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:

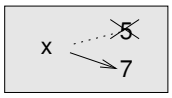

Figura 7.1: Diagramma di stato.

```
>>> 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").

7.5 Radici quadrate

I cicli si usano spesso per calcolare risultati numerici, partendo da un valore approssimativo che viene migliorato iterativamente con approssimazioni successive.

Per esempio, un modo di calcolare le radici quadrate è il metodo di Newton. Supponiamo di voler calcolare la radice quadrata di a. A partire da una qualunque stima, x, possiamo calcolare una stima migliore con la formula seguente:

$$y = \frac{x + a/x}{2}$$

Supponiamo per esempio che a sia 4 e x sia 3:

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

Il risultato è più vicino al valore vero ($\sqrt{4} = 2$). Se ripetiamo il procedimento usando la nuova stima, ci avviciniamo ulteriormente:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

Dopo qualche ulteriore passaggio, la stima diventa quasi esatta:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

In generale, non possiamo sapere *a priori* quanti passaggi ci vorranno per ottenere la risposta esatta, ma sapremo che ci saremo arrivati quando la stima non cambierà più:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

Possiamo fermarci quando `y == x`. Ecco quindi un ciclo che parte da una stima iniziale, `x`, e la migliora fino a quando non cambia più:

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

Per la maggior parte dei valori di `a`, questo codice funziona bene, ma in genere è pericoloso testare l'uguaglianza su valori decimali di tipo `float`, perché sono solo approssimativamente esatti: la maggior parte dei numeri razionali come $1/3$, e irrazionali, come $\sqrt{2}$, non possono essere rappresentati in modo preciso con un `float`.

Piuttosto di controllare se `x` e `y` sono identici, è meglio usare la funzione predefinita `abs` per calcolare il valore assoluto della loro differenza:

```
if abs(y-x) < epsilon:
    break
```

Dove `epsilon` è un valore molto piccolo, come `0.0000001`, che determina quando i due numeri confrontati sono abbastanza vicini da poter essere considerati praticamente uguali.

7.6 Algoritmi

Il metodo di Newton è un esempio di **algoritmo**: è un'operazione meccanica per risolvere un tipo di problema (in questo caso, calcolare la radice quadrata).

Per capire cosa sia un algoritmo, può essere utile iniziare a vedere cosa non è un algoritmo. Quando a scuola vi insegnarono a fare le moltiplicazioni dei numeri a una cifra, probabilmente avevate imparato a memoria le tabelline, che significa ricordare 100 specifiche soluzioni. Una conoscenza di questo tipo non è algoritmica.

Ma se eravate dei bambini un po' pigri, probabilmente avevate imparato qualche trucchetto. Per esempio, per trovare il prodotto tra n e 9, si scrive $n - 1$ come prima cifra e $10 - n$ come seconda cifra. Questo trucco è una soluzione generica per moltiplicare per nove qualunque numero a una cifra. Questo è un algoritmo!

Similmente, le tecniche che avete imparato per l'addizione con riporto, la sottrazione con prestito e le divisioni lunghe sono tutte algoritmi. Una caratteristica degli algoritmi è che non richiedono intelligenza per essere eseguiti. Sono procedimenti meccanici in cui ad ogni passo ne segue un altro, secondo delle semplici regole.

L'esecuzione di un algoritmo, in sé, è una cosa noiosa e ripetitiva. D'altra parte, la procedura di realizzazione di un algoritmo è interessante, intellettualmente stimolante, e una parte cruciale di quella che chiamiamo programmazione.

Alcune delle cose che le persone fanno in modo naturale senza difficoltà o senza nemmeno pensarci, sono le più difficili da esprimere con algoritmi. Capire il linguaggio naturale è un esempio calzante. Lo facciamo tutti, ma finora nessuno è stato in grado di spiegare *come* lo facciamo, almeno non sotto forma di un algoritmo.

7.7 Debug

Quando inizierete a scrivere programmi di grandi dimensioni, impiegherete più tempo per il debug: più codice significa più probabilità di commettere un errore e più posti in cui gli errori possono annidarsi.

Un metodo per ridurre il tempo di debug è il "debug binario". Se nel vostro programma ci sono 100 righe e le controllate una ad una, ci vorranno 100 passaggi.

Provate invece a dividere il problema in due. Cercate verso la metà del programma un valore intermedio che potete controllare. Aggiungete un'istruzione di stampa (o qualcos'altro di controllabile) ed eseguite il programma.

Se il controllo nel punto mediano non è corretto, deve esserci un problema nella prima metà del programma. Se invece è corretto, l'errore sarà nella seconda metà.

Per ogni controllo eseguito in questa maniera, dimezzate le righe da controllare. Dopo 6 passaggi (che sono meno di 100), dovreste teoricamente arrivare a una o due righe di codice.

In pratica, non è sempre chiaro quale sia la "metà del programma" e non è sempre possibile controllare. Non ha neanche molto senso contare le righe e trovare la metà esatta. Meglio considerare i punti del programma dove è più probabile che vi siano errori e quelli dove è facile posizionare dei controlli. Poi, scegliere un punto dove stimate che le probabilità che l'errore sia prima o dopo quel punto siano circa le stesse.

7.8 Glossario

riassegnazione: Assegnazione di un nuovo valore ad una variabile che esiste già.

aggiornamento: Riassegnazione in cui il nuovo valore della variabile dipende da quello precedente.

inizializzazione: Assegnazione che fornisce un valore iniziale ad una variabile da aggiornare successivamente.

incremento: Aggiornamento che aumenta il valore di una variabile (spesso di una unità).

decremento: Aggiornamento che riduce il valore di una variabile.

iterazione: Ripetizione di una serie di istruzioni utilizzando una funzione ricorsiva oppure un ciclo.

ciclo infinito: Ciclo in cui la condizione che ne determina la fine non è mai soddisfatta.

algoritmo: Una procedura generica per risolvere una categoria di problemi.

7.9 Esercizi

Esercizio 7.1. *Copiate il ciclo del Paragrafo 7.5 e incapsulatelo in una funzione di nome* `mia_radq` *che prenda* `a` *come parametro, scelga un valore appropriato di* `x`*, e restituisca una stima del valore della radice quadrata di* `a`*.*

Quale verifica, scrivete una funzione di nome `test_radq` *che stampi una tabella come questa:*

```
a   mia_radq(a)   math.sqrt(a)  diff
-   ----------    ------------  ----
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

La prima colonna è un numero, a; la seconda è la radice quadrata di a calcolata con `mia_radq`*; la terza è la radice quadrata calcolata con* `math.sqrt`*; la quarta è il valore assoluto della differenza tra le due stime.*

Esercizio 7.2. *La funzione predefinita* `eval` *valuta un'espressione sotto forma di stringa, usando l'interprete Python. Ad esempio:*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

Scrivete una funzione di nome `eval_ciclo` *che chieda iterativamente all'utente di inserire un dato, prenda il dato inserito e lo valuti con* `eval`*, infine visualizzi il risultato.*

Deve continuare fino a quando l'utente non scrive `'fatto'`*, e poi restituire il valore dell'ultima espressione che ha valutato.*

Esercizio 7.3. *Il matematico Srinivasa Ramanujan scoprì una serie infinita che può essere usata per generare un'approssimazione di* $1/\pi$*:*

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Scrivete una funzione di nome `stima_pi` *che utilizzi questa formula per calcolare e restituire una stima di* π*. Deve usare un ciclo* `while` *per calcolare gli elementi della sommatoria, fino a quando l'ultimo termine è più piccolo di* `1e-15` *(che è la notazione di Python per* 10^{-15}*). Controllate il risultato confrontandolo con* `math.pi`*.*

Soluzione: `http://thinkpython2.com/code/pi.py`*.*