

Il Viaggio del Programmatore

Dai Fondamenti alla Scrittura di Codice Robusto in Python

Benvenuti in un percorso pensato per trasformarvi in programmatori consapevoli.
Inizieremo con gli strumenti essenziali, impareremo a dare istruzioni precise al computer,
organizzeremo dati complessi e, infine, progetteremo soluzioni eleganti e affidabili.
Ogni passo è un nuovo potere acquisito.



Prima di Scrivere Codice, Impariamo a Pensare

Concetto chiave 1: L'Algoritmo

Un algoritmo è una sequenza precisa e finita di passi elementari per risolvere un problema. Non è ambiguo, è eseguibile e termina in un tempo finito.

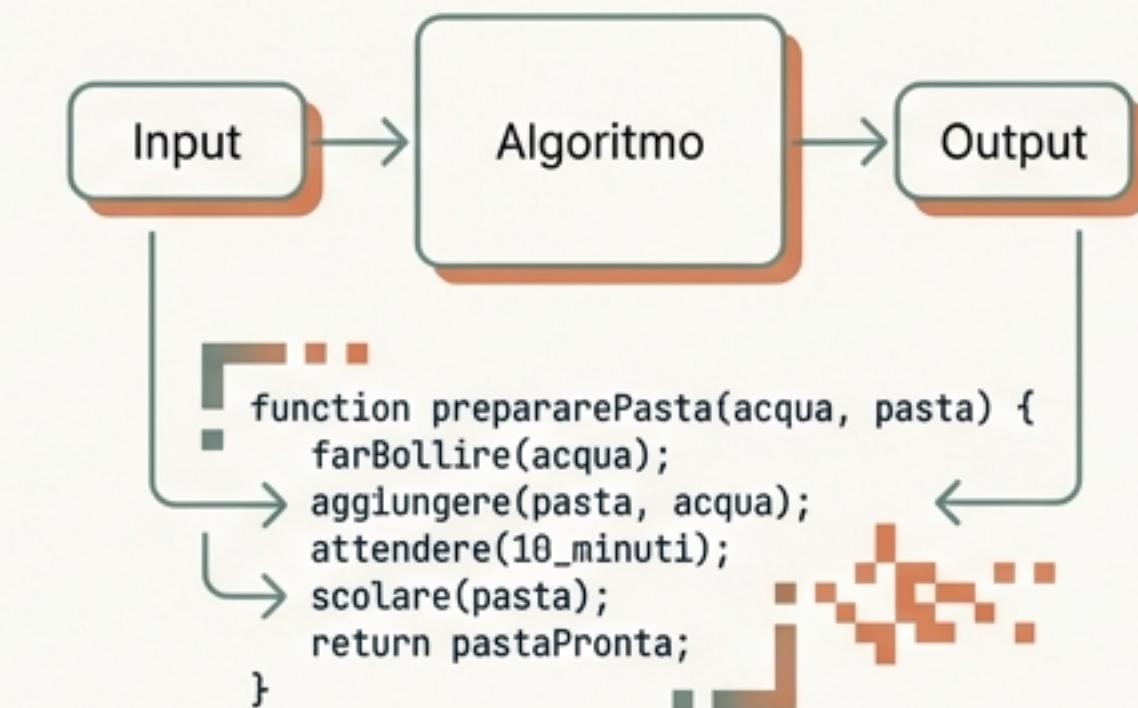
Esempio Quotidiano:

Le istruzioni per montare un mobile o per preparare la pasta.

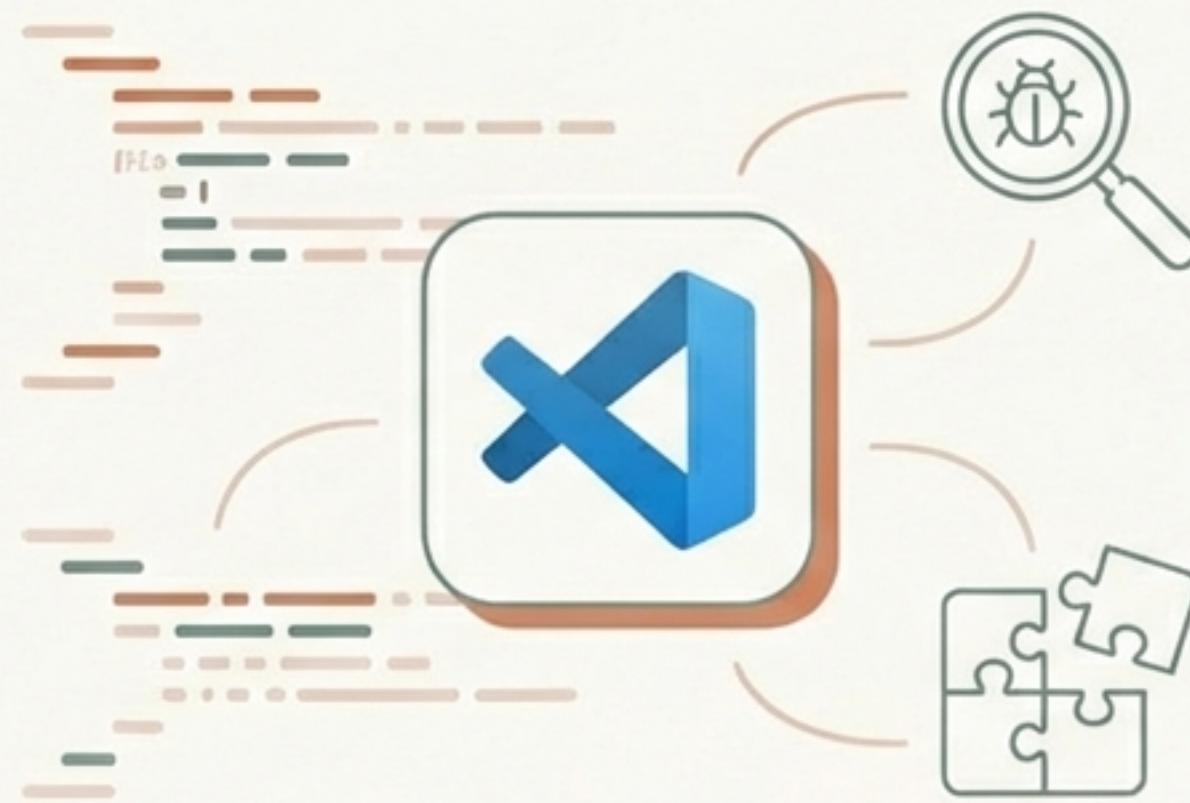


Concetto chiave 2: Il Programma

Un programma è la traduzione di un algoritmo in un linguaggio di programmazione. È il modo in cui comuniciamo la nostra soluzione a un calcolatore.



Allestire il Proprio Laboratorio Digitale



L'IDE (Integrated Development Environment)

Il nostro ambiente di sviluppo. Un software che combina un editor di testo avanzato, un debugger e altri strumenti per scrivere codice in modo efficiente.

La Nostra Scelta: VSCode.

La Nostra Scelta: VSCode. Leggero, potente ed estensibile. Diventerà il vostro centro di comando.

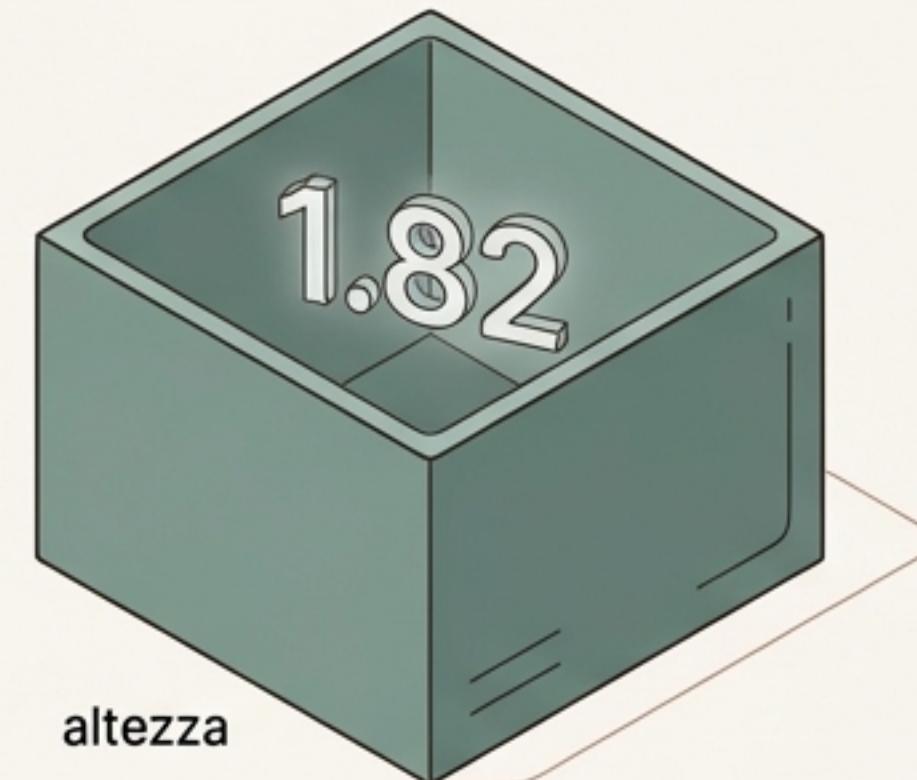
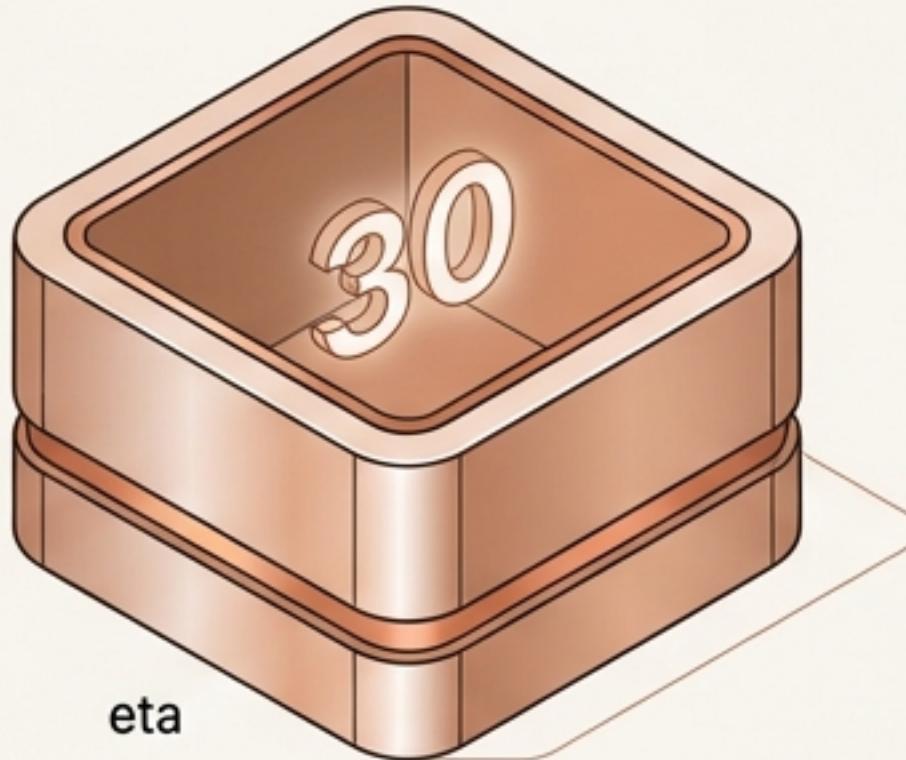


Il Controllo di Versione (Version Control)

Un sistema per tracciare ogni modifica al codice. Non è solo un salvataggio, è una cronologia completa del progetto. **La Scelta: Git.** Lo standard del settore. Git è la vostra rete di sicurezza: permette di sperimentare senza paura, collaborare con altri e tornare indietro a qualsiasi versione precedente del vostro lavoro.

Parte 2: Impartire le Istruzioni

I Mattoni del Codice: Valori, Variabili e Tipi



Variabili

Una variabile è un nome che usiamo per riferirci a un valore, un contenitore di dati. L'assegnazione avviene con l'operatore "=".

```
```python
Assegnazione di valori a variabili
eta = 30
nome = "Mario Rossi"
altezza = 1.82
```
```

Tipi di Dato Fondamentali (Data Types)

Ogni valore ha un tipo, che definisce le operazioni possibili.

- **int**: Numeri interi (es. `5`, `-10`)
- **float**: Numeri con la virgola (es. `3.14`, `2.0`)
- **str**: Sequenze di caratteri, o stringhe (es. "ciao", 'Python')
- **bool**: Valori di verità, `True` o `False`
- **NoneType**: Rappresenta l'assenza di un valore (`None`)

Funzione Utile: `type()` ci permette di scoprire il tipo di dato di una variabile. `print(type(eta))` → <class 'int'>

Dare Vita ai Dati: Espressioni e Operatori

Parte 2: Impartire le Istruzioni

Espressioni vs. Istruzioni:

- **Espressione:** Una combinazione di valori, variabili e operatori che viene valutata per produrre un nuovo valore (es. `5 + 10`).
- **Istruzione:** Un'azione che il programma esegue (es. `a = 5`, `print("Hello")`).

Operatori Aritmetici: Per lavorare con i numeri (`+`, `-`, `*`, `/`, `//` divisione intera, `%` modulo, `**` potenza).

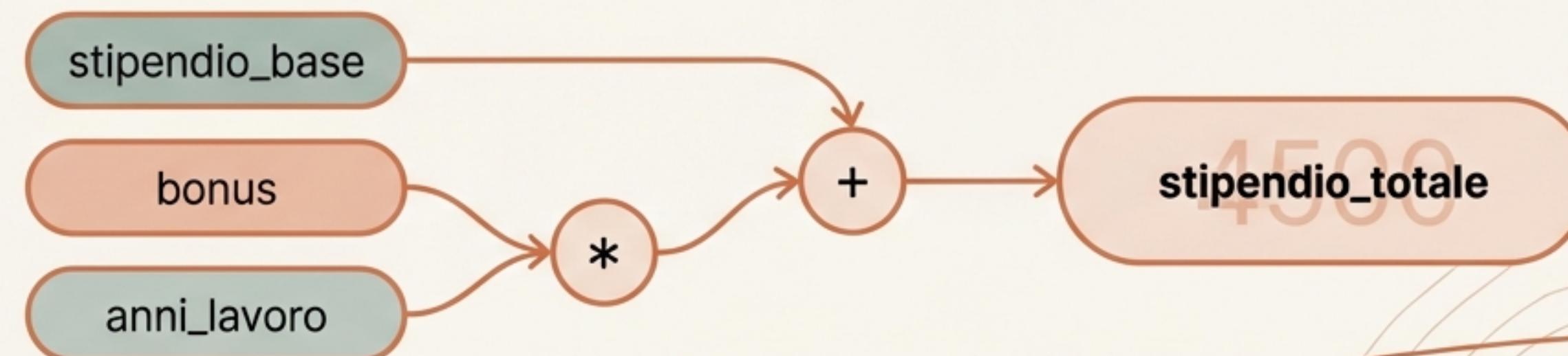
Operatori per Stringhe: `+` (Concatenazione), `*` (Ripetizione).

Operatori di Assegnazione Composti: Un modo più conciso per modificare una variabile (es. `a += 1` è come `a = a + 1`).

Esempio di codice:

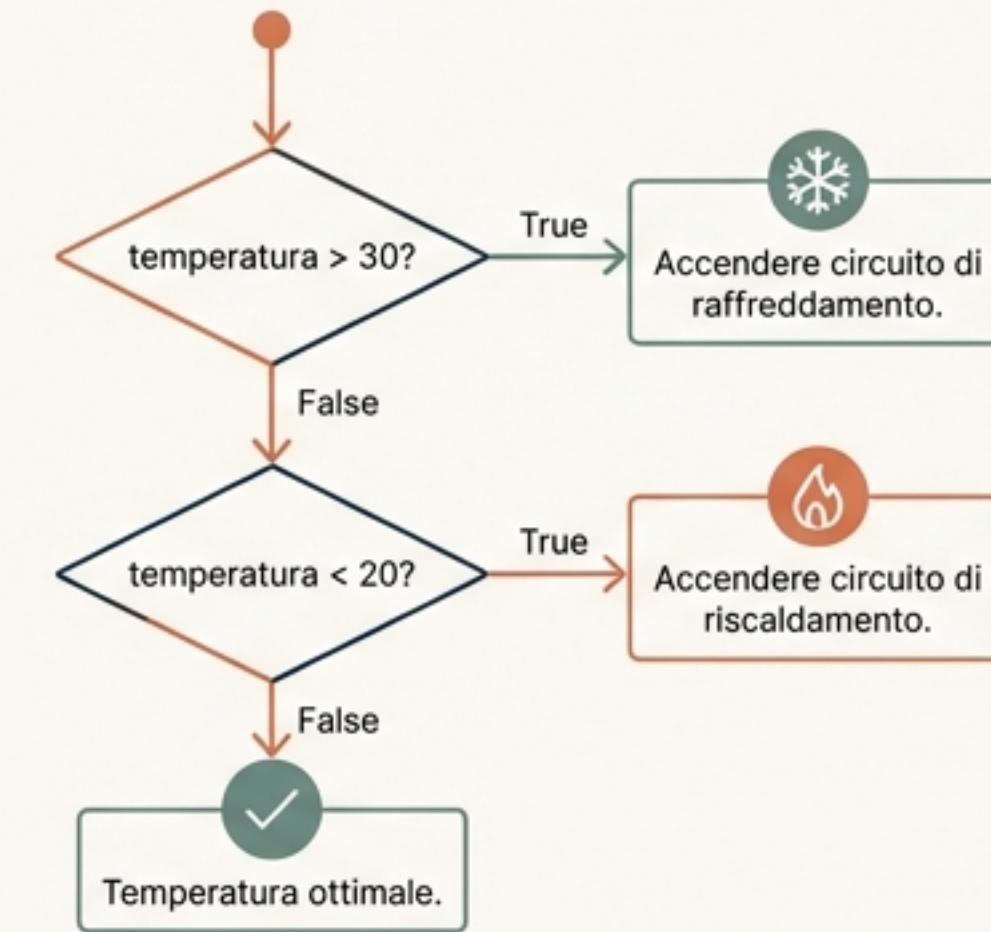
```
anni_lavoro = 5  
stipendio_base = 2000  
bonus = 500
```

```
stipendio_totale = stipendio_base +  
(bonus * anni_lavoro) # Espressione  
print(stipendio_totale) # Istruzione
```



Il Codice che Decide: Logica Booleana ed Esecuzione Condizionale

- **Espressioni Booleane:** Espressioni che valutano a `True` o `False`. Utilizzano operatori di confronto: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- **Operatori Logici:** Per combinare più condizioni:
 - `and`: Vero solo se entrambe le condizioni sono vere.
 - `or`: Vero se almeno una condizione è vera.
 - `not`: Inverte il valore booleano.
- Il Costrutto **`if-elif-else`**: La struttura per controllare il flusso del programma.

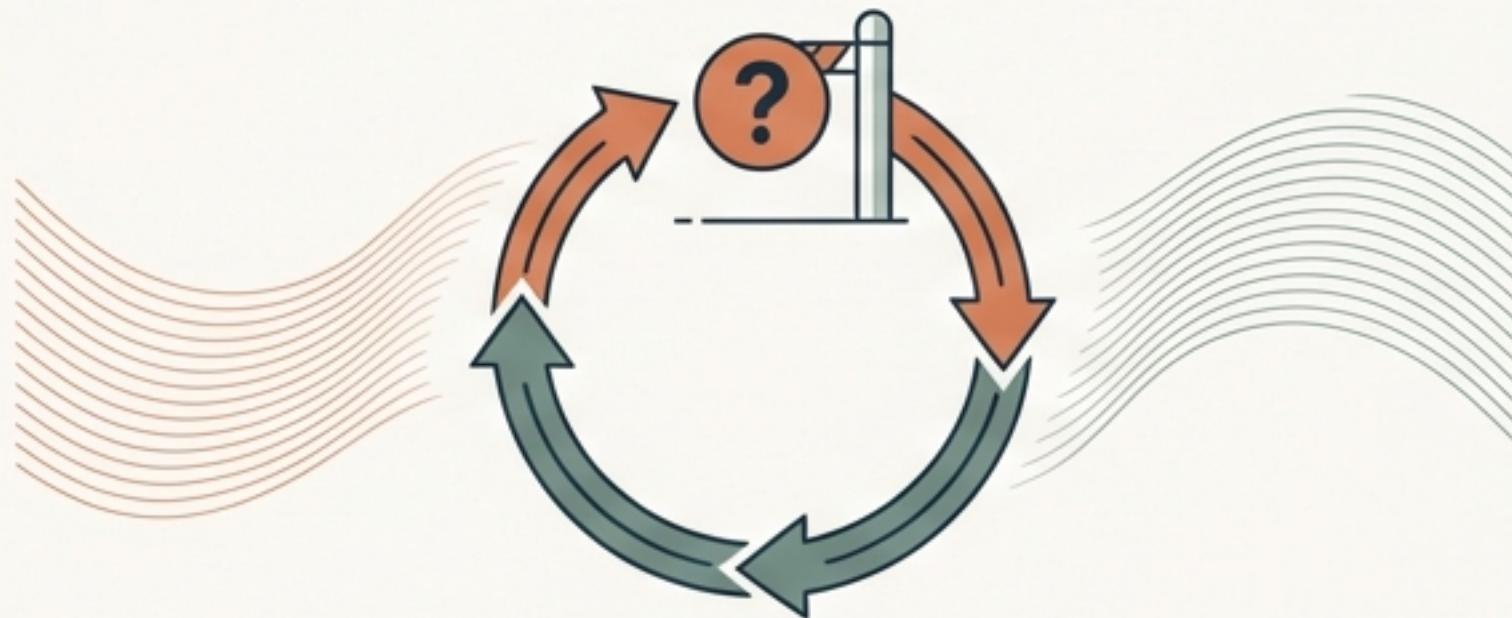


```
temperatura = 25
if temperatura > 30:
    print("Accendere il circuito di raffreddamento.")
elif temperatura < 20:
    print("Accendere il circuito di riscaldamento.")
else:
    print("Temperatura operativa ottimale.")
```

Il Potere dell'Automazione: Cicli `while` e `for`

Iterazione: Ripetere un blocco di codice più volte.

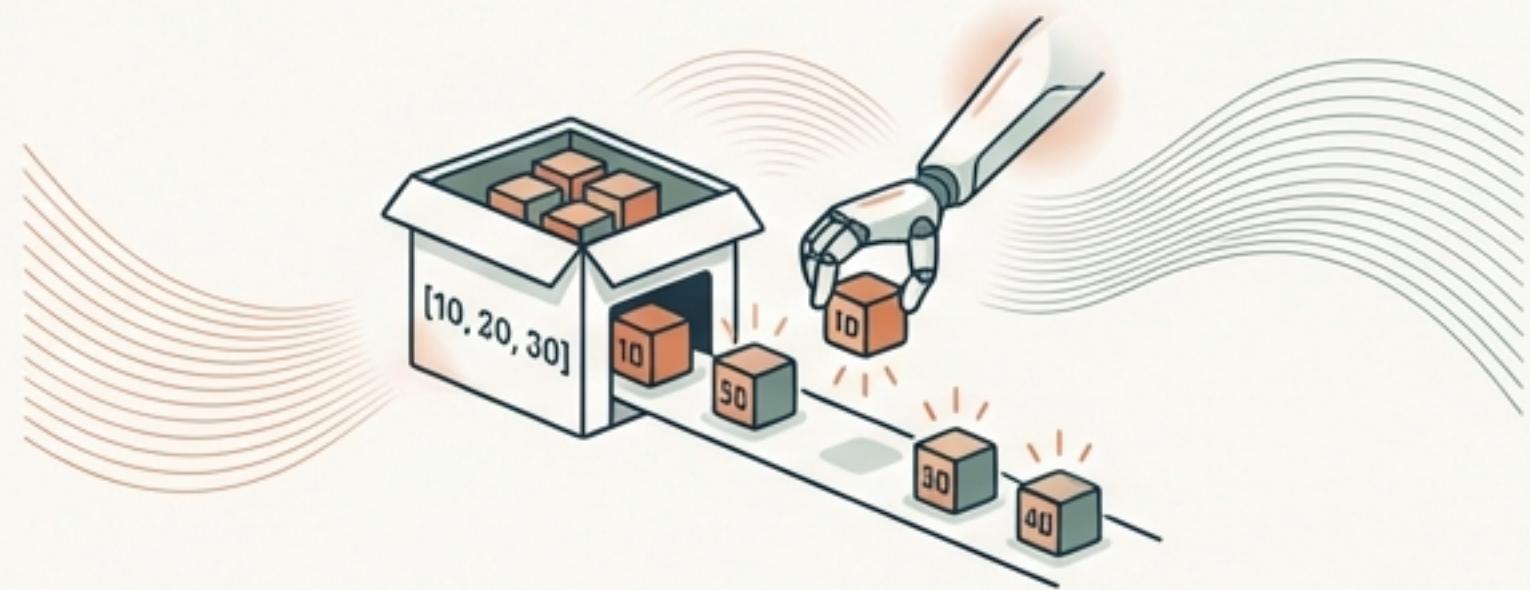
Ciclo `while` ('finché')



Esegue il codice finché una condizione rimane 'True'. È essenziale che il corpo del ciclo modifichi la condizione per evitare cicli infiniti.

```
contatore = 1
while contatore <= 5:
    print("Numero:", contatore)
    contatore += 1
```

Ciclo `for` ('per ogni')

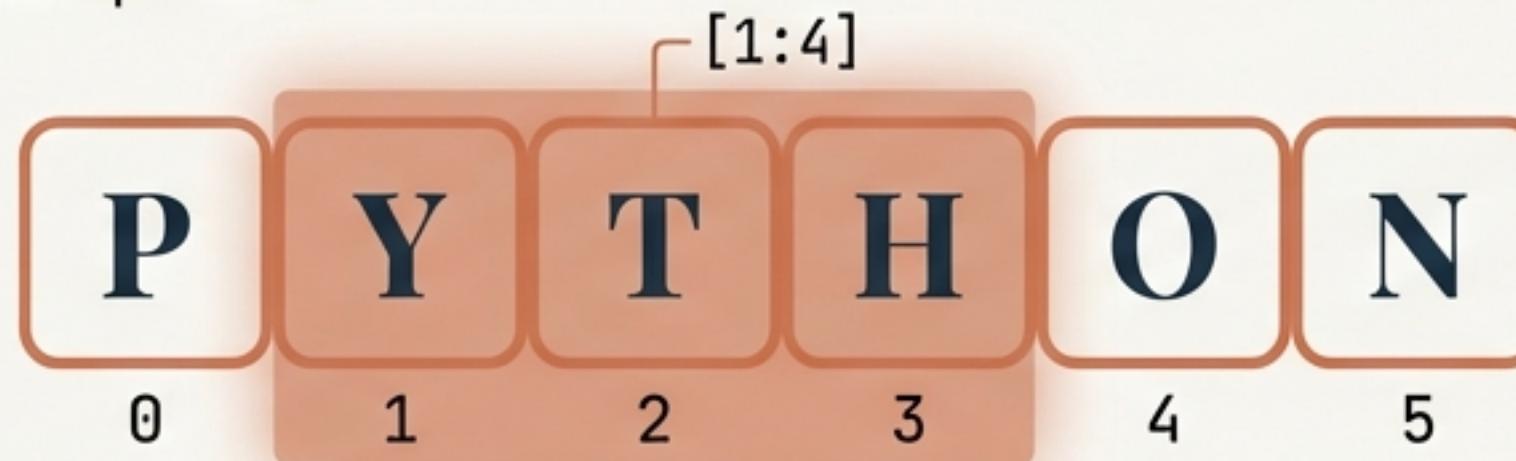


Itera su ogni elemento di una sequenza (come una lista o una stringa). L'attraversamento è automatico.

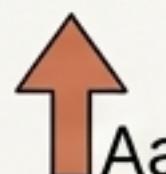
```
prezzi = [10, 20, 30]
for prezzo in prezzi:
    print("Prezzo scontato:", prezzo * 0.8)
```

L'Organizzatore Fondamentale: Le Stringhe

- **Definizione:** Le stringhe sono sequenze immutabili e ordinate di caratteri. "Immutabile" significa che non possono essere modificate dopo la creazione.
- **Creazione:** Con apici singoli ('...') o doppi ("...").
- **Accesso e Slicing:** Si accede ai caratteri con l'indicizzazione (stringa[0]) e a sotto-stringhe con lo slicing (stringa[1:4]). L'indicizzazione parte da 0.



- **Metodi Comuni:**



.upper(), .lower():
Cambiano il case.



.replace(old, new):
Sostituiscono parti della stringa.



.strip(): Rimuovono spazi
bianchi all'inizio e alla fine.



.split(separator): Dividono
la stringa in una lista di
sottostringhe.

```
titolo = " Il Signore degli Anelli "
titolo_pulito = titolo.strip().upper()
# -> "IL SIGNORE DEGLI ANELLI"
parole = titolo_pulito.split(" ")
# -> ["IL", "SIGNORE", "DEGLI", "ANELLI"]
```

Contenitori per Ogni Esigenza: Liste, Tuple, Insiemi e Dizionari

Le 'collections' sono strutture dati per organizzare e manipolare gruppi di elementi. La scelta dipende da cosa devi fare.

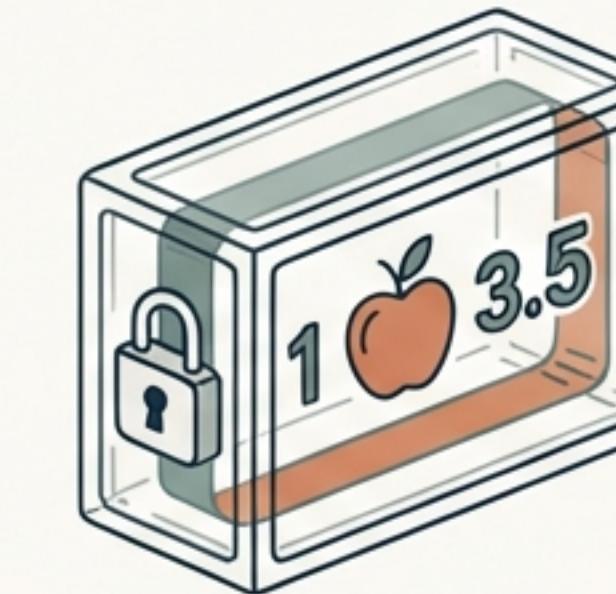


Liste ('list')

Caratteristiche: Ordinate, **modificabili (mutabili)**, consentono duplicati.

Sintassi: [1, "mela", 3.5]

Quando usarle: Quando hai bisogno di una collezione flessibile di elementi che potrebbe cambiare.

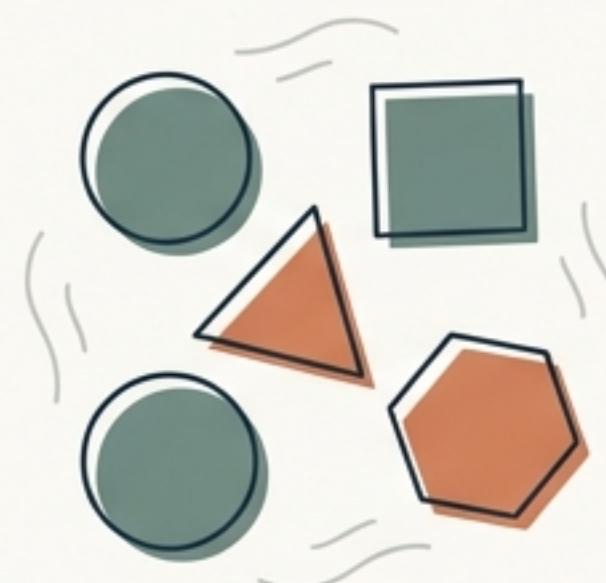


Tuple ('tuple')

Caratteristiche: Ordinate, **non modificabili (immutabili)**, consentono duplicati.

Sintassi: (1, "mela", 3.5)

Quando usarle: Per dati che non devono cambiare, come coordinate o record fissi.

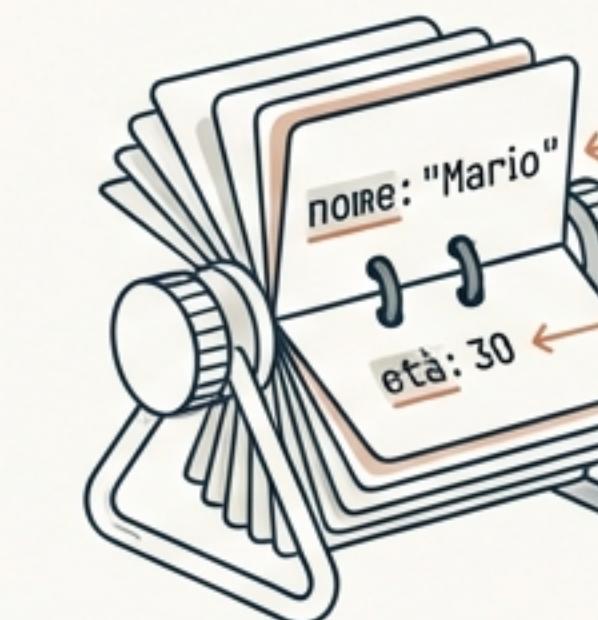


Insiemi ('set')

Caratteristiche: Non ordinate, non indicizzate, **non consentono duplicati**.

Sintassi: {1, "mela", 3.5}

Quando usarli: Per verificare l'appartenenza o rimuovere duplicati in modo efficiente.



Dizionari ('dict')

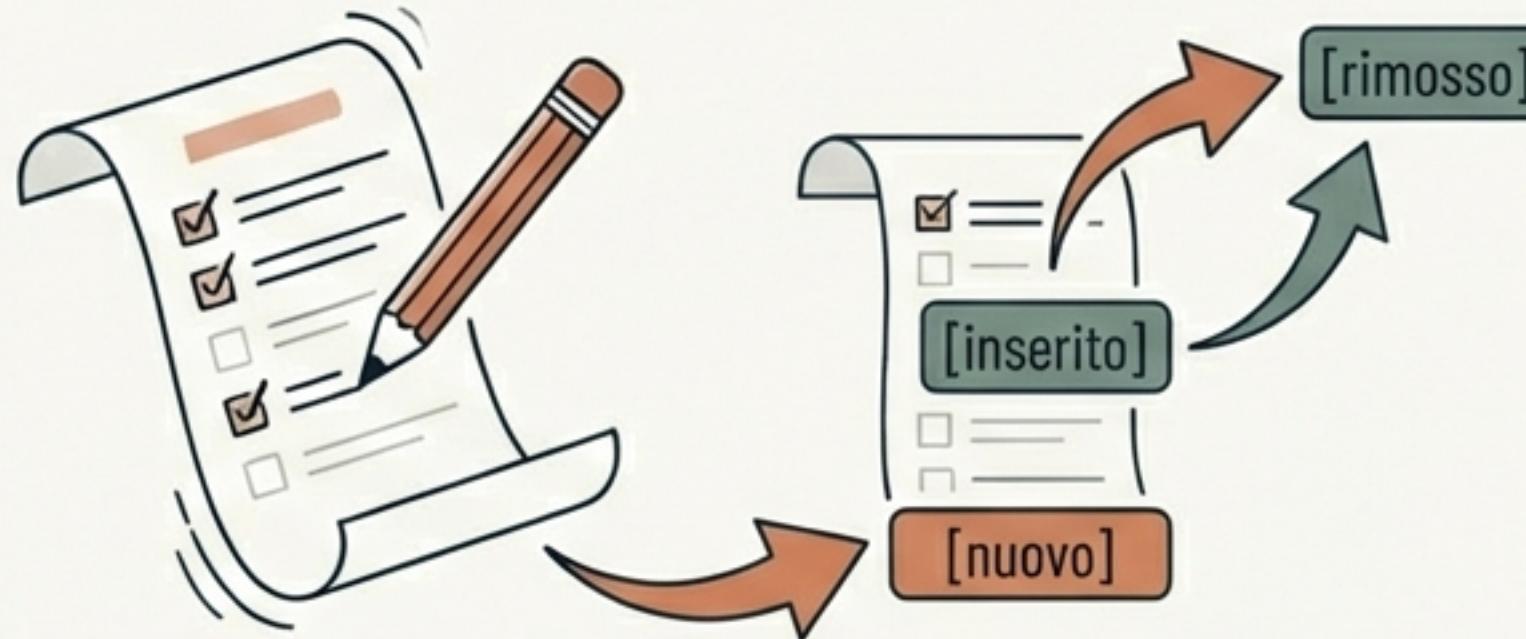
Caratteristiche: Collezione di **coppie chiave-valore**, mutabili. Le chiavi devono essere uniche.

Sintassi: {"nome": "Mario", "età": 30}

Quando usarli: Per associare dati, come un'anagrafica o un catalogo prodotti.

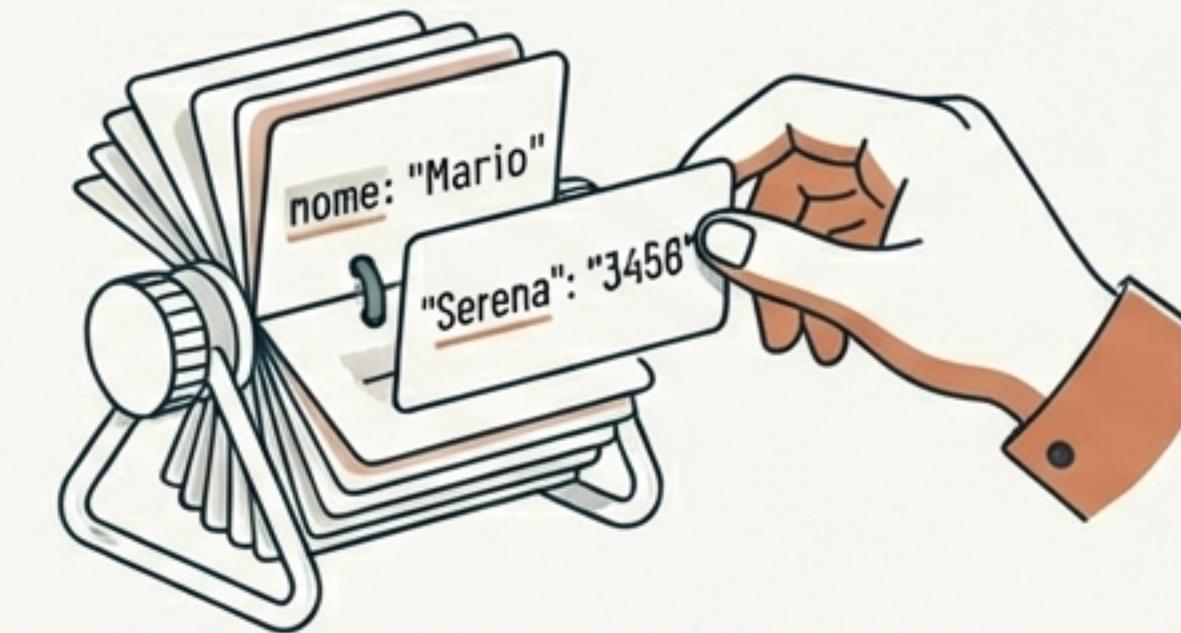
Manipolare le Collezioni: Operazioni Principali

Operazioni sulle Liste:



- **Accesso:** `mia_lista[0], mia_lista[-1]`
- **Aggiungere:** `.append(elemento), .insert(indice, elemento)`
- **Rimuovere:** `.remove(elemento), .pop(indice)`
- **Ordinare:** `.sort(), sorted(mia_lista)`

Operazioni sui Dizionari



- **Accesso:** `mio_dizionario["chiave"], .get("chiave")`
- **Aggiungere/Modificare:** `mio_dizionario["nuova_chiave"] = "nuovo_valore"`
- **Accedere a Chiavi, Valori, Coppie:** `.keys(), .values(), .items()`

Esempio combinato:

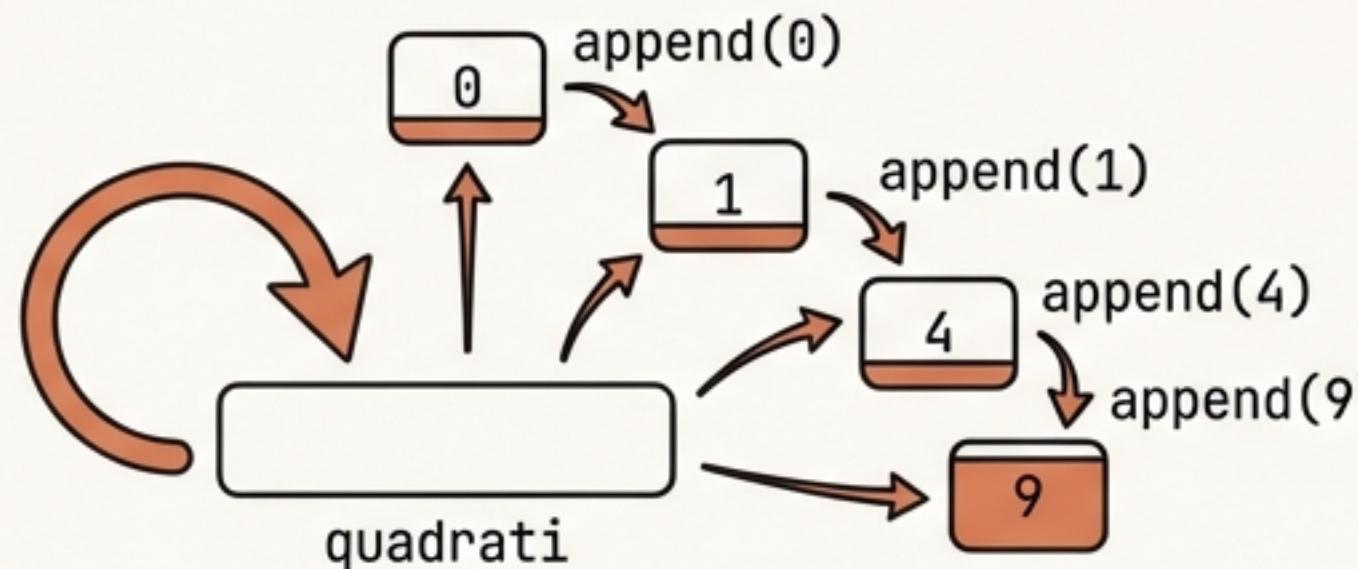
```
rubrica = {"Marco": "1234", "Anna": "5678"}  
rubrica["Serena"] = "3456" # Aggiunge  
print(list(rubrica.keys())) # -> ["Marco", "Anna", "Serena"]
```

Un Modo Più Elegante: List, Set e Dict Comprehension

Il Problema

Spesso creiamo una collezione vuota e la popoliamo con un ciclo `for`.

```
quadrati = []
for x in range(10):
    quadrati.append(x**2)
```



La Soluzione Pythonica

Le 'comprehensions' permettono di fare la stessa cosa in una sola riga, dichiarativa e chiara.

```
# List Comprehension
quadrati = [x**2 for x in range(10)]
```



```
# Set Comprehension
quadrati_unici = {x**2 for x in [1, 2, 2, 3]}
```

```
# Dict Comprehension
quadrati_dict = {x: x**2 for x in range(5)}
```



****Vantaggi**:** Codice più conciso, più leggibile (una volta capita la sintassi) e spesso più veloce.

Parte 4: Costruire come un Architetto

Creare Strumenti Riutilizzabili: Le Funzioni

****Perché le Funzioni?*** Per non ripetere il codice (principio DRY - Don't Repeat Yourself), per dare un nome a un blocco di operazioni e per rendere il programma più modulare e leggibile.

Anatomia di una Funzione:

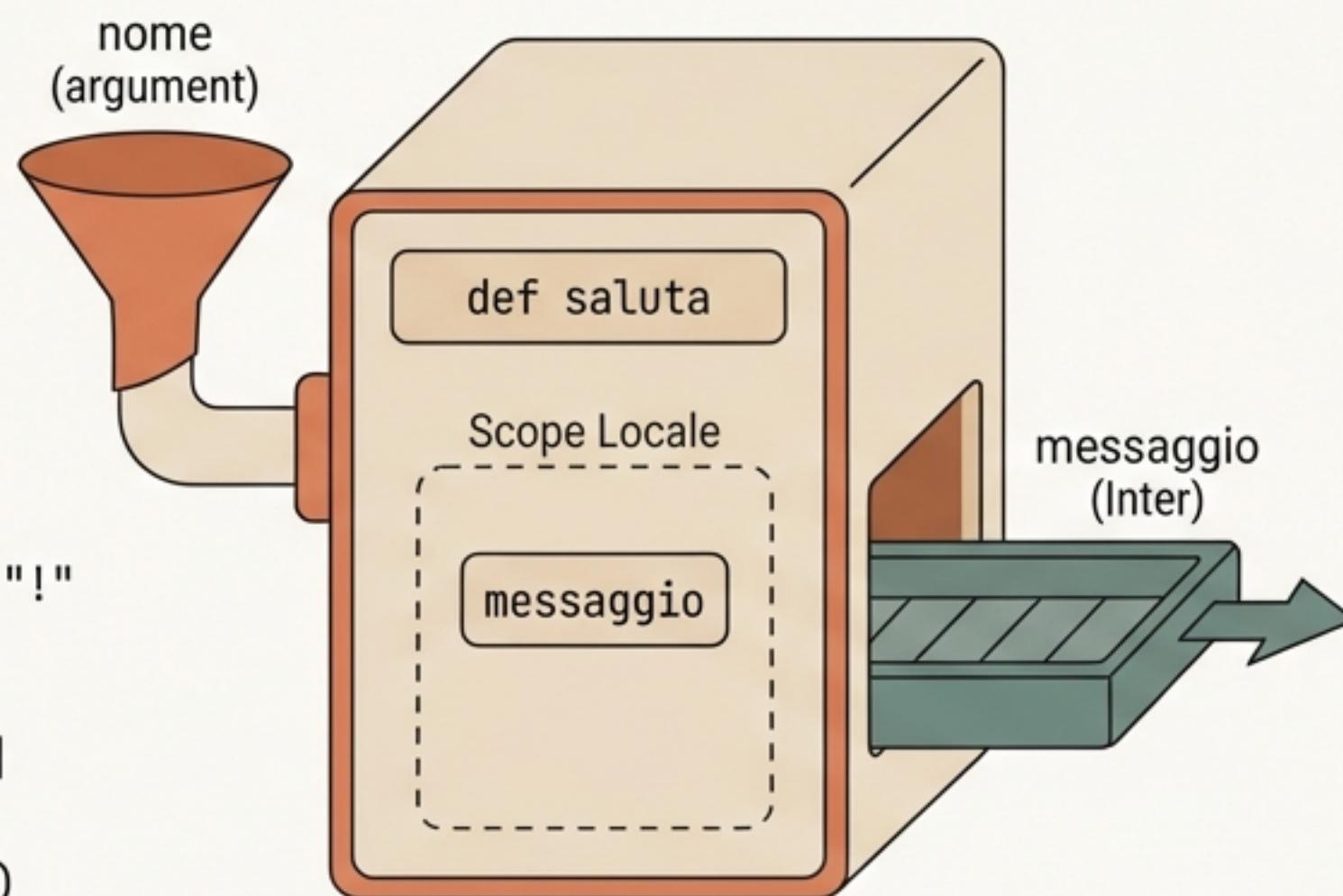
Definizione: Usa la parola chiave `def`...

```
```python
def saluta(nome):
 # Corpo della funzione
 messaggio = "Ciao, " + nome + "!"
 return messaggio
```

```

Invocazione (o Chiamata): Esegue il codice...

```
saluto_per_mario = saluta("Mario")
```



Concetti Chiave:

Argomenti/Parametri: I dati che la funzione riceve in input.

Valore di Ritorno (`return`): Il risultato che la funzione restituisce.

Scope: Le variabili definite all'interno di una funzione esistono solo lì.

Pensare in Modo Diverso: La Ricorsione

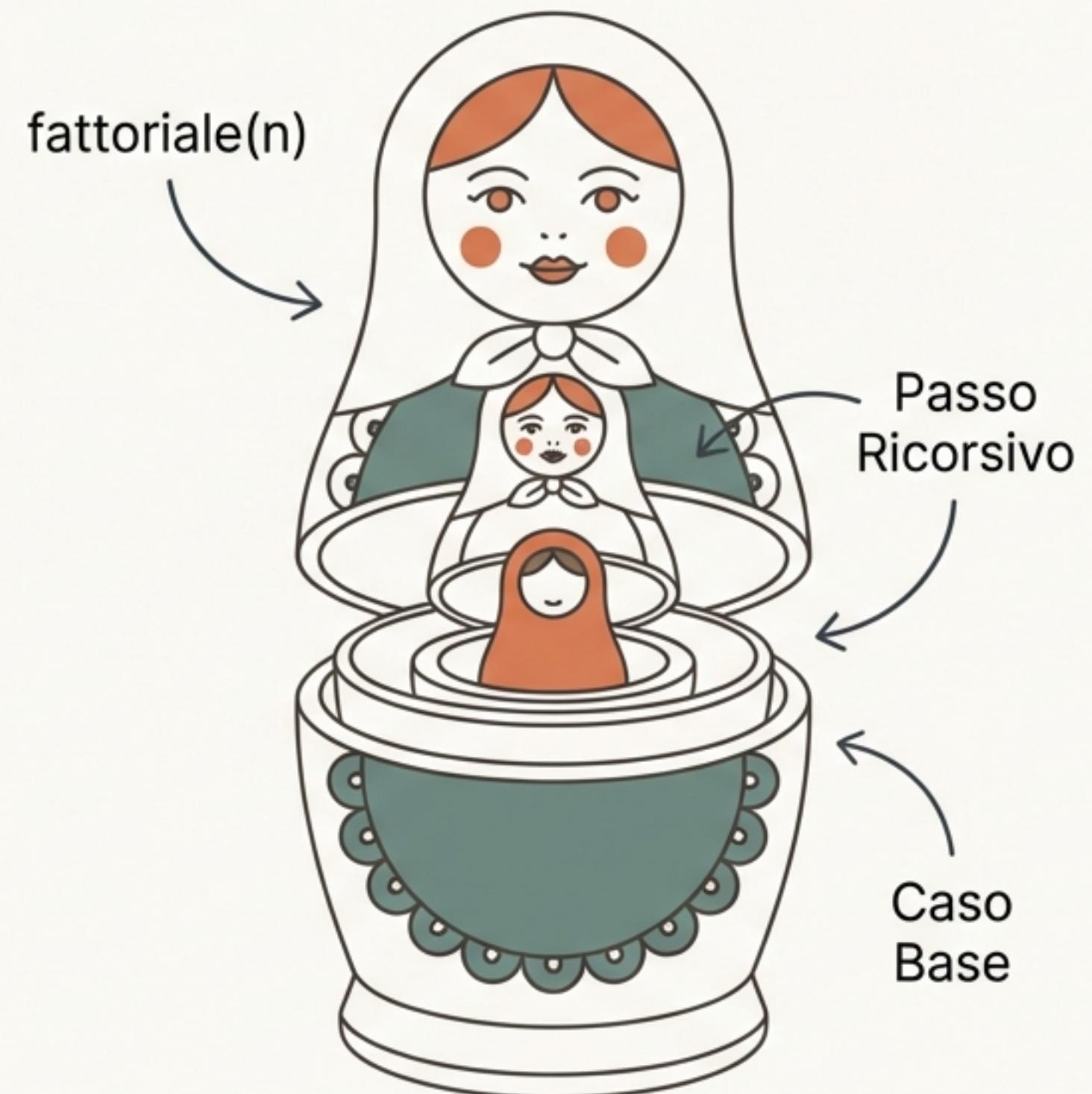
Cos'è la Ricorsione?: Una funzione che chiama sé stessa per risolvere un problema. È un'alternativa all'iterazione per alcuni tipi di problemi.

Due Elementi Fondamentali:

- **Caso Base**: Una condizione che ferma la ricorsione.
- **Passo Ricorsivo**: La parte della funzione che chiama sé stessa, ma con un input che si avvicina al caso base.

Esempio Classico: Il Fattoriale:

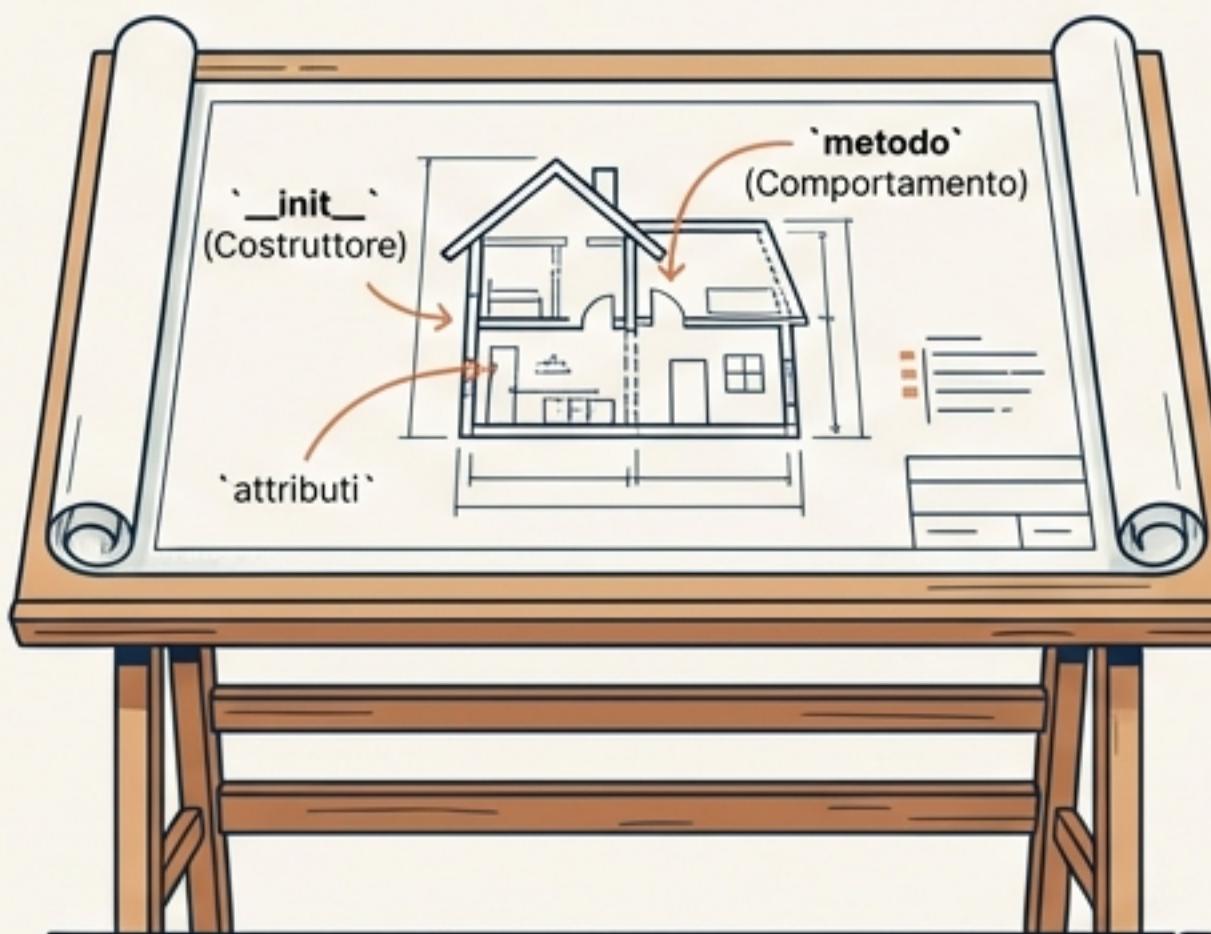
```
def fattoriale(n):  
    if n == 1: # Caso Base  
        return 1  
    else: # Passo Ricorsivo  
        return n * fattoriale(n - 1)
```



I Progetti del Codice: Classi e Oggetti

La programmazione orientata agli oggetti (OOP) organizza il codice raggruppando dati (attributi) e comportamenti (metodi).

Classe (Il Progetto): Un modello per creare oggetti.



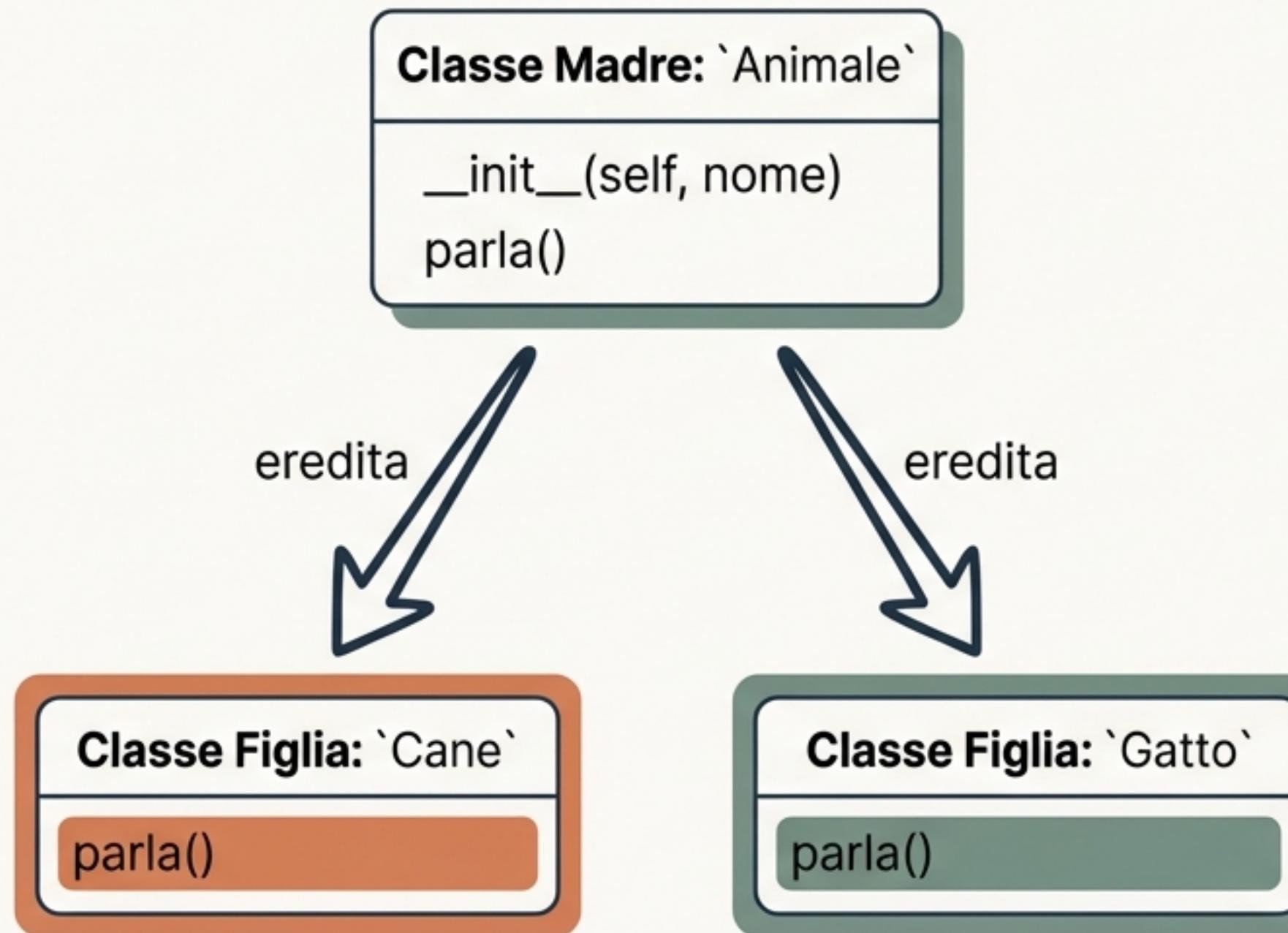
Oggetto (L'Istanza): Una specifica realizzazione di una classe.

```
class Cane:  
    # Metodo costruttore  
    def __init__(self, nome, razza):  
        self.nome = nome      # Attributo  
        self.razza = razza    # Attributo  
  
    # Metodo  
    def abbaia(self):  
        return "Woof!"  
  
# Creazione di due oggetti (istanze)  
mio_cane = Cane("Fido", "Golden Retriever")  
altro_cane = Cane("Rex", "Pastore Tedesco")
```

Concetti Chiave

- **self:** si riferisce all'istanza. `__init__`: il costruttore.
- **Metodo:** una funzione che appartiene a una classe.

Costruire sulle Spalle dei Giganti: Ereditarietà



Il Principio: L'ereditarietà permette a una nuova classe (figlia) di 'ereditare' attributi e metodi da una classe esistente (madre).

Perché Usarla?: Per estendere le funzionalità senza duplicare il codice. Crea una relazione 'è un tipo di'.

Esempio Pratico:

```
# Classe Madre
class Animale:
    def __init__(self, nome):
        self.nome = nome
    def parla(self):
        raise NotImplementedError("...")

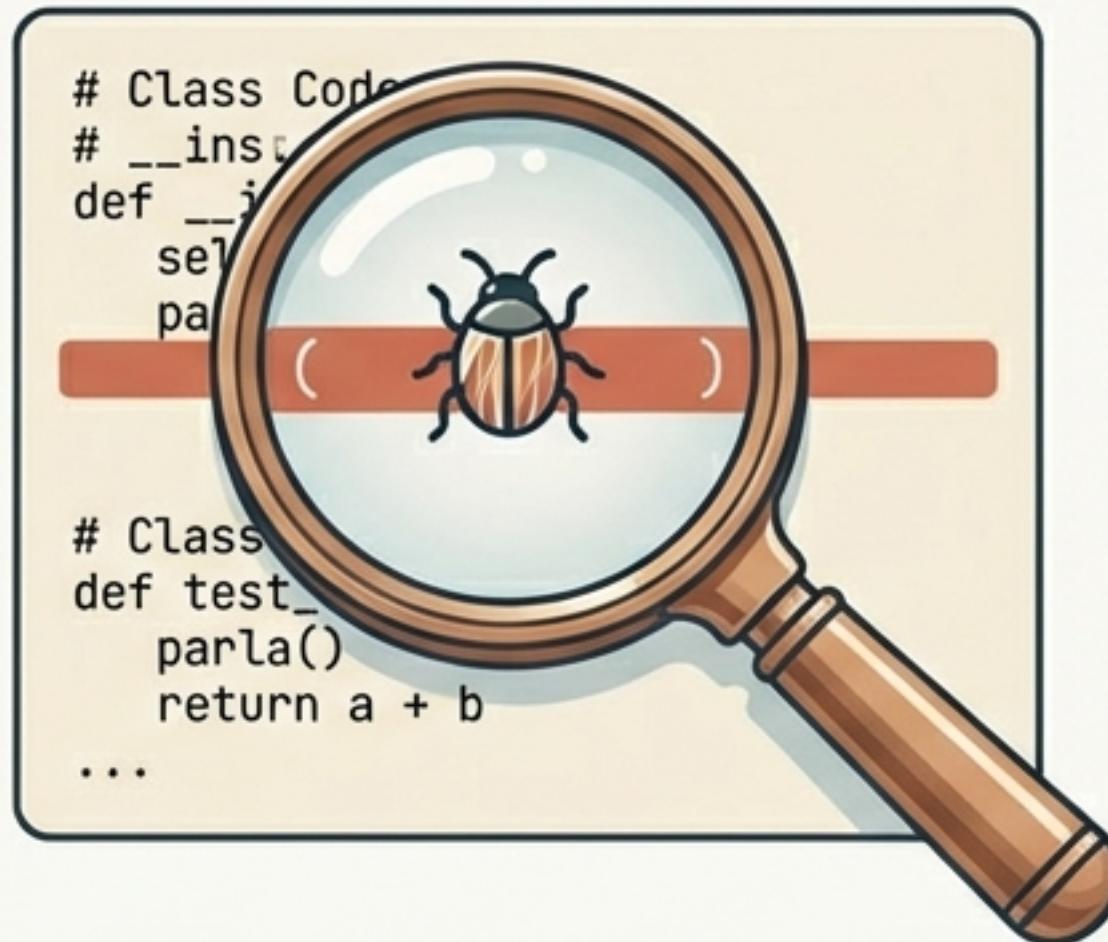
# Classe Figlia che eredita da Animale
class Cane(Animale):
    def parla(self): # Sovrascrive il metodo
        return "Woof!"

# Un'altra Classe Figlia
class Gatto(Animale):
    def parla(self):
        return "Meow!"

fido = Cane("Fido")
print(fido.parla()) # -> "Woof!"
```

Garantire la Qualità: Debugging e Unit Testing

Debugging

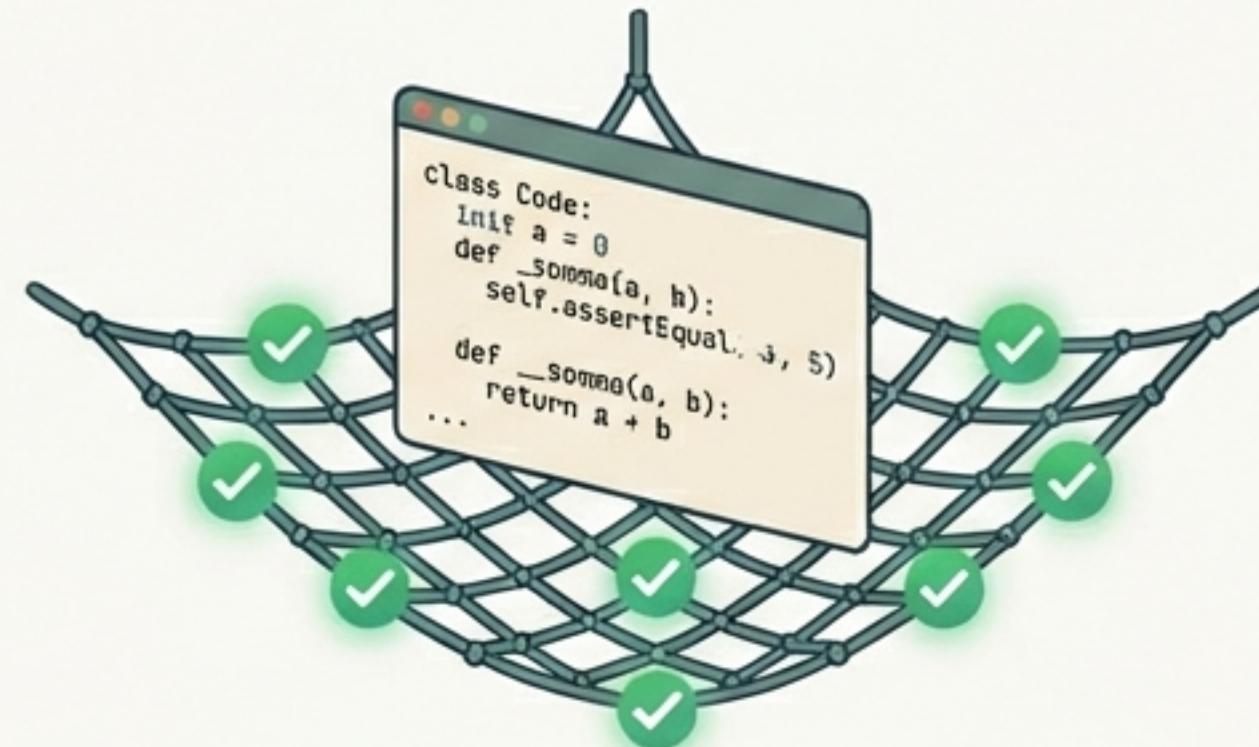


L'arte di trovare e correggere errori (bug) nel codice.

Tecniche Comuni:

- 1. Stampe (`print`):** Semplice ma efficace per tracciare il flusso e i valori.
- 2. Debugger:** Strumenti per eseguire il codice passo-passo, ispezionare le variabili e impostare 'breakpoint'.

Unit Testing



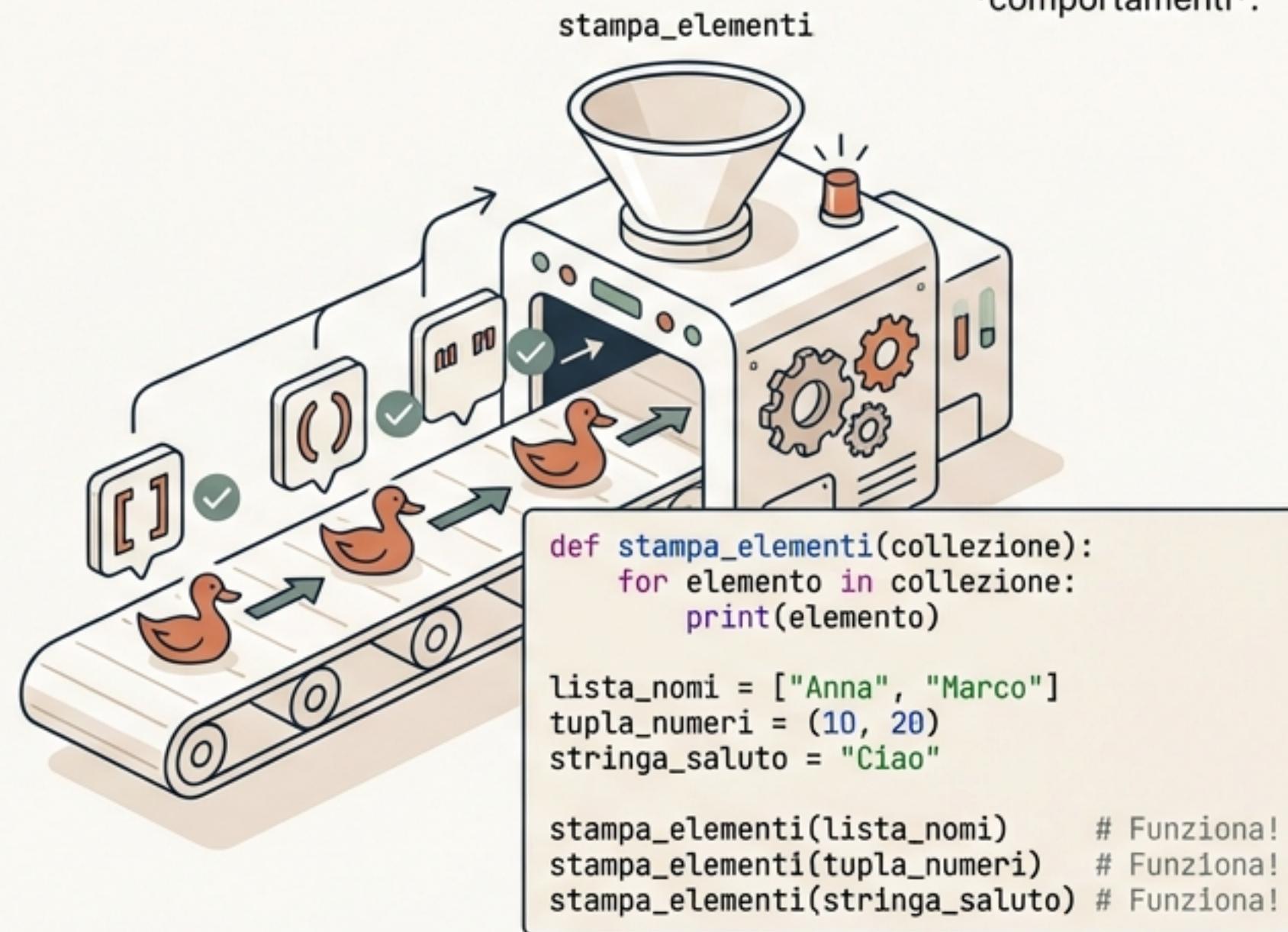
Scrivere codice per testare automaticamente piccole unità (funzioni o metodi) del proprio codice. Garantisce che il codice funzioni, previene regressioni e funge da documentazione vivente.

```
import unittest  
  
def somma(a, b):  
    return a + b  
  
class TestSomma(unittest.TestCase):  
    def test_somma_interi(self):  
        self.assertEqual(somma(2, 3), 5)
```

La Filosofia della Flessibilità: Generic Programming e "Duck Typing"

Programmazione Generica: Scrivere funzioni che operano su una varietà di tipi, purché supportino le operazioni richieste.

Come si Manifesta in Python: Il "Duck Typing": "Se cammina come un'anatra e starnazza come un'anatra, allora deve essere un'anatra." Non ci interessa il ***tipo*** esatto di un oggetto, ma solo i suoi ***comportamenti***.



La Mappa del Vostro Viaggio è Completa

Riepilogo delle Competenze Acquisite

- **Parte 1: La Cassetta degli Attrezzi:** Ora avete un ambiente di sviluppo professionale (IDE) e una rete di sicurezza per il vostro codice (Git).
- **Parte 2: Impartire le Istruzioni:** Sapete manipolare dati con variabili, prendere decisioni con la logica condizionale e automatizzare compiti con i cicli.
- **Parte 3: Organizzare le Informazioni:** Conoscete i contenitori giusti per ogni tipo di dato e sapete crearli in modo elegante.
- **Parte 4: Costruire come un Architetto:** Siete in grado di strutturare il codice in blocchi riutilizzabili (funzioni), creare i vostri tipi di dato (classi) e garantire la qualità (testing).

La programmazione non è una destinazione, ma un viaggio continuo di apprendimento e creazione. Avete ora tutti gli strumenti per iniziare a costruire le vostre soluzioni.

