

# **Corso Python**

**Modulo 2: Strutture Dati Core**



# Nel Modulo Precedente

Nel **Modulo 1** abbiamo imparato:

- Variabili: `nome = "Mario"`
- Tipi primitivi: `str` , `int` , `float` , `bool`
- Operatori: matematici e logici

**Problema:** Come gestiamo MOLTI dati?

```
# Brutto e non scalabile!
studente1 = "Mario"
studente2 = "Luigi"
# ... e se fossero 100?
```

**Soluzione:** Le **Strutture Dati** - collezioni di elementi!

# Agenda del Modulo (2 Ore)

Organizzare i dati è fondamentale quanto elaborarli.

1. **Liste**: Sequenze ordinate e modificabili.
2. **Tuple**: Sequenze immutabili (costanti).
3. **Dizionari**: Associazioni Chiave-Valore.
4. **Set**: Insiemi di elementi unici.
5. **Confronto**: Quando usare cosa?
6. **Laboratorio**: Gestione magazzino dati.

# 1. Le Liste (Lists)

Flessibilità e Ordine

# Cos'è una Lista?

Una **lista** è la struttura dati più versatile e utilizzata in Python. È il punto di partenza per organizzare collezioni di dati.

## Caratteristiche fondamentali:

Proprietà	Significato
<b>Ordinata</b>	Gli elementi mantengono l'ordine di inserimento
<b>Mutabile</b>	Può essere modificata dopo la creazione
<b>Eterogenea</b>	Può contenere tipi diversi (int, str, altre liste...)
<b>Dinamica</b>	Può crescere o ridursi a runtime

```
# Sintassi: parentesi quadre []
lista_vuota = []
numeri = [1, 2, 3, 4, 5]
mista = ["testo", 42, 3.14, True, [1, 2]]
```

# Creazione e Accesso

Una lista è una sequenza ordinata di elementi racchiusa tra parentesi quadre `[]`.

```
# Creazione
numeri = [10, 20, 30, 40, 50]
mista = ["Ciao", 3.14, True, [1, 2]] # Può contenere tipi diversi

# Indicizzazione (parte da 0)
print(numeri[0]) # → 10 (Primo elemento)
print(numeri[-1]) # → 50 (Ultimo elemento)
print(numeri[-2]) # → 40 (Penultimo elemento)
```

Le liste sono **mutabili**: possiamo cambiare i valori dopo averle create.  
`numeri[0] = 99` (Ora la lista inizia con 99).

# Indicizzazione Visuale

```
Lista:  ["A", "B", "C", "D", "E"]
Index:   0   1   2   3   4
Index neg: -5  -4  -3  -2  -1
```

```
lettere = ["A", "B", "C", "D", "E"]

print(lettere[0])    # → "A" (primo)
print(lettere[2])    # → "C" (terzo)
print(lettere[-1])   # → "E" (ultimo)
print(lettere[-3])   # → "C" (terzultimo)
```

**Errore comune:** Accedere a un indice inesistente causa `IndexError`.

# Slicing (Tagliare le liste)

Permette di estrarre sotto-liste. Sintassi: [start : end : step]  
*L'elemento 'end' è escluso!*

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

print(dati[0:3])    # → [0, 1, 2] (Primi 3 elementi)
print(dati[5:])    # → [5, 6, 7, 8, 9] (Dal 5° alla fine)
print(dati[:4])    # → [0, 1, 2, 3] (Dall'inizio al 4°)
print(dati[::2])   # → [0, 2, 4, 6, 8] (Ogni 2 elementi)
print(dati[::-1])  # → [9, 8, ... 0] (Lista invertita!)
```

# Slicing Avanzato

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

# Copia superficiale della lista
copia = numeri[:]

# Ultimi 3 elementi
print(numeri[-3:])    # → [7, 8, 9]

# Dal secondo al penultimo
print(numeri[1:-1])   # → [1, 2, 3, 4, 5, 6, 7, 8]

# Ogni 3 elementi partendo da 1
print(numeri[1::3])   # → [1, 4, 7]

# Modifica tramite slicing
numeri[0:2] = [100, 200] # Sostituisce i primi 2
```

# Metodi Principali: Append & Extend

```
carrello = ["Pane", "Latte"]

# .append(x): Aggiunge UN elemento alla fine
carrello.append("Uova")
# Ora: ["Pane", "Latte", "Uova"]

# .extend(lista): Aggiunge TUTTI gli elementi di un'altra lista
altri = ["Burro", "Formaggio"]
carrello.extend(altri)
# Ora: ["Pane", "Latte", "Uova", "Burro", "Formaggio"]

# Differenza importante:
carrello.append(["A", "B"])  # Aggiunge UNA lista come elemento
carrello.extend(["A", "B"])  # Aggiunge "A" e "B" separatamente
```

# Metodi Principali: Insert, Pop & Remove

```
frutti = ["Mela", "Banana", "Arancia"]

# .insert(index, x): Inserisce in posizione specifica
frutti.insert(1, "Pera")
# Ora: ["Mela", "Pera", "Banana", "Arancia"]

# .pop(index): Rimuove e RESTITUISCE l'elemento
ultimo = frutti.pop()      # Rimuove l'ultimo
print(ultimo)               # → "Arancia"
primo = frutti.pop(0)       # Rimuove il primo
print(primo)                # → "Mela"

# .remove(x): Rimuove la PRIMA occorrenza del valore
frutti.remove("Pera")       # Rimuove "Pera"
```

# Metodi di Ricerca e Conteggio

```
voti = [28, 30, 25, 30, 28, 30, 27]

# .index(x): Trova la posizione della prima occorrenza
pos = voti.index(30)
print(pos) # → 1

# .count(x): Conta le occorrenze
quanti_30 = voti.count(30)
print(quanti_30) # → 3

# Operatore 'in': Verifica presenza
if 30 in voti:
    print("C'è almeno un 30!")

# len(): Lunghezza della lista
print(len(voti)) # → 7
```

# Metodi di Ordinamento

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

# .sort(): Ordina IN PLACE (modifica la lista originale)
numeri.sort()
print(numeri) # → [1, 1, 2, 3, 4, 5, 6, 9]

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

# sorted(): Restituisce una NUOVA lista ordinata
originale = [3, 1, 4]
ordinata = sorted(originale)
print(originale) # → [3, 1, 4] (invariata)
print(ordinata) # → [1, 3, 4]
```

# 2. Le Tuple (Tuples)

**Sicurezza e Immutabilità**

# Definizione e Immutabilità

Le **tuple** sono sequenze simili alle liste, ma con una differenza cruciale: sono **immutabili**.

**Immutabilità significa:**

- Una volta creata, non puoi modificare, aggiungere o rimuovere elementi
- Qualsiasi "modifica" richiede la creazione di una nuova tupla
- Garantisce che i dati rimangano costanti nel tempo

```
coordinate = (45.46, 9.19)
colori_rgb = (255, 0, 0)
singleton = (42,) # Tupla con un solo elemento (nota la virgola!)

# coordinate[0] = 46.0 ← ERRORE! TypeError
```

# Quando Usare le Tuple?

Casi d'uso ideali:

Situazione	Esempio
Dati costanti	Giorni della settimana, configurazioni
Coordinate/punti	(x, y), (lat, lon)
Record di dati	("Mario", 30, "Roma")
Chiavi di dizionario	Quando serve una chiave composta
Return multipli	Restituire più valori da una funzione

Vantaggi tecnici:

- Leggermente più veloci delle liste (ottimizzazione interna)
- Occupano meno memoria
- Hashable (possono essere chiavi di dizionari)

# Perché l'Immutabilità è Utile?

L'immutabilità offre vantaggi importanti:

```
# 1. Protezione da modifiche accidentali
CONFIGURAZIONE = ("localhost", 8080, "produzione")
# Nessuno può modificare accidentalmente questi valori

# 2. Usabili come chiavi di dizionario
posizioni = {
    (0, 0): "origine",
    (1, 0): "est",
    (0, 1): "nord"
}

# 3. Hashable (usabili nei set)
punti_visitati = {(0, 0), (1, 1), (2, 2)}
```

# Unpacking (Spacchettamento)

Assegnare i valori di una tupla a variabili singole in un colpo solo.

```
punto = (10, 20)

# Unpacking
x, y = punto

print(f"X: {x}, Y: {y}")
# Risultato: X: 10, Y: 20

# Scambio valori senza variabile temporanea
a, b = 5, 10
a, b = b, a # Ora a=10, b=5
```

# Unpacking Avanzato

```
# Unpacking con *
numeri = (1, 2, 3, 4, 5)
primo, *resto = numeri
print(primo) # → 1
print(resto) # → [2, 3, 4, 5]

primo, *mezzo, ultimo = numeri
print(mezzo) # → [2, 3, 4]

# Unpacking in funzioni
def calcola(x, y, z):
    return x + y + z

valori = (10, 20, 30)
risultato = calcola(*valori) # Spacchetta la tupla
print(risultato) # → 60
```

# Tuple come Valori di Ritorno

Le funzioni Python possono restituire più valori usando le tuple:

```
def analizza_lista(numeri):
    """Restituisce statistiche sulla lista."""
    return min(numeri), max(numeri), sum(numeri) / len(numeri)

dati = [4, 8, 15, 16, 23, 42]

# Unpacking del risultato
minimo, massimo, media = analizza_lista(dati)

print(f"Min: {minimo}, Max: {massimo}, Media: {media:.2f}")
# → Min: 4, Max: 42, Media: 18.00
```

# Named Tuples (Tuple con Nome)

Per tuple più leggibili, usa `namedtuple` dalla libreria `collections` :

```
from collections import namedtuple

# Definizione
Punto = namedtuple('Punto', ['x', 'y'])
Persona = namedtuple('Persona', ['nome', 'eta', 'citta'])

# Creazione
p = Punto(10, 20)
mario = Persona("Mario", 30, "Roma")

# Accesso per nome (più leggibile!)
print(p.x, p.y)          # → 10 20
print(mario.nome)        # → "Mario"

# Funziona ancora come tupla normale
print(mario[0])          # → "Mario"
```

# **3. I Dizionari (Dictionaries)**

**Dati strutturati Key-Value**

# Cos'è un Dizionario?

Un **dizionario** è una delle strutture dati più potenti di Python. Memorizza coppie **chiave-valore**, permettendo di accedere ai dati tramite un identificatore significativo invece di un indice numerico.

**Analogia:** Come un vero dizionario dove cerchi una parola (chiave) per trovarne la definizione (valore).

## Caratteristiche:

Proprietà	Descrizione
<b>Chiave → Valore</b>	Ogni elemento è una coppia associata
<b>Chiavi uniche</b>	Non possono esistere chiavi duplicate
<b>Accesso O(1)</b>	Ricerca velocissima grazie all'hashing
<b>Ordinato*</b>	Mantiene ordine di inserimento (Python 3.7+)

# Creazione

```
# Sintassi: parentesi graffe {}
vuoto = []
utente = {"nome": "Mario", "eta": 30}
```

**Uso tipico:** Database in memoria, configurazioni, conteggi, cache.

# Struttura Chiave-Valore

Usano le parentesi graffe `{}`. Ogni elemento è una coppia `chiave: valore`.  
È come un vero dizionario (parola -> definizione).

```
utente = {
    "nome": "Mario",
    "eta": 30,
    "admin": True,
    "hobby": ["calcio", "lettura"]  # I valori possono essere liste
}

# Accesso
print(utente["nome"])  # → "Mario"

# Modifica
utente["eta"] = 31

# Aggiunta nuova chiave
utente["email"] = "mario@email.com"
```

# Metodi Sicuri: .get() e .setdefault()

Accedere direttamente con `["chiave"]` dà errore se la chiave non esiste.

```
utente = {"nome": "Mario", "eta": 30}

# ERRORE se la chiave non esiste
# print(utente["telefono"]) # KeyError!

# .get() restituisce None (o un default) se manca la chiave
telefono = utente.get("telefono")
print(telefono) # → None

telefono = utente.get("telefono", "Non disponibile")
print(telefono) # → "Non disponibile"

# .setdefault() restituisce il valore o lo crea se non esiste
utente.setdefault("paese", "Italia")
print(utente["paese"]) # → "Italia"
```

# Iterare sui Dizionari

```
prodotti = {"mela": 1.50, "pane": 2.00, "latte": 1.80}

# Iterare sulle chiavi (default)
for prodotto in prodotti:
    print(prodotto) # mela, pane, latte

# Iterare sui valori
for prezzo in prodotti.values():
    print(prezzo) # 1.50, 2.00, 1.80

# Iterare su chiavi E valori
for prodotto, prezzo in prodotti.items():
    print(f"{prodotto}: €{prezzo}")
```

# Metodi Utili dei Dizionari

```
persona = {"nome": "Luigi", "eta": 25}

# .keys(), .values(), .items()
print(list(persona.keys()))    # → ['nome', 'eta']
print(list(persona.values()))  # → ['Luigi', 25]

# .update() - Unisce due dizionari
extra = {"citta": "Milano", "eta": 26}  # eta verrà sovrascritto
persona.update(extra)
# → {'nome': 'Luigi', 'eta': 26, 'citta': 'Milano'}

# .pop() - Rimuove e restituisce
eta = persona.pop("eta")
print(eta)  # → 26

# 'in' verifica la presenza di una CHIAVE
print("nome" in persona)  # → True
```

# Dictionary Comprehension

Come le list comprehension, ma per dizionari:

```
# Creare un dizionario da liste
nomi = ["Mario", "Luigi", "Peach"]
lunghezze = {nome: len(nome) for nome in nomi}
# → {'Mario': 5, 'Luigi': 5, 'Peach': 5}

# Filtrare un dizionario
prezzi = {"mela": 1.50, "caviale": 100, "pane": 2.00}
economici = {k: v for k, v in prezzi.items() if v < 10}
# → {'mela': 1.50, 'pane': 2.00}

# Invertire chiavi e valori
originale = {"a": 1, "b": 2, "c": 3}
invertito = {v: k for k, v in originale.items()}
# → {1: 'a', 2: 'b', 3: 'c'}
```

# Dizionari Annidati

```
azienda = {  
    "nome": "TechCorp",  
    "dipendenti": {  
        "D001": {"nome": "Mario", "ruolo": "Developer"},  
        "D002": {"nome": "Luigi", "ruolo": "Designer"}  
    "sedi": ["Milano", "Roma"]  
# Accesso annidato  
print(azienda["dipendenti"]["D001"]["nome"]) # → "Mario"  
  
# Modifica annidata  
azienda["dipendenti"]["D001"]["ruolo"] = "Senior Developer"  
  
# Aggiunta  
azienda["dipendenti"]["D003"] = {"nome": "Peach", "ruolo": "Manager"}
```

# **4. I Set (Insiemi)**

**Unicità matematica**

# Cos'è un Set?

Un **set** (insieme) è una collezione che implementa il concetto matematico di **insieme**: una raccolta di elementi **distinti** senza ordine particolare.

## Dal punto di vista matematico:

- `{1, 2, 3}` e `{3, 1, 2}` sono lo stesso insieme
- Un elemento può appartenere o non appartenere, non può "appartenere due volte"

## Caratteristiche in Python:

Proprietà	Significato
<b>Non ordinato</b>	Non esiste un "primo" o "ultimo" elemento
<b>Elementi unici</b>	I duplicati vengono automaticamente rimossi
<b>Mutabile</b>	Si possono aggiungere/rimuovere elementi
<b>Elementi hashable</b>	Solo tipi immutabili (no liste, sì tuple)

# Creazione di Set

```
# Set vuoto - ATTENZIONE alla sintassi!
vuoto = set()      # Corretto
# vuoto = {}        # SBAGLIATO! Questo crea un dizionario vuoto

# Set con elementi
numeri = {1, 2, 3, 4, 5}

# Da lista (rimuove duplicati automaticamente)
da_lista = set([1, 1, 2, 2, 3])  # → {1, 2, 3}

# Da stringa (ogni carattere diventa elemento)
da_stringa = set("hello")  # → {'h', 'e', 'l', 'o'}
```

**Uso principale:** Rimuovere duplicati, verifiche di appartenenza veloci, operazioni insiemistiche.

# Gestione Unicità e Operazioni

I Set sono collezioni non ordinate di elementi **unici**.

```
# Rimuovere duplicati da una lista
numeri = [1, 1, 2, 3, 3, 4]
unici = set(numeri)
print(unici) # → {1, 2, 3, 4}

# Operazioni Insiemistiche
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b) # Unione: {1, 2, 3, 4, 5}
print(a & b) # Intersezione: {3}
print(a - b) # Differenza: {1, 2}
print(a ^ b) # Differenza simmetrica: {1, 2, 4, 5}
```

# Operazioni Insiemistiche Visualizzate

Set A = {1, 2, 3}

Set B = {3, 4, 5}

A   B (Unione)	→ {1, 2, 3, 4, 5}	(tutti)
A & B (Intersezione)	→ {3}	(comuni)
A - B (Differenza)	→ {1, 2}	(solo in A)
B - A (Differenza)	→ {4, 5}	(solo in B)
A ^ B (Simmetrica)	→ {1, 2, 4, 5}	(non comuni)

```
# Metodi equivalenti
a.union(b)
a.intersection(b)
a.difference(b)
a.symmetric_difference(b)
```

# Metodi dei Set

```
colori = {"rosso", "verde", "blu"}  
  
# .add() - Aggiunge un elemento  
colori.add("giallo")  
  
# .remove() - Rimuove (errore se non esiste)  
colori.remove("rosso")  
  
# .discard() - Rimuove (nessun errore se non esiste)  
colori.discard("arancione") # Nessun errore  
  
# .pop() - Rimuove e restituisce un elemento casuale  
elemento = colori.pop()  
  
# .clear() - Svuota il set  
colori.clear()
```

# Verifiche sui Set

```
a = {1, 2, 3}
b = {1, 2, 3, 4, 5}
c = {1, 2}

# Appartenenza
print(2 in a)      # → True
print(10 in a)     # → False

# Sottoinsiemi e Superinsiemi
print(c.issubset(a))    # → True ( $c \subseteq a$ )
print(b.issuperset(a))   # → True ( $b \supseteq a$ )

# Set disgiunti (nessun elemento comune)
d = {10, 20}
print(a.isdisjoint(d))   # → True
```

# Frozen Set (Set Immutabili)

Come le tuple sono alle liste, i `frozenset` sono ai set:

```
# Creazione
fs = frozenset([1, 2, 3])

# Immutabile - non si può modificare
# fs.add(4) # AttributeError!

# Può essere usato come chiave di dizionario
cache = {
    frozenset([1, 2]): "risultato_1",
    frozenset([3, 4]): "risultato_2"
}

# Può essere elemento di un altro set
set_di_set = {frozenset([1, 2]), frozenset([3, 4])}
```

# 5. Confronto Strutture Dati

**Quando usare cosa?**

# Tabella Comparativa

Scegliere la struttura dati giusta è **fondamentale** per scrivere codice efficiente e leggibile. Ogni struttura ha punti di forza specifici.

Caratteristica	Lista	Tupla	Dict	Set
<b>Sintassi</b>	[ ]	( )	{ }	{ } *
<b>Ordinata</b>	✓	✓	✓**	✗
<b>Mutabile</b>	✓	✗	✓	✓
<b>Duplicati</b>	✓	✓	Chiavi: ✗	✗
<b>Indicizzabile</b>	✓	✓	Per chiave	✗
<b>Hashable*</b>	✗	✓	✗	✗

\*Set vuoto: `set()` | \*\*Da Python 3.7+ | \*\*\*Può essere chiave di dict

**Regola d'oro:** Se non sai quale usare, parti con una lista. Poi ottimizza se necessario.

# Guida alla Scelta

Poniti queste **domande** per scegliere la struttura giusta:

1. **I dati devono poter cambiare?** No → Tupla | Sì → continua
2. **Devo associare chiavi a valori?** Sì → Dizionario
3. **Devo eliminare duplicati?** Sì → Set
4. **Ho bisogno di ordine e indici?** Sì → Lista

# Guida alla Scelta: Esempi

```
# LISTA: collezione ordinata modificabile
# Domanda: "Quali elementi ho? In che ordine?"
carrello = ["mela", "pane", "mela", "latte"]

# TUPLA: dati che non devono cambiare
# Domanda: "Questi dati sono fissi/costanti?"
coordinate = (45.46, 9.19)

# DIZIONARIO: associazione chiave-valore
# Domanda: "Devo cercare qualcosa per nome/ID?"
utente = {"id": 1, "nome": "Mario"}

# SET: unicità e appartenenza
# Domanda: "Mi interessano solo valori unici?"
visitati = {1, 3, 5, 7}
```

# Performance a Confronto

La scelta della struttura dati influisce **drasticamente** sulle prestazioni, specialmente con grandi quantità di dati.

## Complessità temporale della ricerca:

Operazione	Lista	Set/Dict
Cerca elemento	$O(n)$	$O(1)$
1 milione di elementi	~1.000.000 confronti	~1 confronto

```
# Esempio pratico
lista = list(range(1000000))
insieme = set(range(1000000))

# Lista: O(n) - deve scorrere tutti gli elementi
999999 in lista      # Lento! Scorre fino alla fine
# Set: O(1) - accesso diretto via hash
999999 in insieme    # Velocissimo! Calcolo hash diretto
```

# Debugging con VS Code

Il **debugger** è il tuo miglior amico per trovare errori!

**Come usarlo:**

1. **Imposta un breakpoint:** Clicca a sinistra del numero di riga (pallino rosso)
2. **Avvia debug:** F5 o icona "Run and Debug"
3. **Controlla variabili:** Nel pannello "Variables" vedi i valori
4. **Step through:** F10 (prossima riga), F11 (entra nella funzione)

```
# Prova a debuggare questo codice
x = 10
y = 20      # ← Metti un breakpoint qui
risultato = x + y
print(risultato)
```

# Debug: Pannelli Utili

Pannello	Uso
<b>Variables</b>	Vedi valori correnti delle variabili
<b>Watch</b>	Aggiungi espressioni da monitorare
<b>Call Stack</b>	Traccia delle funzioni chiamate
<b>Debug Console</b>	Esegui codice durante il debug

## Shortcut fondamentali:

- F5 - Avvia/Continua
- F10 - Step Over (prossima riga)
- F11 - Step Into (entra nella funzione)
- Shift+F5 - Stop

# **Laboratorio Pratico**

**Analizzatore di Testo & Magazzino**

# Esercizio 1: Gestore Magazzino

**Obiettivo:** Usare Liste e Dizionari insieme.

Creare uno script che:

1. Abbia un dizionario `magazzino` con `prodotto: quantità`.
2. Una lista `movimenti` che registra le operazioni (es. "venduto", "rifornito").
3. Chieda all'utente un prodotto da vendere.
4. Aggiorni la quantità e aggiunga l'operazione alla lista.

# Soluzione Esercizio 1

```
# Stato iniziale
magazzino = {
    "Laptop": 10,
    "Mouse": 50,
    "Monitor": 5
}
movimenti = []

# Simulazione Vendita
prodotto_richiesto = "Mouse"
# Verifica disponibilità e aggiornamento
qta_disponibile = magazzino.get(prodotto_richiesto, 0)
if qta_disponibile > 0:
    magazzino[prodotto_richiesto] = qta_disponibile - 1
    # Log operazione (usando una Tupla per immutabilità dati)
    movimenti.append( ("Vendita", prodotto_richiesto, 1) )
    print(f"Venduto: {prodotto_richiesto}")
else:
    print(f"Prodotto non disponibile: {prodotto_richiesto}")
print(f"Stato Magazzino: {magazzino}")
print(f"Log Movimenti: {movimenti}")
```

# Esercizio 2: Analizzatore di Testo

**Obiettivo:** Usare Set e Dizionari per analizzare un testo.  
Creare uno script che:

1. Prenda una stringa di testo
2. Conti la frequenza di ogni parola (dizionario)
3. Trovi le parole uniche (set)
4. Trovi le parole in comune tra due testi

# Soluzione Esercizio 2

```
testo1 = "il gatto e il cane giocano nel giardino"
testo2 = "il cane corre nel parco"

# Conta frequenza parole
def conta_parole(testo):
    parole = testo.lower().split()
    frequenza = {}
    for parola in parole:
        frequenza[parola] = frequenza.get(parola, 0) + 1
    return frequenza

# Parole uniche per testo
parole1 = set(testo1.split())
parole2 = set(testo2.split())

# Analisi
print(f"Frequency testo1: {conta_parole(testo1)}")
print(f"Parole uniche testo1: {parole1}")
print(f"Parole in comune: {parole1 & parole2}")
print(f"Solo in testo1: {parole1 - parole2}")
```

# Esercizio 3: Rubrica Telefonica

**Obiettivo:** Dizionario annidato con operazioni CRUD.

```
rubrica = {}

def aggiungi_contatto(nome, telefono, email=None):
    rubrica[nome] = {"telefono": telefono, "email": email}

def cerca_contatto(nome):
    return rubrica.get(nome, "Contatto non trovato")

def elimina_contatto(nome):
    return rubrica.pop(nome, None)

# Utilizzo
aggiungi_contatto("Mario", "123456", "mario@email.com")
aggiungi_contatto("Luigi", "789012")

print(cerca_contatto("Mario"))
# → {'telefono': '123456', 'email': 'mario@email.com'}
```

# Riepilogo del Modulo

Struttura	Uso Principale	Esempio
<b>Lista</b>	Collezione ordinata modificabile	[1, 2, 3]
<b>Tupla</b>	Dati immutabili, chiavi dict	(x, y)
<b>Dizionario</b>	Mapping chiave-valore	{"nome": "Mario"}
<b>Set</b>	Unicità, operazioni insiemistiche	{1, 2, 3}

# ⚠ Errori Comuni con le Strutture Dati

Errore	Problema	Soluzione
<code>{}</code> è un dict, non set	<code>vuoto = {}</code> crea dizionario	Usa <code>set()</code> per set vuoto
Modifica durante iterazione	Errore a runtime	Itera su una copia
Lista come default	Condivisa tra chiamate	Usa <code>None</code> e crea dentro
Chiave mancante	<code>KeyError</code>	Usa <code>.get()</code> con default

```
# ❌ Modificare durante iterazione
for item in lista:
    lista.remove(item)  # BUG!

# ✅ Itera su copia
for item in lista[:]:  # [:] crea copia
    lista.remove(item)  # OK
```



# Progetto Cross-Modulo: Rubrica Contatti

## Parte 2: Collezioni di Contatti

Ora possiamo gestire **molti contatti!**

```
# Lista di dizionari: ogni contatto è un dict
rubrica = [
    {"nome": "Mario", "telefono": "333-111", "email": "mario@email.com"},
    {"nome": "Luigi", "telefono": "333-222", "email": "luigi@email.com"},
    {"nome": "Peach", "telefono": "333-333", "email": "peach@email.com"}
]
```

```
# Accesso ai dati
print(rubrica[0]["nome"]) # → "Mario"

# Aggiungere un contatto
rubrica.append({"nome": "Toad", "telefono": "333-444", "email": "toad@email.com"})

# Quanti contatti?
print(f"Contatti totali: {len(rubrica)}")
```

**Prossimo Modulo:** Controllo di Flusso (if, for, while)