(elemento) # 20
10 # Ora numeri 'e [10, 40, 50, 30]
11
12 # Metodo pop() senza argomenti - rimuove e restituisce l'ultimo elemento
13 ultimo = numeri.pop() # Rimuove e restituisce l'ultimo elemento (30)
14 print(ultimo) # 30
```

```

15 # Ora numeri 'e [10, 40, 50]
16
17 # Istruzione del - rimuove l'elemento o la sezione specificata
18 del numeri[0] # Rimuove il primo elemento
19 # Ora numeri 'e [40, 50]
20
21 # Metodo clear() - svuota completamente la lista
22 numeri.clear()
23 # 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:

```

1 # Esempio di ValueError con remove()
2 numeri = [1, 2, 3]
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

```

```

1 # Esempio di IndexError con pop()
2 numeri = [1, 2, 3]
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:

```

1 colori = ["rosso", "verde", "blu", "verde", "giallo"]
2
3 # Funzione len() - restituisce il numero di elementi
4 lunghezza = len(colori)
5 print(lunghezza) # 5
6
7 # Metodo count() - conta quante volte un elemento appare nella lista
8 occorrenze_verde = colori.count("verde")
9 print(occorrenze_verde) # 2
10
11 # Metodo index() - trova l'indice della prima occorrenza di un elemento
12 indice_blu = colori.index("blu")
13 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

```

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

```

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

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

```
1 # Creazione di una lista di numeri
2 numeri = [5, 2, 8, 1, 9, 3]
3
4 # Manipolazione della lista
5 numeri.append(7)          # Aggiunge 7 alla fine: [5, 2, 8, 1, 9, 3, 7]
6 numeri.insert(0, 0)       # Inserisce 0 all'inizio: [0, 5, 2, 8, 1, 9, 3, 7]
7 numeri.remove(9)          # Rimuove il 9: [0, 5, 2, 8, 1, 3, 7]
8 elemento = numeri.pop(3)   # Rimuove e restituisce l'elemento all'indice 3 (8)
9                             # numeri ora 'e [0, 5, 2, 1, 3, 7]
10
11 # Informazioni sulla lista
12 lunghezza = len(numeri)   # 6
13 occorrenze = numeri.count(5) # 1
14 posizione = numeri.index(3) # 4
15
16 # Ordinamento e inversione
17 numeri.sort()             # [0, 1, 2, 3, 5, 7]
18 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.

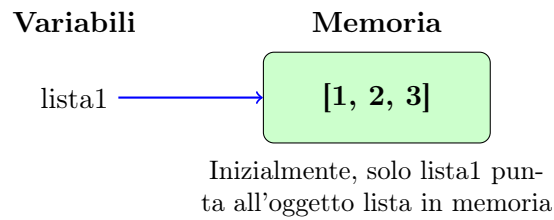
```
1 # Creazione di una lista
2 lista1 = [1, 2, 3]
3
4 # id() mostra l'identificatore univoco dell'oggetto in memoria
5 print(id(lista1)) # Ad esempio: 140424434847752
```

5.2.2 Assegnazione e Riferimenti Multipli

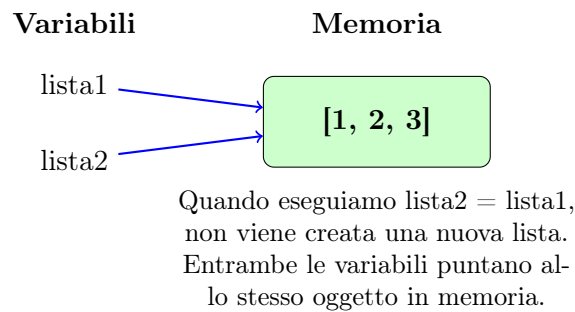
Quando assegni una lista a una seconda variabile, **non viene creata una nuova lista**, ma entrambe le variabili fanno riferimento alla stessa lista in memoria. Modificare la lista attraverso una variabile influenzerà anche l'altra.

```
1 # Creazione di una lista
2 lista1 = [1, 2, 3]
3
4 # Assegnazione a una nuova variabile
5 lista2 = lista1 # Non crea una copia!
6
7 # Modifica attraverso lista1
8 lista1.append(4)
9
10 # Entrambe le liste sono state modificate
11 print(lista1) # [1, 2, 3, 4]
12 print(lista2) # [1, 2, 3, 4]
13
14 # Verifica che puntano allo stesso oggetto
15 print(id(lista1) == id(lista2)) # True
```

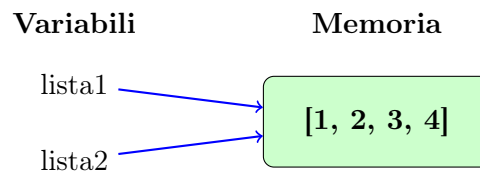
Fase 1: Creazione di lista1



Fase 2: Assegnazione a lista2



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:

```

1 # Creazione di una lista
2 originale = [1, 2, 3]
3
4 # Tre modi per creare copie
5 copia1 = originale.copy()
6 copia2 = originale[:] # Slicing completo
7 copia3 = list(originale)
8
9 # Modifica dell'originale
10 originale.append(4)
11
12 # Le copie non sono influenzate
13 print(originale) # [1, 2, 3, 4]
14 print(copia1)   # [1, 2, 3]
15 print(copia2)   # [1, 2, 3]
16 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:

```

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

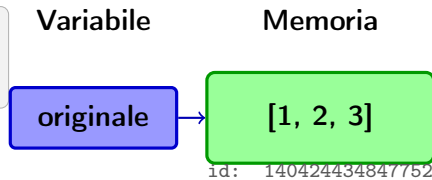
```

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.

COPIA DI LISTE IN PYTHON

Fase 1: Creazione della lista originale

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



Verifica in Python

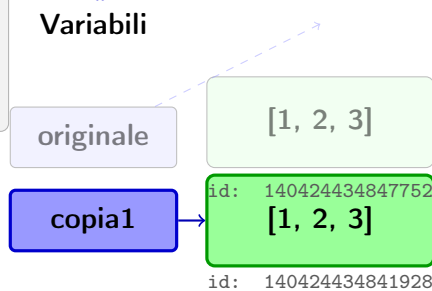
```
>>> id(originale)
140424434847752
```

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 creazione.

Fase 2: Creazione di copia1 con metodo .copy()

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



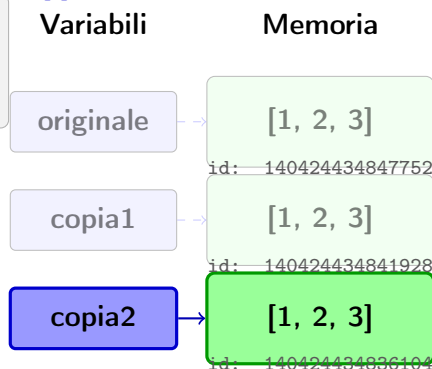
Verifica in Python

```
>>> id(copia1)
140424434841928 >>
copia1 is originale
False >> copia1 ==
originale True
```

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. L'operatore `is` verifica se due variabili fanno riferimento allo stesso oggetto.

Fase 3: Creazione di copia2 con slicing [:]

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



Verifica in Python

```
>>> id(copia2)
140424434836104 >>
copia1 == copia2 True >>
copia1 is copia2 False
```

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

COPIA DI LISTE IN PYTHON (CONTINUAZIONE)

Fase 4: Creazione di copia3 con costruttore list()

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

Variabili

Memoria

Verifica in Python

originale

[1, 2, 3]

id: 140424434847752

copia1

[1, 2, 3]

id: 140424434841928

copia2

[1, 2, 3]

id: 140424434836104

copia3

[1, 2, 3]

id: 140424434830280

```
>> id(copia3)
140424434830280 >>
copia1 == copia2
== copia3 True >>
type(copia3) <class
'list'>
```

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	Sintassi	Casi d'uso / Note
<code>.copy()</code>	<code>lista.copy()</code>	Esplicito, solo Python 3+
Slicing	<code>lista[:]</code>	Idiomatico, tutte le versioni
<code>list()</code>	<code>list(lista)</code>	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: `originale.append(4)` Modifica lista
`print(originale)` [1, 2, 3, 4]
`print(copia1)` [1, 2, 3]
`print(copia2)` [1, 2, 3]
`print(copia3)` [1, 2, 3]

originale

Variabili

Memoria

originale

[1, 2, 3, 4]

id: 140424434847752

copia1

[1, 2, 3]

id: 140424434841928

copia2

[1, 2, 3]

id: 140424434836104

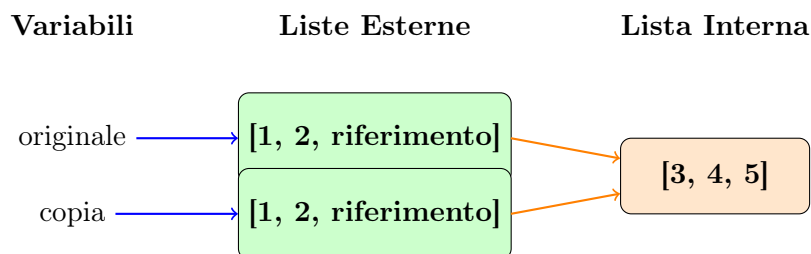
Solo questa lista è modificata!

45

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

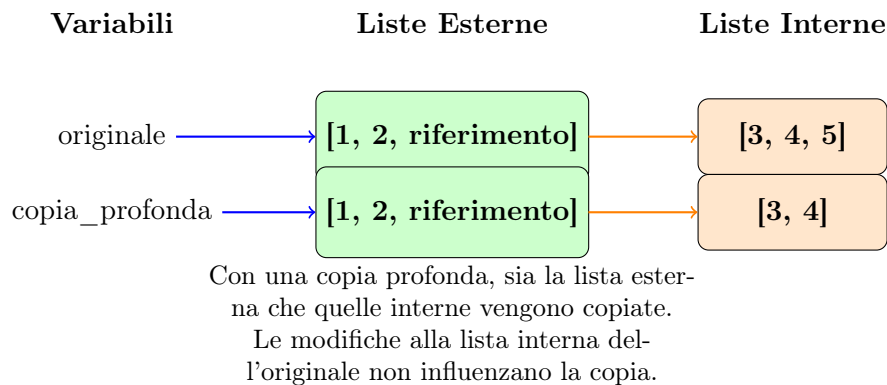
```
1 # Lista con una lista annidata
2 originale = [1, 2, [3, 4]]
3
4 # Creazione di una copia superficiale
5 copia = originale.copy()
6
7 # Modifica della lista interna nell'originale
8 originale[2].append(5)
9
10 # La modifica si riflette anche nella copia!
11 print(originale) # [1, 2, [3, 4, 5]]
12 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.

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

```
1 import copy
2
3 # Lista con una lista annidata
4 originale = [1, 2, [3, 4]]
5
6 # Creazione di una copia profonda
7 copia_profonda = copy.deepcopy(originale)
8
9 # Modifica della lista interna nell'originale
10 originale[2].append(5)
11
12 # La copia profonda non 'e influenzata
13 print(originale)     # [1, 2, [3, 4, 5]]
14 print(copia_profonda) # [1, 2, [3, 4]]
```



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)

```

1 a = [1, 2, 3]
2 b = [1, 2, 3] # Una nuova lista con gli stessi valori
3 c = a        # Riferimento alla stessa lista
4
5 print(a == b) # True - stesso contenuto
6 print(a is b) # False - oggetti diversi in memoria
7 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)

```

1 a = [1, 2, 3]
2 b = [1, 2, 3] # Una nuova lista con gli stessi valori
3 c = a        # Riferimento alla stessa lista
4
5 print(a == b) # True - stesso contenuto
6 print(a is b) # False - oggetti diversi in memoria
7 print(a is c) # True - stesso oggetto in memoria
8
9 # Visualizzazione degli id per conferma
10 print(id(a)) # Ad esempio: 140424434847752
11 print(id(b)) # Diverso da a
12 print(id(c)) # Stesso di a

```


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