Dispensa · Modulo 5

Il modulo 5 introduce uno strumento nuovo: le **funzioni**. Le funzioni non servono a fare cose completamente diverse da quelle già viste — servono a organizzare il codice in modo più pulito. L'idea di fondo è scomporre un problema in sottoproblemi più piccoli, affidare ciascun sottoproblema a una funzione e poi combinare i risultati.

1. Funzioni in matematica

Le funzioni di Python sono direttamente collegate a quelle che abbiamo già incontrato in matematica. Per esempio:

$$\mathbf{f(x) = 2x + 1}$$

Questa funzione prende un numero `x` e restituisce un numero: il doppio di `x` più uno.

x	f(x) = 2x + 1
0	1
1	3
3	7
−2	−3
10	21

Scrivere `f(3)` vuol dire applicare la regola a `3` e ottenere `7`. Tre cose che già sappiamo:

- una funzione ha un **nome** (`f`);
- riceve uno o più **input** (`x`) e produce un **output** (`f(x)`);
- la regola è scritta una volta sola, ma si può applicare a qualunque valore.

In Python si scrive esattamente la stessa cosa:

```
def f(x):
    return 2 * x + 1

f(3)   # → 7
f(10)  # → 21
```

matematica	Python
nome della funzione	`def nome_funzione`
variabile `x`	parametro `x`
`f(x) = ...`	`return ...`
`f(3)`	`f(3)`

La differenza principale: in Python la funzione può contenere variabili, cicli e condizioni — non solo una formula. Ma la struttura è la stessa.

2. Perché servono le funzioni

Negli esercizi sulle liste abbiamo scritto più volte lo stesso blocco di codice per leggere una sequenza di numeri fino alla sentinella `0`. Ecco tre programmi completi degli esercizi precedenti:

```
# Es. 10 - stampa al contrario
numeri = []
n = int(input())
while n != 0:
    numeri.append(n)
    n = int(input())

i = len(numeri) - 1
while i >= 0:
    print(numeri[i])
    i = i - 1
```

```
# Es. 9 - trova il massimo
numeri = []
n = int(input())
while n != 0:
    numeri.append(n)
    n = int(input())

massimo = numeri[0]
i = 1
while i < len(numeri):
    if numeri[i] > massimo:
        massimo = numeri[i]
    i = i + 1
print(massimo)
```

```
# Es. 12 - media e valori sopra la media
numeri = []
n = int(input())
while n != 0:
    numeri.append(n)
    n = int(input())

somma = 0
i = 0
while i < len(numeri):
    somma = somma + numeri[i]
```

```
    i = i + 1
media = somma / len(numeri)
print("Media:", media)
```

Le prime cinque righe sono **identiche** in tutti e tre. Ogni programma fa una cosa diversa con la lista, ma la parte di lettura è sempre la stessa. Se domani vogliamo cambiare la sentinella da `0` a `-1`, o aggiungere un prompt all'`input()`, dobbiamo modificarle in ogni punto in cui compaiono — e rischiamo di dimenticarne uno.

Le funzioni risolvono questo problema: isoliamo quel blocco in un posto solo, gli diamo un nome, e lo chiamiamo ogni volta che serve.

```
def leggi_lista(sentinella):
    numeri = []
    n = int(input())
    while n != sentinella:
        numeri.append(n)
        n = int(input())
    return numeri
```

I tre programmi diventano:

```
# Es. 10 - stampa al contrario
numeri = leggi_lista(0)

i = len(numeri) - 1
while i >= 0:
    print(numeri[i])
    i = i - 1
```

```
# Es. 9 - trova il massimo
numeri = leggi_lista(0)

massimo = numeri[0]
i = 1
while i < len(numeri):
    if numeri[i] > massimo:
        massimo = numeri[i]
    i = i + 1
print(massimo)
```

```
# Es. 12 - media e valori sopra la media
numeri = leggi_lista(0)

somma = 0
i = 0
while i < len(numeri):
    somma = somma + numeri[i]
```

```
    i = i + 1
media = somma / len(numeri)
print("Media:", media)
```

La logica di lettura esiste ora in un posto solo. Se la modifichiamo, cambia ovunque. Lo stesso ragionamento vale per qualsiasi blocco che compare più volte: contare le vocali, trovare il minimo, verificare se una stringa è palindroma. Ogni volta che ci si sorprende a copiare e incollare le stesse righe, è il momento di scrivere una funzione.

Le funzioni ci aiutano a dividere un problema grande in problemi più piccoli e più affrontabili – e che magari abbiamo già risolto!

Pensiamo alla costruzione di una casa. Non facciamo di certo tutto da soli: chiamiamo persone diverse, ognuna esperta nel suo ambito. Chiamiamo l'elettricista e gli diciamo: "ho bisogno di un impianto per una casa da 80 mq con tre stanze". L'elettricista fa il suo lavoro e ci consegna il risultato. E soprattutto non ci interessa "come" lo fa, ci interessa che date le nostre richieste (80mq, 3 stanze) ci restituisca un progetto adeguato. Se poi costruiamo una seconda casa, richiamiamo lo stesso elettricista — questa volta per 120 mq e cinque stanze. Stesso mestiere, parametri diversi.

Abbiamo delegato un pezzo del lavoro a qualcuno di esperto, specializzato nello svolgere quella operazione, di cui non dobbiamo più preoccuparci.

In programmazione funziona esattamente così:

- la **funzione** è l'elettricista — sa fare un lavoro preciso;
- i **parametri** sono le istruzioni che gli diamo ogni volta — cambiano da chiamata a chiamata;
- il **valore di ritorno** è il risultato che ti consegna.

Usare una funzione significa trattare un pezzo del problema un po' come una **scatola nera**: sappiamo cosa riceve in input e cosa restituisce in output, e ci fidiamo del risultato che otteniamo senza doverci preoccupare di come scomporlo in istruzioni più semplici, come avremmo invece fatto se avessimo scritto noi l'intero codice (o costruito l'intera casa).

2. Definizione e chiamata

Con le funzioni bisogna distinguere due momenti completamente diversi: la **definizione** e la **chiamata**.

Torniamo alla metafora dell'elettricista. Prima di poterlo chiamare, salviamo il suo numero in rubrica — e annotiamo anche quali informazioni dobbiamo dargli (metri quadri, numero di stanze) perché senza quelle non può fare nulla. Questo è il momento della **definizione**: descriviamo chi è la funzione, come si chiama e di cosa ha bisogno per lavorare. Ma finché non la chiamiamo, non succede nulla di concreto.

Il momento della **chiamata** è quando prendiamo il telefono, componiamo il numero e gli forniamo le informazioni specifiche: "80 mq, tre stanze". Solo allora l'elettricista inizia a lavorare e alla fine ci consegna il risultato.

La definizione è un'istruzione. Quando Python incontra la definizione, salva il codice della funzione in memoria — ma non lo esegue ancora. La funzione esiste, ma non ha ancora calcolato niente.

La chiamata (o invocazione) è un'espressione. Quando scriviamo ad esempio `area_rettangolo(4, 6)`, Python esegue il corpo della funzione e produce un valore. Quel valore ha un tipo — in questo caso `int` — e può essere usato ovunque si può usare un'espressione: in un `print`, in un assegnamento, in una condizione.

```
risultato = area_rettangolo(4, 6)    # assegnamento
print(area_rettangolo(3, 5))         # dentro print
if area_rettangolo(2, 2) > 3:        # in una condizione
    print("grande")
```

3. Sintassi della definizione

```
def nome_funzione(parametro_1, parametro_2, ..., parametro_n):

    # istruzioni

    return espressione
```

Parti da riconoscere:

- `def` introduce la definizione della funzione;
- il **nome della funzione** segue le stesse regole dei nomi di variabile: lettere, cifre e `_`; non può iniziare con una cifra; non può essere una parola chiave di Python;
- i **parametri** sono nomi di variabile a tutti gli effetti — valgono le stesse regole; non possono essere numeri, stringhe o espressioni;
- i **due punti** `:` aprono il blocco;
- `return` valuta l'espressione che segue, interrompe il corpo della funzione e restituisce quel valore al chiamante.

Esempio.

```
def area_rettangolo(base, altezza):
    area = base * altezza
    return area
```

Questa definizione non calcola ancora nessuna area specifica. Dice soltanto che esiste una funzione chiamata `area_rettangolo`, che prende due parametri e restituisce il loro prodotto.

Conviene accompagnare ogni funzione con un breve commento che dica:

- che cosa fa la funzione;
- che tipo di dati si aspetta in input;
- che cosa restituisce.

```
# Calcola l'area di un rettangolo.
# base, altezza: int o float - le dimensioni del rettangolo
# restituisce: int o float - il prodotto base × altezza
def area_rettangolo(base, altezza):
    area = base * altezza
    return area
```

6. Sintassi della chiamata di funzione

```
nome_funzione(espressione_1, espressione_2, ..., espressione_n)
```

Nella chiamata si passano **espressioni**: valori, variabili, calcoli, o persino un'altra chiamata di funzione — qualunque cosa che Python sappia ridurre a un valore. Ciò che conta è il valore prodotto, non come è scritto.

```
area_rettangolo(4, 6)              # valori letterali
area_rettangolo(base, altezza)     # variabili
area_rettangolo(b * 2, h + 1)      # espressioni aritmetiche
area_rettangolo(successore(7), 5)  # risultato di un'altra funzione
```

Python valuta ogni espressione prima di entrare nella funzione, e passa il valore risultante al parametro corrispondente. Il parametro riceve sempre un valore, indipendentemente da come era scritto fuori.

Il numero di espressioni (detti **argomenti**) nella chiamata deve corrispondere esattamente al numero di parametri nella definizione.

7. Semantica della chiamata di funzione

Quando Python incontra una chiamata di funzione:

1. valuta ciascun argomento **da sinistra a destra**;
2. assegna ogni valore al parametro corrispondente **per posizione**:

```
chiamata:        area_rettangolo( 4  ,    6   )
                                  |       |
                                  ↓       ↓
definizione:  def area_rettangolo(base, altezza):
```

3. esegue il corpo della funzione con quei valori;
4. quando incontra `return`, restituisce il valore e interrompe il corpo;
5. la chiamata nel programma principale si "sostituisce" con il valore restituito.

Esempio con argomenti composti.

Se esiste una funzione `successore(x)` che restituisce `x + 1`, allora:

```
area_rettangolo(successore(7), 5)
```

si valuta così: prima `successore(7)` produce `8`; poi la chiamata diventa `area_rettangolo(8, 5)`, che produce `40`.

8. Tracciare lo stato con le funzioni

Con le funzioni lo stato si divide in **due parti**: la memoria del programma principale e la memoria locale della funzione, che esiste solo durante la chiamata.

Esempio.

```
def triplo_del_successore(x):
    successore = x + 1
    triplo = 3 * successore
    return triplo

n = 15
risultato = triplo_del_successore(n)
print(risultato)
```

passo	riga	stato principale	stato funzione	output
1	`n = 15`	n→15	—	
2	`triplo_del_successore(n)` — chiamata	n→15	x→15	
3	`successore = x + 1`	n→15	x→15, successore→16	
4	`triplo = 3 * successore`	n→15	x→15, successore→16, triplo→48	
5	`return triplo`	n→15	→ restituisce 48, memoria locale chiusa	
6	`risultato = 48`	n→15, risultato→48	—	
7	`print(risultato)`	n→15, risultato→48	—	48

Tre cose da notare:

- al passo 2 la memoria locale si **apre**: `x` riceve il valore di `n`;
- al passo 5 la memoria locale si **chiude**: `successore` e `triplo` spariscono; solo il valore di ritorno passa al chiamante;
- il programma principale non vede mai `successore` né `triplo`: esistono solo dentro la funzione.

9. Scope

Lo **scope** (o ambito) di una variabile è la regione di codice in cui quel nome è definito e può essere usato. Dire che una variabile "è nello scope" significa che Python conosce quel nome e può restituirne il valore; dire che "è fuori scope" significa che il nome non esiste in quel punto — usarlo produce un `NameError`.

In Python ogni funzione ha il proprio **scope locale**: le variabili create dentro la funzione — inclusi i parametri — nascono nel momento in cui la funzione viene chiamata e vengono distrutte nel momento in cui la funzione restituisce il risultato con `return`. Il **programma principale** ha il proprio **scope globale**, che dura per tutta l'esecuzione del programma.

Le due memorie sono **separate e indipendenti**: una variabile locale non è visibile fuori dalla funzione, e una variabile globale non entra automaticamente dentro la funzione.

Questo ha tre conseguenze importanti.

9.1 Le variabili locali non sono visibili fuori dalla funzione

```
def calcola():
    risultato = 42
    return risultato

print(risultato)   # NameError: risultato non esiste qui
```

`risultato` esiste solo dentro `calcola`. Fuori dalla funzione, quel nome non è definito.

9.2 Le variabili esterne non vanno usate direttamente dentro la funzione

```
x = 10

def raddoppia():
    return x * 2     # x non è un parametro: viene presa dallo scope globale

print(raddoppia())   # stampa 20
```

Questo codice funziona, ma è fragile: la funzione dipende in modo nascosto dalla variabile `x` definita fuori. Se `x` cambia, o se la funzione viene usata in un altro programma dove `x` non esiste, il risultato sarà sbagliato o il programma andrà in errore.

La regola pratica è: **tutto ciò di cui la funzione ha bisogno deve entrare come parametro**.

```python
def raddoppia(x):   # corretto: x è un parametro, il valore entra dalla chiamata
    return x * 2

print(raddoppia(10))   # stampa 20
print(raddoppia(7))    # stampa 14
```

Pensate a un muro con una porta tra il programma principale e la funzione: i valori possono passare solo attraverso i parametri (entrando dalla porta) e il valore di ritorno (uscendo dalla porta).

9.3 Stesso nome, variabili diverse

Se una variabile nel programma principale e una variabile dentro la funzione hanno lo stesso nome, sono due variabili distinte che non si influenzano.

```python
def incrementa(x):
    x = x + 1       # questa x è locale alla funzione
    return x

x = 5               # questa x è nel programma principale
y = incrementa(x)
print(x)            # 5 - x nel programma principale non è cambiata
print(y)            # 6
```

passo	riga	stato principale	stato funzione
1	`x = 5`	x=5	—
2	`incrementa(x)` — chiamata	x=5	x=5 (copia)
3	`x = x + 1`	x=5	x=6
4	`return x`	x=5	→ restituisce 6, memoria locale chiusa
5	`y = 6`	x=5, y=6	—

La `x` della funzione è una copia del valore passato, non la stessa variabile. Modificarla dentro la funzione non tocca la `x` del programma principale.

È come avere due persone con lo stesso nome che abitano in città diverse: se una si tinge i capelli, l'altra non cambia.