Dispensa · Modulo 4

Abbiamo introdotto il ciclo `while` come strumento per ripetere un blocco di istruzioni finché una condizione rimane vera.

Adesso riflettiamo meglio su come funzionano queste iterazioni: invece di vedere ogni esercizio con il `while` come un caso a sé, cerchiamo diriconoscere cinque schemi ricorrenti — i paradigmi — che permettono di leggere un problema e capire subito quale struttura serve. Introduciamo poi un nuovo tipo di dato, la lista, che risolve situazioni in cui nessuno dei paradigmi del `while` è sufficiente da solo.

1. Ripasso: tipi, espressioni e operatori

Fino a qui abbiamo visto quattro tipi fondamentali di valori: `int` (interi), `float` (decimali), `str` (stringhe di testo) e `bool` (valori booleani `True` e `False`). Valori come `42`, `3.14`, `"7"` e `True` appartengono a tipi diversi; in particolare, `7` e `"7"` sembrano simili ma per Python non lo sono affatto: il primo è un intero, il secondo una stringa.

Alcune osservazioni importanti:

- `10 == 10.0` è `True`: Python confronta il valore, non il tipo;
- `"mela" < "pera"` è `True`: le stringhe si confrontano in ordine lessicografico;
- `=` è l'assegnamento, `==` è il confronto — i due simboli svolgono funzioni completamente diverse.

L'operatore modulo `%` calcola il resto della divisione intera: `137 % 12` vale `5`, perché `137 = 12 × 11 + 5`. Il suo uso più comune è verificare la parità: `n % 2 == 0` è `True` se e solo se `n` è pari.

La divisione intera `//` restituisce un intero: `10 // 3` vale `3`. Va usata ogni volta che si ha bisogno di un indice intero, per esempio per trovare il carattere centrale di una stringa:

```
s = "programmazione"
s[len(s) // 2]
```

Se si usasse `/` al posto di `//`, Python produrrebbe un `float`, che non è un indice valido.

F-string

Quando si vuole incorporare il valore di una variabile in una stringa di testo, le f-string rendono il codice più leggibile rispetto alla concatenazione con `+`:

```
nome = "Alice"
eta = 20
print(f"Il nome è {nome} e l'età è {eta}")
```

Si può anche formattare il numero di cifre decimali:

```
x = 10
y = 3
print(f"{x} diviso {y} vale circa {x / y:.2f}")
```

che stampa `10 diviso 3 vale circa 3.33`.

2. Tracciare lo stato di un ciclo `while`

Prima di introdurre i paradigmi ripassiamo il metodo che permette di "leggere" un ciclo senza eseguirlo a memoria. Si tratta di costruire la **tabella di stato**: una riga per ogni istruzione eseguita, con il valore di tutte le variabili rilevanti in quel momento.

Esempio: somma dei numeri da `1` a `n` (inclusi), con `n = 3`.

```
n = int(input("Inserisci un numero: "))
totale = 0
i = 1
while i <= n:
    totale = totale + i
    i = i + 1
print(totale)
```

Passo	Istruzione	`n`	`totale`	`i`	Guardia `i <= n`	Output
1	`n = 3`	3	—	—		
2	`totale = 0`	3	0	—		
3	`i = 1`	3	0	1		
4	`while i <= n:`, controllo guardia	3	0	1	True	
5	`totale = 0 + 1`	3	1	1		
6	`i = 1 + 1`	3	1	2		
7	`while i <= n:`, controllo guardia	3	1	2	True	
8	`totale = 1 + 2`	3	3	2		
9	`i = 2 + 1`	3	3	3		
10	`while i <= n:`, controllo guardia	3	3	3	True	
11	`totale = 3 + 3`	3	6	3		
12	`i = 3 + 1`	3	6	4		
13	`while i <= n:`, controllo guardia	3	6	4	**False** → esce	
14	`print(totale)`	3	6	4		→ 6

Tre cose da notare:

- la guardia viene rivalutata **all'inizio di ogni giro**, non solo la prima volta;
- se si dimentica `i = i + 1`, la guardia resta sempre vera e il ciclo non finisce mai;
- il valore stampato è il risultato **dopo** che il ciclo è terminato, non durante, perchè è indentato fuori dal blocco di istruzioni che vengono eseguite all'interno dell'iterazione `while`.

3. Cinque paradigmi del `while`

Negli esercizi fatti fino a questo momento abbiamo notato che la guardia del ciclo `while` può esprimere tipi di condizioni piuttosto diverse. Quando si legge un nuovo problema che richiede un ciclo, la prima domanda da porsi quindi è: qual è la condizione di terminazione dell'iterazione? Ci può aiutare fare riferimento a questi cinque schemi ricorrenti.

3.1 Contatore

Stai leggendo un libro di 300 pagine e vuoi sapere quante iniziano con la lettera "A" maiuscola. Sai già che dovrai scorrere tutte e 300 le pagine: non puoi fermarti prima, perché non sai dove si trovano le pagine che ti interessano. Ogni volta che ne incontri una che inizia con "A", aggiorni un contatore. Quando arrivi all'ultima pagina, il contatore contiene la risposta.

Quando si usa. Il numero di iterazioni è noto in anticipo: il testo dice "da 1 a 10", "cinque volte", "per ogni numero nell'intervallo tra a e b".

Struttura.

```
i = 1                # inizializzazione
while i <= 10:       # guardia: non abbiamo ancora raggiunto il limite
    print(i)         # azione
    i = i + 1        # aggiornamento - senza questa riga il ciclo non termina mai
```

Esempio: stampare i numeri pari fino a n (escluso). Sappiamo che dobbiamo eseguire `n` passi, partendo da `0` fino a `n-1`. L'azione da eseguire è poco più complessa della precedente, perchè va eseguita solo una volta su due.

```
n = int(input("Inserisci un numero: "))
i = 0                # inizializzazione
while i < n:         # guardia: non abbiamo ancora raggiunto il limite
    if i % 2 == 0:   # azione
        print(i)     # azione
    i = i + 1        # aggiornamento - senza questa riga il ciclo non termina mai
```

Esempio: fattoriale. Il fattoriale di `n` (in matematica si scrive `n!`) è il prodotto `1 × 2 × … × n`. L'accumulatore di prodotto parte da `1` (e non da `0`, perché qualunque numero moltiplicato per `0` darebbe `0`):

```
n = int(input("n: "))
fattoriale = 1
i = 1
while i <= n:
    fattoriale = fattoriale * i
    i = i + 1
print(fattoriale)
```

Con `n = 4`, il ciclo esegue quattro giri:

Giro	`i`	`fattoriale = fattoriale * i`	`i` dopo
1	1	$1 \times 1 = 1$	2
2	2	$1 \times 2 = 2$	3
3	3	$2 \times 3 = 6$	4
4	4	$6 \times 4 = 24$	5
stop	5	$5 > 4$	—

Due regole pratiche da ricordare sempre:

1. **L'ordine delle istruzioni dentro il ciclo conta.** Nel fattoriale `fattoriale` va aggiornato **prima** di incrementare `i`. Se si invertissero le due righe, al primo giro il contatore salterebbe a `2` e si perderebbe il fattore `1`. In altri esercizi un cambio di ordine produce errori ancora più evidenti.
2. **Il valore iniziale dell'accumulatore dipende dall'operazione.** Per le somme si parte da `0` (perché `x + 0 = x`); per i prodotti si parte da `1` (perché `x * 0 = 0` e azzerebbe tutto).

Contatore su stringa. Lo stesso schema si usa per scorrere i caratteri di una stringa, dove `i` diventa un indice di posizione. Anche nel caso di una stringa infatti sappiamo quante iterazioni dobbiamo effettuare: tante quante le lettere contenute nella stringa!

Esempio: stampare il numero di vocali presenti in una parola.

```
parola = input("Parola: ")
vocali = 0
i = 0
while i < len(parola):
    if parola[i] in "aeiou":
        vocali = vocali + 1
    i = i + 1
print(vocali)
```

3.2 Soglia su accumulatore

Sei al supermercato con 50 euro in tasca. Prendi un prodotto alla volta e sommi i prezzi man mano che li metti nel carrello. Non sai quanti prodotti riuscirai a comprare: dipende da cosa scegli. Ti fermi quando la somma supera i 50 euro — il numero di prodotti non era noto in anticipo, ed è la soglia a decidere quando smettere.

Quando si usa. Non si conosce in anticipo il numero di iterazioni. C'è una quantità (somma, prodotto, conteggio) che cresce o diminuisce a ogni passo, e il ciclo continua finché quella quantità non raggiunge o supera una soglia.

Struttura.

```
somma = 0                # accumulatore
while somma <= 50:       # guardia: la soglia non è ancora stata raggiunta
    n = int(input())
    somma = somma + n    # aggiornamento dell'accumulatore
print(somma)
```

Nota: la guardia dipende dall'accumulatore, non dall'input. Per questo non serve leggere un valore prima del ciclo.

Se il testo dice "smetti quando la somma supera 50", la guardia è `somma <= 50`: il ciclo continua anche quando la somma è esattamente `50`, e si ferma solo quando diventa maggiore.

3.3 Proprietà matematica

Hai un serbatoio da 1000 litri. Ogni giorno il livello si dimezza: il giorno 1 restano 500 litri, il giorno 2 restano 250, e così via. Vuoi sapere dopo quanti giorni il livello scenderà sotto i 10 litri, per ordinare in tempo una nuova fornitura. Non puoi sapere in anticipo quanti giorni ci vorranno: devi simulare il dimezzamento giorno per giorno e fermarti quando la proprietà "il serbatoio ha ancora almeno 10 litri" cessa di essere vera.

Struttura.

```
valore = ...                    # valore iniziale
while <proprietà su valore>:    # guardia: il valore conserva ancora la proprietà
    valore = <trasformazione>   # il valore si aggiorna secondo una regola
# all'uscita: valore ha perso la proprietà
```

Esempio: serbatoio.

```
litri = 1000
giorni = 0
while litri >= 10:       # guardia: il serbatoio ha ancora abbastanza acqua
    litri = litri // 2   # ogni giorno il livello si dimezza
    giorni = giorni + 1
```

```
print(f"Dopo {giorni} giorni il serbatoio scende sotto i 10 litri.")
```

Quando si usa. Non c'è un contatore né una soglia da accumulare: il valore si trasforma da solo a ogni iterazione secondo una regola, finché non conserva (o perde) una certa proprietà.

Esempio: verificare la divisibilità per 5. Se si sottrae ripetutamente `5` a un numero positivo e si arriva esattamente a `0`, il numero era divisibile per `5`:

```
n = int(input("n: "))
while n > 0:
    n = n - 5
if n == 0:
    print("divisibile per 5")
else:
    print("non divisibile per 5")
```

Dopo l'uscita dal ciclo, il valore di `n` contiene l'informazione che ci interessa: `0` significa divisibile, qualunque altro valore significa non divisibile. Questo è un principio generale: dopo un ciclo di questo tipo bisogna sempre guardare come si è conclusa l'iterazione.

Esempio: cifre di un numero senza usare stringhe. L'ultima cifra di un numero si estrae con `% 10`, e quella cifra si "rimuove" con `// 10`:

```
n = int(input("n: "))
while n > 0:
    cifra = n % 10
    print(cifra)
    n = n // 10
```

Con `n = 3643`, il programma stampa `3`, `4`, `6`, `3` (cioè le cifre da destra verso sinistra).

3.4 Sentinella

Stai leggendo un libro e vuoi sapere quante parole compaiono prima di incontrare la parola `"casa"`. Leggi le parole una alla volta e le conti; quando incontri `"casa"`, ti fermi. La parola `"casa"` non è un valore arbitrario scelto per convenienza — è proprio la parola che stavi cercando, e la sua comparsa è il segnale naturale che il conteggio è finito. `"casa"` non viene contata: è la sentinella, non un dato da elaborare.

Quando si usa. L'utente inserisce una sequenza di dati di lunghezza non nota. Esiste un valore speciale — la sentinella — che non è un dato da elaborare, ma un segnale di arresto.

Struttura.

```
numero = int(input())        # prima lettura - FUORI dal ciclo
somma = 0
while numero != 0:           # guardia: il valore non è la sentinella
    somma = somma + numero
    numero = int(input())    # rilettura - DENTRO il ciclo, prima di tornare alla guardia
print(somma)
```

Due punti critici:

1. La prima lettura va fatta **prima** del ciclo, perché la guardia deve già avere un valore da controllare.
2. La rilettura va fatta **alla fine del corpo del ciclo**, così la guardia può essere rivalutata con il nuovo valore prima del giro successivo.

La sentinella non deve essere elaborata: quando si incontra `0` (o `"fine"`, o `-1`), il ciclo termina e il valore speciale resta fuori dal risultato.

Esempio: conteggio di parole inserite prima di inserire "fine".

```
parola = input()
conteggio = 0
while parola != "fine":
    conteggio = conteggio + 1
    parola = input()
print(conteggio)
```

Se l'utente scrive `gatto`, `cane`, `topo`, `fine`, il programma stampa `3`. La parola `"fine"` non viene contata.

Nota sulla ridondanza. Uscendo da un ciclo con sentinella, lo stato del programma è già noto: la guardia è diventata falsa, quindi il valore della variabile letta è esattamente la sentinella. Considerando l'esempio precedente, quando si esce da `while parola != "fine"` si sa già con certezza che `parola` vale `fine`, altrimenti avremmo proseguito il ciclo! Scrivere subito dopo il ciclo un controllo come questo è quindi inutile:

```
numero = int(input())
somma = 0
while numero != 0:
    somma = somma + numero
    numero = int(input())

if numero == 0:       # ridondante: è sempre vero qui
    print(somma)
```

Aggiungere quel controllo non è sbagliato, ma segnala che non si è compresa la relazione tra la guardia e lo stato del programma all'uscita. Dopo il ciclo, `numero` vale `0` per definizione — non serve verificarlo di nuovo.

Caso speciale: il minimo. Trovare il minimo di una sequenza richiede attenzione al valore iniziale dell'accumulatore. L'errore più comune è inizializzarlo a `0` o a un numero arbitrario "piccolo":

```
# VERSIONE SBAGLIATA
numero = int(input())
minimo = 0              # sbagliato: se tutti i valori sono positivi, 0 non viene mai aggiorn

while numero != 0:
    if numero < minimo:
        minimo = numero
    numero = int(input())

print(minimo)           # stampa 0, non il vero minimo
```

Se l'utente inserisce `5, 3, 8, 0`, il programma stampa `0` invece di `3`, perché nessun valore è minore di `0` e `minimo` non viene mai aggiornato.

La soluzione corretta è inizializzare il minimo **al primo valore letto**, quello che si trova già fuori dal ciclo prima che il ciclo inizi:

```
numero = int(input())
minimo = numero          # il primo valore è il punto di partenza

while numero != 0:
    if numero < minimo:
        minimo = numero
    numero = int(input())

print(minimo)
```

Con questa struttura, il minimo parte da un valore reale della sequenza. Il ciclo poi confronta ogni nuovo valore con il minimo corrente e lo aggiorna solo se trova qualcosa di più piccolo. La sentinella `0` non entra mai nel confronto: viene letta, ma il ciclo termina prima di poterla confrontare.

3.5 Validazione (retry)

Stai creando un account e il sito ti chiede di scegliere un PIN di 4 cifre. Se inserisci un numero troppo corto, troppo lungo, o fuori range, il sito non accetta e ti chiede di riprovare. Non esiste un valore speciale che significa "ho finito": il ciclo si ferma solo quando inserisci qualcosa di valido. Ed è proprio quel valore valido — il PIN che hai scelto — che il programma userà da quel momento in poi.

Quando si usa. Il programma deve ricevere un valore che rispetti certi vincoli. Se l'utente inserisce qualcosa di non valido, la richiesta si ripete.

Struttura.

```python
voto = int(input("Voto (0-30): "))      # prima richiesta
while voto < 0 or voto > 30:            # guardia: il valore non è accettabile
    print("Valore non valido.")
    voto = int(input("Voto (0-30): ")) # nuova richiesta
print("Voto accettato:", voto)
```

La differenza rispetto alla sentinella è fondamentale: qui non esiste un valore speciale di uscita. Il ciclo si ferma non appena l'input soddisfa i vincoli, ed è proprio quel valore valido che si usa nel seguito del programma.

4. Riconoscere il paradigma

Leggere un testo e individuare il paradigma corretto è già metà del lavoro. Alcuni segnali tipici:

Segnale nel testo	Paradigma probabile
"da 1 a 10", "5 volte", "per ogni elemento"	Contatore
"finché la somma non supera", "almeno X"	Soglia su accumulatore
"finché è positivo", "usando solo la sottrazione"	Proprietà matematica
"finché l'utente vuole", "inserisci 0 per terminare"	Sentinella
"controlla che sia valido", "ripeti se non è nel range"	Validazione

I paradigmi non sono gabbie rigide: uno stesso esercizio può combinarne due, o usarne uno come sottoschema di un altro.

Valori iniziali degli accumulatori.

Operazione	Valore iniziale	Motivo
somma	`0`	è l'elemento neutro della somma, `x + 0 = x`
prodotto	`1`	è l'elemento neutro della moltiplicazione, `x * 1 = x`; con `0` il risultato è sempre `0`
contatore	`0`	nessun elemento è stato ancora contato

5. Due problemi che il `while` da solo non riesce a risolvere

Ci sono situazioni in cui i cinque paradigmi non bastano, perché il problema richiede di ricordare una sequenza intera di valori.

Problema 1: stampare i numeri letti al contrario. Leggendo numeri fino a `0` con il paradigma della sentinella, quando arriva lo `0` i numeri precedenti sono andati persi — ogni nuova lettura ha sovrascritto la variabile.

Problema 2: tra tutti i numeri letti, stampare quelli sopra la media. Per sapere se, ad esempio, `4` è sopra la media di `4, 8, 2, 6`, bisogna già conoscere la media. Ma la media si può calcolare solo dopo aver letto tutti i numeri. Bisogna quindi scorrere due volte la stessa sequenza: una volta per leggere e calcolare la media, una seconda volta per confrontare ogni numero con il valore della media, e decidere se stampare o meno.

In entrambi i casi il problema ha la stessa radice: il programma ha bisogno di **ricordare una sequenza di valori** per poterla usare dopo. È esattamente la funzione di una lista.

6. Le liste

Una lista è una sequenza ordinata di valori tenuti insieme sotto un unico nome. È un nuovo tipo di dato, come `int`, `str` e `bool`, ma invece di contenere un solo valore atomico contiene una **collezione** di valori.

Creare una lista

```
voti = [28, 30, 24, 27]        # lista di interi già noti
nomi = ["Alice", "Bruno"]      # lista di stringhe
numeri = []                    # lista vuota - punto di partenza quando non si sa quanti ele
```

Aggiungere elementi: `append`

```
numeri = []
numeri.append(5)
numeri.append(3)
numeri.append(-2)
print(numeri)   # [5, 3, -2]
```

Ogni chiamata ad `append` aggiunge un elemento **in coda**. L'ordine di inserimento viene conservato.

Lunghezza: `len`

```
voti = [28, 30, 24, 27]
print(len(voti))   # 4
```

Una lista vuota ha lunghezza `0`.

Accesso per indice

Gli indici partono da `0`, come nelle stringhe:

```
voti = [28, 30, 24, 27]
#       0   1   2   3
```

```python
print(voti[0])    # 28  (primo elemento)
print(voti[3])    # 27  (quarto elemento)
print(voti[-1])   # 27  (ultimo elemento)
```

Modifica di un elemento

A differenza delle stringhe, le liste sono modificabili:

```python
voti = [28, 30, 24, 27]
print(voti)   # [28, 30, 24, 27]
voti[0] = 29
print(voti)   # [29, 30, 24, 27]
```

Concatenazione

```python
a = [1, 2, 3]
b = [4, 5, 6]
print(a + b)   # [1, 2, 3, 4, 5, 6]
```

L'operatore `+` non modifica `a` né `b`: produce una nuova lista.

Riepilogo operazioni

Operazione	Sintassi	Nota
Lista vuota	`L = []`	
Aggiunta in coda	`L.append(x)`	modifica `L` sul posto
Lunghezza	`len(L)`	
Accesso per indice	`L[i]`	da `0` a `len(L)-1`
Ultimo elemento	`L[-1]`	
Modifica elemento	`L[i] = x`	modifica `L` sul posto
Concatenazione	`L1 + L2`	produce una nuova lista

Nota sui tipi

Python permette liste con elementi di tipo diverso, ma conviene costruire liste omogenee (tutti interi, tutte stringhe, …): è più semplice applicare la stessa operazione a ogni elemento.

7. `while` e liste insieme

Il `while` era già utile per lavorare su un valore alla volta; con le liste diventa capace di costruire, scorrere e modificare strutture che contengono molti dati.

Costruire una lista con il `while`

Il modo più comune di popolare una lista è combinarla con il paradigma della sentinella: si leggono valori uno alla volta e si aggiunge ciascuno alla lista con `append`, finché non arriva il valore speciale che segnala la fine. La lista cresce a ogni iterazione e alla fine contiene tutti i valori inseriti, nell'ordine in cui sono stati letti.

```
numeri = []
n = int(input("Inserisci un numero: "))
while n != 0:
    numeri.append(n)
    n = int(input())

print(numeri)
```

Se l'utente inserisce `5, 3, 8, 0`, al termine del ciclo `numeri` vale `[5, 3, 8]`. Lo `0` non viene aggiunto: è la sentinella, e il ciclo termina prima di eseguire `append` su di esso.

Scorrere una lista con il `while`

Per visitare tutti gli elementi di una lista si usa il paradigma del contatore: `i` parte da `0` (il primo indice valido) e avanza di `1` a ogni giro. La guardia `i < len(numeri)` assicura che ci si fermi prima di uscire dai limiti della lista: l'ultimo indice valido è `len(numeri) - 1`, quindi quando `i` raggiunge `len(numeri)` il ciclo termina.

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

Con `numeri = [5, 3, 8]`, il ciclo stampa `5`, poi `3`, poi `8`, poi si ferma perché `i` vale `3` e `3 < 3` è falso.

Scorrere al contrario

Per visitare la lista dall'ultimo elemento al primo si inverte la direzione: `i` parte dall'indice dell'ultimo elemento (`len(numeri) - 1`) e decresce di `1` a ogni giro. La guardia `i >= 0` assicura che ci si fermi dopo aver visitato il primo elemento (indice `0`).

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

Con `numeri = [5, 3, 8]`, il ciclo stampa `8`, poi `3`, poi `5`, poi si ferma perché `i`

vale `-1` e `-1 >= 0` è falso.

Soluzione ai problemi aperti

Con `append` il Problema 1 (stampare al contrario) è ora risolvibile:

```
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
```

Il primo ciclo accumula; il secondo scorre la lista in ordine inverso.

Anche il Problema 2 (stampare i numeri sopra la media) è ora risolvibile. Serve fare due passate sulla stessa sequenza: la prima per raccogliere i valori e calcolare la media, la seconda per confrontare ogni elemento con essa.

```
numeri = []
n = int(input())
while n != 0:
    numeri.append(n)
    n = int(input())

# prima passata: calcola la media
somma = 0
i = 0
while i < len(numeri):
    somma = somma + numeri[i]
    i = i + 1
media = somma / len(numeri)

# seconda passata: stampa solo i valori sopra la media
i = 0
while i < len(numeri):
    if numeri[i] > media:
        print(numeri[i])
    i = i + 1
```

Con input `4, 8, 2, 6, 0`, la media è `(4 + 8 + 2 + 6) / 4 = 5.0`. Il secondo ciclo stampa `8` e `6`. Senza la lista questo programma sarebbe impossibile da scrivere: al momento di fare i confronti, i valori originali non sarebbero più disponibili.

È possibile anche una versione più compatta con soli due cicli: la somma si accumula già durante la lettura, così la media è disponibile subito dopo il primo ciclo senza bisogno di riscorrere la lista.

```
numeri = []
somma = 0
n = int(input())
while n != 0:
    numeri.append(n)
    somma = somma + n
    n = int(input())

media = somma / len(numeri)

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

8. La funzione `split`: da stringa a lista

Ora che conosciamo il nuovo tipo `list` possiamo anche introdurre una nuova operazione che prende in input una stringa e restituisce una lista di sottostringhe.

```
frase = "era una notte incantevole"
parole = frase.split()
print(parole)   # ["era", "una", "notte", "incantevole"]
```

Senza argomenti, la funzione `split()` divide di default sugli spazi presenti, ma si può specificare un separatore diverso:

```
s = "1,2,3,4"
s.split(",")   # ["1", "2", "3", "4"]
```

9. Come scrivere lo stato durante un ciclo con lista

Come per il `while` elementare, costruire la tabella dello stato aiuta a seguire l'esecuzione. Con le liste la tabella mostra il contenuto aggiornato della lista a ogni passo.

Esempio con input `5, 3, 0`:

```
numeri = []
n = int(input("Inserisci un numero: "))
while n != 0:
    numeri.append(n)
    n = int(input("Inserisci un numero: "))
```

Passo	Istruzione	`numeri`	`n`
1	`numeri = []`	`[]`	—
2	`n = input()` ← `5`	`[]`	5
3	`while n != 0` → True	`[]`	5
4	`numeri.append(5)`	`[5]`	5
5	`n = input()` ← `3`	`[5]`	3
6	`while n != 0` → True	`[5]`	3
7	`numeri.append(3)`	`[5, 3]`	3
8	`n = input()` ← `0`	`[5, 3]`	0
9	`while n != 0` → **False**	`[5, 3]`	0