Mini-HOWTO sull'ordinamento dei dati

Release 0.01

Andrew Dalke

dalke@bioreason.com

3 aprile 2004

Sommario

Questo documento è un piccolo resoconto su una mezza dozzina di metodi di organizzare una lista di dati con il metodo built-in `sort()`. Traduzione italiana a cura di Carlos (enne.enne at tiscalinet.it).

È disponibile presso la pagina degli HOWTO Python all'indirizzo http://www.python.org/doc/howto, la traduzione presso la pagina http://www.zonapython.it/doc/howto/.

Indice

Le liste in Python hanno il metodo built-in `sort()`. Siccome manca una descrizione univoca e specifica sui molti modi in cui è possibile ordinare una lista, questo documento cerca di fornirne una descrizione.

1 Ordinare dati di tipo elementare

Per un ordinamento in ordine crescente basta chiamare il metodo `sort()` di una lista.

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> print a
[1, 2, 3, 4, 5]
```

Sort prende una funzione facoltativa che può essere chiamata per operare confronti. La routine predefinita di sort è equivalente a

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort(cmp)
>>> print a
[1, 2, 3, 4, 5]
```

ove `cmp` è la funzione built-in che confronta gli oggetti `x` e `y`, restituendo -1, 0 o 1 a seconda se `x < y`, `x == y` o `x > y`. Affinché la lista finale abbia senso, durante l'ordinamento i rapporti devono restare costanti.

Volendo, per il confronto si può definire una propria funzione; per i numeri interi (ed i numeri in generale) ad esempio possiamo scrivere:

```
>>> def compara_numeri(x, y):
>>>    return x-y
>>>
>>> a = [5, 2, 3, 1, 4]
>>> a.sort(compara_numeri)
>>> print a
[1, 2, 3, 4, 5]
```

Per inciso, la funzione non lavora se il risultato dell'operazione è fuori dai limiti, come in `sys.maxint - (-1)`.

Se non si vuole definire una funzione con un nome nuovo, se ne può creare una anonima mediante la funzione `lambda`, come questa:

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort(lambda x, y: x-y)
>>> print a
[1, 2, 3, 4, 5]
```

Se si desidera l'ordine inverso è sufficiente scrivere:

```
>>> a = [5, 2, 3, 1, 4]
>>> def inverti_numeri(x, y):
>>>     return y-x
>>>
>>> a.sort(inverti_numeri)
>>> print a
[5, 4, 3, 2, 1]
```

(un'implementazione più generale potrebbe restituire `cmp(y,x)` o `-cmp(x,y)`).

Però, dato che si fa prima a non obbligare Python a chiamare una funzione per ogni confronto, se si vuole una lista ordinata all'inverso per dati di tipo elementari, è meglio eseguire prima l'ordinamento diretto e poi usare il metodo `reverse()`.

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a.reverse()
>>> print a
[1, 2, 3, 4, 5]
```

Ecco una stringa di confronto insensibile a maiuscole e minuscole con la funzione `lambda`:

```
>>> import string
>>> a = string.split("Trattasi di una stringa di prova, da NN.")
>>> a.sort(lambda x, y: cmp(string.lower(x), string.lower(y)))
>>> print a
['da', 'di', 'di', 'NN.', 'prova,', 'stringa', 'Trattasi', 'una']
```

Questo evita di dover convertire le maiuscole di una parola ad ogni confronto. Talora, può essere più spiccio conteggiarle una volta e poi usare i valori ottenuti, come nel seguente esempio.

```
>>> parole = string.split("Trattasi di una stringa di prova, da NN.")
>>> offsets = []
>>> for i in range(len(parole)):
...     offsets.append( (string.lower(parole[i]), i) )
...
>>> offsets.sort()
>>> nuove_parole = []
>>> for nonticurare, i in offsets:
...      nuove_parole.append(parole[i])
...
>>> print nuove_parole
```

La lista `offsets` viene inizializzata in una tupla di stringhe composte di caratteri in lettere minuscole, alla corretta posizione, nella lista `parole`, e quindi ordinata. Il metodo di ordinamento in Python ordina le tuple confrontandone i termini; dati `x` e `y`, confronta `x[0]` e `y[0]`, `x[1]` e `y[1]`, ecc. finché ci sia differenza.

Ne risulta che la lista `offsets` è ordinata a partire dal suo primo termine e il secondo fa capire dove fosse disposto il dato in origine. (Il ciclo `for` assegna `nonticurare` e `di` ai due campi di ogni termine della lista, ma basta già il valore dell'indice.)

Un altro modo di implementare ciò è memorizzare il dato d'origine come secondo termine della lista `offsets`, come in:

```
>>> parole = string.split("Trattasi di una stringa di prova, da NN.")
>>> offsets = []
>>> for parola in parole:
...     offsets.append( (string.lower(parola), parola) )
...
>>> offsets.sort()
>>> nuove_parole = []
>>> for parola in offsets:
...     nuove_parole.append(parola[1])
...
>>> print nuove_parole
```

Questo non è sempre corretto, poiché i secondi termini nella lista (la parola, in questo esempio) vengono confrontati quando i primi sono uguali. Se ciò succede molte volte, un confronto dei due oggetti risulta sempre superfluo — il che, se per lo più i termini si equivalgono e gli oggetti definiscono i propri metodi `__cmp__` (ma con l'ulteriore incombenza di determinare se `__cmp__` sia definito), costituisce un grande spreco.

Ancora, per liste molto estese, o nelle quali il calcolo delle informazioni di confronto sia gravoso, gli ultimi due esempi sono probabilmente i modi più rapidi per ordinarle; non funzioneranno, però, con dati difficili a disporsi, come i numeri complessi (la cui conoscenza, peraltro, non è essenziale).

2 Confrontare classi

Il confronto fra due tipi elementari di dati — come interi o stringhe — è incorporato in Python ed ha un senso. C'è un modo predefinito di confrontare istanze di classi, anche se di solito non è molto utile; con il metodo `__cmp__` si possono definire dei confronti personalizzati, come in:

```
>>> class Spam:
>>>     def __init__(self, spam, eggs):
>>>         self.spam = spam
>>>         self.eggs = eggs
>>>     def __cmp__(self, other):
>>>         return cmp(self.spam+self.eggs, other.spam+other.eggs)
>>>     def __str__(self):
>>>         return str(self.spam + self.eggs)
>>>
>>> a = [Spam(1, 4), Spam(9, 3), Spam(4,6)]
>>> a.sort()
>>> for spam in a:
>>>   print str(spam)
5
10
12
```

Se si desidera un ordinamento in base ad un attributo specifico della classe, correttamente basterebbe definire il metodo `__cmp__` per confrontare quei valori, ma non si può, volendo un confronto fra attributi diversi in tempi diversi. Invece, bisogna tornare indietro e tradurre la funzione di confronto in criterio di ordinamento, come in:

```
>>> a = [Spam(1, 4), Spam(9, 3), Spam(4,6)]
>>> a.sort(lambda x, y: cmp(x.eggs, y.eggs))
>>> for spam in a:
>>>   print spam.eggs, str(spam)
3 12
4 5
6 10
```

Volendo confrontare due attributi a scelta (senza preoccuparsi troppo delle prestazioni), si può addirittura definire il proprio oggetto per il confronto, usando la capacità delle classi di emulare funzioni definendo il metodo `__call__`, come in:

```
>>> class CmpAttr:
>>>     def __init__(self, attr):
>>>         self.attr = attr
>>>     def __call__(self, x, y):
>>>         return cmp(getattr(x, self.attr), getattr(y, self.attr))
>>>
>>> a = [Spam(1, 4), Spam(9, 3), Spam(4,6)]
>>> a.sort(CmpAttr("spam"))  # ordina in base all'attributo "spam"
>>> for spam in a:
>>>    print spam.spam, spam.eggs, str(spam)
1 4 5
4 6 10
9 3 12

>>> a.sort(CmpAttr("eggs"))   # riordina in base all'attributo "eggs"
>>> for spam in a:
>>>    print spam.spam, spam.eggs, str(spam)
9 3 12
1 4 5
4 6 10
```

Naturalmente, se si cerca un ordinamento più rapido, si possono estrarre gli attributi in una lista intermedia e ordinare quest'ultima.

E, così, ecco qua sei modi diversi di definire come ordinare una lista:

- con il metodo predefinito;
- con una funzione per il confronto;
- al rovescio, senza l'uso di funzioni per il confronto;
- tramite una lista intermedia (due forme);
- usando una classe dove si è definito il proprio metodo __cmp__;
- utilizzando la funzione sort di un oggetto.