# Dispensa · Modulo 3

## 1. Che cosa cambia nel modulo 3

I moduli precedenti hanno introdotto gli strumenti di base per costruire programmi: tipi, variabili, espressioni, istruzioni, condizioni. A questo punto è naturale chiedersi non solo *come si scrive* un programma, ma *come si verifica che funzioni davvero*.

Il modulo 3 introduce due nuove competenze. La prima è metodologica: imparare a distinguere i programmi che si comportano correttamente da quelli che producono risultati sbagliati senza segnalare errori. La seconda è strutturale: imparare a usare il ciclo `while`, che permette di ripetere un blocco di istruzioni finché una condizione rimane vera.

## 2. Due tipi di errori

Quando un programma non funziona come previsto, l'errore può essere di due tipi fondamentalmente diversi.

Il primo tipo è l'**errore di esecuzione** (o errore a runtime). Il programma non parte oppure si interrompe a metà: Python mostra un messaggio di errore che indica il tipo di problema, la riga in cui si è verificato e l'operazione che ha fallito. Questi errori sono in un certo senso i più facili: forniscono informazioni concrete su che cosa è andato storto.

Il secondo tipo è l'**errore logico**. Il programma viene eseguito senza messaggi di errore, ma produce un risultato sbagliato rispetto a quello atteso. Questo tipo di errore è più insidioso perché non fornisce alcun segnale automatico: bisogna accorgersene confrontando il comportamento del programma con quello che ci si aspettava.

Questa distinzione è fondamentale: un programma che "gira" non è necessariamente un programma corretto.

## 3. Leggere i messaggi di errore

Quando Python segnala un errore a runtime, il messaggio contiene sempre almeno tre informazioni utili:

| Informazione | Domanda |
|---|---|
| tipo di errore | che genere di problema si è verificato? |
| riga segnalata | dove si manifesta? |
| contesto | quale operazione stava tentando Python? |

I tipi di errore più comuni in questa fase del corso sono quattro.

Un `SyntaxError` indica che il codice non è scritto correttamente: Python non riesce nemmeno a leggerlo. Compare quando mancano due punti, parentesi non bilanciate o istruzioni incomplete.

Un `NameError` indica che si sta usando un nome di variabile che non esiste nello stato del programma in quel momento. Succede per esempio quando si fa un refuso nel nome, o quando si usa una variabile prima di averla assegnata.

```
print(eta)
eta = int(input("Età: "))
```

Un **TypeError** indica che si sta tentando un'operazione tra valori di tipo incompatibile. Il caso più frequente è cercare di sommare o concatenare una stringa con un numero senza convertire prima.

```
eta = input("Età: ")
print("Tra 10 anni avrai " + eta + 10)
```

Un **ValueError** indica che si sta tentando una conversione di tipo su un valore che non può essere convertito: per esempio `int("ciao")`.

Imparare a leggere questi messaggi è una competenza essenziale. Non bisogna scoraggiarsi davanti a un errore: è il programma che sta comunicando dov'è il problema.

## 4. Testing e casi limite

Anche quando un programma non genera errori, potrebbe comunque produrre risultati sbagliati. L'unico modo per verificare la correttezza è **testarlo**: scegliere degli input, stabilire in anticipo quale output ci si aspetta, eseguire il programma e confrontare i risultati.

Il metodo si articola in questi passi:

1. definire chiaramente il problema e le specifiche;
2. scegliere input rappresentativi;
3. scrivere l'output atteso prima di eseguire;
4. eseguire il programma;
5. confrontare output atteso e output reale;
6. se differiscono, individuare il bug, correggere e ripetere.

### I casi limite

Un errore molto comune è dimenticare i **casi limite**: i valori di input al confine del dominio che il programma dovrebbe saper gestire. Per un numero intero, i casi limite tipici sono zero, uno, meno uno, valori negativi. Per una stringa, una stringa vuota o una stringa di un solo carattere.

### Esempio: stampa di un numero con segno

Supponiamo di voler scrivere un programma che stampa un numero con il suo segno esplicito: `+7`, `-3`, `0`.

Una prima versione potrebbe essere:

```
n = int(input("Numero: "))
if n > 0:
    segno = "+"
else:
    segno = "-"
print(segno + str(n))
```

Testando questo programma:

| Input | Output atteso | Output reale |
|---|---|---|
| 7 | +7 | +7 |
| -3 | -3 | --3 |
| 0 | 0 | -0 |

Per `7` funziona. Per `-3` produce `--3`: `str(-3)` restituisce già `"-3"`, e concatenarci davanti un altro `"-"` raddoppia il segno. Per `0` produce `-0`: il ramo `else` si attiva per qualsiasi valore non positivo, zero incluso.

Il programma non ha mai generato errori, eppure due casi su tre sono sbagliati.

**Correzione:**

```
n = int(input("Numero: "))
if n > 0:
    print("+" + str(n))
elif n < 0:
    print(str(n))
else:
    print("0")
```

Questa versione tratta esplicitamente tutti e tre i casi: positivo, negativo e zero.

## 5. Valutazione lazy

Gli operatori `and` e `or` non valutano sempre entrambe le parti dell'espressione. Python si ferma appena il risultato è già determinabile: questo comportamento si chiama **valutazione lazy** o **short-circuit**.

Con `or`: se la prima condizione è `True`, il risultato sarà `True` indipendentemente dalla seconda. Python non valuta la seconda.

Con `and`: se la prima condizione è `False`, il risultato sarà `False` indipendentemente dalla seconda. Python non valuta la seconda.

Per esempio:

```
x = 5

if x > 0 or y > 0:
    print("Almeno uno positivo")
```

Questo codice non genera un `NameError` anche se `y` non è definita, perché `x > 0` vale già `True` e Python non arriva a valutare `y > 0`.

Al contrario:

```
if x > 0 and y > 0:
    print("Entrambi positivi")
```

Qui Python valuta `x > 0` (True), poi deve valutare `y > 0` per decidere il risultato dell'`and` — e a quel punto genera `NameError` perché `y` non esiste.

La valutazione lazy non è un'anomalia: è una caratteristica del linguaggio che può essere sfruttata intenzionalmente, per esempio per evitare divisioni per zero nel secondo operando.

## 6. Dal `if` al `while`

Il costrutto `if` esegue un blocco al massimo una volta: se la condizione è vera, esegue il blocco; altrimenti lo salta. Non c'è modo di tornare indietro.

Il ciclo `while` risolve il problema di quando si vuole ripetere un blocco più volte. La struttura è identica all'`if`, ma con una differenza cruciale: dopo aver eseguito il blocco, il `while` **torna a valutare la condizione**. Se è ancora vera, il blocco viene eseguito di nuovo. Se è falsa, il ciclo termina.

Questa è la differenza fondamentale: con `while`, il blocco deve **modificare lo stato** in modo da portare prima o poi la condizione a diventare falsa. Se non lo fa, il ciclo non si ferma mai — si parla di **ciclo infinito**.

## 7. Sintassi e semantica del `while`

### Sintassi

```
while espressione_booleana:
    istruzione-1
    istruzione-2
    ...
    istruzione-k
```

- `while` introduce una ripetizione condizionata;
- la guardia è un'espressione booleana valutata prima di ogni iterazione;
- i due punti `:` aprono il blocco;
- il blocco va indentato.

### Semantica

1. Python valuta la guardia.
2. Se vale `True`, esegue il blocco.
3. Torna al punto 1.
4. Se vale `False`, esce dal ciclo e continua con le istruzioni successive.

### Esempio: contatore

```
i = 0
while i < 5:
    print(i)
    i = i + 1
```

Questo programma stampa `0`, `1`, `2`, `3`, `4`. Le parti da riconoscere sono:

- `i = 0` è l'**inizializzazione**: definisce il punto di partenza;
- `i < 5` è la **guardia**: il ciclo continua finché vale `True`;
- `print(i)` è l'**azione** ripetuta;

- `i = i + 1` è l'**aggiornamento** che modifica lo stato e avvicina il programma alla terminazione.

Se si dimentica l'aggiornamento `i = i + 1`, la guardia non cambia mai e il ciclo non finisce. Su terminale si può interrompere con `Ctrl+C`.

## 8. Pattern comuni del `while`

### Contatore

Il pattern più semplice: una variabile intera parte da un valore iniziale e viene incrementata o decrementata a ogni iterazione fino a raggiungere una soglia.

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
```

### Valore sentinella

Il ciclo continua finché l'utente non inserisce un valore speciale (la "sentinella") che segnala la fine.

```
numero = int(input("Numero (-1 per terminare): "))
somma = 0
while numero != -1:
    somma = somma + numero
    numero = int(input("Numero (-1 per terminare): "))
print(somma)
```

Il numero di iterazioni non è noto in anticipo: dipende dall'utente.

### Validazione dell'input

Il ciclo viene usato per ripetere la richiesta finché l'utente non fornisce un valore valido.

```
n = int(input("Inserisci un numero positivo: "))
while n < 0:
    n = int(input("Numero non valido. Riprova: "))
```

Questo schema garantisce che dopo il ciclo `n` contenga sempre un valore valido.

## 9. Scansione di stringhe con `while`

Una stringa è una sequenza di caratteri accessibili tramite indice: `s[0]` è il primo carattere, `s[1]` il secondo, fino a `s[len(s)-1]` l'ultimo. Combinando un contatore con la funzione `len()`, si può scorrere ogni carattere.

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

L'ordine delle operazioni è importante: bisogna **prima usare `i`**, poi aggiornarlo. Se si incrementa `i` prima di accedere al carattere, si salta il primo elemento.

**Esempio: caratteri nelle posizioni pari**

```
parola = input("Parola: ")
i = 0
while i < len(parola):
    print(parola[i])
    i = i + 2
```

Incrementando `i` di 2 a ogni passo si visitano solo gli indici `0`, `2`, `4`, …

## 10. Cicli annidati

Un ciclo `while` può contenere nel suo corpo un altro ciclo `while`. Il ciclo interno viene eseguito per intero a ogni iterazione del ciclo esterno.

```
p = 1
while p <= 3:
    r = 1
    while r <= 3:
        print(p, r)
        r = r + 1
    p = p + 1
```

Due punti fondamentali da osservare:

- ogni ciclo ha la propria variabile di controllo e il proprio aggiornamento;
- la variabile del ciclo interno (`r`) deve essere **reinizializzata** all'inizio di ogni iterazione del ciclo esterno, altrimenti alla seconda passata risulterebbe già al valore finale.

## 11. Tracciare lo stato del programma durante un `while`

Il metodo di tracciamento già introdotto per le istruzioni semplici si applica direttamente anche ai cicli: si costruisce la tabella riga per riga, ripetendo le colonne della guardia e del blocco per ogni iterazione.

**Esempio: contatore**

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```

| Passo | Istruzione | Stato della memoria | Output |
|---|---|---|---|
| 1 | `i = 0` | `i` → `0` | |
| 2 | `while i < 3` → True | `i` → `0` | |
| 3 | `print(i)` | `i` → `0` | `0` |
| 4 | `i = i + 1` | `i` → `1` | |

| Passo | Istruzione | Stato della memoria | Output |
|---|---|---|---|
| 5 | `while i < 3` → True | `i` → 1 | |
| 6 | `print(i)` | `i` → 1 | 1 |
| 7 | `i = i + 1` | `i` → 2 | |
| 8 | `while i < 3` → True | `i` → 2 | |
| 9 | `print(i)` | `i` → 2 | 2 |
| 10 | `i = i + 1` | `i` → 3 | |
| 11 | `while i < 3` → **False** | `i` → 3 | |

**Esempio: valore sentinella**

(ipotesi: l'utente inserisce `4`, poi `7`, poi `-1`)

```
numero = int(input("Numero: "))
somma = 0
while numero != -1:
    somma = somma + numero
    numero = int(input("Numero: "))
```

| Passo | Istruzione | Stato della memoria | Output |
|---|---|---|---|
| 1 | `numero = input(...)` ← 4 | `numero` → 4 | |
| 2 | `somma = 0` | `numero` → 4, `somma` → 0 | |
| 3 | `while numero != -1` → True | `numero` → 4, `somma` → 0 | |
| 4 | `somma = somma + numero` | `numero` → 4, `somma` → 4 | |
| 5 | `numero = input(...)` ← 7 | `numero` → 7, `somma` → 4 | |
| 6 | `while numero != -1` → True | `numero` → 7, `somma` → 4 | |
| 7 | `somma = somma + numero` | `numero` → 7, `somma` → 11 | |
| 8 | `numero = input(...)` ← -1 | `numero` → -1, `somma` → 11 | |
| 9 | `while numero != -1` → **False** | `numero` → -1, `somma` → 11 | |