Python per sopravvivere

Parte dell'opera «Informatica per sopravvivere»

Massimo Piai ⟨*pxam67 (@) virgilio·it*⟩

2006.02.01

Massimo Piai è un matematico appassionato di informatica, che ha trovato nel software libero e nella libertà delle informazioni l'unica possibilità di sviluppare tale passione. I suoi campi di interesse attuali sono la matematica e le scienze, la diffusione della Cultura Informatica, la didattica e la pedagogia.

Il presente lavoro è stato realizzato utilizzando Alml, il sistema di composizione SGML realizzato da Daniele Giacomini per la gestione dei suoi *Appunti di informatica libera*.

Indice generale

Introduzione

In questa parte verranno introdotti i principali elementi della sintassi e dell'idioma del linguaggio Python,[1] a partire da semplici esempi di programmazione. Il principale vantaggio di Python per un uso didattico è che la sintassi è molto semplice, tanto da far sembrare i programmi scritti in Python quasi come se fossero stati scritti in uno pseudolinguaggio.

Ma Python non è un linguaggio «giocattolo»: si tratta infatti di un ambiente per lo sviluppo rapido di prototipi e anche di applicazioni complesse, è dotato di un'ampia gamma di moduli per scopi specializzati (anche sviluppati da terze parti), è stato adattato da più soggetti ed a diverse piattaforme, è stato utilizzato per lo sviluppo di alcune «killer application», ad esempio Google. Si tratta inoltre di un linguaggio orientato agli oggetti fin dalla progettazione, il che può sembrare vantaggioso a qualcuno.

Come prerequisito per la lettura è necessario essere stati introdotti ai rudimenti della programmazione strutturata, mediante linguaggi specifici oppure pseudocodifica.

---

[1] **Python** software con licenza GPL-compatibile

Capitolo 1

Primo esempio di programma Python

Per cominciare a cogliere un'idea delle caratteristiche[1] di Python, si consideri il problema della somma di due numeri positivi espressa attraverso il concetto dell'incremento unitario: ***n***+***m*** equivale a incrementare ***m***, di un'unità, per ***n*** volte, oppure incrementare ***n*** per ***m*** volte. L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli; il listato 1.1 presenta la soluzione tramite pseudocodifica.

Listato 1.1. Somma ciclica (pseudocodifica).

```
SOMMA (X, Y)

    LOCAL Z INTEGER
    LOCAL I INTEGER

    Z := X
    FOR I := 1; I <= Y; I++
        Z++
    END FOR

    RETURN Z

END SOMMA
```

Tenendo presente che in Python l'incremento unitario della variabile '***variabile***'; si esprime con l'idioma

```
variabile += 1
```

il listato 1.2 traduce l'algoritmo in un programma Python completo e commentato.

Listato 1.2. Primo esempio in Python.

```
#!/usr/bin/python
##
## somma.py <x> <y>
## Somma esclusivamente valori positivi.
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# somma(<x>, <y>)
#
def somma(x, y):
  z = x
  for i in range(y):
    z += 1
  return z
#
# Inizio del programma.
#
```

```
x = int(sys.argv[1])
y = int(sys.argv[2])
z = somma(x, y)
print x, "+", y, "=", z
```

Si noti in particolare la compattezza del programma Python: solo dieci righe «utili», contro le sedici dell'equivalente programma in Perl (listato 1.3) e addirittura ventidue dell'equivalente programma in C (listato 1.4).

Listato 1.3. Somma ciclica (Perl).

```
#!/usr/bin/perl
##
## somma.pl <x> <y>
## Somma esclusivamente valori positivi.
##
#
# &somma (<x>, <y>)
#
sub somma
{
    local ($x) = $_[0];
    local ($y) = $_[1];
    #
    local ($z) = $x;
    local ($i);
    #
    for ($i = 1; $i <= $y; $i++)
      {
        $z++;
      }
    #
    return $z;
}
#
# Inizio del programma.
#
$x = $ARGV[0];
$y = $ARGV[1];
#
$z = &somma ($x, $y);
#
print "$x + $y = $z\n";
#
```

Listato 1.4. Somma ciclica (C).

```
/* ================================================================= */
/* somma <x> <y>                                                     */
/* Somma esclusivamente valori positivi.                             */
/* ================================================================= */

#include <stdio.h>

/* ================================================================= */
/* somma (<x>, <y>)                                                  */
/* ----------------------------------------------------------------- */
int somma (int x, int y)
{
    int z = x;
    int i;

    for (i = 1; i <= y; i++)
      {
        z++;
      };

    return z;
}

/* ================================================================= */
/* Inizio del programma.                                             */
/* ----------------------------------------------------------------- */
int main (int argc, char *argv[])
{
    int x;
    int y;
    int z;

    /* Converte le stringhe ottenute dalla riga di comando in
       numeri interi e li assegna alle variabili x e y.           */
    sscanf (argv[1], "%d", &x);
    sscanf (argv[2], "%d", &y);

    z = somma (x, y);

    printf ("%d + %d = %d\n", x, y, z);

    return 0;
}
```


[1] Salvo diverso avviso, nel seguito si farà riferimento alla versione 2.3 del linguaggio.

Capitolo 2

Studio dettagliato del precedente esempio

Si procede ora ad analizzare il testo del semplice programma Python visto in precedenza (listato 1.2).

La riga

```
#!/usr/bin/python
...
```

come di consueto indica che quanto segue deve essere considerata una successione di comandi da «dare in pasto» all'interprete indicato, in questo caso '**`python`**' ossia l'eseguibile corrispondente a Python.

Le righe successive che inizino con il carattere cancelletto, '`#`', vanno considerati dei commenti ad uso del lettore e vengono ignorate dall'interprete.

La riga

```
...
import sys
...
```

serve a importare nel programma quanto contenuto (definizioni di funzioni, variabili, eccetera) nel ***modulo*** '**`sys`**', il quale è parte della libreria standard di Python.

Per accedere alle componenti del modulo si ricorre alla ***notazione puntata***, come si vedrà nel seguito.

La riga

```
...
def somma(x, y):
...
```

costituisce l'intestazione di una funzione, mediante la quale si definisce la funzione stessa, in questo caso '**`somma`**'.

Fra parentesi vanno indicati i parametri formali della funzione; l'intestazione termina obbligatoriamente con i due punti ('`:`').

Il blocco

```
...
  z = x
  for i in range(y):
    z += 1
  return z
...
```

costituisce il corpo della funzione '**somma**'.

Come si può notare l'appartenenza di un blocco al corpo di una funzione è stabilita dall'allineamento delle righe successive: la prima riga del corpo deve rientrare a destra di almeno uno spazio rispetto all'intestazione, e le righe del blocco devono essere allineate con la prima; il corpo della funzione termina con la prima riga che sia allineata con l'intestazione oppure rientri a sinistra rispetto a questa. Questo sistema di rientri costituisce la convenzione mediante cui in Python si raggruppano le righe in blocchi e si indica il controllo di un istruzione su una riga o blocco; in generale lo schema è:

```
istruzione_che_controlla:
    istruzione_controllata
    istruzione_controllata
    istruzione_controllata
    ...

istruzione_non_controllata
```

Si noti in particolare la presenza del carattere '**:**' nella riga di controllo.

La riga

```
...
  z = x
...
```

rappresenta un'istruzione di assegnamento alla variabile locale '**z**'.

La riga

```
...
  for i in range(y):
...
```

è l'istruzione di controllo di un ciclo enumerativo.

Si tratta di un tipico idioma Python: il costrutto

```
for variabile in lista:
  ...
```

indica che durante l'esecuzione del ciclo la variabile assume in successione i valori di una ***lista***, quindi il ciclo ha termine; l'idioma

```
range(valore)
```

restituisce una lista di valori da 0 a ***valore***−1.

La riga

```
...
    z += 1
...
```

è l'idioma Python che descrive l'incremento unitario di una variabile.

La riga

```
...
  return z
...
```

permette alla funzione '**somma**' di restituire il valore della variabile locale '**z**' al chiamante.

Dalla riga

```
...
x = int(sys.argv[1])
...
```

inizia il corpo principale del programma; mediante la notazione '**sys.argv**' si accede alla variabile '**argv**' definita nel modulo '**sys**'; la variabile '**argv**' contiene una lista di valori corrispondenti rispettivamente al nome del programma eseguito ('**sys.argv[0]**') e agli argomenti forniti allo stesso sulla linea di comando ('**sys.argv[1]**', '**sys.argv[2]**', eccetera). Poiché gli argomenti vengono passati al programma sottoforma di stringhe, è necessario convertirli nel caso in cui debbano essere intesi diversamente, per esempio come valori numerici interi; a tal fine si utilizza proprio la notazione

```
int(valore_non_intero)
```

la quale restituisce per l'appunto il valore intero corrispondente all'argomento fornito (sempre che ciò sia possibile); il valore ottenuto viene poi assegnato alla variabile globale '**x**'. A questo punto il significato della riga successiva dovrebbe risultare evidente.

La riga

```
...
z = somma(x, y)
...
```

rappresenta la chiamata della funzione '**somma**' precedentemente definita, con parametri '**x**' e '**y**', e successivamente l'assegnamento del valore restituito alla funzione alla variabile globale '**z**'.

Infine, la riga

```
...
print x, "+", y, "=", z
...
```

serve a inviare allo standard output (ossia generalmente lo schermo del terminale) i risultati

dell'elaborazione; come si vede, la parola chiave '**`print`**' può essere seguita da più argomenti, variabili o costanti di tipo diverso, separati dal carattere virgola ('**`,`**').

Un modo alternativo per generare lo stesso output è quello di utilizzare l'***operatore di formato*** ('**`%`**'), la cui sintassi rivela un evidente eredità dal linguaggio C:

```
print "%d + %d = %d" % (x, y, z)
```

Per concludere, ecco un esempio di esecuzione del programma Python da riga di comando:

```
$ ls -l somma.py [ Invio ]

  -rw-r--r--    1 max2     max2          349 2004-09-19 22:06 somma.py


$ chmod +x somma.py [ Invio ]

$ ls -l somma.py [ Invio ]

  -rwxr-xr-x    1 max2     max2          349 2004-09-19 22:06 somma.py


$ ./somma.py 23 34 [ Invio ]

  23 + 34 = 57


$
```

Capitolo 3

Ulteriori esempi di programmi Python

Nelle sezioni seguenti sono descritti alcuni problemi elementari attraverso cui si insegnano le tecniche di programmazione ai principianti. Assieme ai problemi vengono proposte le soluzioni in forma di programma Python. Per le soluzioni in forma di pseudocodifica si rimanda agli *Appunti di informatica libera* di Daniele Giacomini.

3.1 Problemi elementari di programmazione

3.1.1 Somma attraverso incremento unitario: versione con ciclo iterativo

```
...
def somma(x, y):
  z = x
  i = 1
  while i <= y:
    z += 1
    i += 1
  return z
...
```

3.1.2 Moltiplicazione di due numeri positivi attraverso la somma

La moltiplicazione di due numeri positivi, può essere espressa attraverso il concetto della somma: ***n**m*** equivale a sommare ***m*** volte ***n***, oppure ***n*** volte ***m***. L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli.

Listato 3.2. Moltiplicazione ciclica.

```
#!/usr/bin/python
##
## moltiplica.py <x> <y>
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# moltiplica(<x>, <y>)
#
def moltiplica(x, y):
  z = 0
  for i in range(y):
    z = z+x
  return z
#
# Inizio del programma.
#
x = int(sys.argv[1])
y = int(sys.argv[2])
z = moltiplica(x, y)
print x, "*", y, "=", z
```

Nel listato 3.2 viene mostrata una soluzione per mezzo di un ciclo enumerativo. Il ciclo viene ripetuto '**y**' volte, incrementando la variabile '**z**' del valore di '**x**'. Alla fine, '**z**' contiene il ri-

sultato del prodotto di '**x**' per '**y**'. Il frammento seguente mostra invece la traduzione del ciclo enumerativo in un ciclo iterativo:

```
...
def moltiplica(x, y):
  z = 0
  i = 1
  while i <= y:
    z = z+x
    i += 1
  return z
...
```

3.1.3 Divisione intera tra due numeri positivi

La divisione di due numeri positivi, può essere espressa attraverso la sottrazione: ***n***/***m*** equivale a sottrarre ***m*** da ***n*** fino a quando ***n*** diventa inferiore di ***m***. Il numero di volte in cui tale sottrazione ha luogo, è il risultato della divisione.

Listato 3.4. Divisione ciclica.

```
#!/usr/bin/python
##
## dividi.py <x> <y>
## Divide esclusivamente valori positivi.
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# dividi(<x>, <y>)
#
def dividi(x, y):
  z = 0
  i = x
  while i >= y:
    i = i-y
    z += 1
  return z
#
# Inizio del programma.
#
x = int(sys.argv[1])
y = int(sys.argv[2])
z = dividi(x, y)
print "Divisione intera -> %d/%d = %d" % (x, y, z)
```

Il listato 3.4 realizza l'algoritmo; si noti anche l'uso dell'operatore di formato.

3.1.4 Elevamento a potenza

L'elevamento a potenza, utilizzando numeri positivi, può essere espresso attraverso il concetto della moltiplicazione: ***n*****m*** equivale a moltiplicare ***m*** volte ***n*** per se stesso.

Listato 3.5. Potenza ciclica.

```
#!/usr/bin/python
##
## exp.py <x> <y>
## Eleva a potenza.
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# exp(<x>, <y>)
#
def exp(x, y):
  z = 1
  for i in range(y):
    z = z*x
  return z
#
# Inizio del programma.
#
x = int(sys.argv[1])
y = int(sys.argv[2])
z = exp(x, y)
print "%d ** %d = %d" % (x, y, z)
```

Nel listato 3.5 viene mostrata una soluzione per mezzo di un ciclo enumerativo. Il ciclo viene ripetuto '**y**' volte; ogni volta la variabile '**z**' viene moltiplicata per il valore di '**x**', a partire da '**1**'. Alla fine, '**z**' contiene il risultato dell'elevamento di '**x**' a '**y**'. Il frammento seguente mostra invece la traduzione del ciclo enumerativo in un ciclo iterativo:

```
...
def exp(x, y):
  z = 1
  i = 1
  while i <= y:
    z = z*x
    i += 1
  return z
...
```

Il frammento seguente mostra una soluzione ricorsiva:

```
...
def exp(x, y):
  if x == 0:
    return 0
```

```
  elif y == 0:
    return 1
  else:
    return x*exp(x, y-1)
...
```

3.1.5 Radice quadrata

Il calcolo della parte intera della radice quadrata di un numero si può fare per tentativi, partendo da 1, eseguendo il quadrato fino a quando il risultato è minore o uguale al valore di partenza di cui si calcola la radice.

Il listato 3.8 realizza l'algoritmo.

Listato 3.8. Radice quadrata ciclica.

```
#!/usr/bin/python
##
## radice.py <x>
## Radice quadrata.
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# radice(<x>)
#
def radice(x):
  z = 0
  t = 0
  while True:
    t = z*z
    if t > x:
      #
      # E' stato superato il valore massimo.
      #
      z -= 1
      return z
    z += 1
  #
  # Teoricamente, non dovrebbe mai arrivare qui.
  #
#
# Inizio del programma.
#
x = int(sys.argv[1])
z = radice(x)
print "radq(%d) = %d" % (x, z)
```

3.1.6 Fattoriale

Il fattoriale è un valore che si calcola a partire da un numero positivo. Può essere espresso come il prodotto di ***n*** per il fattoriale di ***n***−1, quando ***n*** è maggiore di 1, mentre equivale a 1 quando ***n*** è uguale a 1. In pratica, ***n***! = ***n*** * (***n***−1) * (***n***−2)... *1.

Listato 3.9. Fattoriale.

```
#!/usr/bin/python
##
## fatt.py <x>
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# fatt(<x>)
#
def fatt(x):
  i = x-1
  while i > 0:
    x = x*i
    i -= 1
  return x
#
# Inizio del programma.
#
x = int(sys.argv[1])
fatt = fatt(x)
print "%d! = %d" % (x, fatt)
```

La soluzione mostrata nel listato 3.9 fa uso di un ciclo iterativo in cui l'indice '**i**', che inizialmente contiene il valore di '**x-1**', viene usato per essere moltiplicato al valore di '**x**', riducendolo ogni volta di un'unità. Quando '**i**' raggiunge lo '**0**', il ciclo termina e '**x**' contiene il valore del fattoriale. L'esempio seguente mostra invece una soluzione ricorsiva che dovrebbe risultare più intuitiva:

```
...
def fatt(x):
  if x == 1:
    return 1
  else:
    return x*fatt(x-1)
...
```

3.1.7 Massimo comune divisore

Il massimo comune divisore tra due numeri può essere ottenuto sottraendo a quello maggiore il valore di quello minore, fino a quando i due valori sono uguali. Quel valore è il massimo comune divisore.

Il listato 3.11 realizza l'algoritmo.

Listato 3.11. Massimo comune divisore.

```
#!/usr/bin/python
##
## mcd.py <x> <y>
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# mcd(<x>, <y>)
#
def mcd(x, y):
  while x != y:
    if x > y:
      x = x-y
    else:
      y = y-x
  return x
#
# Inizio del programma.
#
x = int(sys.argv[1])
y = int(sys.argv[2])
z = mcd(x, y)
print "Il massimo comune divisore di %d e %d e' %d" %  (x, y, z)
```

3.1.8 Numero primo

Un numero intero è numero primo quando non può essere diviso per un altro intero diverso dal numero stesso e da 1, generando un risultato intero.

Il listato 3.12 realizza l'algoritmo.

Listato 3.12. Numero primo.

```
#!/usr/bin/python
##
## primo.py <x>
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
```

```
# primo(<x>)
#
def primo(x):
  primo = True
  i = 2
  while i < x and primo:
    j = x/i
    j = x-(j*i)
    if j == 0:
      primo = False
    else:
      i += 1
  return primo
#
# Inizio del programma.
#
x = int(sys.argv[1])
if primo(x):
  print x, "e‘ un numero primo"
else:
  print x, "non e‘ un numero primo"
```

3.2 Scansione di array

Nelle sezioni seguenti sono descritti alcuni problemi legati alla scansione di array. Assieme ai problemi vengono proposte le soluzioni in forma di programmi Python.

3.2.1 Ricerca sequenziale

La ricerca di un elemento all'interno di un array disordinato può avvenire solo in modo sequenziale, cioè controllando uno per uno tutti gli elementi, fino a quando si trova la corrispondenza cercata. La tabella 3.13 presenta la descrizione delle variabili più importanti che appaiono nei programmi Python successivi.

Tabella 3.13. Ricerca sequenziale: variabili utilizzate.

Variabile	Descrizione
`lista`	È l'array su cui effettuare la ricerca.
`x`	È il valore cercato all'interno dell'array.
`a`	È l'indice inferiore dell'intervallo di array su cui si vuole effettuare la ricerca.
`z`	È l'indice superiore dell'intervallo di array su cui si vuole effettuare la ricerca.

Il listato 3.14 presenta un esempio di programma Python che risolve il problema in modo iterativo.

Listato 3.14. Ricerca sequenziale.

```
#!/usr/bin/python
##
## ricercaseq.py <elemento-cercato> <valore>...
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# ricercaseq(<lista>, <elemento>, <inizio>, <fine>)
#
def ricercaseq(lista, x, a, z):
  for i in range(a, z+1):
    if x == lista[i]:
      return i
  return -1
#
# Inizio del programma.
#
x = sys.argv[1]
lista = sys.argv[2:]
i = ricercaseq(lista, x, 0, len(lista)-1)
if i != -1:
  print "L'elemento", x, "si trova nella posizione", i
else:
  print "L'elemento", x, "non e' stato trovato"
```

L'algoritmo usato dovrebbe risultare abbastanza chiaro.

Per quanto concerne le caratteristiche del linguaggio usate, si noti che Python offre nativamente supporto per le liste, le quali sono strutture dati più complesse degli array, ma che ovviamente possono essere utilizzate per simularli, proprio come nell'esempio; le liste hanno gli indici che vanno da '**0**' a '**len(*lista*)-1**', ove la funzione '**len**' restituisce il numero di elementi della lista '***lista***'; per accedere ai singoli elementi di una lista si usa la notazione

***lista*[*indice*]**

inoltre è possibile estrarre una sottolista mediante la notazione

***lista*[*indice_iniziale*:*indice_finale*]**

eventualmente tralasciando uno dei due indici (o anche entrambi), che quindi assumono valori predefiniti (rispettivamente '**0**' e '**len(*lista*)**'; si tenga presente che in tale notazione l'elemento corrispondente all'indice finale si intende da escludersi.

Esiste anche una soluzione ricorsiva che viene mostrata nel frammento seguente:

```
...
def ricercaseq(lista, x, a, z):
  if a > z:
```

```
    return -1
  elif x == lista[a]:
    return a
  else:
    return ricercaseq(lista, x, a+1, z)
...
```

3.2.2 Ricerca binaria

La ricerca di un elemento all'interno di un array ordinato può avvenire individuando un elemento centrale: se questo corrisponde all'elemento cercato, la ricerca è terminata, altrimenti si ripete nella parte di array precedente o successiva all'elemento, a seconda del suo valore e del tipo di ordinamento esistente.

Il problema posto in questi termini è ricorsivo. Il programma mostrato nel listato 3.16 utilizza le stesse variabili già descritte per la ricerca sequenziale.

Listato 3.16. Ricerca binaria.

```
#!/usr/bin/python
##
## ricercabin.py <elemento-cercato> <valore>...
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# ricercabin(<lista>, <elemento>, <inizio>, <fine>)
#
def ricercabin(lista, x, a, z):
  #
  # Determina l'elemento centrale.
  #
  m = (a+z)/2
  if m < a:
    #
    # Non restano elementi da controllare: l'elemento cercato
    # non c'e'.
    #
    return -1
  elif x < lista[m]:
    #
    # Si ripete la ricerca nella parte inferiore.
    #
    return ricercabin(lista, x, a, m-1)
  elif x > lista[m]:
    #
    # Si ripete la ricerca nella parte superiore.
    #
    return ricercabin(lista, x, m+1, z)
  else:
    #
```

```
    # m rappresenta l'indice dell'elemento cercato.
    #
    return m
#
# Inizio del programma.
#
x = sys.argv[1]
lista = sys.argv[2:]
i = ricercabin(lista, x, 0, len(lista)-1)
if i != -1:
  print "L'elemento", x, "si trova nella posizione", i
else:
  print "L'elemento", x, "non e' stato trovato"
```

3.3 Problemi classici di programmazione

Nelle sezioni seguenti sono descritti alcuni problemi classici attraverso cui si insegnano le tecniche di programmazione. Assieme ai problemi vengono proposte le soluzioni in forma di programma Python.

3.3.1 Bubblesort

Il Bubblesort è un algoritmo relativamente semplice per l'ordinamento di un array, in cui ogni scansione trova il valore giusto per l'elemento iniziale dell'array stesso. Una volta trovata la collocazione di un elemento, si ripete la scansione per il segmento rimanente di array, in modo da collocare un altro valore. Il testo del programma dovrebbe chiarire il meccanismo. La tabella 3.17 presenta la descrizione delle variabili più importanti utilizzate dal programma.

Tabella 3.17. Bubblesort: variabili utilizzate.

Variabile	Descrizione
`lista`	È l'array da ordinare.
`a`	È l'indice inferiore del segmento di array da ordinare.
`z`	È l'indice superiore del segmento di array da ordinare.

Nel listato 3.18 viene mostrata una soluzione iterativa.

Listato 3.18. Bubblesort.

```
#!/usr/bin/python
##
## bsort.py <valore>...
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# bsort(<lista>, <inizio>, <fine>)
```

```
#
def bsort(lista, a, z):
  #
  # Inizia il ciclo di scansione dell'array.
  #
  for j in range(a, z):
    #
    # Scansione interna dell'array per collocare nella posizione
    # j l'elemento giusto.
    #
    for k in range(j+1, z+1):
      if lista[k] < lista[j]:
        #
        # Scambia i valori
        #
        scambio = lista[k]
        lista[k] = lista[j]
        lista[j] = scambio
#
# Inizio del programma.
#
lista = sys.argv[1:]
bsort(lista, 0, len(lista)-1)
for elemento in lista:
  print elemento,
```

Vale la pena di osservare nelle ultime due righe alcuni aspetti idiomatici di Python: avendo a disposizione una lista è possibile utilizzare i suoi elementi come indici di un ciclo enumerativo utilizzando la consueta parola chiave '**`in`**'; inoltre, se si intende che l'output non vada a capo ma prosegua sulla medesima riga, si può usare il carattere vigola ('**`,`**') in coda all'istruzione '**`print`**'.

Segue la funzione '**`bsort`**' in versione ricorsiva:

```
...
def bsort(lista, a, z):
  if a < z:
    #
    # Scansione interna dell'array per collocare nella posizione
    # a l'elemento giusto.
    #
    for k in range(a+1, z+1):
      if lista[k] < lista[a]:
        #
        # Scambia i valori
        #
        scambio = lista[k]
        lista[k] = lista[a]
        lista[a] = scambio
    bsort(lista, a+1, z)
...
```

3.3.1.1 Alcune osservazioni aggiuntive

Si tenga presente che in questa sezione, come in quelle sugli algoritmi di ricerca, l'istruzione di lettura degli argomenti della linea di comando non prevede la trasformazione da stringa a intero; questo perché, in generale, ci si può aspettare che una lista sia costituita da elementi non numerici.

Tuttavia, si consideri la seguente situazione:

```
$ ./bsort.py 3 5 8 2 9 4512 7 67431 3 6 3 [ Invio ]

  2 3 3 3 4512 5 6 67431 7 8 9
```

Ovviamente non è quello che ci si aspetta; in realtà, l'output è comprensibile se si tiene presente che gli argomenti vengono trattati come stringhe, perciò la lista viene ordinata secondo l'ordine lessicografico basato sul codice ASCII (in pratica l'ordine alfabetico esteso).

Se si vuole correggere il comportamanto del programma, è possibile sostituire la riga

```
...
lista = sys.argv[1:]
...
```

con la riga

```
...
lista = map(int, sys.argv[1:])
...
```

Con l'occasione, si noti un'ulteriore interessante aspetto idiomatico di Python: è possibile applicare una funzione per trasformare tutti i membri di una lista utilizzando il costrutto:

```
map(funzione, lista)
```

Si invita il lettore interessato a consultare la documentazione di Python per ulteriori aspetti riguardanti la gestione delle liste.

Ecco il comportamento del programma modificato:

```
$ ./bsort.py 3 5 8 2 9 4512 7 67431 3 6 3 [ Invio ]

  2 3 3 3 5 6 7 8 9 4512 67431
```

Vale la pena notare che, così modificato, il programma non funziona se gli argomenti passatigli non possono essere interpretati come numeri interi.

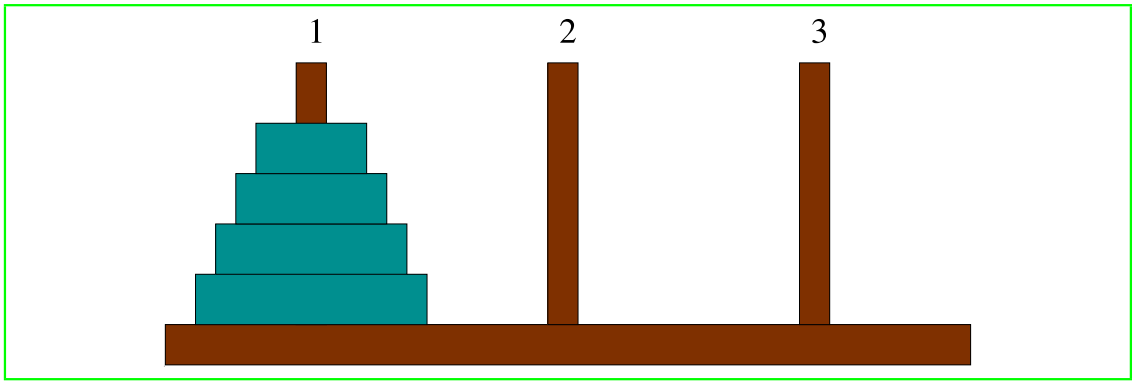
3.3.2 Torre di Hanoi

La torre di Hanoi è un gioco antico: si compone di tre pioli identici conficcati verticalmente su una tavola e di una serie di anelli di larghezze differenti. Gli anelli sono più precisamente dei dischi con un foro centrale che permette loro di essere infilati nei pioli.

Il gioco inizia con tutti gli anelli collocati in un solo piolo, in ordine, in modo che in basso ci sia l'anello più largo e in alto quello più stretto. Si deve riuscire a spostare tutta la pila di anelli in un dato piolo muovendo un anello alla volta e senza mai collocare un anello più grande sopra uno più piccolo.

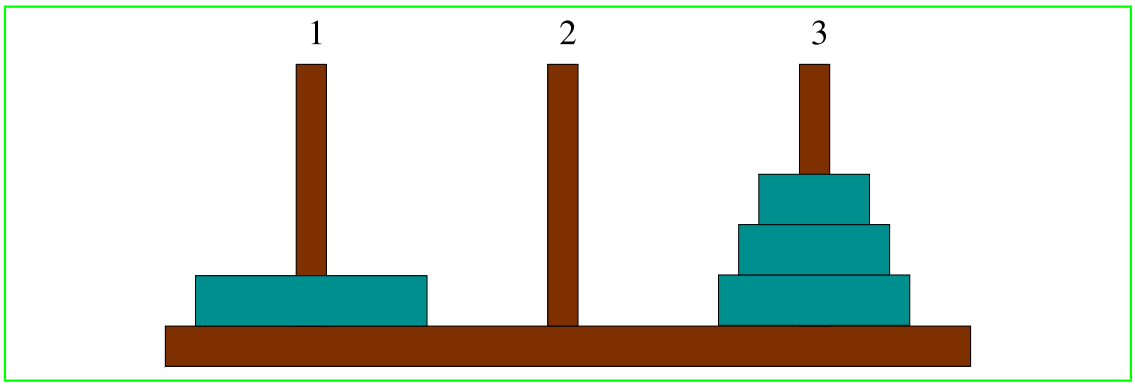
La figura 3.24 illustra la situazione iniziale della torre di Hanoi all'inizio del gioco.

Figura 3.24. Situazione iniziale della torre di Hanoi all'inizio del gioco.



Nella figura 3.24 gli anelli appaiono inseriti sul piolo 1; si supponga che questi debbano essere spostati sul piolo 2. Si può immaginare che tutti gli anelli, meno l'ultimo, possano essere spostati in qualche modo corretto, dal piolo 1 al piolo 3, come nella situazione della figura 3.25.

Figura 3.25. Situazione dopo avere spostato $n-1$ anelli.



A questo punto si può spostare l'ultimo anello rimasto (l'n-esimo), dal piolo 1 al piolo 2; quindi, come prima, si può spostare in qualche modo il gruppo di anelli posizionati attualmente nel piolo 3, in modo che finiscano nel piolo 2 sopra l'anello più grande.

Pensando in questo modo, l'algoritmo risolutivo del problema deve essere ricorsivo e potrebbe essere gestito da un'unica funzione che può essere chiamata opportunamente '**`hanoi`**', i cui parametri sono presentati nella tabella 3.26.

Tabella 3.26. Parametri sella funzione '**hanoi**'.

Parametro	Descrizione
`n`	È la dimensione della torre espressa in numero di anelli: gli anelli sono numerati da 1 a n.
`p1`	È il numero del piolo su cui si trova inizialmente la pila di n anelli.
`p2`	È il numero del piolo su cui deve essere spostata la pila di anelli.
`6-p1-p2`	È il numero dell'altro piolo. Funziona così se i pioli sono numerati da 1 a 3.

Il listato 3.27 presenta il programma Python con funzione ricorsiva per la soluzione del problema.

Listato 3.27. Torre di Hanoi.

```
#!/usr/bin/python
##
## hanoi.py <n-anelli> <piolo-iniziale> <piolo-finale>
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# hanoi(<n-anelli>, <piolo-iniziale>, <piolo-finale>)
#
def hanoi(n, p1, p2):
  if n > 0:
    hanoi(n-1, p1, 6-p1-p2)
    print "Muovi l'anello %d dal piolo %d al piolo %d" % (n, p1, p2)
    hanoi(n-1, 6-p1-p2, p2)
##
## Inizio del programma.
##
(n, p1, p2) = map(int, sys.argv[1:4])
hanoi(n, p1, p2)
```

Si colga l'occasione per osservare un'ulteriore aspetto idiomatico di Python, ossia la possibilità di assegnare «in parallelo» gli elementi di una lista a più variabili (o, come si dice in gergo Python, una ***tupla***) con una singola istruzione di assegnamento.

Tornando al problema, ecco l'analisi dell'algoritmo risolutivo: se '**n**', il numero degli anelli da spostare, è minore di '**1**', non si deve compiere alcuna azione. Se '**n**' è uguale a '**1**', le istruzioni controllate dalla struttura condizionale vengono eseguite, ma nessuna delle chiamate ricorsive fa alcunché, dato che '**n-1**' è pari a '**0**'. In questo caso, supponendo che '**n**' sia uguale a '**1**', che '**p1**' sia pari a '**1**' e '**p2**' pari a '**2**', il risultato è semplicemente:

```
Muovi l'anello 1 dal piolo 1 al piolo 2
```

Il risultato è quindi corretto per una pila iniziale consistente di un solo anello.

Se '**n**' è uguale a '**2**', la prima chiamata ricorsiva sposta un anello ('**n-1**' = '**1**') dal piolo 1 al piolo 3 (ancora assumendo che i due anelli debbano essere spostati dal primo al terzo piolo) e si sa che questa è la mossa corretta. Quindi viene stampato il messaggio che dichiara lo spostamento del secondo piolo (l'***n***-esimo) dalla posizione 1 alla posizione 2. Infine, la seconda chiamata ricorsiva si occupa di spostare l'anello collocato precedentemente nel terzo piolo, nel secondo, sopra a quello che si trova già nella posizione finale corretta.

In pratica, nel caso di due anelli che devono essere spostati dal primo al secondo piolo, appaiono i tre messaggi seguenti:

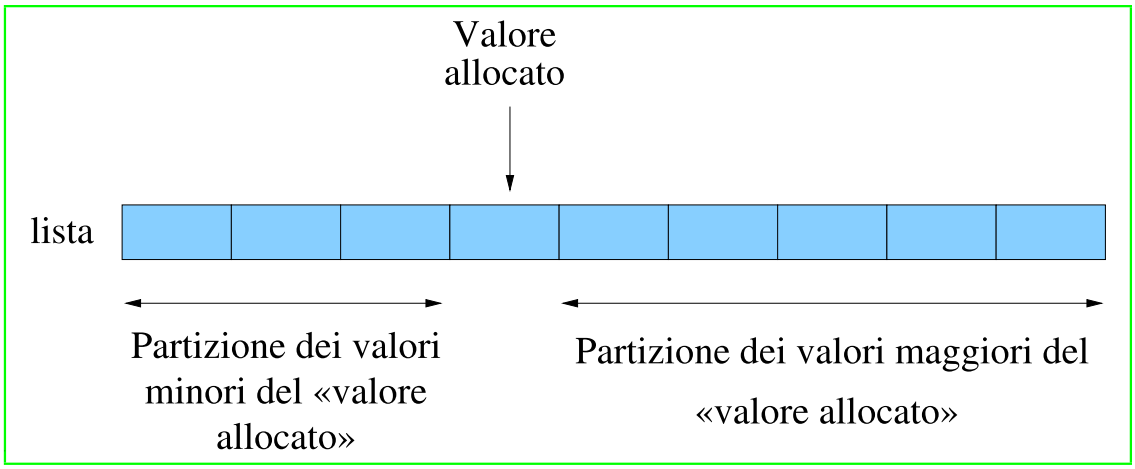
```
Muovi l'anello 1 dal piolo 1 al piolo 3
Muovi l'anello 2 dal piolo 1 al piolo 2
Muovi l'anello 1 dal piolo 3 al piolo 2
```

Nello stesso modo si potrebbe dimostrare il funzionamento per un numero maggiore di anelli.

3.3.3 Quicksort (ordinamento non decrescente)

L'ordinamento degli elementi di un array è un problema tipico che si può risolvere in tanti modi. Il Quicksort è un algoritmo sofisticato, ottimo per lo studio della gestione degli array, oltre che per quello della ricorsione. Il concetto fondamentale di questo tipo di algoritmo è rappresentato dalla figura 3.30.

Figura 3.30. Il concetto base dell'algoritmo del Quicksort: suddivisione dell'array in due gruppi disordinati, separati da un valore piazzato correttamente nel suo posto rispetto all'ordinamento.



Una sola scansione dell'array è sufficiente per collocare definitivamente un elemento (per esempio il primo) nella sua destinazione finale e allo stesso tempo per lasciare tutti gli elementi con un valore inferiore a quello da una parte, anche se disordinati, e tutti quelli con un valore maggiore, dall'altra.

In questo modo, attraverso delle chiamate ricorsive, è possibile elaborare i due segmenti dell'array rimasti da riordinare.

L'algoritmo può essere descritto grossolanamente come:

1. localizzazione della collocazione finale del primo valore, separando in questo modo i valori;
2. ordinamento del segmento precedente all'elemento collocato definitivamente;

3. ordinamento del segmento successivo all'elemento collocato definitivamente.

Viene qui indicata con '**`part`**' la funzione che esegue la scansione dell'array, o di un suo segmento, per determinare la collocazione finale (indice '**`cf`**') del primo elemento (dell'array o del segmento in questione).

Sia '**`lista`**' l'array da ordinare. Il primo elemento da collocare corrisponde inizialmente a '**`lista[a]`**' e il segmento di array su cui intervenire corrisponde a '**`lista[a:z+1]`**' (cioè a tutti gli elementi che vanno dall'indice '**`a`**' all'indice '**`z`**').

Alla fine della prima scansione, l'indice '**`cf`**' rappresenta la posizione in cui occorre spostare il primo elemento, cioè '**`lista[a]`**'. In pratica, '**`lista[a]`**' e '**`lista[cf]`**' vengono scambiati.

Durante la scansione che serve a determinare la collocazione finale del primo elemento, '**`part`**' deve occuparsi di spostare gli elementi prima o dopo quella posizione, in funzione del loro valore, in modo che alla fine quelli inferiori o uguali a quello dell'elemento da collocare si trovino nella parte inferiore e gli altri dall'altra. In pratica, alla fine della prima scansione, gli elementi contenuti in '**`lista[a:cf]`**' devono contenere valori inferiori o uguali a '**`lista[cf]`**', mentre quelli contenuti in '**`lista[cf+1:z+1]`**' devono contenere valori superiori.

Indichiamo con '**`qsort`**' la funzione che esegue il compito complessivo di ordinare l'array. Il suo lavoro consisterebbe nel chiamare '**`part`**' per collocare il primo elemento, continuando poi con la chiamata ricorsiva di se stessa per la parte di array precedente all'elemento collocato e infine alla chiamata ricorsiva per la parte restante di array.

Assumendo che '**`part`**' e le chiamate ricorsive di '**`qsort`**' svolgano il loro compito correttamente, si potrebbe fare un'analisi informale dicendo che se l'indice '**`z`**' non è maggiore di '**`a`**', allora c'è un elemento (o nessuno) all'interno di '**`lista[a:z+1]`**' e inoltre, '**`lista[a:z+1]`**' è già nel suo stato finale. Se '**`z`**' è maggiore di '**`a`**', allora (per assunzione) '**`part`**' ripartisce correttamente '**`lista[a:z+1]`**'. L'ordinamento separato dei due segmenti (per assunzione eseguito correttamente dalle chiamate ricorsive) completa l'ordinamento di '**`lista[a:z+1]`**'.

Le figure 3.31 e 3.32 mostrano due fasi della scansione effettuata da part all'interno dell'array o del segmento che gli viene fornito.

Figura 3.31. La scansione dell'array da parte di '**`part`**' avviene portando in avanti l'indice '**`i`**' e portando indietro l'indice '**`cf`**'. Quando l'indice '**`i`**' localizza un elemento che contiene un valore maggiore di '**`lista[a]`**' e l'indice '**`cf`**' localizza un elemento che contiene un valore inferiore o uguale a '**`lista[a]`**', gli elementi cui questi indici fanno riferimento vengono scambiati, quindi il processo di avvicinamento tra '**`i`**' e '**`cf`**' continua.

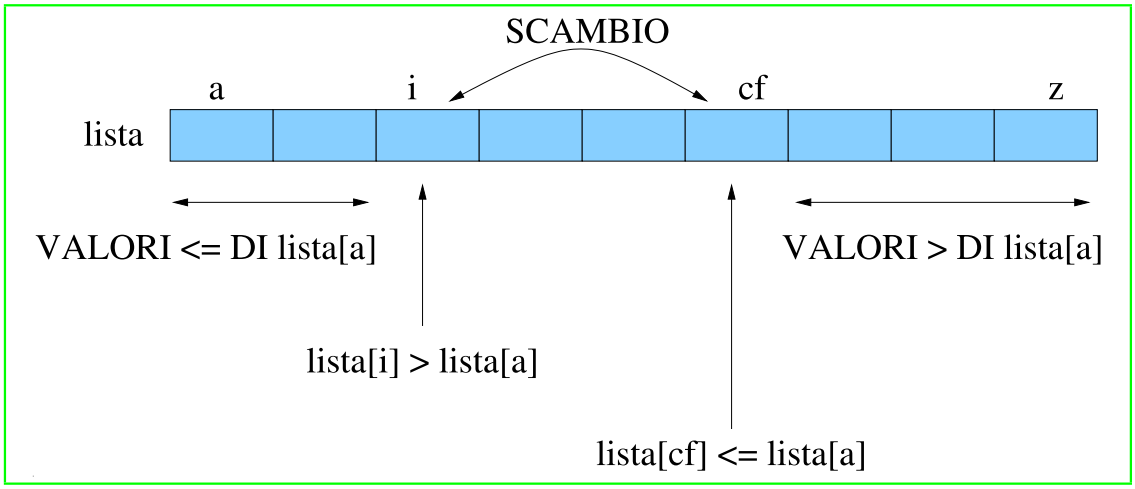
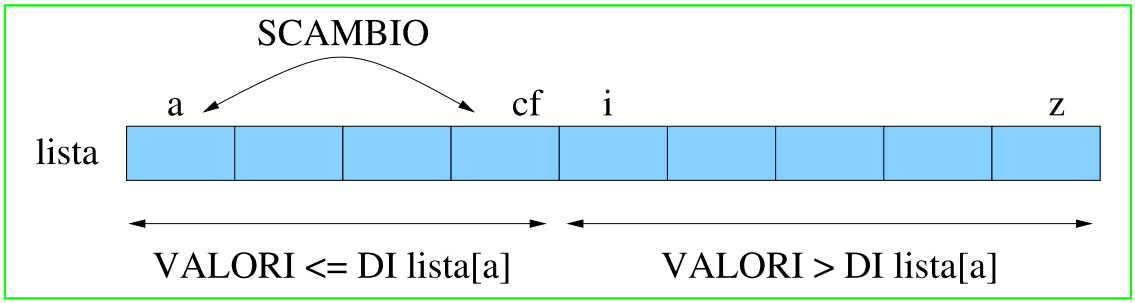


Figura 3.32. Quando la scansione è giunta al termine, quello che resta da fare è scambiare l'elemento '**lista[a]**' con '**lista[cf]**'.



In pratica, l'indice '**i**', iniziando dal valore '**a+1**', viene spostato verso destra fino a che viene trovato un elemento maggiore di '**lista[a]**', quindi è l'indice '**cf**' a essere spostato verso sinistra, iniziando dalla stessa posizione di '**z**', fino a che viene incontrato un elemento minore o uguale a '**lista[a]**'. Questi elementi vengono scambiati e lo spostamento di '**i**' e '**cf**' riprende. Ciò prosegue fino a che '**i**' e '**cf**' si incontrano, momento in cui '**lista[a:z+1]**' è stata ripartita e '**cf**' rappresenta la collocazione finale per l'elemento '**lista[l]**'.

La tabella 3.33 riassume la descrizione delle variabili utilizzate.

Tabella 3.33. Quicksort: variabili utilizzate.

Variabile	Descrizione
`lista`	L'array da ordinare in modo crescente.
`a`	L'indice inferiore del segmento di array da ordinare.
`z`	L'indice superiore del segmento di array da ordinare.
`cf`	Sta per «collocazione finale» ed è l'indice che cerca e trova la posizione giusta di '**lista[l]**' nell'array.
`i`	È l'indice che insieme a '**cf**' serve a ripartire l'array.

Il listato 3.34 presenta il programma Python che include le due funzioni.

Listato 3.34. Quicksort.

```
#!/usr/bin/python
##
## qsort.py <valore>...
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# part(<lista>, <inizio>, <fine>)
#
def part(lista, a, z):
  #
  # Viene preparata una variabile che serve per scambiare due valori.
  #
  scambio = 0
  #
  # Si assume che a sia inferiore a z.
```

```
  #
  i = a+1
  cf = z
  #
  # Inizia il ciclo di scansione dell'array.
  #
  while True:
    while True:
      #
      # Sposta i a destra.
      #
      if lista[i] > lista[a] or i >= cf:
        break
      else:
        i += 1
    while True:
      #
      # Sposta cf a sinistra.
      #
      if lista[cf] <= lista[a]:
        break
      else:
        cf -= 1
    if cf <= i:
      #
      # E' avvenuto l'incontro tra i e cf.
      #
      break
    else:
      #
      # Vengono scambiati i valori.
      #
      scambio = lista[cf]
      lista[cf] = lista[i]
      lista[i] = scambio
      i += 1
      cf -= 1
  #
  # A questo punto lista[a:z+1] e' stata ripartita e cf e' la
  # collocazione di lista[a].
  #
  scambio = lista[cf]
  lista[cf] = lista[a]
  lista[a] = scambio
  #
  # A questo punto, lista[cf] e' un elemento (un valore) nella
  # giusta posizione.
  #
  return cf
#
# quicksort(<lista>, <inizio>, <fine>)
#
def quicksort(lista, a, z):
```

```
  #
  # Viene preparata la variabile cf.
  #
  cf = 0
  #
  if z > a:
    cf = part(lista, a, z)
    quicksort(lista, a, cf-1)
    quicksort(lista, cf+1, z)
##
## Inizio del programma.
##
lista = sys.argv[1:]
#
quicksort(lista, 0, len(lista)-1);
#
for elemento in lista:
  print elemento,
#
```

Vale la pena di osservare che l'array viene indicato nelle chiamate in modo che alla funzione sia inviato un riferimento a quello originale, perché le variazioni fatte all'interno delle funzioni devono riflettersi sull'array originale.

3.3.3.1 Alcune osservazioni aggiuntive

In Python, non è necessario alcun particolare accorgimento sintattico per garantire questo comportamento: infatti, le liste costituiscono un tipo di dato ***mutabile*** (secondo la terminologia Python), alla stessa stregua di altri tipi come i ***dizionari*** (per i quali si rinvia il lettore alla documentazione del linguaggio[1]); quando un oggetto mutabile viene passato come argomento a una funzione, avviene un assegnamento al corrispondente parametro formale; l'assegnamento di un oggetto mutabile a una variabile è realizzato in Python mediante il cosiddetto meccanismo dell'***aliasing***: in pratica la nuova variabile coincide in tutto e per tutto con l'oggetto assegnato[2], e in particolare se quest'ultimo cambia valore tale cambiamento si riflette sulla nuova variabile. Pertanto, le variazioni fatte sui parametri formali all'interno di funzioni che ricevono come argomenti delle liste, si riflettono sulle liste originali.

Ecco un esempio che può aiutare a chiarire la questione (si tratta di una sessione interattiva dell'interprete Python):

```
$ python [ Invio ]

  Python 2.3.4 (#2, Jul  5 2004, 09:15:05)
  [GCC 3.3.4 (Debian 1:3.3.4-2)] on linux2
  Type "help", "copyright", "credits" or "license" for more information.


>>> lista = [1, 3, 7, 0, 10] [ Invio ]

>>> altra_lista = lista [ Invio ]

>>> altra_lista[3] = 1000 [ Invio ]
```

```
>>> print lista [ Invio ]

  [1, 3, 7, 1000, 10]


>>> [ Ctrl d ]

$
```

3.3.4 Permutazioni

La permutazione è lo scambio di un gruppo di elementi posti in sequenza. Il problema che si vuole analizzare è la ricerca di tutte le permutazioni possibili di un dato gruppo di elementi.

Se ci sono ***n*** elementi in un array, allora alcune delle permutazioni si possono ottenere bloccando l'***n***-esimo elemento e generando tutte le permutazioni dei primi elementi. Quindi l'***n***-esimo elemento può essere scambiato con uno dei primi ***n***−1, ripetendo poi la fase precedente. Questa operazione deve essere ripetuta finché ognuno degli ***n*** elementi originali è stato usato nell'***n***-esima posizione.

Tabella 3.37. Permutazioni: variabili utilizzate.

Variabile	Descrizione
`lista`	L'array da permutare.
`a`	L'indice inferiore del segmento di array da permutare.
`z`	L'indice superiore del segmento di array da permutare.
`k`	È l'indice che serve a scambiare gli elementi.

Il listato 3.38 presenta il programma Python, le cui variabili più importanti sono descritte nella tabella 3.37.

Listato 3.38. Permutazioni.

```
#!/usr/bin/python
##
## permuta.py <valore>...
##
#
# Importa il modulo sys, per usare sys.argv
#
import sys
#
# permuta(<lista>, <inizio>, <fine>)
#
def permuta(lista, a, z):
  #
  # Se il segmento di array contiene almeno due elementi, si
  # procede.
  #
  if z-a >= 1:
    #
```

```
    # Inizia un ciclo di scambi tra l'ultimo elemento e uno degli
    # altri contenuti nel segmento di array.
    #
    for k in range (z, a-1, -1):
      #
      # Scambia i valori.
      #
      scambio = lista[k]
      lista[k] = lista[z]
      lista[z] = scambio
      #
      # Esegue una chiamata ricorsiva per permutare un segmento
      # piu' piccolo dell'array.
      #
      permuta(lista, a, z-1)
      #
      # Scambia i valori.
      #
      scambio = lista[k]
      lista[k] = lista[z]
      lista[z] = scambio
  else:
    #
    # Visualizza la situazione attuale dell'array.
    #
    for elemento in lista:
      print elemento,
    print
##
## Inizio del programma.
##
lista = sys.argv[1:]
#
permuta(lista, 0, len(lista)-1)
#
```

3.3.4.1 Alcune osservazioni aggiuntive

Si colga l'occasione per notare un paio di aspetti idiomatici di Python. Per prima cosa, è possibile usare la funzione '**range**' per costruire un ciclo enumerativo decrescente, poiché '**range**' accetta un terzo argomento che in pratica rappresenta il passo con cui viene generata la successione di valori che popola la lista; ecco alcuni esempi:

```
$ python [ Invio ]

  Python 2.3.4 (#2, Jul  5 2004, 09:15:05)
  [GCC 3.3.4 (Debian 1:3.3.4-2)] on linux2
  Type "help", "copyright", "credits" or "license" for more information.


>>> range(10) [ Invio ]
```

```
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(1, 11) [ Invio ]

  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> range(0, 30, 5) [ Invio ]

  [0, 5, 10, 15, 20, 25]

>>> range(0, 10, 3) [ Invio ]

  [0, 3, 6, 9]

>>> range(0, -10, -1) [ Invio ]

  [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

>>> range(0) [ Invio ]

  []

>>> range(1, 0) [ Invio ]

  []

>>> [ Ctrl d ]

$
```

si noti in particolare che, al solito, il secondo argomento denota il primo valore escluso dalla successione. Si noti infine l'uso dell'istruzione '**`print`**' isolata, allo scopo di andare a capo (ciò è necessario poiché nel ciclo enumerativo precedente si era usato il carattere vigola ('**`,`**') in coda, il quale sopprime il carattere di *newline*).

3.4 Un programma interattivo: «numeri.py»

In questa sezione viene presentato e commentato un semplice programma interattivo, '**`numeri.py`**', con lo scopo di illustrare alcune ulteriori caratteristiche (sintattiche e idiomatiche) del linguaggio Python.

3.4.1 Una sessione d'esempio

Prima del sorgente, è interessante vedere un esempio di esecuzione del programma, allo scopo di comprenderne meglio il funzionamento. Trattandosi di un programma interattivo, la sessione presentata dovrebbe commentarsi da sé.

```
$ ./numeri.py [ Invio ]

  ********
   Numeri
  ********

  Con questo programma e' possibile effettuare alcune
  operazioni matematiche su operandi scelti dall'utente.


Scegliere su quanti operandi operare [2-10]: pippo [ Invio ]

  Non e' un numero intero. Riprova...


Scegliere su quanti operandi operare [2-10]: 1 [ Invio ]

  Il numero dev'essere compreso fra 2 e 10. Riprova...


Scegliere su quanti operandi operare [2-10]: 3 [ Invio ]

  L'utente ha scelto di operare su 3 operandi.
  Inserire gli operandi su cui operare:


Operando 0: -7.9 [ Invio ]

Operando 1: pippo [ Invio ]

  Non e' un numero. Riprova...


Operando 1: 0.25 [ Invio ]

Operando 2: 123456 [ Invio ]

  L'utente ha scelto di operare sui seguenti operandi: -7.9, 0.25, 123456.0
  Operazioni consentite:
  0 - Termina
  1 - Addizione
  2 - Moltiplicazione
  3 - Massimo
  4 - Minimo
  5 - Media


Scegliere un'operazione [1-5, 0 per terminare]: 2 [ Invio ]
```

```
Scegliere un'operazione [1-5, 0 per terminare]: 1 [ Invio ]

Scegliere un'operazione [1-5, 0 per terminare]: 9 [ Invio ]

  Il numero dev'essere compreso fra 0 e 5. Riprova...


Scegliere un'operazione [1-5, 0 per terminare]: 5 [ Invio ]

Scegliere un'operazione [1-5, 0 per terminare]: 0 [ Invio ]

  Il prodotto degli operandi e' -243825.6
  La somma degli operandi e' 123448.35
  La media degli operandi e' 61726.0875


Desideri ricominciare da capo? [s/N]: n [ Invio ]

  Grazie per aver utilizzato il programma numeri.py!


$
```

3.4.2 Il codice sorgente

Il listato 3.56 presenta il codice sorgente del programma '**numeri.py**'.

Listato 3.56. '**numeri.py**', un programma interattivo in Python.

```
1  #!/usr/bin/python
2  #============================================================================
3  # Copyright (C) 2005 Massimo Piai <pxam67 (at) virgilio (dot) it>
4  #============================================================================
5  ###
6  ### numeri.py
7  ###
8  ##
9  ## Importazione moduli essenziali
10 ##
11 import operator
12 import os
13 import sys
14 ##
15 ## Definizione delle varie funzioni
16 ##
17 #
18 # Calcola la media aritmetica
19 #
20 def med(operandi):
21   return reduce(operator.add,operandi)/len(operandi)
22 #
23 # Emette un'intestazione
24 #
25 def Intestazione():
```

```
26    print
27    print "********"
28    print " Numeri "
29    print "********"
30    print
31    print "Con questo programma e' possibile effettuare alcune"
32    print "operazioni matematiche su operandi scelti dall'utente."
33    print
34  #
35  # Elenca gli operandi
36  #
37  def Elenca_operandi():
38
39    for operando in operandi[:len(operandi)-1]:
40      print str(operando) + ",",
41
42    print operandi[len(operandi)-1]
43  #
44  # Elenca le operazioni possibili
45  #
46  def Elenca_operazioni_possibili():
47    print "Operazioni consentite:"
48
49    for i in range(len(nomi_operazioni)):
50      print i, "-", nomi_operazioni[i]
51  #
52  # Richiede all'utente l'indicazione del numero
53  # di operandi
54  #
55  def Richiesta_numero_operandi():
56    quanti_operandi = 0
57
58    while True:
59
60      try:
61        quanti_operandi = int(raw_input(
62          "Scegliere su quanti operandi operare [2-10]: "
63          ))
64      except ValueError:
65        print "Non e' un numero intero. Riprova..."
66      else:
67
68        if quanti_operandi in range(2,11):
69          break
70        else:
71          print "Il numero dev'essere compreso fra 2 e 10. Riprova..."
72
73    print
74    return quanti_operandi
75  #
76  # Richiede all'utente di inserire gli operandi
77  #
78  def Richiesta_inserimento_operandi():
```

```
79    print "L'utente ha scelto di operare su", quanti_operandi, "operandi."
80    print "Inserire gli operandi su cui operare:"
81    operandi = []
82
83    for i in range(quanti_operandi):
84
85      while True:
86
87        try:
88          operando = float(raw_input("Operando " + str(i) + ": "))
89          operandi.append(operando)
90          break
91        except ValueError:
92          print "Non e' un numero. Riprova..."
93
94    print
95    return operandi
96  #
97  # Richiede all'utente di scegliere le operazioni
98  #
99  def Richiesta_scelta_operazioni():
100   print "L'utente ha scelto di operare sui seguenti operandi:",
101   Elenca_operandi()
102   Elenca_operazioni_possibili()
103   operazioni = []
104
105   while True:
106
107     try:
108       opz_n = int(raw_input(
109         "Scegliere un'operazione [1-5, 0 per terminare]: "
110         ))
111     except ValueError:
112       print "Non e' un numero intero. Riprova..."
113     else:
114
115       if opz_n in range(len(operazioni_possibili)+1):
116
117         if not opz_n:
118           break
119         else:
120
121           if nomi_operazioni[opz_n] not in operazioni:
122             operazioni.append(nomi_operazioni[opz_n])
123           else:
124             print "Operazione gia' scelta."
125
126       else:
127         print "Il numero dev'essere compreso fra 0 e 5. Riprova..."
128
129   print
130   return operazioni
131 #
```

```
# Presenta i risultati e offre la possibilita‘ di
# ricominciare o terminare
#
def Calcolo_e_presentazione_risultati():

  for operazione in operazioni:
    print nomi_risultati_operazioni[operazione], "degli operandi e‘",

    if operazione in operazioni_n_arie:
      print operazioni_possibili[operazione](operandi)
    else:
      print reduce(operazioni_possibili[operazione], operandi)

  print

  return raw_input(
    "Desideri ricominciare da capo? [s/N]: "
    ).lower().startswith("s")
#
# Gestione ringraziamenti e saluti
#
def Commiato():
  print "Grazie per aver utilizzato il programma",
  print os.path.basename(sys.argv[0]) + "!"
  print
##
## Inizio programma principale
##
#
# Variabili globali
#
continua = True
operandi = []
operazioni = []
quanti_operandi = 0
operazioni_possibili = {"Addizione": operator.add,
                        "Moltiplicazione": operator.mul,
                        "Massimo": max,
                        "Minimo": min,
                        "Media": med}
nomi_risultati_operazioni = {"Addizione": "La somma",
                             "Moltiplicazione": "Il prodotto",
                             "Massimo": "Il massimo",
                             "Minimo": "Il minimo",
                             "Media": "La media"}
nomi_operazioni = ["Termina", "Addizione", "Moltiplicazione",
                   "Massimo", "Minimo", "Media"]
operazioni_n_arie = ["Massimo", "Minimo", "Media"]
#
#
#
Intestazione()

```

```
185  while continua:
186    quanti_operandi = Richiesta_numero_operandi()
187    #
188    # L'utente ha scelto su quanti operandi operare,
189    # quindi si procede a chiederne l'inserimento
190    #
191    operandi = Richiesta_inserimento_operandi()
192    #
193    # L'utente ha inserito gli operandi, quindi
194    # si procede a chiedergli che operazione eseguire
195    #
196    operazioni = Richiesta_scelta_operazioni()
197    #
198    # L'utente ha scelto le operazioni,
199    # si procede quindi al calcolo e alla
200    # presentazione dei risultati, offrendo
201    # la possibilita' di ricominciare o terminare
202    #
203    continua = Calcolo_e_presentazione_risultati()
204    print
205  #
206  # Commiato dall'utente
207  #
208  Commiato()
```

3.4.3 Analisi e commento

Commentiamo gli aspetti principali del programma. Cominciamo l'analisi dal livello più esterno:

- **Righe 157-176**

  Vengono definite alcune variabili (o ***nomi*** secondo la terminologia Python) utilizzate dal programma principale.

  La variabile '**`operazioni_possibili`**' è un dizionario che serve a mantenere una corrispondenza fra stringhe (che contengono dei nomi convenzionali di funzioni) e funzioni (le quali possono essere incorporate oppure appartenenti a un modulo esterno oppure ancora definite altrove nel programma stesso). '**`nomi_risultati_operazioni`**' è un dizionario che mantiene una corrispondenza fra fra stringhe che contengono dei nomi convenzionali di funzioni e stringhe che contengono i nomi tradizionali che esprimono il risultato del calcolo delle funzioni stesse. '**`nomi_operazioni`**' è una lista che contiene i nomi delle funzioni che l'utente può di volta in volta scegliere di utilizzare. '**`operazioni_n_arie`**' è una lista che contiene i nomi delle funzioni che sono definite in modo da operare su 2 o più argomenti (le rimanenti funzioni operano esattamente su 2 argomenti).

- **Righe 177-205**

  È il programma principale. Si compone essenzialmente di un ciclo iterativo che si ripete fintantoché la variabile booleana '**`continua`**' risulta vera. La successione dei passi che compongono il corpo del ciclo è la seguente:

  – richiedere all'utente su quanti operandi operare, e assegnazione di tale valore a '**`quanti_operandi`**';

  - richiedere all'utente su quali operandi operare, e assegnazione di tali valori alla lista '**`operandi`**';
  - richiedere all'utente con quali operazioni operare, e assegnazione di tali valori alla lista '**`operazioni`**';
  - calcolo e presentazione dei risultati; richiesta all'utente se vuole continuare con una nuova iterazione e assegnazione corrispondente alla variabile '**`continua`**'.

All'uscita del ciclo il programma termina dopo essersi accomiatato dall'utente.

I vari passi che costituiscono il ciclo principale sono realizzati mediante funzioni che restituiscono un valore adeguato alla necessità. Procediamo ad analizzare gli aspetti maggiormente qualificanti di tali funzioni.

- **Righe 31-39**

  La funzione '**`Elenca_operandi`**' elenca i membri della lista '**`operandi`**' separati da virgole.

  Si noti l'idioma tipico per iterare su di una lista fino al penultimo membro:

  ```
  for membro in lista[:len(lista)-1]:
    ...
  ```

  Ciò è necessario per trattare l'ultimo membro come caso speciale.

- **Righe 40-47**

  La funzione '**`Elenca_operazioni_possibili`**' emette un elenco numerato delle operazioni possibili per permettere all'utente la scelta.

  Si noti il tipico idioma per iterare su di una lista attraverso gli indici invece che attraverso i membri:

  ```
  for indice in range(len(lista)):
    ...
  ```

- **Righe 48-71**

  La funzione '**`Richiesta_numero_operandi()`**' riceve in input il numero di operandi su cui operare e lo restituisce al chiamante; effettua anche un controllo piuttosto dettagliato dell'input, tramite i costrutti Python per la gestione delle ***eccezioni***[3]: '**`try`**', '**`except`**' e '**`else`**'. In pratica la funzione esegue un ciclo infinito ('**`while True:`**') all'interno del quale riceve l'input e cerca ('**`try:`**') di convertirlo in un intero ('**`...int(raw_input(...`**'); se la conversione fallisce ('**`except ValueError:`**') il ciclo continua; se la conversione ha successo ma il valore non è consentito il ciclo continua; altrimenti il ciclo termina ('**`break`**'[4]).

- **Righe 72-92**

- La funzione '**`Richiesta_inserimento_operandi`**' costruisce la lista degli operandi ricevuti in input e la restituisce al chiamante; il controllo dell'input viene effettuato con la tecnica già vista della gestione delle eccezioni.

Si noti l'idioma tipico per l'estensione di una lista con un membro in coda:

```
lista.append(membro)
```

- **Righe 93-127**

  La funzione '**`Richiesta_scelta_operazioni`**' chiama a sua volta '**`Elenca_operandi`**' e '**`Elenca_operazioni_possibili`**', dopodiché prepara e restituisce al chiamante la lista delle operazioni (ossia delle stringhe contenenti i nomi delle operazioni come da dizionario '**`operazioni_possibili`**') richiesta dall'utente. L'utente sceglie le operazioni mediante il numero progressivo mostrato da '**`Elenca_operazioni_possibili`**' e l'input viene via via controllato mediante una tecnica simile a quelle già incontrate. Si noti inoltre:

  – la struttura condizionale '**`if not opz_n:`**' la quale ha successo esattamente quando '**`opz_n`**' vale 0[5]; in tal caso il ciclo termina;

  – l'istruzione di estensione della lista delle operazioni richieste è controllata dalla struttura condizionale '**`if nomi_operazioni[opz_n] not in operazioni:`**' per far sì che non venano inseriti doppioni nella lista.

  Si tratta di un idioma Python tipico per controllare la presenza di un membro in una lista:

  ```
  membro in lista
  ```

  oppure:

  ```
  membro not in lista
  ```

- **Righe 128-146**

  La funzione '**`Calcolo_e_presentazione_risultati`**' calcola le operazioni richieste sugli operandi indicati, presenta i risultati e chiede all'utente se desidera ricominciare. L'applicazione delle funzioni che realizzano le operazioni agli operandi viene effettuata in due modi diversi a seconda che la funzione sia ***binaria*** (ossia lavori esattamente su 2 argomenti) oppure ***ternaria*** o più: nel primo caso su utilizza la funzione Python '**`reduce`**' la quale per l'appunto estende le funzioni binarie al caso di più argomenti[6].

  '**`raw_input(...`**' chiede all'utente di inserire un'input; la stringa viene convertita in minuscole tramite il metodo '**`lower`**'; il metodo '**`startswith("s")`**' restituisce un valore booleano a seconda che la stringa cui viene applicato inizi o meno con '**`"s"`**'.

3.4.3.1 Alcune osservazioni aggiuntive

Le variabili di cui alle righe 157-176, essendo dichiarate nel blocco più esterno del sorgente, queste potrebbero essere chiamate «variabili globali», secondo una tradizione consolidata: in effetti tali nomi sono accessibili (in lettura) in tutto il programma, mentre l'accesso in scrittura è possibile solamente nel livello più esterno.

Più precisamente, in Python è possibile accedere - a un livello più interno - a un nome dichiarato al livello più esterno, anche in scrittura, ma:

- il valore associato al nome è quello del livello esterno fino alla modifica, e
- la modifica perde effetto appena il controllo ritorna al livello esterno.

È possibile alterare quest'ultimo comportamento utilizzando la parola chiave '**`global`**': dichiarando come ***global*** un nome, ogni modifica al livello interno si riflette al livello esterno. Ad esempio:

```
def funz():
  global x
  x = 0
  print "sono funz"
  print "x: ", x

x = 42
print "sono il programma principale"
print "x: ", x
funz()
print "sono il programma principale"
print "x: ", x
```

`$ python tmp/global.py` [*Invio*]

```
sono il programma principale
x:  42
sono funz
x:  0
sono il programma principale
x:  0
```

Senza la dichiarazione '**`global x`**' l'esecuzione avrebbe fornito il seguente output:

```
sono il programma principale
x:  42
sono funz
x:  0
sono il programma principale
x:  42
```

Per evitare ambiguità, nel seguito chiameremo nomi o variabili ***global*** i nomi Python dichiarati con la parola chiave '**`global`**'.

---

[1] Essenzialmente un dizionario è un tipo speciale di array: in pratica si tratta di una collezione di coppie chiave-valore, in cui le chiavi possono essere di qualsiasi tipo immutabile, e non solamente numeri interi; corrispondono in pratica agli array associativi del linguaggio Perl.

[2] In altri termini: argomento e parametro formale sono due nomi dello stesso oggetto Python.

[3] Trattasi di errori non sintattici non necessariamente fatali intercettati in fase di esecuzione.

[4] Si tenga presente che la parola chiave può comparire solo nel corpo di un ciclo (enumerativo oppure iterativo) e ha l'effetto di terminare immediatamente il ciclo più interno fra quelli che la includono.

[5] In Python sono considerati valori booleani falsi ('`False`'): '`None`', lo zero numerico di qualunque tipo, la stringa vuota ('`""`'), la tupla vuota ('`()`'), la lista vuota ('`[]`'), il dizionario vuoto ('`{}`'). Tutti gli altri valori sono interpretati come veri ('`True`').

[6] '`reduce`' viene utilizzata anche nelle righe 11-18 ove viene definita la funzione '`med`' la quale calcola la media artitmetica dei membri della lista che le viene passata come argomento.

Appendici

Appendice A

Informazioni aggiuntive sul software e altre opere citate

Python, IV

⟨*http://www.python.org/license.html* ⟩

Indice analitico

Massimo Piai *(... circa 1975)*
⟨*pxam67 (ad) virgilio·it* ⟩