Capitolo 7

Iterazione

In questo capitolo parleremo dell'iterazione, che è la capacità di eseguire ripetutamente uno stesso blocco di istruzioni. Abbiamo visto una sorta di iterazione nel Paragrafo 5.8, usando la ricorsione. Ne abbiamo visto un tipo nel Paragrafo 4.2, dove abbiamo utilizzato un ciclo `for`. Qui ne vedremo un tipo ulteriore, che usa l'istruzione `while`. Ma prima, qualche altro dettaglio sull'assegnazione delle variabili.

7.1 Riassegnazione

Vi sarete forse già accorti che è possibile effettuare più assegnazioni ad una stessa variabile. Una nuova assegnazione fa sì che la variabile faccia riferimento ad un nuovo valore, cessando di riferirsi a quello vecchio.

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

La prima volta che visualizziamo `x`, il suo valore è 5; la seconda volta è 7.

La Figura 7.1 illustra il diagramma di stato per questa **riassegnazione**.

Ora, è bene chiarire un punto che è frequente motivo di confusione. Dato che Python usa (=) per le assegnazioni, potreste interpretare l'istruzione `a = b` come un'espressione matematica di uguaglianza, cioè una proposizione per cui `a` e `b` sono uguali. Questo non è corretto.

In primo luogo, l'equivalenza è una relazione simmetrica, cioè vale in entrambi i sensi, mentre l'assegnazione non lo è: in matematica se $a = 7$ allora è anche $7 = a$. Ma in Python l'istruzione `a = 7` è valida mentre `7 = a` non lo è.

Inoltre, in matematica un'uguaglianza è o vera o falsa, e rimane tale: se ora $a = b$ allora a sarà sempre uguale a b. In Python, un'assegnazione può rendere due variabili temporaneamente uguali, ma non è affatto detto che l'uguaglianza permanga:

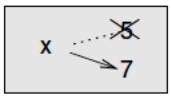

Figura 7.1: Diagramma di stato.

```
>>> a = 5
>>> b = a    # a e b ora sono uguali
>>> a = 3    # a e b non sono più uguali
>>> b
5
```

La terza riga modifica il valore di `a` ma non quello di `b`, quindi `a` e `b` non sono più uguali.

Anche se le riassegnazioni di variabile sono spesso utili, vanno usate con cautela. Se il valore di una variabile cambia di frequente, può rendere il codice difficile da leggere e correggere.

7.2 Aggiornare le variabili

Una delle forme più comuni di riassegnazione è l'**aggiornamento**, dove il nuovo valore della variabile dipende da quello precedente.

```
>>> x = x + 1
```

Questo significa: "prendi il valore attuale di `x`, aggiungi uno, e aggiorna `x` al nuovo valore."

Se tentate di aggiornare una variabile inesistente, si verifica un errore perché Python valuta il lato destro prima di assegnare un valore a `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Prima di aggiornare una variabile occorre quindi **inizializzarla**, di solito con una comune assegnazione:

```
>>> x = 0
>>> x = x + 1
```

L'aggiornamento di una variabile aggiungendo 1 è detto **incremento**; sottrarre 1 è detto invece **decremento**.

7.3 L'istruzione `while`

Spesso i computer sono usati per automatizzare dei compiti ripetitivi: ripetere operazioni identiche o simili senza fare errori, è qualcosa che i computer fanno molto bene e le persone piuttosto male. Nella programmazione, la ripetizione è chiamata anche **iterazione**.

Abbiamo già visto due funzioni, `contoallarovescia` e `stampa_n`, che iterano usando la ricorsione. Dato che l'iterazione è un'operazione molto frequente, Python fornisce varie caratteristiche del linguaggio per renderla più semplice da implementare. Una è l'istruzione `for`, che abbiamo già visto nel Paragrafo 4.2 e sulla quale torneremo.

Un'altra istruzione è `while`. Ecco una variante di `contoallarovescia` che usa l'istruzione `while`:

```
def contoallarovescia(n):
    while n > 0:
        print(n)
        n = n-1
    print('Via!')
```

Si può quasi leggere il programma con l'istruzione `while` come fosse scritto in inglese: significa "Finché (while) `n` è maggiore di 0, stampa il valore di `n` e poi decrementa `n` di 1. Quando arrivi a 0, stampa la stringa `Via!`"

In modo più formale, questo è il flusso di esecuzione di un'istruzione `while`:

1. Determina se la condizione è vera (`True`) o falsa (`False`).
2. Se la condizione è falsa, esce dal ciclo `while` e continua l'esecuzione dalla prima istruzione successiva.
3. Se la condizione è vera, esegue il blocco di istruzioni nel corpo del ciclo while e vi rimane, ritornando al punto 1.

Questo tipo di flusso è chiamato ciclo, (in inglese *loop*), perché il terzo punto ritorna ciclicamente da capo.

Il corpo del ciclo deve cambiare il valore di una o più variabili in modo che la condizione prima o poi diventi falsa e il ciclo abbia termine. Altrimenti, il ciclo verrebbe ripetuto continuamente, dando luogo ad un **ciclo infinito**. Una fonte inesauribile di divertimento per gli informatici, è osservare che le istruzioni dello shampoo: "lava, risciacqua, ripeti" sono un ciclo infinito.

Nel caso di `contoallarovescia`, è evidente che il ciclo terminerà: se `n` è zero o negativo, il ciclo non viene mai eseguito. Altrimenti, `n` diventa via via più piccolo ad ogni ripetizione del ciclo stesso, fino a diventare, prima o poi, zero.

In altri cicli, può non essere così evidente. Per esempio:

```
def sequenza(n):
    while n != 1:
        print(n)
        if n % 2 == 0:        # n è pari
            n = n / 2
        else:               # n è dispari
            n = n*3+1
```

La condizione di questo ciclo è `n != 1`, per cui il ciclo si ripeterà fino a quando `n` non sarà uguale a `1`, cosa che rende falsa la condizione.

Ad ogni ripetizione del ciclo, il programma stampa il valore di `n` e poi controlla se è pari o dispari. Se è pari, `n` viene diviso per 2. Se è dispari, `n` è moltiplicato per 3 e al risultato viene aggiunto 1. Se per esempio il valore passato a `sequenza` è 3, i valori risultanti di `n` saranno nell'ordine 3, 10, 5, 16, 8, 4, 2, 1.

Dato che `n` a volte sale e a volte scende, non c'è modo di stabilire che `n` raggiungerà 1 in modo da terminare il ciclo. Solo per qualche particolare valore di `n`, possiamo dimostrarlo: ad esempio, se il valore di partenza è una potenza di 2, `n` sarà per forza un numero pari ad ogni ciclo, fino a raggiungere 1. L'esempio precedente finisce proprio con una sequenza simile, a partire dal numero 16.

La domanda difficile è se il programma giunga a termine per *qualsiasi* valore positivo di `n`. Sinora, nessuno è riuscito a dimostrarlo *né* a smentirlo! (Vedere `http://it.wikipedia.org/wiki/Congettura_di_Collatz`.)

Come esercizio, riscrivete la funzione `stampa_n` del Paragrafo 5.8 usando l'iterazione al posto della ricorsione.

7.4 `break`

Vi può capitare di poter stabilire il momento in cui è necessario terminare un ciclo solo mentre il flusso di esecuzione si trova nel bel mezzo del corpo. In questi casi potete usare l'istruzione `break` per interrompere il ciclo e saltarne fuori.

Per esempio, supponiamo che vogliate ricevere delle risposte dall'utente, fino a quando non viene digitata la parola `fine`. Potete scrivere:

```
while True:
    riga = input('> ')
    if riga == 'fine':
        break
    print(riga)

print('Finito!')
```

La condizione del ciclo è `True`, che è sempre vera per definizione, quindi il ciclo è destinato a continuare, a meno che non incontri l'istruzione `break`.

Ad ogni ripetizione, il programma mostra come prompt il simbolo >. Se l'utente scrive `fine`, l'istruzione `break` interrompe il ciclo, altrimenti il programma ripete quello che l'utente ha scritto e ritorna da capo. Ecco un esempio di esecuzione:

```
> non ho finito
non ho finito
> fine
Finito!
```

Questo modo di scrivere i cicli `while` è frequente, perché vi permette di controllare la condizione ovunque all'interno del ciclo (e non solo all'inizio) e di esprimere la condizione di stop in modo affermativo ("fermati quando succede questo") piuttosto che negativo ("continua fino a quando non succede questo").