

# Modulo 6 Funzioni

---



---

## 31. Functions

### Obiettivi di apprendimento:

- Definire una funzione in Python usando la parola chiave **def**.
- Comprendere l'uso dei parametri (**input**) e dei valori di ritorno (**return**).
- Distinguere tra funzioni che restituiscono un valore e funzioni che eseguono solo un'azione.

### Contenuto teorico:

Una **Funzione** è un blocco di codice organizzato e riutilizzabile che esegue un'unica azione correlata. Le funzioni rendono il codice modulare e più facile da leggere e *debuggare*.

- **Definizione:** Si usa **def** seguito dal nome della funzione e dalle parentesi per i parametri.
- **Corpo:** Il codice all'interno della funzione deve essere indentato.
- **return:** La parola chiave **return** viene utilizzata per restituire un valore dalla funzione all'ambiente chiamante. Se non è presente, la funzione restituisce implicitamente **None**.

```
# 1. Funzione che esegue un'azione (non restituisce un valore significativo)
def saluta(nome):
    """Stampa un messaggio di saluto."""
    print(f"Ciao, {nome}! Benvenuto.")

# Chiamata della funzione
saluta("Alice")
saluta("Bob")

# 2. Funzione che restituisce un valore
def somma_due_numeri(a, b):
    """Restituisce la somma di due numeri."""
    risultato = a + b
    return risultato

# Uso del valore di ritorno
valore1 = 10
valore2 = 5
somma = somma_due_numeri(valore1, valore2)
print(f"\nLa somma di {valore1} e {valore2} è: {somma}")

# 3. Restituire più valori (Python li incapsula in una Tupla)
def calcola_stats(lista_numeri):
    if not lista_numeri:
        return 0, 0, 0

    somma = sum(lista_numeri)
    media = somma / len(lista_numeri)
```

```

massimo = max(lista_numeri)
return somma, media, massimo

# Assegnazione multipla (unpacking della tupla)
dati = [10, 20, 30, 40]
tot, avg, mx = calcola_stats(dati)
print(f"\nDati: {dati}")
print(f"Somma: {tot}, Media: {avg}, Max: {mx}")

```

## 32. Default Arguments

### Obiettivi di apprendimento:

- Definire valori predefiniti per i parametri di una funzione.
- Capire come i parametri predefiniti influenzano la flessibilità della chiamata.

### Contenuto teorico:

Gli **Argomenti Predefiniti** (Default Arguments) permettono di assegnare un valore di *default* a un parametro direttamente nella definizione della funzione.

- Se l'utente **non** specifica un valore per quel parametro durante la chiamata, verrà utilizzato il valore predefinito.
- Se l'utente **specificava** un valore, il valore predefinito viene ignorato.
- I parametri con valori predefiniti devono sempre essere definiti **dopo** i parametri posizionali (quelli senza default).

```

def genera_report(nome_cliente, valuta="EUR", tasso_tasse=0.22):
    """Genera un report applicando un tasso di tasse predefinito (22%
    IVA)."""

    # Questo parametro è obbligatorio
    print(f"\nReport generato per: {nome_cliente}")

    # Se la valuta non è specificata, usa EUR
    print(f"Valuta utilizzata: {valuta}")

    # Se il tasso non è specificato, usa 0.22
    print(f"Tasso Tasse applicato: {tasso_tasse:.0%}")

# 1. Chiamata senza parametri opzionali (usa i default)
genera_report("Azienda Alpha")
# Output: Tasso Tasse applicato: 22%

# 2. Chiamata specificando solo il secondo parametro (valuta)
genera_report("Azienda Beta", valuta="USD")
# Output: Valuta utilizzata: USD, Tasso Tasse applicato: 22%

# 3. Chiamata specificando tutti i parametri

```

```
genera_report("Azienda Gamma", valuta="GBP", tasso_tasse=0.05)
# Output: Tasso Tasse applicato: 5%
```

## 33. Keyword Arguments

### Obiettivi di apprendimento:

- Capire come usare gli argomenti per parola chiave (**keyword arguments**) per migliorare la leggibilità.
- Conoscere la differenza tra argomenti posizionali e argomenti per parola chiave.

### Contenuto teorico:

Gli **Argomenti per Parola Chiave** (Keyword Arguments) permettono di passare valori a una funzione specificando il nome del parametro a cui sono destinati.

- **Vantaggio:** Rende le chiamate a funzioni con molti parametri molto più chiare, poiché si capisce immediatamente a cosa serve ogni valore.
- **Ordine:** Quando si usano i keyword arguments, l'ordine dei parametri nella chiamata **non è importante**, a differenza degli argomenti posizionali.
- **Regola:** Tutti gli argomenti posizionali devono essere forniti **prima** di qualsiasi keyword argument.

```
def crea_email(destinatario, oggetto, corpo, firma="Team Python"):
    """Crea una bozza di email usando tutti i parametri specificati."""
    print("--- NUOVA EMAIL ---")
    print(f"A: {destinatario}")
    print(f"Oggetto: {oggetto}")
    print(f"Corpo: {corpo}")
    print(f"Firma: {firma}")
    print("-----")

# 1. Chiamata con argomenti posizionali (standard, l'ordine è cruciale)
crea_email("anna@mail.com", "Riunione", "Si terrà lunedì alle 10:00")

# 2. Chiamata con Keyword Arguments (l'ordine non è importante, la
# leggibilità migliora)
crea_email(
    corpo="Il codice è pronto per la revisione.",
    oggetto="Code Review",
    destinatario="dev@comp.com",
    firma="Luca (Sviluppatore")
)

# 3. Combinazione (Posizionale prima, Keyword dopo)
# Questo funziona:
crea_email("boss@mail.com", "Aggiornamento settimanale", corpo="...")

# Questo NON funziona (SyntaxError: Positional argument follows keyword
# argument):
# crea_email(destinatario="boss@mail.com", "Aggiornamento", corpo="...")
```

## 34. \*args & \*\*kwargs

### Obiettivi di apprendimento:

- Comprendere l'uso di `*args` per accettare un numero variabile di argomenti posizionali.
- Comprendere l'uso di `**kwargs` per accettare un numero variabile di argomenti per parola chiave.

### Contenuto teorico:

Questi due simboli speciali permettono di creare funzioni estremamente flessibili che possono accettare un numero arbitrario di argomenti.

#### `*args` (Arbitrary Positional Arguments)

- Raccoglie tutti gli argomenti posizionali aggiuntivi (non esplicitamente definiti) in una **tupla**.
- Il nome `args` è convenzionale; l'asterisco (\*) è ciò che conta.

```
def calcola_somma(*numeri):
    """Accetta un numero qualsiasi di argomenti e ne calcola la somma."""
    # 'numeri' è ora una tupla: (10, 20, 30, 40)
    print(f"Tipo di dati ricevuto (*args): {type(numeri)}")

    risultato = sum(numeri)
    return risultato

print(f"\nSomma di 10, 20, 30: {calcola_somma(10, 20, 30)}")
print(f"Somma di 1, 2, 3, 4, 5, 6: {calcola_somma(1, 2, 3, 4, 5, 6)}")
```

#### `**kwargs` (Arbitrary Keyword Arguments)

- Raccoglie tutti gli argomenti per parola chiave aggiuntivi (non esplicitamente definiti) in un **dizionario**.
- Il nome `kwargs` è convenzionale; il doppio asterisco (\*\*) è ciò che conta.

```
def registra_utente(nome, email, **dati_aggiuntivi):
    """Registra l'utente con i dati di base e qualsiasi dato aggiuntivo."""
    print(f"\nRegistrazione utente: {nome} ({email})")

    # 'dati_aggiuntivi' è ora un dizionario: {'eta': 30, 'citta': 'Milano'}
    print(f"Tipo di dati ricevuto (**kwargs): {type(dati_aggiuntivi)}")

    if dati_aggiuntivi:
        print("Dati aggiuntivi:")
        for chiave, valore in dati_aggiuntivi.items():
```

```

print(f"- {chiave}: {valore}")

# Chiamata con argomenti di base e kwargs
registra_utente("Marco Verdi", "marco@mail.it", eta=30, citta="Milano",
hobby="trekking")

```

## 35. Iterables

### Obiettivi di apprendimento:

- Comprendere la definizione di **iterable** e **iterator**.
- Capire come funzionano i cicli **for** "sotto il cofano".

### Contenuto teorico:

Un **Iterable** è qualsiasi oggetto Python che può essere "iterato", ovvero su cui si può eseguire un ciclo elemento per elemento.

Esempi comuni di iterables includono: **Lists**, **Tuples**, **Sets**, **Dictionaries**, **Strings**, e **range**.

- **Iterables:** Sono oggetti che hanno un metodo `__iter__()` che restituisce un **iterator**.
- **Iterators:** Sono oggetti che mantengono lo stato di avanzamento e hanno un metodo `__next__()` che restituisce l'elemento successivo nella sequenza. Quando non ci sono più elementi, sollevano l'eccezione `StopIteration`.

### Come funziona **for** loop (dietro le quinte):

1. Quando si scrive `for elemento in lista:`, Python chiama internamente `iter(lista)` per ottenere un **iterator**.
2. Ad ogni ciclo, Python chiama `next(iterator)` per ottenere l'elemento successivo.
3. Quando `next()` solleva `StopIteration`, il ciclo **for** termina.

```

# Esempio manuale di iterazione
lista_numeri = [10, 20, 30]

# 1. Ottenere l'iterator
iteratore = iter(lista_numeri)

# 2. Chiamare next() manualmente
print(f"Primo elemento: {next(iteratore)}")
print(f"Secondo elemento: {next(iteratore)}")
print(f"Terzo elemento: {next(iteratore)}")

# 3. La prossima chiamata a next() solleverebbe StopIteration
# print(next(iteratore))

```

## 36. Membership Operators

## Obiettivi di apprendimento:

- Utilizzare gli operatori `in` e `not in` per verificare l'appartenenza a una collezione.
- Capire l'efficienza degli operatori di appartenenza su diversi tipi di collezioni (List, Set, Dictionary).

## Contenuto teorico:

Gli **Operatori di Appartenenza** (Membership Operators) vengono usati per verificare se un valore è presente o assente in una sequenza (stringa, lista, tupla, set o dizionario).

- `in`: Restituisce `True` se l'elemento specificato è presente nella sequenza.
- `not in`: Restituisce `True` se l'elemento specificato **non** è presente nella sequenza.

```
lista_frutti = ["mela", "kiwi", "banana"]
testo = "Ciao mondo!"
dati_utente = {"nome": "Anna", "eta": 25}

print("== MEMBERSHIP OPERATORS ==")

# 1. Liste
print(f"'kiwi' in lista_frutti? {'kiwi' in lista_frutti}") # True
print(f"'arancia' not in lista_frutti? {'arancia' not in lista_frutti}") # True

# 2. Stringhe (verifica la presenza di una sottostringa)
print(f"'mon' in testo? {'mon' in testo}") # True

# 3. Dizionari (verifica la presenza solo nelle CHIAVI)
print(f"'nome' in dati_utente? {'nome' in dati_utente}") # True
print(f"'Anna' in dati_utente? {'Anna' in dati_utente}") # False (cerca nelle chiavi)
print(f"'Anna' in dati_utente.values()? {'Anna' in dati_utente.values()}") # True (cerca nei valori)

# Performance (Cenni)
# La ricerca in un **Set** o nelle **chiavi di un Dizionario** è molto veloce (O(1)).
# La ricerca in una **Lista** o **Tupla** è più lenta (O(n)), perché deve scorrere tutti gli elementi.
```

## 37. List Comprehensions

### Obiettivi di apprendimento:

- Scrivere codice più compatto e leggibile usando le List Comprehensions.
- Creare liste da altre iterables, applicando trasformazioni e filtri.

### Contenuto teorico:

Una **List Comprehension** offre un modo conciso per creare una nuova lista basata su una sequenza esistente, applicando tipicamente una trasformazione o un filtro.

- **Sintassi Base:** [espressione for elemento in iterabile]
- **Sintassi con Condizione:** [espressione for elemento in iterabile if condizione]

Le List Comprehensions sono generalmente **più veloci** e sempre **più leggibili** dei cicli **for** tradizionali per la creazione di liste.

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

print("== LIST COMPREHENSIONS ==")

# 1. Trasformazione: Quadrati di tutti i numeri
quadrati = [x ** 2 for x in numeri]
print(f"Quadrati: {quadrati}") # [1, 4, 9, 16, 25, ...]

# 2. Filtro: Solo i numeri pari
numeri_pari = [x for x in numeri if x % 2 == 0]
print(f"Pari: {numeri_pari}") # [2, 4, 6, 8]

# 3. Trasformazione e Filtro combinati
# Trova i quadrati solo dei numeri dispari
quadrati_dispari = [x ** 2 for x in numeri if x % 2 != 0]
print(f"Quadrati dispari: {quadrati_dispari}") # [1, 9, 25, 49, 81]

# 4. Creazione da Stringhe
parola = "python"
lettere_maiuscole = [c.upper() for c in parola]
print(f"Lettere maiuscole: {lettere_maiuscole}") # ['P', 'Y', 'T', 'H',
'0', 'N']
```

## 38. Match-Case Statements

### Obiettivi di apprendimento:

- Utilizzare la struttura **match-case** come alternativa moderna ai complessi **if-elif-else** per il *pattern matching*.
- Capire come gestire i *pattern* con valori, letterali e il *wildcard* (`_`).

### Contenuto teorico:

Lo statement **match-case** (introdotto in Python 3.10) è l'equivalente del **switch** di altri linguaggi e permette di eseguire codice basato sul valore (il *pattern*) di un'espressione.

- **match:** Prende un'espressione (es. una variabile) da confrontare.
- **case:** Definisce i *pattern* di confronto.
- **case \_ (Wildcard):** Agisce come il **default** di **switch**; se nessun case precedente corrisponde, questo blocco viene eseguito.

```

def assegna_livello_permessi(codice_utente):
    """Assegna un livello di accesso basato su un codice
numerico/stringa."""

    match codice_utente:
        # Match di un valore letterale
        case 100:
            livello = "Amministratore"

        # Match di un altro valore letterale
        case 200 | 201: # Match multiplo
            livello = "Editor"

        # Match di una stringa
        case "GUEST":
            livello = "Utente Ospite"

        # Match complesso con variabili (per utenti con nome specifico)
        # Nota: i pattern sono più potenti di un semplice confronto di
        uguaglianza
        # case Utente(nome="Luca"): ...

        # Match Wildcard (default)
        case _:
            livello = "Utente Standard"

    print(f"Codice '{codice_utente}' assegnato a: {livello}")

assegna_livello_permessi(100)
assegna_livello_permessi(201)
assegna_livello_permessi("GUEST")
assegna_livello_permessi(550) # Cade nel case _

```

## 39. Modules

### Obiettivi di apprendimento:

- Comprendere cos'è un modulo e come promuove la riusabilità del codice.
- Usare la parola chiave `import` per accedere a funzionalità esterne (librerie standard o file personali).
- Distinguere tra `import modulo`, `import modulo as alias` e `from modulo import funzione`.

### Contenuto teorico:

Un **Modulo** è semplicemente un file Python (`.py`) contenente definizioni di classi, funzioni e variabili che possono essere usate in altri programmi.

### 1. Importazione di base

```
# Importa l'intero modulo math
import math

# Per usare le sue funzioni, bisogna premettere il nome del modulo
radice_quadrata = math.sqrt(25)
pi_value = math.pi
print(f"Radice di 25: {radice_quadrata}")
```

## 2. Importazione con alias

Utile per moduli con nomi lunghi.

```
import random as rnd # Alias rnd
numero_casuale = rnd.randint(1, 100)
print(f"Numero casuale (con alias): {numero_casuale}")
```

## 3. Importazione selettiva (`from...import`)

Importa solo le funzioni o classi specifiche, rendendole accessibili direttamente senza il prefisso del modulo.

```
from time import sleep
from random import choice

print("Inizio pausa...")
sleep(1) # Chiamata diretta a sleep
print("Fine pausa!")

scelta = choice(["A", "B", "C"]) # Chiamata diretta a choice
print(f"Scelta diretta: {scelta}")

# Attenzione: from time import * (importare tutto) è sconsigliato
# perché può causare conflitti di nomi (namespace pollution).
```

## 40. Scope Resolution 🎧

### Obiettivi di apprendimento:

- Comprendere il concetto di **scope** (ambito) e il meccanismo **LEGB** in Python.
- Distinguere tra variabili locali, racchiuse (enclosing) e globali.
- Utilizzare le parole chiave **global** e **nonlocal** per modificare variabili in ambiti superiori.

### Contenuto teorico:

Lo **Scope** definisce dove una variabile è accessibile all'interno di un programma. Python usa la regola **LEGB** per risolvere gli ambiti:

1. **L (Local)**: Ambito definito all'interno della funzione corrente.
2. **E (Enclosing/Nonlocal)**: Ambito delle funzioni esterne racchiudenti (nested functions).
3. **G (Global)**: Ambito definito nel corpo principale del modulo (`.py`).
4. **B (Built-in)**: Ambito dei nomi predefiniti di Python (es. `print`, `len`, `str`).

Python cerca sempre una variabile partendo dal livello locale (L) e risalendo fino al livello built-in (B).

```
# G - Ambito Globale
nome_globale = "Mondo"

def funzione_esterna():
    # E - Ambito Enclosing/Racchiuso
    nome_esterno = "Python"

    def funzione_interna():
        # L - Ambito Locale
        nome_locale = "Funzione"

        # Accesso (L -> E -> G)
        print(f'L: {nome_locale}') # Trovato in L
        print(f'E: {nome_esterno}') # Trovato in E
        print(f'G: {nome_globale}') # Trovato in G

    funzione_interna()

funzione_esterna()

# Uso di 'global' e 'nonlocal'
def modifica_ambiti():
    x = 10 # Ambito locale (funzione esterna)
    y = 50 # Variabile racchiusa

    def modifica():
        # Usa la parola chiave 'global' per modificare la variabile
        'globale'
        global x_globale
        x_globale = 999

        # Usa la parola chiave 'nonlocal' per modificare la variabile
        'racchiusa'
        nonlocal y
        y = 55

    z = 10 # Variabile LOCALE a 'modifica'

    modifica()
    print(f'\nVariabile \'y\' (nonlocal) modificata a: {y}') # 55

x_globale = 100
modifica_ambiti()
print(f'Variabile \'x_globale\' (global) modificata a: {x_globale}') # 999
```

## 41. if name == '\_\_main\_\_': 📩

### Obiettivi di apprendimento:

- Comprendere il ruolo della variabile speciale `__name__`.
- Sapere quando e perché utilizzare il blocco `if __name__ == '__main__':`

### Contenuto teorico:

Quando un file Python viene eseguito, la variabile speciale `__name__` viene impostata in base al contesto:

- Esecuzione Diretta:** Se il file viene eseguito come programma principale (es. `python miofile.py`), Python imposta `__name__` a "`__main__`".
- Importazione:** Se il file viene importato in un altro file (es. `import myfile`), `__name__` viene impostato al **nome del modulo** (es. "`"myfile"`").

Il blocco `if __name__ == '__main__':` è il punto di ingresso standard del programma e serve a:

- **Evitare l'Esecuzione Involontaria:** Impedisce che il codice di esecuzione (come chiamate a funzioni di test, demo o il ciclo principale del programma) venga eseguito quando il modulo viene semplicemente importato da un altro file.
- **Chiarezza:** Identifica chiaramente il codice che deve essere eseguito quando il file viene lanciato direttamente.

```
# file: utility.py

def calcola_area(lato):
    return lato * lato

def main():
    print("--- ESECUZIONE MAIN ---")

    # Questo codice verrà eseguito solo se il file viene lanciato
    # direttamente
    print(f'L\'area di un quadrato di lato 5 è: {calcola_area(5)}')

# Punto di ingresso standard:
if __name__ == '__main__':
    # Quando esegui 'python utility.py', __name__ è '__main__'
    main()
else:
    # Quando un altro file fa 'import utility', __name__ è 'utility'
    print(f"Modulo 'utility' importato. __name__ è: {__name__}")

# Se un altro file importa 'utility.py', può comunque usare calcola_area()
# senza eseguire la funzione main().
```

043\_Slot\_Machine

---

044\_Encryption\_Program

---

045\_Hangman\_Game