**Procedure e funzioni**

* Sottoprogrammi
* Definizione di procedura e funzione
* Sintassi di base in python
* Modulo dove inserire le funzioni ed importato nel programma main
* Argomenti di una funzione
* Visibilità delle variabili
* Parametri attuali e formali
* Parametri opzionali
* Funzioni polimorfe, cioè con numero di argomenti variabili
* Valori di ritorno da funzione tramite una lista
* Funzioni ricorsive
* Funzioni come variabili e passate come argomenti ad altre funzioni
* Funzioni anonime lambda

In Python, una funzione è un blocco di codice riutilizzabile che esegue un'attività specifica. Le funzioni ti permettono di organizzare il codice, renderlo più leggibile e riutilizzabile. Ecco come si definiscono le funzioni in Python:

**Sintassi di base**

def nome\_funzione(parametri):  
    return valore\_di\_ritorno # Opzionale

* **def**: La parola chiave che indica l'inizio della definizione di una funzione.
* **nome\_funzione**: Il nome della funzione, che deve seguire le convenzioni di denominazione di Python (ad esempio, snake\_case).
* **parametri**: Valori di input opzionali che la funzione può accettare.
* **:**: I due punti che indicano l'inizio del blocco di codice della funzione.
* **Docstring**: Una stringa di documentazione opzionale che descrive lo scopo della funzione.
* **Corpo della funzione**: Il blocco di codice che esegue l'attività della funzione.
* **return valore\_di\_ritorno**: Un'istruzione opzionale che restituisce un valore dalla funzione.

**Esempi**

1. **Procedura, funzione senza parametri e senza valore di ritorno:**

def saluta():  
    """Stampa un messaggio di saluto."""  
    print("Ciao!")  
  
saluta() # Chiamata alla funzione

1. **Procedura, funzione con parametri e senza valore di ritorno:**

def saluta\_nome(nome):  
    """Stampa un messaggio di saluto personalizzato."""  
    print(f"Ciao, {nome}!")  
  
saluta\_nome("Alice") # Chiamata alla funzione con un argomento

1. **Funzione con parametri e con valore di ritorno:**

def somma(a, b):  
    """Restituisce la somma di due numeri."""  
    return a + b  
  
risultato = somma(5, 3) # Chiamata alla funzione e assegnazione del valore di ritorno  
print(risultato) # Stampa 8

1. **Funzione con parametri opzionali:**

def potenza(base, esponente=2):  
    """Calcola la potenza di un numero (l'esponente predefinito è 2)."""  
    return base \*\* esponente  
  
print(potenza(3)) # Stampa 9 (3^2)  
print(potenza(3, 3)) # Stampa 27 (3^3)

**Chiamata di una funzione**

Per eseguire il codice all'interno di una funzione, è necessario chiamarla utilizzando il suo nome seguito da parentesi tonde. Se la funzione accetta parametri, è necessario passare i valori corrispondenti all'interno delle parentesi.

**Funzioni con argomenti variabili**

In Python, \*args e \*\*kwargs sono usati nelle definizioni di funzione per consentire un numero variabile di argomenti. Funzioni polimorfe

**Perché sono utili?**

* **Flessibilità**: Permettono di creare funzioni che possono gestire un numero variabile di input, rendendo il codice più adattabile.
* **Codice più pulito**: Invece di dover definire molteplici versioni di una funzione per diversi

numeri di argomenti, si può usare \*args e \*\*kwargs.

* **Compatibilità**: Sono spesso usati in librerie e framework per permettere agli utenti di passare opzioni aggiuntive alle funzioni.

In Python, \*args e \*\*kwargs sono usati nelle definizioni di funzione per consentire un numero variabile di argomenti. Ecco una spiegazione dettagliata:

**\*args (Argomenti Posizionali Variabili)**

* **Scopo**:
* \*args permette a una funzione di accettare un numero arbitrario di argomenti posizionali.
* Gli argomenti posizionali sono quelli passati alla funzione senza un nome specifico.
* **Funzionamento**:
* Quando una funzione è chiamata con \*args, tutti gli argomenti posizionali extra vengono raccolti in una tupla.
* All'interno della funzione, args è una tupla che contiene questi argomenti.
* **Esempio**:

    def somma(\*args):  
        risultato = 0  
        for numero in args:  
            risultato += numero  
        return risultato  
  
    print(somma(1, 2, 3))  # Output: 6  
    print(somma(1, 2, 3, 4, 5))  # Output: 15

**\*\*kwargs (Argomenti con Nome Variabili)**

* **Scopo**:
* \*\*kwargs consente a una funzione di accettare un numero arbitrario di argomenti con nome (o argomenti "keyword").
* Gli argomenti con nome sono passati alla funzione utilizzando la sintassi nome=valore.
* **Funzionamento**:
* Quando una funzione è chiamata con \*\*kwargs, tutti gli argomenti con nome extra vengono raccolti in un dizionario.
* All'interno della funzione, kwargs è un dizionario che contiene questi argomenti.
* **Esempio**:

    def stampa\_info(\*\*kwargs):  
        for chiave, valore in kwargs.items():  
            print(f"{chiave}: {valore}")  
  
    stampa\_info(nome="Alice", eta=30, citta="Roma")  
    # Output:  
    # nome: Alice  
    # eta: 30  
    # citta: Roma

**Uso Combinato**

È possibile utilizzare sia \*args che \*\*kwargs nella stessa definizione di funzione:

    def funzione\_mista(\*args, \*\*kwargs):  
        print("Argomenti posizionali:", args)  
        print("Argomenti con nome:", kwargs)  
  
    funzione\_mista(1, 2, 3, nome="Bob", professione="Programmatore")  
    # Output:  
    # Argomenti posizionali: (1, 2, 3)  
    # Argomenti con nome: {'nome': 'Bob', 'professione': 'Programmatore'}

Python offre un modo per gestire un numero arbitrario di argomenti in una funzione attraverso simboli speciali.

Questi simboli sono \* e \*\* , e vengono utilizzati davanti ai nomi delle variabili per indicare rispettivamente \*args (posizionali) e \*\*kwargs(parole chiavi).

Esempio

\*args

def add(\*args):

return sum(args)

#add(1,2,3,4)

#10

def supporter(\*\*kwargs):

for k,v in kwargs.items():

print(f"{k} tifa per la squadra {v}")

supporter(davide="Napoli")

link per scaricare i file di esempio

https://github.com/albertocesaretti/python--funzioni

**Funzioni Ricorsive**

Le **funzioni ricorsive** in Python sono funzioni che richiamano se stesse direttamente o indirettamente per risolvere un problema. Sono particolarmente utili quando un problema può essere scomposto in sottoproblemi più piccoli e simili.

Ecco alcuni esempi comuni di funzioni ricorsive in Python:

**1. Fattoriale di un numero**

Il fattoriale di un numero intero non negativo n, indicato con n\!, è il prodotto di tutti gli interi positivi minori o uguali a n. La definizione ricorsiva è:

* 0\! = 1
* n\! = n \\times (n-1)\! per n \> 0

<!-- end list -->

def fattoriale(n):  
    if n == 0:  # Caso base: condizione di terminazione della ricorsione  
        return 1  
    else:  # Passo ricorsivo  
        return n \* fattoriale(n - 1)  
  
print(f"Il fattoriale di 5 è: {fattoriale(5)}")  # Output: Il fattoriale di 5 è: 120  
print(f"Il fattoriale di 0 è: {fattoriale(0)}")  # Output: Il fattoriale di 0 è: 1

**2. Successione di Fibonacci**

La successione di Fibonacci è una sequenza di numeri in cui ogni numero è la somma dei due precedenti, con i primi due numeri che sono 0 e 1 (o 1 e 1, a seconda della convenzione). La definizione ricorsiva è:

* F(0) = 0
* F(1) = 1
* F(n) = F(n-1) + F(n-2) per n \> 1

<!-- end list -->

def fibonacci(n):  
    if n <= 1:  # Caso base  
        return n  
    else:  # Passo ricorsivo  
        return fibonacci(n - 1) + fibonacci(n - 2)  
  
print(f"Il 7° numero di Fibonacci è: {fibonacci(7)}")  # Output: Il 7° numero di Fibonacci è: 13  
# La sequenza inizia con 0, 1, 1, 2, 3, 5, 8, 13... quindi il 7° elemento (indice 6) è 8,  
# mentre il 7° elemento (se iniziamo a contare da 1) è 13. L'output è corretto per la convenzione F(0)=0.

**Attenzione:** La funzione di Fibonacci ricorsiva mostrata sopra è molto intuitiva, ma è **inefficiente** per valori elevati di n a causa delle molteplici ricalcolazioni degli stessi valori. Per problemi reali, si preferirebbe un'implementazione iterativa o la memorizzazione dei risultati (memoization).

**3. Somma degli elementi di una lista**

Questo esempio mostra come una funzione ricorsiva può essere usata per attraversare una struttura dati come una lista.

def somma\_lista(lista):  
    if not lista:  # Caso base: lista vuota  
        return 0  
    else:  # Passo ricorsivo: somma il primo elemento con la somma del resto della lista  
        return lista[0] + somma\_lista(lista[1:])  
  
my\_list = [1, 2, 3, 4, 5]  
print(f"La somma degli elementi della lista {my\_list} è: {somma\_lista(my\_list)}")  # Output: La somma degli elementi della lista [1, 2, 3, 4, 5] è: 15

**4. Calcolo della potenza (base elevata a esponente)**

Per calcolare base^{esponente}, possiamo usare la definizione ricorsiva:

* base^0 = 1
* base^{esponente} = base \\times base^{esponente-1} per esponente \> 0

<!-- end list -->

def potenza(base, esponente):  
    if esponente == 0:  # Caso base  
        return 1  
    else:  # Passo ricorsivo  
        return base \* potenza(base, esponente - 1)  
  
print(f"2 elevato alla 3 è: {potenza(2, 3)}")  # Output: 2 elevato alla 3 è: 8  
print(f"5 elevato alla 0 è: {potenza(5, 0)}")  # Output: 5 elevato alla 0 è: 1

**5. Inversione di una stringa**

def inverti\_stringa(s):  
    if len(s) == 0:  # Caso base: stringa vuota  
        return ""  
    else:  # Passo ricorsivo: il primo carattere va alla fine della stringa invertita del resto  
        return inverti\_stringa(s[1:]) + s[0]  
  
my\_string = "hello"  
print(f"La stringa '{my\_string}' invertita è: '{inverti\_stringa(my\_string)}'")  # Output: La stringa 'hello' invertita è: 'olleh'

**Quando usare la ricorsione?**

La ricorsione è spesso una scelta elegante quando:

* Il problema ha una **struttura ricorsiva** intrinseca (es. alberi, grafi, strutture matematiche come il fattoriale).
* La soluzione ricorsiva è più **leggera e leggibile** rispetto a una iterativa.
* I **casi base** (condizioni di terminazione) sono chiaramente definiti.

**Limitazioni della ricorsione**

* **Profondità massima di ricorsione (Recursion Depth Limit):** Python ha un limite predefinito al numero di chiamate ricorsive per prevenire errori di stack overflow. Questo limite può essere aumentato, ma con cautela.

**Inefficienza:** Come visto con Fibonacci, alcune soluzioni ricorsive possono essere computazionalmente costose a causa di ricalcoli ridondanti. In questi casi, tecniche come la **memoization** (memorizzazione dei risultati) o la trasformazione in un algoritmo iterativo possono migliorare le prestazioni.

**Funzioni lambda**

Le **funzioni lambda** in Python sono piccole funzioni anonime, cioè senza nome. Sono definite con la parola chiave lambda e possono avere un numero qualsiasi di argomenti, ma **una sola espressione**. Il risultato di questa espressione è il valore restituito dalla funzione.

Sono particolarmente utili per operazioni semplici e quando hai bisogno di una funzione per un breve periodo, spesso come argomento per funzioni di ordine superiore come map(), filter(), o sorted().

Ecco qualche esempio pratico:

**1. Funzione Lambda Semplice**

# Una funzione lambda che somma due numeri  
somma = lambda a, b: a + b  
print(somma(5, 3)) # Output: 8  
  
# Una funzione lambda che moltiplica un numero per se stesso  
quadrato = lambda x: x \* x  
print(quadrato(7)) # Output: 49

In questi esempi, somma e quadrato sono variabili a cui abbiamo assegnato le funzioni lambda.

**2. Lambda con filter()**

La funzione filter() costruisce un iteratore dagli elementi di un iterabile per cui una funzione restituisce True. Le lambda sono perfette per questo.

numeri = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
# Filtra solo i numeri pari  
numeri\_pari = list(filter(lambda x: x % 2 == 0, numeri))  
print(numeri\_pari) # Output: [2, 4, 6, 8, 10]  
  
# Filtra solo i numeri maggiori di 5  
numeri\_maggiori\_di\_5 = list(filter(lambda x: x > 5, numeri))  
print(numeri\_maggiori\_di\_5) # Output: [6, 7, 8, 9, 10]

**3. Lambda con map()**

La funzione map() applica una data funzione a tutti gli elementi di un iterabile e restituisce un iteratore dei risultati.

numeri = [1, 2, 3, 4, 5]  
  
# Raddoppia ogni numero  
numeri\_raddoppiati = list(map(lambda x: x \* 2, numeri))  
print(numeri\_raddoppiati) # Output: [2, 4, 6, 8, 10]  
  
# Converte ogni numero in una stringa aggiungendo "n\_"  
numeri\_stringhe = list(map(lambda x: "n\_" + str(x), numeri))  
print(numeri\_stringhe) # Output: ['n\_1', 'n\_2', 'n\_3', 'n\_4', 'n\_5']

**4. Lambda con sorted()**

La funzione sorted() restituisce una nuova lista ordinata dagli elementi dell'iterabile specificato. Puoi usare una lambda per definire una chiave di ordinamento personalizzata.

# Lista di tuple (nome, età)  
persone = [("Alice", 30), ("Bob", 25), ("Charlie", 35)]  
  
# Ordina per età (il secondo elemento della tupla)  
persone\_ordinate\_per\_eta = sorted(persone, key=lambda persona: persona[1])  
print(persone\_ordinate\_per\_eta) # Output: [('Bob', 25), ('Alice', 30), ('Charlie', 35)]  
  
# Lista di stringhe  
parole = ["banana", "mela", "kiwi", "arancia"]  
  
# Ordina per lunghezza della stringa  
parole\_ordinate\_per\_lunghezza = sorted(parole, key=lambda parola: len(parola))  
print(parole\_ordinate\_per\_lunghezza) # Output: ['mela', 'kiwi', 'banana', 'arancia']

**5. Lambda come argomento per una funzione personalizzata**

Puoi passare una lambda anche a una tua funzione, se questa accetta una funzione come argomento.

def applica\_operazione(lista, operazione):  
    risultati = []  
    for elemento in lista:  
        risultati.append(operazione(elemento))  
    return risultati  
  
numeri = [1, 2, 3, 4, 5]  
  
# Applica una lambda per elevare al cubo ogni numero  
cubi = applica\_operazione(numeri, lambda x: x\*\*3)  
print(cubi) # Output: [1, 8, 27, 64, 125]

**Quando usare le lambda?**

* **Brevi e Semplici:** Quando la logica della funzione è molto concisa e può essere espressa in una singola riga.
* **Funzioni di Ordine Superiore:** Come visto negli esempi, sono perfette con map(), filter(), sorted(), min(), max() e altre funzioni che accettano un altro funzione come argomento.
* **Evitare di Definire una Funzione Nascosta:** Se hai bisogno di una funzione che userai solo una volta in un contesto specifico, usare una lambda evita di inquinare il namespace globale con una funzione definita in modo formale (con def).

**Quando *non* usare le lambda?**

* **Logica Complessa:** Se la funzione richiede più espressioni, istruzioni condizionali (if/else) complesse, cicli (for/while) o annotazioni di tipo, è meglio usare una funzione normale definita con def.
* **Chiarezza:** Se l'uso di una lambda rende il codice meno leggibile per chi lo legge (incluso il te stesso futuro!), preferisci una funzione def.
* **Documentazione:** Le funzioni def possono avere docstrings, il che è essenziale per codice ben documentato e manutenibile.

Spero che questi esempi ti abbiano dato un'idea chiara di come e quando usare le funzioni lambda in Python! Hai qualche scenario specifico in mente per cui vorresti vedere un esempio?