Dispensa · Modulo 6

Il modulo 6 approfondisce come Python gestisce la memoria, e in particolare cosa succede davvero quando scriviamo `x = 42` o passiamo una lista a una funzione. Capire il modello di memoria è fondamentale per evitare errori sottili — specialmente quando si lavora con le liste — e per distinguere due categorie di funzioni che si comportano in modo molto diverso.

1. Le variabili non sono barattoli

Fino ad ora abbiamo spesso immaginato le variabili come **barattoli** (o scatole): si mette un valore dentro, poi si può cambiare il contenuto del barattolo.

Questa immagine è comoda, ma non corrisponde a come funziona Python. In Python le variabili sono **etichette**: un nome che *punta* a un oggetto che esiste in memoria.

```
x = 42
```

Questa istruzione non "mette 42 nel barattolo x". Crea l'oggetto `42` in memoria e attacca a quell'oggetto l'etichetta `x`.

La funzione `id()` restituisce un identificatore univoco che Python assegna a ogni oggetto. Ci permette di osservare direttamente questo meccanismo:

```
x = 42
print(id(x))    # es. 140234567890
```

L'identificatore non cambia finché l'oggetto esiste in memoria e continuiamo a puntarci con la stessa etichetta.

2. Due etichette sullo stesso oggetto

Cosa succede quando scriviamo `y = x`?

```
x = 42
y = x
print(id(x))    # es. 140234567890
print(id(y))    # stesso numero!
```

`y = x` **non** crea una copia del valore. Crea una seconda etichetta che punta allo *stesso oggetto* in memoria. In questo momento `x` e `y` sono due nomi diversi per la stessa cosa.

3. Tipi immutabili: spostare l'etichetta

Proviamo a modificare `x`:

```
x = 42
print(id(x))    # es. 140234567890
y = x
```

```
print(id(y))    # stesso numero!
x = x + 1
print(x)        # 43
print(y)        # 42
print(id(x))    # diverso da prima!
print(id(y))    # uguale a prima!
```

Perché `y` rimane `42`? Perché `int`, `float`, `str` e `bool` sono tipi **immutabili**. L'oggetto `42` non può essere modificato dall'interno — è come una pallina del lotto: non puoi cambiarle il numero stampato sopra.

Scrivere `x = x + 1` non modifica l'oggetto `42`. Crea un nuovo oggetto `43` e *sposta* l'etichetta `x` su di esso. L'etichetta `y` resta dov'era, a puntare a `42`.

Regola: con i tipi immutabili, qualunque operazione che "cambia il valore" in realtà sposta l'etichetta su un nuovo oggetto. Le altre etichette rimangono invariate.

4. Tipi mutabili: modificare senza spostare

Le **liste** si comportano diversamente. Sono tipi **mutabili**: è possibile modificarle dall'interno senza creare un nuovo oggetto.

```
a = [1, 2, 3]
print(id(a))    # es. 140234111222

a.append(4)
print(a)        # [1, 2, 3, 4]
print(id(a))    # stesso numero di prima!
```

Metodi come `append()`, `sort()`, `reverse()`, `pop()` e l'istruzione `del` modificano la lista *in-place*: l'oggetto lista in memoria viene alterato, ma l'etichetta non si sposta. L'identificatore rimane uguale.

Regola: con le liste, le operazioni in-place cambiano il contenuto dell'oggetto. L'etichetta non si sposta. L'`id` non cambia.

5. Alias: due nomi per la stessa lista

```
a = [1, 2, 3]
b = a
```

Come con gli interi, `b = a` crea una seconda etichetta che punta alla *stessa lista*. `a` e `b` sono due nomi — due **alias** — dello stesso oggetto.

La conseguenza è importante:

```
a = [1, 2, 3]
b = a
b.append(4)
```

```
print(a)    # [1, 2, 3, 4]
print(b)    # [1, 2, 3, 4]
```

Modificare la lista tramite `b` è identico a modificarla tramite `a`, perché sono la stessa lista. Questo si chiama **aliasing**.

Con gli interi non era un problema: l'unico modo per "modificare" `x` è spostare l'etichetta su un nuovo oggetto, e le altre etichette non si spostano. Con le liste invece le modifiche in-place si propagano a tutti gli alias.

6. Copiare una lista

Per ottenere una lista indipendente bisogna copiare elemento per elemento:

```
a = [1, 2, 3]
copia = []
i = 0
while i < len(a):
    copia.append(a[i])
    i = i + 1

copia.append(4)
print(a)      # [1, 2, 3]   - invariata
print(copia)  # [1, 2, 3, 4]
```

Dopo la copia, `a` e `copia` puntano a oggetti distinti: modificare uno non influisce sull'altro.

7. Identità e uguaglianza

Python mette a disposizione due operatori di confronto distinti:

Operatore	Cosa controlla	Domanda
`==`	uguaglianza di valore	I contenuti sono uguali?
`is`	identità	Sono lo stesso oggetto in memoria?

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]   # lista separata, stesso contenuto

print(a == b)   # True  - stessi valori
print(a is b)   # True  - stesso oggetto

print(a == c)   # True  - stessi valori
print(a is c)   # False - oggetti diversi
```

`is` è equivalente a controllare `id(a) == id(b)`. Di solito nei programmi si usa `==`; `is` serve quando vogliamo verificare esplicitamente se due variabili puntano allo stesso oggetto.

8. Tracciamento della memoria

Per analizzare il comportamento di un programma passo per passo usiamo la solita **tabella di tracciamento**.

La colonna **Stato del programma** però deve registrare se due variabili puntano allo stesso oggetto in modo da tenere traccia delle modifiche che avvengono tramite alias.

Esempio — aliasing con una lista:

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)
print(b)
```

Passo	Codice sorgente	Stato del programma	Output
1	a = [1, 2, 3]	a->[1,2,3]	
2	b = a	a,b->[1,2,3]	
3	b.append(4)	a,b->[1,2,3,4]	
4	print(a)		[1,2,3,4]
5	print(b)		[1,2,3,4]

Al passo 2 `a` e `b` compaiono insieme perché puntano allo stesso oggetto. Al passo 3 la modifica tramite `b` è visibile anche da `a`.

Esempio — riassegnazione con tipi immutabili:

```
x = 5
y = x
y = y + 3
print(x)
print(y)
```

Passo	Codice sorgente	Stato del programma	Output
1	x = 5	x->5	
2	y = x	x,y->5	
3	y = y + 3	x->5, y->8	
4	print(x)		5
5	print(y)		8

Al passo 3 `y = y + 3` crea un nuovo oggetto `8` e sposta solo l'etichetta `y`. `x` rimane ancorata al vecchio oggetto `5`.

9. Passaggio degli argomenti

Quando chiamiamo una funzione, il parametro parte come un **alias** dell'argomento. Per tracciare programmi con funzioni si aggiungono due colonne separate per lo scope globale e quello della funzione:

Passo	Codice	Stato - globale	Stato - funzione	Output

Nella colonna funzione, un parametro che fa alias dell'argomento si scrive `parametro->variabile_globale->valore`.

Con tipi immutabili

```
def incrementa(n):
    n = n + 1
    print("dentro:", n)
    return n

x = 10
x = incrementa(x)
print("fuori:", x)
```

Passo	Codice	Stato - globale	Stato - funzione	Output
	x = 10	x->10		
	x = incrementa(x)	x->10	n->x->10	
	n = n + 1	x->10	n->11 (x non è più collegata!)	
	print("dentro:", n)	x->10	n->11	dentro: 11
	return n	x->11	-	
	print("fuori:", x)	x->11		fuori: 11

`n = n + 1` sposta solo l'etichetta `n` su un nuovo oggetto; `x` rimane invariata.

Con tipi mutabili

```
def aggiungi(lista, elemento):
    lista.append(elemento)
```

```
numeri = [1, 2, 3]
aggiungi(numeri, 4)
print(numeri)
```

Passo	Codice	Stato - globale	Stato - funzione	Output
	numeri = [1, 2, 3]	numeri->[1,2,3]		
	aggiungi(numeri, 4)	numeri->[1,2,3]	lista->numeri->[1,2,3], elemento->4	
	lista.append(elemento)	numeri->[1,2,3,4]	lista->numeri->[1,2,3,4], elemento->4	
	(fine funzione)	numeri->[1,2,3,4]	-	
	print(numeri)			[1,2,3,4]

`lista.append(elemento)` modifica in-place l'oggetto condiviso: la modifica avviene su `numeri` anche durante l'esecuzione della funzione.

10. Effetti collaterali

Un **effetto collaterale** è qualsiasi modifica che una funzione produce al di fuori del suo scope locale. I casi più comuni:

- modificare una lista (o altro oggetto mutabile) passato come argomento
- modificare una variabile globale
- stampare qualcosa a schermo
- scrivere su un file

Gli effetti collaterali non sono sbagliati in sé — a volte sono proprio lo scopo della funzione — ma è importante esserne consapevoli. Una funzione con effetti collaterali *nascosti* è una fonte comune di bug.

Esempio: la funzione seguente produce un effetto collaterale inaspettato:

```
def trova_minimo(dati):
    dati.sort()          # modifica la lista originale!
    minimo = dati[0]
    return minimo

voti = [7, 3, 9, 5]
m = trova_minimo(voti)
print(voti)   # [3, 5, 7, 9] - era [7, 3, 9, 5]
```

Chi chiama `trova_minimo` potrebbe non aspettarsi che i dati vengano riordinati.

11. Funzioni pure

Una **funzione pura** è una funzione che:

1. prende input (parametri)
2. restituisce output (con `return`)
3. **non produce effetti collaterali**

Le funzioni pure sono facili da testare e da riutilizzare. Dato un certo input producono sempre lo stesso output, senza dipendere da — o modificare — nulla all'esterno.

```
def raddoppia(lista):
    risultato = []
    i = 0
    while i < len(lista):
        risultato.append(lista[i] * 2)
        i = i + 1
    return risultato

numeri = [1, 2, 3]
doppi = raddoppia(numeri)
print(numeri)   # [1, 2, 3]  - invariata
print(doppi)    # [2, 4, 6]
```

`raddoppia` non tocca la lista originale: costruisce e restituisce una lista nuova.

12. Funzioni void

Una **funzione void** è una funzione che:

1. non usa `return`
2. opera **in-place** su un oggetto mutabile passato come argomento

Il risultato non viene restituito: viene scritto direttamente nell'oggetto condiviso.

```
def raddoppia_in_place(lista):
    i = 0
    while i < len(lista):
        lista[i] = lista[i] * 2
        i = i + 1

numeri = [1, 2, 3]
raddoppia_in_place(numeri)
print(numeri)   # [2, 4, 6]
```

Notare che non scriviamo `numeri = raddoppia_in_place(numeri)`: la funzione non restituisce nulla, e la lista è già stata modificata in-place.

Espressione o istruzione?

La distinzione tra funzione pura e void si riflette anche su come la chiamata va usata nel codice:

- La chiamata a una **funzione pura** è un'**espressione**: produce un valore con un tipo. Si può assegnare a una variabile, usare come guardia di un `if`, come argomento di un'altra funzione, ecc.

```
doppi = raddoppia(numeri)          # assegnazione - OK
if raddoppia(numeri) == [...]:     # guardia - OK
```

- La chiamata a una **funzione void** è un'**istruzione**: non produce nessun valore. Si usa da sola, come si usano `print` o un assegnamento.

```
raddoppia_in_place(numeri)           # istruzione - OK
x = raddoppia_in_place(numeri)       # x varrebbe None - errore concettuale
if raddoppia_in_place(numeri):       # non ha senso - non c'è valore
```

Confronto: pura vs void

Caratteristica	Funzione pura	Funzione void
Usa `return`	Sì	No
Modifica l'argomento	No	Sì
La chiamata è	un'espressione	un'istruzione
Come si usa	`y = f(x)`	`f(x)`
La lista originale	Rimane invariata	Viene modificata

Quando scegliere l'una o l'altra

La scelta dipende dal requisito: bisogna chiedersi se vogliamo **modificare direttamente l'originale** oppure **ottenere una versione nuova** senza toccarlo.

Voglio…	Tipo di funzione
Calcolare un valore	pura
Produrre una nuova lista	pura
Modificare la lista in-place	void
Solo stampare (effetto visibile)	void

Una nota sulle funzioni ibride

In Python nulla vieta di avere funzioni che *sia* modificano un parametro *sia* restituiscono un valore. Un esempio della libreria standard è `pop()`: toglie l'ultimo elemento dalla lista (effetto collaterale) e lo restituisce.

```
numeri = [10, 20, 30]
ultimo = numeri.pop()
print(ultimo)   # 30
print(numeri)   # [10, 20]
```

`pop()` si comporta in modo "ibrido": non è né pura né void nel senso stretto. Per i primi programmi è utile tenere le due categorie separate, per rendere il comportamento del codice più prevedibile.

13. Import e librerie

Man mano che i programmi crescono è utile raccogliere le funzioni riutilizzabili in un file separato e importarle dove servono. Supponiamo di avere un file `utils.py` con funzioni sulle liste:

```
# utils.py
def somma_lista(lista):
    ...

def inverti_lista(lista):
    ...
```

Nel file principale si importa il modulo e si chiamano le sue funzioni con il prefisso:

```
# main.py
import utils

dati = [3, 1, 4, 1, 5]
totale = utils.somma_lista(dati)
```

Il prefisso (`utils.`) serve a distinguere funzioni con lo stesso nome che potrebbero venire da moduli diversi. Per importare, basta che i due file si trovino nella stessa cartella; si usa il nome del file senza l'estensione `.py`.

Python mette a disposizione anche molte **librerie standard** già installate. Non vengono caricate automaticamente per non rallentare l'avvio: vanno importate esplicitamente. Un esempio è `math`:

```
import math

print(math.sqrt(16))   # 4.0
print(math.pi)         # 3.1415926535897793
```

In alternativa si può importare direttamente il nome che serve:

```
from math import sqrt

print(sqrt(16))   # 4.0
```

Il modulo `random`

Il modulo `random` genera numeri pseudo-casuali, utili per simulazioni, giochi e test.

```
import random

random.randint(1, 6)    # numero intero tra 1 e 6 (inclusi) - funzione pura
random.random()         # float tra 0.0 e 1.0 (escluso 1.0) - funzione pura
random.shuffle(lista)   # rimescola la lista in-place - funzione void
```

`random.randint` e `random.random` sono funzioni **pure**: restituiscono un valore e non modificano nulla. `random.shuffle` è invece **void**: modifica la lista in-place senza restituire nulla, come se si mischiasse un mazzo di carte.

Esempio: simulare il lancio di un dado dieci volte e stampare i risultati:

```
import random

i = 0
while i < 10:
    lancio = random.randint(1, 6)
    print(lancio)
    i = i + 1
```

Esempio: estrarre cinque numeri del lotto (da 1 a 90) senza ripetizioni:

```
import random

estratti = []
while len(estratti) < 5:
    n = random.randint(1, 90)
    if n not in estratti:
        estratti.append(n)

print(estratti)
```

Riepilogo

Concetto	Sintesi
Variabile	Etichetta che punta a un oggetto in memoria
`id(x)`	Identificatore univoco dell'oggetto puntato da `x`
Tipo immutabile	Operazioni di "modifica" creano un nuovo oggetto e spostano l'etichetta
Tipo mutabile	Operazioni in-place modificano l'oggetto esistente; `id` non cambia

Concetto	Sintesi
Alias	Due etichette che puntano allo stesso oggetto
`==`	Confronto di valore (stesso contenuto?)
`is`	Confronto di identità (stesso oggetto?)
Passaggio argomenti	Il parametro parte come alias dell'argomento
Effetto collaterale	Modifica prodotta fuori dallo scope locale della funzione
Funzione pura	Riceve input, restituisce output, nessun effetto collaterale; chiamata = espressione
Funzione void	Opera in-place su un mutabile; non usa `return`; chiamata = istruzione
`import modulo`	Rende disponibili le funzioni di una libreria (standard o file custom)
`random.shuffle`	Funzione void del modulo random: rimescola una lista in-place