lez_05

January 21, 2026

1 Funzioni

In questa lezione ci occuperemo di funzioni.

Una definizione formale di funzione ci viene data da Wikipedia: >**Funzione**: Una funzione (detta anche routine, subroutine, procedura, sottoprogramma o metodo), in informatica e nell'ambito della programmazione, è un particolare costrutto sintattico di un determinato linguaggio di programmazione che permette di raggruppare, all'interno di un programma, una sequenza di istruzioni in un unico blocco, espletando così una specifica (e in generale più complessa) operazione, azione (o elaborazione) sui dati del programma stesso in modo tale che, a partire da determinati input, restituisca determinati output.

In soldoni, possiamo dire che una funzione è una raccolta di operazioni elementari che eseguite sequenzialmente portano a termine un compito ben specificato. Generalmente questo compito viene svolto lavorando su delle variabili per modificarle o produrre un output.

Abbiamo già utilizzato diverse funzioni nel corso delle lezioni, come ad esempio *print()*, *type()* o *input()*, e queste ricadono nella categoria delle funzioni **built-in**, ossia presenti di default nel linguaggio di programmazione Python.

Abbiamo anche visto come sia possibile importare funzioni da moduli o librerie esterne, con la keyword **import**, quando abbiamo importanto la funzione *randint()* dal modulo **random**, per simulare il lancio di un dado.

Il passo successivo è quello di utilizzare funzioni scritte da noi. Questo è comodo per suddividere un programma complesso in unità funzionali, che svolgano compiti ben determinati e che nel loro complesso portino a termine il compito che ci siamo prefissati. Questo porta dei vantaggi evidenti in termini di organizzazione, compattezza e leggibilità del codice.

Prima di vedere come si definiscono e si utilizzano le funzioni in python, un pò di **best practice**: 1. **Nomi delle funzioni**: - il nome di una funzione dovrebbe essere scritto in lettere minuscole; - se composto da più parole, queste dovrebbero essere separate da un underscore (**_**); - il nome della funzione dovrebbe ricordare un predicato, un azione, e dovrebbe riflettere lo scopo della funzione. 2.** Compito di una funzione**: - una funzione dovrebbe rispettare il principio di** single responsibility**. Se il nostro programma deve scaricare una pagina web, cercare un valore al suo interno e poi stamparlo, è consigliabile che ognuno di questi 3 compiti sia svolto da una funzione a sé stante. Questo migliora la leggibilità del codice e facilita la sua manutenzione. - Il compito di una funzione dovrebbe essere descritto nel modo più chiaro e conciso possibile nella** docstring**, vedi più avanti.

Al fine di poter utilizzare una funzione è necessario PRIMA **definirla** e POI **chiamarla**.

La sintassi per la definizione di una funzione è

```
def nome_funzione(parametri):     codice
```

Quindi utilizziamo la *keyword* **def** per definire una funzione.

Per chiamare una funzione basta scrivere il suo nome e fornirle gli opportuni parametri.

Vediamo ora una semplice definizione e chiamata di una funzione che prende in input un numero intero e ci restituisce il suo quadrato:

[3]:
```
def square(n):
    """funzione che restituisce il quadrato di un numero"""
    quadrato = n ** 2
    return(quadrato)

numero_al_quadrato = square(4)
print(numero_al_quadrato)
print(square.__doc__)
```

```
16
funzione che restituisce il quadrato di un numero
```

Analizziamo uno per uno gli elementi di questa funzione: - "def square(n):" stiamo definendo la funzione, usando la keyword **def**, specificando il suo nome (notare come il nome sia un verbo), e stabilendo quali argomenti siano necessari affinché funzioni. In questo caso è sufficiente un parametro, che sarà il numero da elevare al quadrato. I due punti stabiliscono che da quel momento in poi scriveremo quello che la funzione **farà**, ossia il codice che verrà eseguito quando la funzione verrà chiamata. Questo codice sarà rappresentato da tutte le righe indentate dopo la riga di definizione.

- la " " "docstring" " ": questa riga è la **docstring**. La docstring descrive il comportamento della funzione, ed è visualizzabile stampando *nome_funzione.___doc___. *Non è obbligatoria, e nel nostro caso è veramente superflua, ma per funzioni più complesse sarebbe buona norma inserirla. >**NOTA**: quando il nome di un metodo inizia e finisce con due caratteri underscore, significa che è un metodo* speciale*. Vedremo cosa significa quando vedremo la programmazione ad oggetti.
- corpo della funzione —-> nel nostro caso si tratta di una singola istruzione, ossia definire una variabile chiamata quadrato, e assegnarle il valore n^2. Ovviamente per funzioni più complesse il corpo della funzione si popolerà di molte più righe di codice
- il **return**: la keyword **return** stabilisce quale sarà il valore che la funzione ci restituirà in output. È "grazie" al return se siamo riusciti ad assegnare l'output della funzione alla variabile numero_al_quadrato

Vediamo ora una funzione leggermente più complessa, che prende in input due numeri e ci restituisce il più grande.

La scriveremo prima in una forma più prolissa, e poi in una forma più concisa

[4]:
```
def get_bigger(a, b):
    if a > b:
        return a
```

```
    else:
        return b

a = 2
b = 1
maggiore = get_bigger(a, b)
print(maggiore)
```

```
2
```

[5]:
```
def get_bigger_v2(a, b):
    return a if a > b else b
a = 1
b = 2
print(get_bigger_v2(a, b))
```

```
2
```

Come abbiamo visto (giusto per rimarcare l'ovvio) una funzione può prendere più di un parametro.

Tutti i parametri che abbiamo specificato nella definizione della funzione sono **obbligatori**, quindi se cerchiamo di chiamare la funzione passando un numero non sufficiente di parametri riceveremo un errore:

[6]:
```
get_bigger(a)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_6683/1749365761.py in <module>
----> 1 get_bigger(a)

TypeError: get_bigger() missing 1 required positional argument: 'b'
```

In questo caso non potevamo aspettarci nulla di diverso: affinché si possano confrontare dei numeri è indispensabile che ce ne siano almeno due.

Tuttavia python ci offre la possibilità di definire dei parametri **opzionali**. Per spiegare questo concetto supponiamo di voler scrivere una funzione che stampa i numeri da 0 ad n, dove n è un parametro fornito dall'utente. Tuttavia, se n non viene fornito, la funzione stamperà i numeri da 0 a 9 di default.

Per ottenere questo risultato passiamo l'argomento n nel modo seguente:

[8]:
```
def stampa_serie(n = 10):
    for i in range(n):
        print(i, end = '  ') #end ci permette di specificare il carattere␣
 ↪finale della stampa. Di default print va a capo
print('Stampa con parametro fornito:')
stampa_serie(5)
```

```
print('\nStampa senza parametro fornito: ')
stampa_serie()
```

```
Stampa con parametro fornito:
0  1  2  3  4
Stampa senza parametro fornito:
0  1  2  3  4  5  6  7  8  9
```

Quindi possiamo stabilire che un argomento sia opzionale assegnandogli un valore di default in fase di definizione della funzione. In questo caso non risolve nessun particolare problema poiché la funzione in sé non serve a molto, ma in vari casi questo è molto utile. > Diamo ad esempio un occhiata alla funzione *linakge()* definita nel modulo **cluster.hierarchy** della libreria **scipy**: > >`linkage(y, method='single', metric='euclidean', optimal_ordering=False)` > >I parametri di questa funzione definiscono in che modo sarà effettuato il cluster. L'utente può decidere se modificarne alcuni o tutti, ma se si limitasse a chiamare la funzione passando solo i dati da clusterizzare (y), la funzione lavorerebbe con i parametri definiti di default.

È importante ricordare che gli argomenti opzionali vanno **SEMPRE** passati **DOPO** gli argomenti obbligatori.

Supponiamo di voler definire una funzione che stabilisce qual'è il numero più grande non tra due valori, ma tra un insieme di valori. Per ottenere questo abbiamo due possibili strade. La prima e la più intuitiva è quella fornire alla funzione un vettore di numeri:

[9]:
```
def get_bigger_in_list(list):
    maggiore = list[0]
    for x in list:
        if x > maggiore: maggiore = x
    return maggiore

lista = [1, 2, 5, 134, 1, 5, 15, 194]
print(get_bigger_in_list(lista))
```

```
194
```

L'alternativa è quella di usare l'argomento ***args**.

*args è un argomento speciale che raccoglie tutti i parametri aggiuntivi che passiamo ad una funzione. Con aggiuntivi intendo argomenti che non sono stati "*previsti*" in fase di definizione della funzione. Per chiarire questo aspetto riscriviamo la funzione sopra utilizzando il parametro *args, e poi scriviamo una funzione che organizza una festa, perché siamo gente allegra!

[11]:
```
def get_bigger_with_args(*args):
    maggiore = args[0]
    for x in args:
        if x > maggiore: maggiore = x
    return maggiore

print(get_bigger_with_args(1, 2, 3, 145, 8))
```

```
145
```

[14]:
```
seq = ['A', 'T', 'C', 'G', 'T', 'G', 'A', 'G']
def count_g(*args):
    cont = 0
    for nuc in args:
        if nuc == 'G':
            cont += 1
    return(cont)
numero = 165
print(count_g('A', 'T', 'C', 'G', 'T', 'G', 'A', 'G', 'gatto', numero))
```

```
3
```

***args** ha raccolto tutti i numeri che abbiamo passato alla funzione in fase di chiamata, e li ha infilati in un vettore chiamato **args**. Successivamente abbiamo ciclato su quel vettore per trovare il numero più grande. Il vantaggio di questo approccio è che la funzione avrebbe funzionato anche con più o meno numeri.

Sostanzialmente non c'è una grande differenza tra passare un vettore di elementi o utilizzare *args, quindi la scelta su cosa utilizzare è lasciata allo sviluppatore.

Ma ci eravamo promessi una festa, ed è giusto farla:

[16]:
```
def throw_party(host, *invitati):
    print("{} sta dando una festa".format(host))
    for invitato in invitati:
        print("{} si è unito/a alla festa!".format(invitato))

throw_party('Lorenzo', 'Viviana', 'Dario', 'Adriano', 'Alessia', 'Paolo',␣
 ↪'Blasco')
```

```
Lorenzo sta dando una festa
Viviana si è unito/a alla festa!
Dario si è unito/a alla festa!
Adriano si è unito/a alla festa!
Alessia si è unito/a alla festa!
Paolo si è unito/a alla festa!
Blasco si è unito/a alla festa!
```

Ragioniamo sulla festa che abbiamo organizzato e facciamo alcune considerazioni: 1. Non è indispensabile utilizzare la parola *args*, quello che conta è l'asterisco. Infatti abbiamo sostituito **args* con **invitati*; 2. Se vogliamo utilizzare *args, dobbiamo farlo dopo le variabili obbligatorie e dopo quelle opzionali: va messo in fondo ai parametri; 3. Una funzione non deve necessariamente restituire un valore, infatti qui non c'è nessun return. Questo è tipico delle funzioni che stampano semplicemente delle informazioni a schermo; 4. Ci siamo divertiti :)

Un ultimo tipo di parametro che possiamo passare ad una funzione è ****kwargs**.

Questo è molto simile a *args, tuttavia invece di generare un vettore con i valori che gli vengono forniti, genera un dizionario. Il suo nome esteso è infatti **keyword-arguments**. Vediamolo in azione prima di commentarlo: la funzione seguente stampa i punteggi ottenuti da N giocatori in un videogioco.

[19]:
```
def get_scoreboard(game, **kwargs):
    print(game, 'scores:')
    for key in kwargs:
        print(key, kwargs[key])

get_scoreboard('Space Invaders', player_1 = 98, player_2 = 76, player_3 = 132)
```

```
Space Invaders scores:
player_1 98
player_2 76
player_3 132
```

Quello che fa kwargs, è raccogliere tutti i parametri passati dopo il primo parametro obbligatorio (game) e li raccoglie in un dizionario. Quindi nel nostro caso genera una struttura dati di questo tipo: `>kwargs = { > 'player_1' : 98, > 'player_2' : 76, > 'player_3' : 132}`

Dovremmo riconoscere questa struttura come un dizionario, e con questa informazione dovremmo capire cosa succede nel corpo della funzione. Se così non fosse, suggerisco di tornare alla lezione 4!

1.1 Esercizi

1. Scrivere una funzione che restituisca il fattoriale di un numero n dato come parametro.
2. Scrivere una funzione che restituisca un vettore contenente i numeri della serie di fibonacci partendo da 1 fino ad arrivare al primo numero maggiore del numero n passato come parametro. Se nessun parametro viene fornito, assume che n sia 50.
3. Scrivere un programma che chieda il nome all'utente, per poi inserirlo in una frase di benvenuto e stampare la frase. Suddividere questo compito in 3 funzioni separate e poi chiamarle nella corretta sequenza.
4. Scrivere lo stesso programma, sempre suddiviso in 3 funzioni, ma che dia il benvenuto a più persone, fornite separatamente, utilizzando *args

1.2 Soluzioni

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

1.3 Esercizio 1

[1]:
```
#prima soluzione:

def calc_factorial(n):
    factorial = n
    for i in range(1, n):
        factorial *= i
    return factorial

print(calc_factorial(4))
print(calc_factorial(5))

#seconda soluzione

def calc_factorial_v2(n):
    factorial = 1
    for i in range(n, 1, -1):
        factorial *= i
    return factorial

print(calc_factorial_v2(4))
print(calc_factorial_v2(5))
```

```
24
120
24
120
```

1.4 Esercizio 2

[21]:
```
def fibonacci(n = 100):
    i = 1
    j = 1
    fibonacci = 1
    lista_fib = [1, 1]
    while fibonacci < n:
        fibonacci = i + j
        lista_fib.append(fibonacci)
```

```
        i = j
        j = fibonacci
    return lista_fib
print(fibonacci(100))
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

1.5 Esercizio 3

[24]:
```
def get_name():
    return input("What's your name? ")

def make_welcome(name):
    return('Welcome {}!'.format(name))

def say_hello(sentence):
    print(sentence)

welcome = make_welcome(name)
say_hello(welcome)

#O in alternativa:

say_hello(make_welcome(get_name()))
```

```
Welcome Peppino di Capri!
```

```
What's your name?  Lorenzo
```

```
Welcome Lorenzo!
```

1.6 Esercizio 4

[55]:
```
def get_name_v2(*args):
    return list(args)

def make_welcome_v2(names):
    saluti = []
    for name in names:
        saluti.append('Welcome, ' + name)
    return saluti

def say_hello_v2(salutes):
    for salute in salutes:
        print(salute)

nomi = get_name_v2('Lorenzo', 'Blasco', 'Viviotty')
saluti = make_welcome_v2(nomi)
say_hello_v2(saluti)
```

```
Welcome, Lorenzo
Welcome, Blasco
Welcome, Viviotty
```