

Corso Python

Modulo 3: Flusso, Logica e Funzioni

Collegamento al Modulo Precedente

Nel **Modulo 2** abbiamo imparato:

-  Liste: [1, 2, 3]
-  Dizionari: {"nome": "Mario"}
-  Tuple e Set

Ora aggiungiamo:

-  **Logica:** Decidere COSA eseguire
-  **Cicli:** Ripetere operazioni
-  **Funzioni:** Organizzare e riutilizzare codice

Agenda del Modulo (2 Ore)

Diamo vita ai dati con la logica decisionale e la modularità.

1. **Controllo di Flusso:** Prendere decisioni (`if` , `else`).
2. **Operatori:** Confronto e logici.
3. **Cicli (Loops):** Ripetere operazioni (`for` , `while`).
4. **Funzioni:** Creare blocchi di codice riutilizzabili.
5. **Scope:** Dove vivono le variabili.
6. **Laboratorio:** Validatore Password e altri esercizi.

1. Controllo di Flusso

If, Elif, Else e Indentazione

Prendere Decisioni: if

L'istruzione `if` è il **fondamento del controllo di flusso** in ogni linguaggio di programmazione. Permette al programma di "ragionare" e prendere decisioni diverse in base alle circostanze.

Senza condizioni: Il codice esegue sempre le stesse istruzioni, dall'alto al basso.

Con condizioni: Il programma può adattarsi, scegliendo percorsi diversi.

```
eta = 18

if eta ≥ 18:
    print("Sei maggiorenne")
    print("Puoi votare")

print("Fine programma") # Sempre eseguito
```

Il blocco indentato viene eseguito **solo** se la condizione è `True`.

If-Else: Due Alternative

`else` gestisce il caso in cui la condizione è falsa.

```
temperatura = 15

if temperatura > 25:
    print("Fa caldo!")
    print("Prendi la crema solare")
else:
    print("Non fa così caldo")
    print("Prendi una giacca")
```

If-Elif-Else: Multiple Alternative

`elif` (else if) permette di testare condizioni multiple in sequenza.

```
voto = 85

if voto >= 90:
    print("Eccellente!")
elif voto >= 80:
    print("Ottimo!")
elif voto >= 60:
    print("Promosso.")
else:
    print("Bocciato.")
```

Nota: Solo UN blocco viene eseguito. Appena una condizione è vera, le altre vengono ignorate.

L'Importanza dell'Indentazione

In Python, gli spazi non sono estetici: **sono sintassi**.

Perché questa scelta?

- **Leggibilità forzata:** Il codice ben indentato è più facile da leggere
- **Meno errori:** Impossibile dimenticare una parentesi }
- **Uniformità:** Tutto il codice Python ha lo stesso aspetto

Come indentare?

Linguaggio	Delimitatore Blocchi
C, Java, JavaScript	{ }
Python	Indentazione

```
if True:  
print("Errore!")      # IndentationError!  
    print("Corretto")
```

Convenzione PEP 8: Usa sempre **4 spazi** (non tab). VS Code lo fa automaticamente.

If Annidati

Possiamo mettere `if` dentro altri `if` per logiche complesse.

```
eta = 25
ha_patente = True

if eta ≥ 18:
    print("Sei maggiorenne")
    if ha_patente:
        print("Puoi guidare!")
    else:
        print("Prendi la patente prima")
else:
    print("Sei minorenne")
```

Attenzione: Troppi livelli di annidamento rendono il codice difficile da leggere.

Esercizio 1.1: Verifica Triangolo

Obiettivo: Verifica se tre lati possono formare un triangolo.

Regola: La somma di due lati qualsiasi deve essere maggiore del terzo.

```
lato1, lato2, lato3 = 3, 4, 5  
  
# Verifica se formano un triangolo valido  
# Se sì, stampa anche se è equilatero, isoscele o scaleno
```

Suggerimento: Usa `if` annidati o condizioni multiple con `and`.

2. Operatori

Confronto e Logici

Operatori di Confronto

Gli operatori di confronto sono il **cuore delle condizioni**. Confrontano due valori e restituiscono sempre un valore booleano: `True` o `False`. Questi operatori funzionano con **numeri, stringhe, liste** e altri tipi comparabili.

Operatore	Significato	Esempio
<code>=</code>	Uguale a	<code>5 = 5</code> → <code>True</code>
<code>≠</code>	Diverso da	<code>5 ≠ 3</code> → <code>True</code>
<code>></code>	Maggiore di	<code>5 > 3</code> → <code>True</code>
<code><</code>	Minore di	<code>5 < 3</code> → <code>False</code>
<code>≥</code>	Maggiore o uguale	<code>5 ≥ 5</code> → <code>True</code>
<code>≤</code>	Minore o uguale	<code>3 ≤ 5</code> → <code>True</code>

Nota: Le stringhe vengono confrontate in ordine **lessicografico** (alfabetico):

`"apple" < "banana"` → `True`

Attenzione: = vs ==

Errore comune dei principianti!

```
x = 5      # ASSEGNAZIONE: mette 5 in x

x == 5     # CONFRONTO: chiede "x è uguale a 5?"
            # Restituisce True o False

# Errore tipico
if x = 5:  # SyntaxError! Volevi =
            print("OK")
```

Operatori Logici: and, or, not

Combinano più condizioni booleane.

```
eta = 25
ha_biglietto = True

# AND: entrambe devono essere vere
if eta ≥ 18 and ha_biglietto:
    print("Puoi entrare")

# OR: almeno una deve essere vera
if eta < 12 or eta > 65:
    print("Biglietto scontato")

# NOT: inverte il valore
if not ha_biglietto:
    print("Compra un biglietto!")
```

Tabella di Verità

AND (entrambe vere)

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

OR (almeno una vera)

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

NOT (inverte)

A	not A
True	False
False	True

Concatenare Confronti

Python permette di concatenare confronti in modo elegante.

```
eta = 25

# Modo classico
if eta ≥ 18 and eta ≤ 65:
    print("Età lavorativa")

# Modo Pythonico (più leggibile!)
if 18 ≤ eta ≤ 65:
    print("Età lavorativa")

# Funziona anche con più valori
x = 5
if 0 < x < 10:
    print("x è tra 1 e 9")
```

Operatori di Identità e Appartenenza

```
# is / is not: verifica se sono lo STESSO oggetto
a = [1, 2, 3]
b = [1, 2, 3]
c = a

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

# in / not in: verifica appartenenza
lista = [1, 2, 3, 4, 5]
print(3 in lista)    # True
print(10 not in lista)  # True
print("a" in "ciao")   # True
```

Valori "Truthy" e "Falsy"

In Python, **ogni valore può essere valutato come booleano**. Questo è un concetto potente che rende il codice più conciso e leggibile.

- **Falsy:** Valori "vuoti" o "nulli" → considerati `False`
- **Truthy:** Tutto il resto → considerato `True`

Tipo	Valore Falsy	Valore Truthy
Booleano	<code>False</code>	<code>True</code>
Numeri	<code>0</code> , <code>0.0</code>	Qualsiasi altro numero
Stringhe	<code>""</code> (vuota)	Qualsiasi stringa non vuota
Liste/Tuple	<code>[]</code> , <code>()</code>	Collezioni con elementi
Dizionari/Set	<code>{}</code> , <code>set()</code>	Con almeno un elemento
Speciale	<code>None</code>	-

Truthy e Falsy: Esempi Pratici

```
if []:
    print("Mai eseguito") # Lista vuota è Falsy

if [1, 2, 3]:
    print("Eseguito!")     # Lista non vuota è Truthy

# Uso pratico: verificare se una variabile ha contenuto
nome = ""
if nome:
    print(f"Ciao {nome}")
else:
    print("Nome non inserito")

# Equivalente più esplicito (ma meno pythonico)
if len(nome) > 0:
    print(f"Ciao {nome}")
```

Operatore Ternario (Inline If)

Scrivere un if-else in una sola riga.

```
# Sintassi: valore_se_vero if condizione else valore_se_falso

eta = 20
status = "maggiorenne" if eta ≥ 18 else "minorenne"
print(status) # → "maggiorenne"

# Equivalente a:
if eta ≥ 18:
    status = "maggiorenne"
else:
    status = "minorenne"

# Utile per assegnazioni rapide
prezzo = 100
sconto = 0.1 if prezzo > 50 else 0
```

Match-Case (Python 3.10+)

Alternativa elegante a lunghe catene di if-elif.

```
comando = "start"

match comando:
    case "start":
        print("Avvio programma")
    case "stop":
        print("Arresto programma")
    case "pause":
        print("Pausa")
    case _: # Default (come else)
        print("Comando non riconosciuto")
```

Match-Case Avanzato

```
# Pattern matching con valori
punto = (0, 5)

match punto:
    case (0, 0):
        print("Origine")
    case (0, y):
        print(f"Sull'asse Y a {y}")
    case (x, 0):
        print(f"Sull'asse X a {x}")
    case (x, y):
        print(f"Punto generico ({x}, {y})")

# Con guardie (condizioni extra)
match punto:
    case (x, y) if x == y:
        print("Sulla diagonale")
```



Esercizi Intermedi: Operatori

Pratica Prima di Proseguire

Esercizio 2.1: Controllo Range

Obiettivo: Verifica se un numero è in un intervallo valido.

Regole:

- Il voto deve essere tra 0 e 100 (inclusi)
- Se il voto è 60 o superiore, lo studente è "Promosso"

```
voto = 75
```

```
# Usa il concatenamento dei confronti (modo pythonico)
# Es: if 0 <= voto <= 100:
```

Esercizio 2.2: Valori Truthy/Falsy

Obiettivo: Senza eseguire il codice, prevedi l'output.

```
lista = []
nome = "Mario"
numero = 0

if lista:
    print("A")
if nome:
    print("B")
if numero:
    print("C")
if nome and not lista:
    print("D")
```

Domanda: Quali lettere vengono stampate?

3. I Cicli (Loops)

Automatizzare la ripetizione

While Loop

Il ciclo `while` rappresenta l'**iterazione condizionale**: esegue un blocco di codice finché una condizione rimane vera.

Struttura concettuale:

1. Verifica la condizione
2. Se `True` : esegui il blocco, poi torna al punto 1
3. Se `False` : esci dal ciclo

⚠ Attenzione ai loop infiniti! Se la condizione non diventa mai `False`, il programma si blocca.

```
contatore = 0

while contatore < 3:
    print(f"Giro numero {contatore}")
    contatore += 1 # Fondamentale: aggiornare la condizione!
```

While: Input dell'Utente

Caso d'uso classico: chiedere input finché non è valido.

```
while True:  
    risposta = input("Scrivi 'esci' per uscire: ")  
    if risposta.lower() == "esci":  
        print("Arrivederci!")  
        break # Esce dal ciclo  
    print(f"Hai scritto: {risposta}")
```

```
# Alternativa con condizione  
risposta = ""  
while risposta != "esci":  
    risposta = input("Comando: ")  
    print(f"Eseguo: {risposta}")
```

For Loop: Iterare su Sequenze

Itera su ogni elemento di una sequenza (lista, stringa, range...).

```
# Iterare su una lista
frutti = ["mela", "banana", "arancia"]
for frutto in frutti:
    print(f"Mi piace la {frutto}")

# Iterare su una stringa
for lettera in "Python":
    print(lettera) # P, y, t, h, o, n

# Iterare su un dizionario
prezzi = {"mela": 1.5, "pane": 2.0}
for prodotto in prezzi:
    print(prodotto) # Stampa le chiavi
```

List Comprehension

Le **list comprehension** sono una delle caratteristiche più distinctive di Python.
Permettono di creare liste in modo **conciso, leggibile e performante**.

Filosofia: Esprimere in una riga l'intento "crea una lista applicando questa trasformazione a questi elementi".

Sintassi: [espressione for elemento in iterabile if condizione]

Parte	Obbligatoria	Descrizione
espressione	✓	Cosa mettere nella nuova lista
for elemento in iterabile	✓	Da dove prendere i dati
if condizione	✗	Filtro opzionale

List Comprehension: Confronto

```
# MODO TRADIZIONALE (4 righe)
quadrati = []
for n in range(5):
    quadrati.append(n ** 2)

# LIST COMPREHENSION (1 riga!)
quadrati = [n ** 2 for n in range(5)]
# → [0, 1, 4, 9, 16]

# Con condizione (solo numeri pari)
pari = [n for n in range(10) if n % 2 == 0]
# → [0, 2, 4, 6, 8]
```

Regola pratica: Se la comprehension diventa troppo complessa da leggere, usa un ciclo normale.

List Comprehension Avanzata

```
# Trasformare stringhe
nomi = ["mario", "luigi", "peach"]
maiuscoli = [nome.upper() for nome in nomi]
# → ["MARIO", "LUIGI", "PEACH"]

# Filtrare e trasformare insieme
numeri = [-3, -1, 0, 2, 5, -4]
positivi_raddoppiati = [n * 2 for n in numeri if n > 0]
# → [4, 10]

# Nested comprehension (matrice)
matrice = [[i * j for j in range(3)] for i in range(3)]
# → [[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

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'}
```

Range: Generare Sequenze Numeriche

`range(start, stop, step)` genera numeri senza creare una lista in memoria.

```
# range(stop) - da 0 a stop-1
for i in range(5):
    print(i) # 0, 1, 2, 3, 4

# range(start, stop)
for i in range(2, 6):
    print(i) # 2, 3, 4, 5

# range(start, stop, step)
for i in range(0, 10, 2):
    print(i) # 0, 2, 4, 6, 8

# Contare all'indietro
for i in range(5, 0, -1):
    print(i) # 5, 4, 3, 2, 1
```

Enumerate: Indice + Valore

Quando serve sia l'indice che il valore durante l'iterazione.

```
nomi = ["Anna", "Bruno", "Carla"]

# Modo NON pythonico
for i in range(len(nomi)):
    print(f"{i}: {nomi[i]}")

# Modo PYTHONICO con enumerate
for indice, nome in enumerate(nomi):
    print(f"{indice}: {nome}")

# Partire da un indice diverso
for i, nome in enumerate(nomi, start=1):
    print(f"{i}. {nome}") # 1. Anna, 2. Bruno, 3. Carla
```

Zip: Iterare su Più Liste

Combina elementi di più sequenze in parallelo.

```
nomi = ["Anna", "Bruno", "Carla"]
eta = [25, 30, 28]
citta = ["Roma", "Milano", "Napoli"]

# Iterare su due liste insieme
for nome, anni in zip(nomi, eta):
    print(f"{nome} ha {anni} anni")

# Iterare su tre liste
for nome, anni, luogo in zip(nomi, eta, citta):
    print(f"{nome}, {anni} anni, da {luogo}")

# Creare un dizionario da due liste
anagrafica = dict(zip(nomi, eta))
# {'Anna': 25, 'Bruno': 30, 'Carla': 28}
```

Break e Continue

Controllano il comportamento dentro il ciclo.

```
# BREAK: Interrompe completamente il ciclo
for n in range(10):
    if n == 5:
        print("Trovato 5, esco!")
        break
    print(n) # 0, 1, 2, 3, 4

# CONTINUE: Salta al prossimo giro
for n in range(5):
    if n == 2:
        continue # Salta il 2
    print(n) # 0, 1, 3, 4
```

Else nei Cicli (Poco Conosciuto!)

Il blocco `else` viene eseguito se il ciclo termina **normalmente** (senza `break`).

```
# Cercare un elemento
numeri = [1, 3, 5, 7, 9]
cerca = 4

for n in numeri:
    if n == cerca:
        print(f"Trovato {cerca}!")
        break
else:
    # Eseguito solo se NON c'è stato break
    print(f"{cerca} non trovato nella lista")

# Output: "4 non trovato nella lista"
```

Cicli Annidati

Un ciclo dentro un altro ciclo.

```
# Tabellina pitagorica
for i in range(1, 4):
    for j in range(1, 4):
        print(f"{i} x {j} = {i*j}")
    print("---")

# Iterare su matrice (lista di liste)
matrice = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for riga in matrice:
    for elemento in riga:
        print(elemento, end=" ")
    print() # Nuova riga
```

While vs For: Quando Usare Quale?

Situazione	Usa
Sai quante iterazioni fare	for
Iteri su una collezione	for
Non sai quando fermarti	while
Loop basato su condizione	while
Input utente fino a validazione	while
Contare o enumerare	for + range

```
# FOR: so che voglio 10 iterazioni
for i in range(10):
    print(i)
```

```
# WHILE: non so quando l'utente scriverà "esci"
while input() != "esci":
    pass
```



Esercizi Intermedi: Cicli

Pratica Prima di Proseguire

Esercizio 3.1: Conta le Vocali

Obiettivo: Conta quante vocali ci sono in una stringa.

```
frase = "Python e fantastico"  
vocali = "aeiouAEIOU"  
conteggio = 0  
  
# Usa un ciclo for per contare le vocali  
# Output atteso: "Vocali trovate: 6"
```

Esercizio 3.2: Numeri Primi

Obiettivo: Stampa tutti i numeri primi da 1 a 50.

Suggerimento: Un numero è primo se è divisibile solo per 1 e se stesso.

```
for numero in range(2, 51):
    # Verifica se 'numero' è primo
    # Suggerimento: usa un altro ciclo e break/else
    pass
```

Esercizio 3.3: List Comprehension

Obiettivo: Riscrivi questi cicli come list comprehension.

```
# 1. Estrai solo le parole che iniziano con 'p'  
parole = ["python", "java", "perl", "ruby", "php"]  
risultato = []  
for p in parole:  
    if p.startswith("p"):  
        risultato.append(p)  
  
# 2. Crea una lista dei quadrati dei numeri pari da 1 a 10  
quadrati_pari = []  
for n in range(1, 11):  
    if n % 2 == 0:  
        quadrati_pari.append(n ** 2)
```

4. Le Funzioni

Modularità e Riutilizzo

Perché le Funzioni?

Le funzioni sono uno dei **pilastri della programmazione moderna**. Rappresentano il passaggio da "scrivere codice" a "progettare software".

Principio DRY: *Don't Repeat Yourself* - Non ripetere lo stesso codice!

Problema	Soluzione con Funzioni
Codice duplicato	Scrivi una volta, usa ovunque
File enormi illeggibili	Dividi in blocchi logici
Bug difficili da trovare	Testa ogni funzione isolatamente
Modifiche rischiose	Cambia in un solo punto

Funzioni: Prima e Dopo

```
# SENZA funzione: codice ripetuto, difficile da mantenere
print("=" * 40)
print("TITOLO 1")
print("=" * 40)

print("=" * 40)
print("TITOLO 2")
print("=" * 40)

# CON funzione: riutilizzo, una modifica aggiorna tutto!
def stampa_titolo(testo):
    print("=" * 40)
    print(testo)
    print("=" * 40)

stampa_titolo("TITOLO 1")
stampa_titolo("TITOLO 2")
```

Definizione e Chiamata

Usiamo `def` per definire una funzione.

```
# DEFINIZIONE (crea la funzione)
def saluta():
    print("Ciao!")
    print("Benvenuto!")

# CHIAMATA (esegue la funzione)
saluta()
saluta() # Posso chiamarla quante volte voglio

# Con parametri
def saluta_persona(nome):
    print(f"Ciao, {nome}!")

saluta_persona("Mario") # Ciao, Mario!
saluta_persona("Luigi") # Ciao, Luigi!
```

Return: Restituire Valori

`return` termina la funzione e restituisce un valore.

```
def somma(a, b):
    risultato = a + b
    return risultato

# Il valore restituito può essere usato
x = somma(3, 5)
print(x) # 8

# O usato direttamente
print(somma(10, 20)) # 30

# Senza return, la funzione restituisce None
def solo_stampa(msg):
    print(msg)

risultato = solo_stampa("Test")
print(risultato) # None
```

Return Multipli

Una funzione può restituire più valori (come tupla).

```
def statistiche(numeri):
    """Calcola min, max e media di una lista."""
    minimo = min(numeri)
    massimo = max(numeri)
    media = sum(numeri) / len(numeri)
    return minimo, massimo, media # Tupla implicita

# Unpacking del risultato
dati = [4, 8, 15, 16, 23, 42]
mi, ma, me = statistiche(dati)
print(f"Min: {mi}, Max: {ma}, Media: {me:.2f}")

# O come tupla singola
risultato = statistiche(dati)
print(risultato) # (4, 42, 18.0)
```

Docstring: Documentare le Funzioni

La **docstring** (documentation string) è una stringa speciale che documenta cosa fa una funzione. È fondamentale per il **codice professionale**.

Perché documentare?

- Il codice viene letto più spesso di quanto viene scritto
- Tra 6 mesi non ricorderai cosa fa quella funzione
- Altri sviluppatori devono capire il tuo codice
- Gli IDE mostrano la docstring come tooltip

Convenzione: Usa triple virgolette `"""` subito dopo la definizione.

Docstring: Formato Standard

```
def calcola_area(base, altezza):
    """
    Calcola l'area di un triangolo.

    Args:
        base: La base del triangolo.
        altezza: L'altezza del triangolo.

    Returns:
        L'area del triangolo (base * altezza / 2).
    """
    return base * altezza / 2

# Accedere alla docstring
print(calcola_area.__doc__)
help(calcola_area) # Mostra documentazione formattata
```

Parametri Default

Possiamo dare valori predefiniti agli argomenti.

```
def saluta(nome, saluto="Ciao"):  
    print(f"{saluto}, {nome}!")  
  
saluta("Mario")          # Ciao, Mario!  
saluta("Luigi", "Buongiorno") # Buongiorno, Luigi!  
  
# Esempio pratico  
def calcola_prezzo(base, tassa=0.22, sconto=0):  
    prezzo_tassato = base * (1 + tassa)  
    return prezzo_tassato * (1 - sconto)  
  
print(calcola_prezzo(100))          # 122.0  
print(calcola_prezzo(100, 0.10))    # 110.0  
print(calcola_prezzo(100, 0.22, 0.1)) # 109.8
```

Argomenti Nominati (Keyword Arguments)

Passare argomenti specificando il nome.

```
def crea_utente(nome, eta, citta="Sconosciuta"):
    return {"nome": nome, "eta": eta, "citta": citta}

# Argomenti posizionali (ordine conta)
u1 = crea_utente("Mario", 30)

# Argomenti nominati (ordine non conta)
u2 = crea_utente(eta=25, nome="Luigi", citta="Roma")

# Mix (posizionali PRIMA dei nominati)
u3 = crea_utente("Peach", citta="Milano", eta=28)

print(u2) # {'nome': 'Luigi', 'eta': 25, 'citta': 'Roma'}
```

*args: Argomenti Variabili Posizionali

Quando non sai quanti argomenti arriveranno.

```
def somma_tutto(*numeri):
    # 'numeri' è una TUPLA
    print(f"Ricevuti: {numeri}")
    totale = 0
    for n in numeri:
        totale += n
    return totale

print(somma_tutto(1, 2))          # 3
print(somma_tutto(1, 2, 3, 4, 5)) # 15
print(somma_tutto())              # 0

# Modo più pythonico
def somma_tutto(*numeri):
    return sum(numeri)
```

**kwargs: Argomenti Variabili Nominati

Raccoglie argomenti nominati extra in un dizionario.

```
def stampa_info(**dati):
    # 'dati' è un DIZIONARIO
    for chiave, valore in dati.items():
        print(f"{chiave}: {valore}")

stampa_info(nome="Mario", eta=30, citta="Roma")
# nome: Mario
# eta: 30
# citta: Roma

# Combinare tutto
def funzione_completa(obbligatorio, *args, **kwargs):
    print(f"Obbligatorio: {obbligatorio}")
    print(f"Args: {args}")
    print(f"Kwargs: {kwargs}")
```

Spacchettare Argomenti

Usare `*` e `**` per passare collezioni come argomenti.

```
def somma(a, b, c):
    return a + b + c

# Spacchettare una lista/tupla
numeri = [1, 2, 3]
print(somma(*numeri)) # Equivale a somma(1, 2, 3) → 6

# Spacchettare un dizionario
dati = {"a": 10, "b": 20, "c": 30}
print(somma(**dati)) # Equivale a somma(a=10, b=20, c=30) → 60

# Utile per passare configurazioni
config = {"timeout": 30, "retry": 3}
connetti(**config)
```

Funzioni Lambda (Anonime)

Funzioni brevi definite in una riga.

```
# Sintassi: lambda parametri: espressione

# Funzione normale
def quadrato(x):
    return x ** 2

# Lambda equivalente
quadrato = lambda x: x ** 2

print(quadrato(5)) # 25

# Lambda con più parametri
somma = lambda a, b: a + b
print(somma(3, 4)) # 7

# Lambda con condizione
pari_dispari = lambda n: "pari" if n % 2 == 0 else "dispari"
```

Lambda: Casi d'Uso Tipici

Le lambda sono utili come argomenti di altre funzioni.

```
# Ordinare una lista di tuple per il secondo elemento
studenti = [("Anna", 85), ("Bruno", 92), ("Carla", 78)]

# Con lambda
ordinati = sorted(studenti, key=lambda x: x[1])
# [('Carla', 78), ('Anna', 85), ('Bruno', 92)]

# Filtrare con filter()
numeri = [1, 2, 3, 4, 5, 6]
pari = list(filter(lambda x: x % 2 == 0, numeri))
# [2, 4, 6]

# Trasformare con map()
quadrati = list(map(lambda x: x**2, numeri))
# [1, 4, 9, 16, 25, 36]
```



Esercizi Intermedi: Funzioni (Parte 1)

Pratica Prima di Proseguire

Esercizio 4.1: Funzione Saluto Personalizzato

Obiettivo: Crea una funzione con parametri default.

```
def saluto_personalizzato(nome, saluto="Ciao", emoji="👋"):  
    """  
    Restituisce un saluto personalizzato.  
  
    Args:  
        nome: Il nome della persona  
        saluto: Il tipo di saluto (default: "Ciao")  
        emoji: Emoji da aggiungere (default: "👋")  
  
    Returns:  
        Stringa con il saluto completo  
    """  
    # Completa la funzione  
    pass  
  
# Test:
```

Esercizio 4.2: Lambda e Ordinamento

Obiettivo: Ordina liste usando lambda.

```
prodotti = [
    {"nome": "Laptop", "prezzo": 999, "disponibilita": 5},
    {"nome": "Mouse", "prezzo": 25, "disponibilita": 50},
    {"nome": "Tastiera", "prezzo": 75, "disponibilita": 20},
]

# 1. Ordina per prezzo crescente
per_prezzo = sorted(prodotti, key=lambda x: ____)

# 2. Ordina per disponibilità decrescente
per_disponibilita = sorted(prodotti, key=lambda x: ____, reverse=____)

# 3. Ordina per nome (alfabetico)
per_nome = sorted(prodotti, key=____)
```

Funzioni come Oggetti

In Python, le funzioni sono oggetti di prima classe.

```
# Assegnare funzione a variabile
def saluta(nome):
    return f"Ciao, {nome}!"

mia_funzione = saluta # Senza parentesi!
print(mia_funzione("Mario")) # Ciao, Mario!

# Passare funzione come argomento
def applica(func, valore):
    return func(valore)

def raddoppia(x):
    return x * 2

print(applica(raddoppia, 5)) # 10
print(applica(len, "Python")) # 6
```

Type Hints (Suggerimenti di Tipo)

Annotazioni per indicare i tipi (non obbligatorie, ma utili).

```
def saluta(nome: str) → str:  
    return f"Ciao, {nome}!"  
  
def somma(a: int, b: int) → int:  
    return a + b  
  
def elabora(dati: list[int]) → dict[str, float]:  
    return {  
        "somma": sum(dati),  
        "media": sum(dati) / len(dati)  
    }  
  
# Python NON blocca tipi sbagliati (a runtime)  
# Ma gli IDE li usano per suggerimenti e errori
```

6. Scope delle Variabili

Dove vivono le variabili

Scope Locale vs Globale

Lo **scope** (ambito) determina **dove una variabile è visibile** e accessibile nel codice. È un concetto fondamentale per evitare bug difficili da trovare.

Due tipi principali:

- **Scope Globale:** Variabili definite a livello di modulo (fuori dalle funzioni). Visibili ovunque.
- **Scope Locale:** Variabili definite dentro una funzione. Esistono solo durante l'esecuzione della funzione.

Scope: Esempio Pratico

```
# Variabile GLOBALE (definita fuori dalle funzioni)
messaggio = "Sono globale"

def funzione():
    # Variabile LOCALE (esiste solo dentro la funzione)
    messaggio = "Sono locale"
    print(messaggio)

funzione()      # Stampa: "Sono locale"
print(messaggio) # Stampa: "Sono globale"
```

La variabile locale "**nasconde**" (**shadowing**) quella globale dentro la funzione. Sono due variabili completamente separate!

Leggere vs Modificare Variabili Globali

```
contatore = 0

def incrementa():
    # ERRORE! Python pensa sia una nuova variabile locale
    contatore += 1 # UnboundLocalError
    pass

def leggi():
    # Leggere è OK
    print(contatore) # 0

# Per MODIFICARE, serve 'global'
def incrementa_corretto():
    global contatore
    contatore += 1

incrementa_corretto()
print(contatore) # 1
```

La Parola Chiave **global**

Permette di modificare variabili globali dall'interno di una funzione.

```
punteggio = 0

def aggiungi_punti(punti):
    global punteggio
    punteggio += punti

def reset():
    global punteggio
    punteggio = 0

aggiungi_punti(10)
print(punteggio) # 10
aggiungi_punti(5)
print(punteggio) # 15
reset()
print(punteggio) # 0
```

⚠️ Consiglio: Evita **global** quando possibile. Usa **return** invece.

Nonlocal: Scope Annidato

Per funzioni dentro funzioni.

```
def esterna():
    x = "esterna"

def interna():
    nonlocal x  # Modifica la x di 'esterna', non crea locale
    x = "modificata da interna"

print(f"Prima: {x}")
interna()
print(f"Dopo: {x}")

esterna()
# Prima: esterna
# Dopo: modificata da interna
```

Closure: Funzioni che "Ricordano"

Una **closure** è una funzione che "cattura" e ricorda le variabili dell'ambiente in cui è stata creata, anche dopo che quell'ambiente non esiste più.

Concetto chiave: La funzione interna mantiene un riferimento alle variabili della funzione esterna.

Casi d'uso comuni:

- Factory di funzioni (creare funzioni personalizzate)
- Decoratori (argomento avanzato)
- Callback con stato

Closure: Esempio Factory

```
def crea_moltiplicatore(fattore):
    def moltiplica(numero):
        return numero * fattore # 'fattore' è ricordato!
    return moltiplica

# Creiamo funzioni specializzate
doppio = crea_moltiplicatore(2) # fattore=2 "congelato"
triplo = crea_moltiplicatore(3) # fattore=3 "congelato"

print(doppio(5)) # 10 (5 * 2)
print(triplo(5)) # 15 (5 * 3)
print(doppio(10)) # 20 (10 * 2)
```

Ogni closure mantiene la propria copia di **fattore** !

Regola LEGB

Quando Python incontra un nome di variabile, lo cerca seguendo un **ordine preciso** chiamato **LEGB**:

Livello	Nome	Descrizione
L	Local	Dentro la funzione corrente
E	Enclosing	Nelle funzioni esterne (closure)
G	Global	A livello di modulo
B	Built-in	Funzioni predefinite (print, len, etc.)

Python si ferma appena trova il nome. Se non lo trova in nessun livello: **NameError**.

LEGB: Esempio Visivo

```
x = "globale"          # G - Global

def esterna():
    x = "enclosing"    # E - Enclosing

    def interna():
        x = "locale"   # L - Local
        print(x)        # Cerca: L → trovato! Stampa "locale"

    interna()

esterna()  # Output: "locale"
```

Attenzione: Non sovrascrivere nomi built-in come `list`, `str`, `print`!



Esercizi Intermedi: Scope

Pratica Prima di Proseguire

Esercizio 5.1: Prevedi l'Output

Obiettivo: Senza eseguire, prevedi cosa stampa questo codice.

```
x = 10

def funzione_a():
    x = 20
    print(f"A: {x}")

def funzione_b():
    print(f"B: {x}")

def funzione_c():
    global x
    x = 30
    print(f"C: {x}")

funzione_a()
funzione_b()
funzione_c()
print(f"Finale: {x}")
```

Esercizio 5.2: Contatore con Closure

Obiettivo: Crea una factory di contatori usando closure.

```
def crea_contatore(inizio=0):
    """
    Restituisce una funzione che ad ogni chiamata
    restituisce il prossimo numero della sequenza.
    """
    conteggio = inizio

    def conta():
        nonlocal conteggio
        # Completa: incrementa e restituisce
        pass

    return conta
```

Esercizio 5.3: Correggere il Bug

Obiettivo: Questo codice ha un bug. Trovalo e corregilo.

```
totale_vendite = 0

def aggiungi_vendita(importo):
    totale_vendite += importo
    print(f"Vendita registrata: {importo}€")

def mostra_totale():
    print(f"Totale vendite: {totale_vendite}€")

aggiungi_vendita(100)
aggiungi_vendita(50)
mostra_totale()
```

Suggerimento: Considera due soluzioni: una con `global`, una senza.

7. Laboratorio Pratico

Esercizi su Flusso, Cicli e Funzioni

Esercizio 1: Password Checker

Obiettivo: Funzione che valida una password.

Regole:

1. Lunghezza minima 8 caratteri
2. Deve contenere almeno un numero
3. Deve contenere almeno una lettera maiuscola

Main Loop: Chiedere password finché non è valida.

Soluzione Esercizio 1

```
def check_password(pwd):
    # Regola 1: lunghezza
    if len(pwd) < 8:
        return False, "Troppo corta (min 8 caratteri)"

    # Regola 2: almeno un numero
    if not any(c.isdigit() for c in pwd):
        return False, "Deve contenere almeno un numero"

    # Regola 3: almeno una maiuscola
    if not any(c.isupper() for c in pwd):
        return False, "Deve contenere almeno una maiuscola"

    return True, "Password valida!"

# Main loop
while True:
    pwd = input("Inserisci password: ")
    valida, messaggio = check_password(pwd)
    print(messaggio)
    if valida:
        break
```

Esercizio 2: FizzBuzz

Classico esercizio di programmazione.

Stampa i numeri da 1 a N, ma:

- Se divisibile per 3, stampa "Fizz"
- Se divisibile per 5, stampa "Buzz"
- Se divisibile per entrambi, stampa "FizzBuzz"

Soluzione Esercizio 2

```
def fizzbuzz(n):
    for i in range(1, n + 1):
        if i % 3 == 0 and i % 5 == 0:
            print("FizzBuzz")
        elif i % 3 == 0:
            print("Fizz")
        elif i % 5 == 0:
            print("Buzz")
        else:
            print(i)

fizzbuzz(15)
# 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz,
# 11, Fizz, 13, 14, FizzBuzz
```

Esercizio 3: Calcolatrice

Obiettivo: Creare una calcolatrice con funzioni.

```
def somma(a, b):
    return a + b
def sottrai(a, b):
    return a - b
def moltiplica(a, b):
    return a * b
def dividi(a, b):
    if b == 0:
        return "Errore: divisione per zero"
    return a / b

# Dizionario di operazioni
operazioni = {
    "+": somma,
    "-": sottrai,
    "*": moltiplica,
    "/": dividi
}
```

Soluzione Esercizio 3 (Continua)

```
def calcolatrice():
    while True:
        expr = input("Calcolo (es: 5 + 3) o 'esci': ")
        if expr.lower() == "esci":
            break

        try:
            parti = expr.split()
            a = float(parti[0])
            op = parti[1]
            b = float(parti[2])

            if op in operazioni:
                risultato = operazioni[op](a, b)
                print(f"\n= {risultato}")
            else:
                print("Operatore non valido")
        except:
            print("Formato non valido. Usa: numero operatore numero")

calcolatrice()
```

Riepilogo del Modulo

Concetto	Uso
if/elif/else	Decisioni condizionali
for	Iterare su sequenze note
while	Iterare finché condizione vera
Funzioni	Blocchi riutilizzabili
return	Restituire valori
***args, **kwargs**	Argomenti variabili
Scope	Visibilità delle variabili

⚠ Errori Comuni: Flusso e Funzioni

Errore	Problema	Soluzione
Loop infinito	<code>while True</code> senza <code>break</code>	Assicura condizione di uscita
Scope variabili	Modificare globale senza <code>global</code>	Usa <code>return</code> invece
Default mutabile	<code>def f(lista=[])</code> condivide lista	Usa <code>None</code> come default
Dimenticare <code>return</code>	Funzione restituisce <code>None</code>	Aggiungi <code>return valore</code>

```
# ❌ Default mutabile
def aggiungi(item, lista=[]):
    lista.append(item) # BUG: stessa lista ogni volta!
    return lista
```

```
# ✅ Corretto
def aggiungi(item, lista=None):
    if lista is None:
        lista = []
    lista.append(item)
    return lista
```



Progetto Cross-Modulo: Rubrica Contatti

Parte 3: Funzioni per la Rubrica

```
rubrica = [] # Lista globale di contatti

def aggiungi_contatto(nome, telefono, email):
    """Aggiunge un nuovo contatto alla rubrica."""
    contatto = {"nome": nome, "telefono": telefono, "email": email}
    rubrica.append(contatto)
    print(f"Aggiunto: {nome}")

def cerca_contatto(nome):
    """Cerca un contatto per nome."""
    for contatto in rubrica:
        if contatto["nome"].lower() == nome.lower():
            return contatto
    return None
```

```
def mostra_tutti():
    """Mostra tutti i contatti."""
    if not rubrica:
        print("Rubrica vuota!")
    for c in rubrica:
        print(f"{c['nome']}: {c['telefono']}")
```

Prossimo Modulo: Moduli, OOP e Gestione Errori