Il Tutorial di Python

Autori:
Guido van Rossum
Fred L. Drake, Jr., editor

BeOpen PythonLabs
E-mail: python-docs@python.org

Traduzione:
Riccardo Fabris, ?
E-mail: python.it@tiscalinet.it

Traduzione in revisione!

basata sulla versione del:
16 Ottobre, 2000
Release 2.0

Premesse

[le parti legali ecc. non tradotte NdT]

BEOPEN.COM TERMS AND CONDITIONS FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen''), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee'') accessing and otherwise using this software in source or binary form and its associated documentation (``the Software'').
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an ``AS IS'' basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the ``BeOpen Python'' logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI OPEN SOURCE LICENSE AGREEMENT

Python 1.6 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1012. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1012.

CWI PERMISSIONS STATEMENT AND DISCLAIMER

Sommario:

Python è un linguaggio di programmazione potente e di facile apprendimento. Utilizza efficienti strutture dati di alto livello e un semplice ma efficace approccio alla programmazione orientata agli oggetti. L'elegante sintassi di Python e la tipizzazione dinamica, unite alla sua natura di linguaggio interpretato, lo rendono ideale per lo scripting e lo sviluppo rapido di applicazioni in molte aree diverse e sulla maggior parte delle piattaforme.

L'interprete Python e l'ampia libreria standard sono liberamente disponibili, in file sorgenti o binari, per tutte le principali piattaforme sul sito web di Python http://www.python.org, e possono essere liberamente distribuiti. Lo stesso sito contiene anche, oltre alle distribuzioni, puntatori a molti moduli Python liberi e gratuiti (`free') di terzi, interi programmi, strumenti di sviluppo, e documentazione addizionale.

L'interprete Python è facilmente estendibile con nuove funzioni o tipi di dato implementati in C o C++} (o altri linguaggi richiamabili dal C). Python è anche adatto come linguaggio di estensione per applicazioni personalizzabili.

Questo tutorial introduce informalmente il lettore ai concetti e alle caratteristiche base del linguaggio e del sistema Python. È di aiuto avere un interprete Python a portata di mano per fare esperienza diretta, ma tutti gli esempi sono autoesplicativi, quindi il tutorial può essere letto anche a elaboratore spento.

Per una descrizione degli oggetti e dei moduli standard, si veda il documento *Python Library Reference*. Il *Python Reference Manual* fornisce una definizione più formale del linguaggio. Se s'intendono scrivere estensioni in C o C++, si legga *Extending and Embedding the Python Interpreter* e *Python/C API Reference*. Ci sono anche numerosi libri che si occupano in modo approfondito di Python.

Questo tutorial non si propone di essere onnicomprensivo e di coprire ogni singola funzionalità o anche solo quelle più comunemente usate. Vuole essere piuttosto un'introduzione alle caratteristiche più notevoli di Python e vuole fornire un'idea precisa dello stile del linguaggio. Dopo averlo letto si sarà capaci di leggere e scrivere moduli e programmi in Python, e si sarà pronti ad imparare di più sui vari moduli della libreria Python descritta nel *Python Library Reference*.

1. Per Stimolarvi l'Appetito

Se in qualche occasione avete scritto uno script di shell di grosse dimensioni, è probabile che la sensazione seguente vi sia familiare. Vorreste tanto aggiungere ancora un'altra funzione, ma è già così lento, e così grosso, e così complicato; oppure la funzionalità che avete in mente necessita di una chiamata di sistema o di un'altra funzione accessibile solo da C... Di solito il problema in esame non è abbastanza rilevante da giustificare una riscrittura dello script in C; magari richiede stringhe di lunghezza variabile o altri tipi di dati (come liste ordinate di nomi di file) che sono semplici da gestire dalla shell ma richiedono molto lavoro per essere implementati in C, o forse non avete familiarità sufficiente col C.

Un'altra situazione: forse dovete lavorare con parecchie librerie C, e la solita procedura ciclica di scrittura/compilazione/test/ricompilazione è troppo lenta. Avete la necessità di sviluppare i programmi in tempi più brevi. Magari avete scritto un programma che potrebbe usare un linguaggio di estensione, e non volete stare a progettare un linguaggio, scrivere e testare un interprete per esso e poi congiungerlo alla vostra applicazione.

In casi simili, Python potrebbe essere quello che fa per voi. Python è semplice da usare ma è un vero linguaggio di programmazione, che offre molte più strutture e supporto per programmi di grandi dimensioni che i linguaggi di shell. D'altra parte, offre anche il controllo degli errori del C e, essendo un *linguaggio di altissimo livello*, ha tipi di dato built-in di alto livello, come array flessibili e dizionari, che prenderebbero molto tempo per essere implementati in maniera efficiente in C. Grazie ai suoi tipi di dati di applicazione più generale, Python è applicabile a un insieme di problemi molto più vasto di *Awk* o anche *Perl*, cionondimeno molte cose sono semplici in Python almeno quanto in questi linguaggi.

Python permette di suddividere i vostri programmi in moduli che possono essere riutilizzati in altri programmi Python. è accompagnato da un'ampia raccolta di moduli standard che potete usare come basi per i vostri programmi o come esempi utili nell'apprendimento della programmazione in Python. Ci sono anche moduli built-in che forniscono il supporto per cose come I/O su file, chiamate di sistema, socket, e anche interfacce a toolkit GUI (Interfaccia Utente Grafica) come Tk.

Python è un linguaggio interpretato, e questo può far risparmiare molto tempo durante lo sviluppo del programma, poiché non sono necessari compilazione e linking. L'interprete può essere utilizzato interattivamente, il che rende semplice fare esperimenti con le funzionalità del linguaggio, scrivere programmi usa-e-getta o testare funzioni durante lo sviluppo bottom-up di programmi. è anche un'utile calcolatrice.

Python consente di scrivere programmi molto compatti e di facile lettura. Tipicamente i programmi scritti in Python sono molto più brevi degli equivalenti in C, per numerose ragioni:

- i tipi di dati di alto livello consentono di esprimere operazioni complesse in una singola istruzione;
- le istruzioni vengono raggruppate tramite indentazione invece che con parentesi di inizio/fine;
- non è necessario dichiarare variabili ed argomenti.

Python è *estendibile*: se sapete programmare in C è facile aggiungere all'interprete nuove funzioni o moduli built-in, per eseguire operazioni critiche alla massima velocità o per lincare a programmi Python delle librerie che possono essere disponibili solo in forma di file binari (ad esempio librerie grafiche proprietarie). Quando sarete veramente smaliziati, potrete lincare l'interprete Python a un'applicazione scritta in C e usarlo come linguaggio di estensione o di comando per tale applicazione.

A proposito, l'origine del nome dipende dallo show televisivo della BBC ``Monty Python's Flying Circus'' e non ha niente a che fare con i pericolosi rettili omonimi. Fare riferimento alle caratteristiche burle dei Monty Python nella documentazione non solo è permesso, è incoraggiato!

1.1 E adesso?

Ora che la vostra curiosità nei confronti di Python è stata stimolata, vorrete esaminarlo in maggior dettaglio. Dato che il modo migliore di imparare un linguaggio è usarlo, siete invitati a farlo.

Nel prossimo capitolo verranno spiegati i meccanismi per utilizzare l'interpete. Si tratta di informazioni abbastanza banali, ma essenziali per lavorare sugli esempi che verranno mostrati più avanti.

Il resto del tutorial introdurrà varie caratteristiche e funzionalità del linguaggio (e sistema) Python attraverso esempi, iniziando con semplici espressioni, istruzioni e tipi di dati, passando per funzioni e moduli, per finire col toccare concetti avanzati come le eccezioni e le classi definite dall'utente.

2. Usare l'Interprete Python

2.1 Invocazione dell'Interprete

L'interprete Python sulle macchine UNIX sulle quali è disponibile di solito è installato come /usr/local/bin/python; aggiungendo /usr/local/bin nel percorso di ricerca della shell è possibile farlo partire digitando il comando:

```
python
```

dalla shell. Dato che la directory in cui collocare l'interprete può essere scelta al momento dell'installazione, è possibile collocarlo altrove; in caso si consulti il proprio guru Python locale o l'amministratore di sistema. (Per esempio, /usr/local/python è un'alternativa diffusa).

Digitare un carattere di EOF (Control-D su Unix, Control-Z su DOS o Windows) al prompt primario fa sì che si l'interprete esca con uno status pari a zero. Se non funziona, si può uscire digitando i seguenti comandi: "`import sys; sys.exit()`".

Le funzioni di editing di riga dell'interprete di solito non sono molto sofisticate. Su Unix, chiunque abbia installato l'interprete può avere abilitato il supporto per la libreria GNU readline, che aggiunge funzionalità di storico e di editing interattivo più avanzate. Forse il modo più rapido di controllare se sia supportato l'editing della riga di comando è di digitare Control-P al primo prompt Python che si riceve. Se viene emesso un `beep', l'editing è abilitato; si veda l'Appendice A per un'introduzione ai comandi da tastiera. Se sembra che non accada nulla, o se si ha un eco di `P`, allora l'editing della riga di comando non è disponibile; si potrà solamente utilizzare il tasto Indietro (`backspace') per cancellare caratteri dalla riga corrente.

L'interprete opera all'incirca come una shell Unix: quando viene lanciato con lo standard input connesso a un terminale legge ed esegue interattivamente dei comandi; quando viene invocato con il nome di un file come argomento o con un file come standard input legge ed esegue uno *script* da quel file.

Un terzo modo di lanciare l'interprete è tramite "**`python -c`** *comando* `[arg] ...`", che esegue la/e istruzione/i contenuta/e in *comando*}, analogamente a quanto succede per l'opzione **`-c`** della shell. Dato che spesso le istruzioni Python contengono spazi o altri caratteri che la shell considera speciali, è molto meglio racchiudere integralmente *comando* tra doppie virgolette.

Si noti che c'è una differenza tra "`python file`" e "`python <file`". Nel secondo caso le richieste di input del programma, a esempio chiamate a `input()` e `raw_input()`, vengono soddisfatte da *file*. Dato che questo file è gia stato letto fino alla fine dall'analizzatore sintattico (` parser') prima che il programma venga effettivamente eseguito, il programma si imbatterà immediatamente in EOF. Nel primo caso (che di solito è quello più utile) le richieste vengono soddisfatte da qualunque file o dispositivo sia connesso all'input standard dell'interprete Python.
Quando si usa un file di script, talvolta è utile poter lanciare lo script e successivamente entrare in modalità interattiva. Lo si può ottenere passando l'opzione `-i` prima dello script. (Non funziona se lo script è letto dallo standard input, per lo stesso motivo illustrato nel paragrafo precedente).

2.1.1 Passaggio di Argomenti

Quando noti all'inteprete, il nome dello script e poscia argomenti addizionali sono passati allo script tramite la variabile `sys.argv`, che è una lista di stringhe. La sua lunghezza minima è uno; quando non vengono forniti nè script nè argomenti, `sys.argv[0]` è una stringa vuota. Quando il nome dello script è fornito come `'-'` (che significa b standard input), `sys.argv[0]` viene posto a `'-'`. Allorché viene usato **-c** *comando*, `sys.argv[0]` viene posto a `'-c'`. Le opzioni trovate dopo **-c** *comando* non sono consumate nell'elaborazione delle opzioni da parte dell'interprete Python, ma lasciate in `sys.argv` per essere gestite dal comando.

2.1.2 Modo Interattivo

Quando i comandi vengono letti da un terminale, si dice che l'inteprete è in *modo interattivo*. In questa modalità esso presenta, in attesa del prossimo comando, un *prompt primario*, composto di solito da tre segni consecutivi di maggiore ("`>>> `"); per le righe di continuazione il prompt è quello *secondario*, per default tre punti consecutivi ("`. . . `").
L'interprete stampa a video un messaggio di benvenuto in cui compare il numero di versione e un avviso di copyright prima del prompt iniziale, p.e.:

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Le righe di continuazione sono necessarie quando si introduce un costrutto multiriga. Come esempio si prenda questo blocco `if`:

```
>>> il_mondo_è_piatto = 1
>>> il_mondo_è_piatto:
...     print "Occhio a non caderne fuori!"
...
Occhio a non caderne fuori!
```

2.2 L'Interprete e il Suo Ambiente

2.2.1 Gestione degli Errori

Quando sopravviene un errore, l'inteprete stampa un messaggio di errore e una traccia dello stack. Se si trova in modalità interattiva ritorna poi al prompt primario; se l'input proveniva da un file, esce con uno stato diverso da zero dopo aver stampato la traccia dello stack. (Eccezioni gestite da una clausola `except` in un'istruzione `try` non sono errori in questo contesto.) Alcuni errori sono incondizionatamente fatali e provocano un'uscita con uno stato diverso da zero; ciò accade quando si tratta di problemi interni dell'interprete e di alcuni casi di esaurimento della memoria. Tutti i messaggi di errore vengono scritti nello stream di errore standard; l'output normale dei comandi eseguiti viene scritto nell'output standard.

Digitando il carattere di interruzione (di solito Ctrl-C o CANC) al prompt primario o secondario si cancella l'input e si ritorna al prompt primario. [2.1]Digitando un'interruzione mentre un comando è in esecuzione viene sollevata un'eccezione `KeyboardInterrupt`, che può essere gestita tramite un'istruzione `try`.

2.2.2 Script Python Eseguibili

Sui sistemi Unix in stile BSD, gli script Python possono essere resi direttamente eseguibili, come per gli script di shell, ponendo all'inizio dello script la riga

```
#! /usr/bin/env python
```

((dato per scontato che l'interprete si trovi nel $PATH dell'utente) e dando al file il permesso di esecuzione. I caratteri "`#!`" devono essere i primi due del file. Si noti che il carattere `hash' "`#`" [in italiano `cancelletto', `fregno', `diesis' N.d.T.] è usato in Python per iniziare un commento).

2.2.3 Il File di Avvio in Modo Interattivo

Quando si usa Python interattivamente, risulta spesso comodo che alcuni comandi standard vengano eseguiti ad ogni lancio dell'interprete. È possibile farlo configurando appositamente una variabile d'ambiente chiamata $PYTHONSTARTUP con il nome di un file contenente i propri comandi di avvio, all'incirca come si fa con .profile per le shell Unix.
Tale file viene letto solo nelle sessioni interattive, non quando Python legge comandi da uno script, e nemmeno quando il file di dispositivo /dev/tty è fornito espressamente come fonte dei comandi (altrimenti si comporterebbe come in una sessione interattiva). Il file viene eseguito nello stesso spazio di nomi dove vengono eseguiti i comandi interattivi, cosicché gli oggetti che definisce o importa possono essere usati senza essere qualificati nella sessione interattiva. È anche possibile cambiare i prompt, primario (`sys.ps1`) e secondario (`sys.ps2`).
Se si vuole far leggere all'interprete un file di avvio addizionale dalla directory corrente, è possibile farlo tramite il file di avvio globale, p.e. "`if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`". Se si vuole utilizzare il file di avvio in uno script, bisogna indicarlo esplicitamente nello script:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(nome_del_file):
    execfile(nome_del_file)
```

3. Un'Introduzione Informale a Python

Negli esempi seguenti, l'input e l'output sono distinguibili per la presenza o meno dei prompt ("`>>> `" e "`. . . `"): per sperimentare l'esempio è necessario digitare tutto quello che segue il prompt, quando esso compare; le righe che non iniziano con un prompt sono output dell'interprete. Si noti che se in un esempio compare un prompt secondario isolato su una riga significa che si deve introdurre una riga vuota; questo viene usato per terminare un comando che si estende su più righe.
Molti degli esempi di questo manuale, anche quelli immessi al prompt interattivo, presentano commenti. In Python i commenti iniziano con il carattere `hash', "`#`", e continuano fino alla fine della riga. Un commento può comparire all'inizio di una riga o dopo degli spazi bianchi o del codice, ma non dentro una stringa letterale [i `letterali' sono elementi di programma, notazioni per valori costanti di alcuni tipi built-in; possono essere

numeri o stringhe, visto che i caratteri sono stringhe di lunghezza unitaria NdT]. Un carattere `hash' dentro una stringa letterale è solo un carattere `hash'.
Alcuni esempi:

```
# questo è il primo commento
SPAM = 1                 # e questo è il secondo
                    # ... and now a third!
STRING = "# Questo non è un commento."
```

3.1 Usare Python come Calcolatrice

Proviamo con qualche semplice comando Python. Si avvii l'inteprete e si aspetti il prompt primario, "`>>> `". (Non dovrebbe metterci molto).

3.1.1 Numeri

L'interprete si comporta come una semplice calcolatrice: si può digitare un'espressione ed esso fornirà il valore risultante. La sintassi delle espressioni è chiara: gli operatori `+`, `-`, `*` e `/` funzionano come nella maggior parte degli altri linguaggi (p.e. Pascal o C); le parentesi possono essere usate per raggruppare operatori e operandi. A esempio:

```
>>> 2+2
4
>>> # Questo è un commento
... 2+2
4
>>> 2+2  ## e un commento sulla stessa riga del codice
4
>>> (50-5*6)/4
5
>>> # Una divisione tra interi restituisce solo il quoziente:
... 7/3
2
>>> 7/-3
-3
```

Come in C, il segno di uguale ("`=`") è usato per assegnare un valore a una variabile. Il valore di un assegnamento non viene stampato:

```
>>> larghezza = 20
```

```
>>> altezza = 5*9
>>> larghezza * altezza
900
```

Un valore può essere assegnato simultaneamente a variabili diverse:

```
>>> x = y = z = 0  # Zero x, y e z
>>> x
0
>>> y
0
>>> z
0
```

Le operazioni in virgola mobile sono pienamente supportate; in presenza di operandi di tipo misto gli interi vengono convertiti in virgola mobile:

```
>>> 4 * 2.5 / 3.3
3.0303030303
>>> 7.0 / 2
3.5
```

Anche i numeri complessi sono supportati; per contrassegnare i numeri immaginari si usa il suffisso "`j`" o "`J`". I numeri complessi con una componente reale non nulla sono indicati scritti come "`(`*real*`+`*imag*`j)`", o possono essere creati con la funzione "`complex(`*real*`, `*imag*`)`".

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

I numeri complessi sono sempre rappresentati come due numeri in virgola mobile, la parte reale e quella immaginaria. Per estrarre queste parti da un numero complesso *z* si usino `z.real` e `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Le funzioni di conversione a virgola mobile e intero (`float()`, `int()` e `long()`) non funzionano con i numeri complessi: non c'è alcun modo corretto per convertire un numero complesso in numero reale. Si usi `abs(z)` per ottenere la sua grandezza (in virgola mobila) o `z.real` per ottenere la sua parte reale.

```
>>> a=1.5+0.5j
>>> float(a)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
1.5
>>> abs(a)
1.58113883008
```

In modo interattivo, l'ultima espressione stampata è assegnata alla variabile _. Ciò facilita alquanto calcoli in successione quando si sta usando Python come calcolatrice da tavolo, a esempio:

```
>>> tassa = 17.5 / 100
>>> prezzo = 3.50
>>> prezzo * tassa
0.61249999999999993
>>> prezzo + _
4.1124999999999998
>>> round(_, 2)
4.1100000000000003
```

Questa variabile dev'essere trattata dall'utente come a sola lettura. Non le si deve assegnare esplicitamente un valore, si creerebbe una variabile locale indipendente con lo stesso nome che maschererebbe la variabile built-in e il suo comportamento particolare.

3.1.2 Stringhe

Oltre ai numeri, Python può anche manipolare stringhe, che possono essere espresse in diversi modi. Possono essere racchiuse tra apici singoli o virgolette:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
```

```
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Le stringhe letterali possono estendersi su più righe in modi diversi. Si possono proteggere i newline tramite le barre oblique inverse [il cosiddetto `escape' NdT], p.e.:

```
ciao = "Questa è una stringa abbastanza lunga che contiene\n\
parecchie linee di testo proprio come si farebbe in C.\n\
    Si noti che gli spazi bianchi all'inizio della riga sono\
 significativi.\n"
print ciao
```

che stamperà quanto segue:

```
Questa è una stringa abbastanza lunga che contiene
parecchie linee di testo proprio come si farebbe in C.
    Si noti che gli spazi bianchi all'inizio della riga sono significativi.
```

Oppure le stringhe possono essere circondate da un paio di virgolette o apici tripli corrispondenti: `"""` o `'''`. Non è necessario proteggere i caratteri di fine riga quando si usano le triple virgolette, essi saranno inclusi nella stringa.

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

produrrà il seguente output:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

L'interprete stampa il risultato delle operazioni sulle stringhe nello stesso modo in cui vengono inserite in input: comprese tra virgolette, e con virgolette [interne NdT] e altri caratteri particolari protetti da barre oblique inverse [`backslash'], per mostrare il loro esatto valore. La stringa è racchiusa tra doppie virgolette se contiene un apice singolo e

non virgolette doppie, altrimenti è racchiusa tra apici singoli. (L'istruzione `print`, descritta più avanti, può essere usata per scrivere stringhe senza virgolette o caratteri di escape).
Le stringhe possono essere concatenate (incollate assieme) tramite l'operatore `+`, e ripetute tramite `*`:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Due stringhe letterali consecutive vengono concatenate automaticamente; la prima linea dell'esempio precedente poteva anche essere scritta come "`word = 'Help' 'A'`"; questo funziona solo con due stringhe di testo, non con espressioni arbitrarie che comprendano stringhe:

```
>>> import string
>>> 'str' 'ing'                   # <- Questo è corretto
'string'
>>> string.strip('str') + 'ing'   # <- Questo è corretto
'string'
>>> string.strip('str') 'ing'     # <- Questo invece è errato
  File "<stdin>", line 1
    string.strip('str') 'ing'
                        ^
SyntaxError: invalid syntax
```

Le stringhe possono essere indicizzate come in C, il primo carattere di una stringa ha indice 0. Non c'è alcun tipo carattere distinto; un carattere è semplicemente una stringa di lunghezza uno. Come in Icon, possono essere specificate sottostringhe con la *notazione a fette* [`slice']: due indici separati da due punti.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

A differenza di quanto avviene in C, le stringhe Python non possono essere modificate. Un'assegnazione effettuata su un indice di una stringa costituisce un errore:

```
>>> word[0] = 'x'
Traceback (innermost last):
```

```
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:-1] = 'Splat'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Comunque creare una nuova stringa combinando i contenuti è semplice ed efficiente:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[-1:]
'SplatA'
```

Gli indici di fetta hanno utili comportamenti di default. Il primo indice, se omesso, viene posto uguale a 0 per default. Se viene tralasciato il secondo, viene posto alla dimensione della stringa affettata.

```
>>> word[:2]    # I primi due caratteri
'He'
>>> word[2:]    # Tutto tranne i primi due caratteri
'lpA'
```

Ecco un utile invariante delle operazioni di affettamento: `s[:i] + s[i:]` equivale a `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Gli indici di fetta degeneri vengono gestiti con eleganza: un indice troppo grande viene rimpiazzato con la dimensione della stringa, un indice destro minore di quello sinistro fa sì che venga restituita una stringa vuota.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Gli indici possono essere numeri negativi, per iniziare il conteggio da destra. A esempio:

```
>>> word[-1]     # L'ultimo carattere
'A'
>>> word[-2]     # Il penultimo carattere
'p'
>>> word[-2:]    # Gli ultimi due caratteri
```

```
'pA'
>>> word[:-2]    # Tutta la stringa eccetto i due ultimi caratteri
'Hel'
```

Si noti però che -0 è la stessa cosa di 0, quindi non si conta dall'estremità destra!

```
>>> word[-0]     # (dato che -0 è la stessa cosa di 0)
'H'
```

Gli indici di fetta negativi vengono troncati se sono fuori intervallo, ma non si tenti di farlo con indici a singolo elemento:

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # error
Traceback (innermost last):
  File "<stdin>", line 1
IndexError: string index out of range
```

Il modo migliore di tenere a mente come lavorano gli slice è di pensare che gli indici puntano *tra* i caratteri, con l'indice 0 posto al margine sinistro del primo carattere. Quindi il margine destro dell'ultimo carattere di una stringa di n caratteri ha indice n, per esempio:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

La prima riga di numeri dà la posizione degli indici 0...5 nella stringa; la seconda fornisce i corrispondenti indici negativi. La fetta da i a j consiste di tutti i caratteri compresi tra i margini contrassegnati$i$ e j, rispettivamente.

Per indici non negativi, l'ampiezza di una fetta è pari alla differenza tra gli indici, se entrambi sono compresi nei limiti, p.e. la lunghezza di `word[1:3]` è 2.

La funzione built-in `len()` restituisce la lunghezza di una stringa:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

3.1.3 Stringhe Unicode

A partire da Python 2.0 il programmatore ha a sua disposizione un nuovo tipo di dato testo: l'oggetto Unicode. Può essere utilizzato per immagazzinare e manipolare dati Unicode (si veda http://www.unicode.org). Esso si integra al meglio con gli oggetti stringa esistenti, garantendo conversioni in automatico ove necessario.

Unicode ha il vantaggio di fornire un unico ordinale per ogni carattere che possa comparire in un qualsiasi testo. In precedenza c'erano solo 256 ordinali possibili per i caratteri scrivibili e ciascun testo era tipicamente collegato a una code page che mappava gli ordinali sui caratteri scrivibili. Ciò provocava molta confusione specialmente riguardo al problema dell'internazionalizzazione del software (di solito indicata come "`i18n`" -- "`i`" + 18 caratteri + "`n`"). Unicorde ha risolto tali problemi definendo un'unica code page per tutti gli script.

Creare stringhe Unicode in Python è semplice quanto creare stringhe normali:

```
>>> u'Hello World !'
u'Hello World !'
```

Il carattere "`u`" minuscolo davanti agli apici indica che si vuole creare una stringa Unicode. Se si desidera includere nella stringa caratteri speciali, lo si può fare usando la codifica Python *Unicode-Escape*. Il seguente esempio mostra come fare:

```
>>> u'Hello\\u0020World !'
u'Hello World !'
```

La sequenza di escape `u0020` inserisce il carattere Unicode con ordinale esadecimale 0x0020 (il carattere di spazio) nella posizione indicata.

Gli altri caratteri sono interpretati usando il valore del loro rispettivo ordinale direttamente come Unicode. Grazie al fatto che i primi 256 caratteri Unicode sono gli stessi della codifica standard Latin-1 usata in molti paesi occidentali, l'inserimento Unicode risulta grandemente semplificato.

Per gli esperti, c'è anche una modalità `raw', proprio come per le stringhe normali. Si deve prefissare una 'r' minuscola alla stringa per far sì che Python usi la codifica *Raw-Unicode-Escape*. Non farà altro che applicare la conversione `uXXXX` di cui sopra nel caso ci sia un numero dispari di barre oblique inverse davanti alla 'u' minuscola.

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

La modalità `raw' è utile perlopiù quando si devono introdurre un sacco di barre oblique inverse, p.e. nelle espressioni regolari.

A parte queste codifiche standard, Python fornisce un insieme completo di altri mezzi per creare stringhe Unicode partendo da una codifica conosciuta.

La funzione built-in `unicode()` permette l'accesso a tutti i codec (COdificatori e DECodificatori) Unicode ufficiali. Alcune delle codifiche più note nelle quali tali codecs possono effettuare la conversione sono: *Latin-1*, *ASCII*, *UTF-8* e *UTF-16*. Le ultime due sono codifiche a lunghezza variabile che permettono di memorizzare caratteri Unicode in 8 o 16 bit. Python use UTF-8 come codifica di default. Questo diventa più evidente quando si stampano o si scrivono su file stringhe Unicode.

```
>>> u"äöü"
u'\344\366\374'
>>> str(u"äöü")
'\303\244\303\266\303\274'
```

Se si hanno dei dati in una codifica specifica e si vuole produrre una stringa Unicode corrispondente, si può usare la funzione built-in `unicode()`, con il nome della codifica come secondo argomento.

```
>>> unicode('\303\244\303\266\303\274','UTF-8')
u'\344\366\374'
```

Per riconvertire la stringa Unicode in una stringa che usa la codifica originale, gli oggetti forniscono un metodo `encode()`.

```
>>> u"äöü".encode('UTF-8')
'\303\244\303\266\303\274'
```

3.1.4 Liste

Python riconosce una certa quantità di tipi di dati *composti*, usati per raggruppare insieme altri valori. Il più versatile è il tipo *lista*, che può essere scritto come una lista, compresa tra parentesi quadre, di valori (gli elementi della lista) separati da virgole. Gli elementi della lista non devono essere necessariamente tutti dello stesso tipo.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Come per gli indici delle stringhe, gli indici delle liste iniziano da 0, e anche le liste possono essere affettate, concatenate e così via:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

Al contrario delle stringhe, che sono *immutabili*, è possibile modificare gli elementi individuali di una lista:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

È anche possibile assegnare valori alle fette, e questo può pure modificare le dimensioni della lista:

```
>>> # Rimpiazza alcuni elementi:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Rimuove alcuni elementi:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Inserisce alcuni elementi:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a     # Inserisce (una copia di) se stesso all'inizio
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

La funzione built-in `len()` si applica anche alle liste:

```
>>> len(a)
8
```

È possibile avere delle liste annidate (contenenti cioè altre liste), a esempio:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Si veda la sezione 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Si noti che nell'ultimo esempio, `p[1]` e `q` si riferiscono proprio allo stesso oggetto! Ritorneremo più avanti sulla *semantica degli oggetti*.

3.2 Primi Passi Verso la Programmazione

Di certo possiamo usare Python per compiti più complessi che fare 2+2. Per esempio, possiamo scrivere una sottosuccessione iniziale della serie di *Fibonacci* facendo come segue:

```
>>> # Serie di Fibonacci:
... # la somma di due elementi definisce l'elemento successivo
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Questo esempio introduce parecchie nuove funzionalità.

- La prima riga contiene un *assegnamento multiplo*: le variabili `a` e `b` ricevono simultaneamente i nuovi valori 0 e 1. Ciò viene fatto di nuovo nell'ultima linea, a dimostrazione che le espressioni sul lato destro dell'assegnamento vengono tutte valutate prima di effettuare qualsiasi assegnamento.
- Il ciclo `while` viene eseguito fino a quando la condizione (in questo caso `b < 10`) rimane vera. In Python, come in C, qualsiasi valore diverso da zero è vero; zero è falso. La condizione può anche essere il valore di una stringa o di una lista, di fatto di una sequenza qualsiasi; qualsiasi cosa con una lunghezza diversa da zero ha valore logico vero, sequenze vuote hanno valore logico falso. Il test usato nell'esempio è una semplice comparazione. Gli operatori standard di confronto sono scritti come in C: < (minore di), > (maggiore di), == (uguale a), <= (minore o uguale a), >= (maggiore o uguale a) e `!=` (diverso da).
- Il *corpo* del ciclo è *indentato*: l'indentazione è il sistema con cui Python raggruppa le istruzioni. Python non possiede (per il momento!) un servizio di editing intelligente dell'input da riga di comando, per cui si devono introdurre una tabulazione o uno o più spazi per ogni riga indentata. In pratica si dovranno trattare gli input per Python più complicati con un editor di testo; la maggior parte degli editor di testo hanno una funzione di auto-indentazione. Quando un'istruzione composta viene immessa interattivamente, dev'essere seguita da una riga vuota per indicarne il completamento (dato che il parser non può sapere quando si è digitata l'ultima linea). Si osservi che ogni linea all'interno di un blocco base dev'essere indentata in ugual misura.
- L'istruzione `print` stampa a video il valore dell'espressione, o delle espressioni, che le viene passata. Differisce dal semplice scrivere l'espressione che si vuole (come abbiamo fatto precedentemente negli esempi sull'uso come calcolatrice) per il modo in cui gestisce espressioni multiple e stringhe. Le stringhe vengono stampate senza virgolette, e viene inserito uno spazio tra gli elementi, quindi si possono formattare elegantemente i risultati, come in questo esempio:
- >>> i = 256*256
- >>> print 'Il valore di i è', i
- Il valore di i è 65536

Una virgola finale evita l'andare a capo (newline) dopo l'output:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Si noti che l'interprete inserisce un newline prima di stampare il prompt successivo se l'ultima linea non è stata completata.

4. Di più sugli strumenti di controllo del flusso

Oltre all'istruzione `while` appena introdotta, Python riconosce le solite istruzioni di controllo del flusso presenti in altri linguaggi, con qualche particolarità.

4.1 L'Istruzione `if`

Forse il tipo di istruzione più conosciuta è `if`. Per esempio:

```
>>> x = int(raw_input("Introdurre un numero: "))
>>> if x < 0:
...      x = 0
...      print 'Numero negativo cambiato in zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Uno'
... else:
...      print 'Più di uno'
...
```

Possono essere presenti, o meno, una o più parti `elif`, e la parte `else` è opzionale. La parola chiave `elif' è un'abbreviazione di `else if', e serve ad evitare un eccesso di indentazioni. Una sequenza `if` ... `elif` ... `elif` ... sostituisce le istruzioni *switch* o *case* che si trovano in altri linguaggi.

4.2 L'Istruzione `for`

L'istruzione `for` di Python differisce un po' da quella a cui si è abituati in C o Pascal. Piuttosto che iterare sempre su una progressione aritmetica (come in Pascal), o dare all'utente la possibilità di definire sia il passo iterativo che la condizione di arresto (come in C), in Python l'istruzione `for` compie un'iterazione sugli elementi di una qualsiasi sequenza (p.e. una lista o una stringa), nell'ordine in cui appaiono nella sequenza. A esempio (senza voler fare giochi di parole!):

```
>>> # Misura la lunghezza di alcune stringhe:
... a = ['gatto', 'finestra', 'defenestrare']
>>> for x in a:
...      print x, len(x)
...
gatto 5
```

```
finestra 8
defenestrare 12
```

Non è prudente modificare all'interno del ciclo la sequenza su cui avviene l'iterazione (può essere fatto solo per tipi di sequenze mutabili, per dire, liste). Se è necessario modificare la lista su cui si effettua l'iterazione, p.e. duplicare elementi scelti, si deve iterare su una copia. La notazione a fette rende questo procedimento particolarmente conveniente:

```
>>> for x in a[:]: # fa una copia tramite affettamento dell'intera lista
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrare', 'gatto', 'finestra', 'defenestrare']
```

4.3 La Funzione `range()`

Se è necessario iterare su una successione di numeri, viene in aiuto la funzione built-in `range()`, che genera liste contenenti progressioni aritmetiche, p.e.:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

L'estremo destro passato alla funzione non fa mai parte della lista generata; `range(10)` genera una lista di 10 valori, esattamente gli indici leciti per gli elementi di una sequenza di lunghezza 10. È possibile far partire l'intervallo da un altro numero, o specificare un incremento diverso (persino negativo, talvolta è chiamato "passo" [`step']):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Per effettuare un'iterazione sugli indici di una sequenza, si usino in combinazione `range()` e `len()` come segue:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
```

```
4 lamb
```

4.4 Le Istruzioni `break` e `continue`, e la Clausola `else` nei Cicli

L'istruzione `break`, come in C, esce immediatamente dal ciclo `for` o `while` più interno che la racchiude.

L'istruzione `continue`, anch'essa presa a prestito dal C, prosegue con l'iterazione seguente del ciclo.

Le istruzioni di ciclo possono avere una clausola `else` che viene eseguita quando il ciclo termina per esaurimento della lista (con `for`) o quando la condizione diviene falsa (con `while`), ma non quando il ciclo è terminato da un'istruzione `break`. Ciò è esemplificato nel ciclo seguente, che ricerca numeri primi:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'è uguale a', x, '*', n/x
...             break
...     else:
...         print n, 'è un numero primo'
...
2 è un numero primo
3 è un numero primo
4 è uguale a 2 * 2
5 è un numero primo
6 è uguale a 2 * 3
7 è un numero primo
8 è uguale a 2 * 4
9 è uguale a 3 * 3
```

4.5 L'Istruzione `pass`

L'istruzione `pass` non fa nulla. Può essere usata quando un'istruzione è necessaria per sintassi ma il programma non richiede venga svolta alcuna azione. Per esempio:

```
>>> while 1:
...       pass # In attesa di un interrupt da tastiera
...
```

4.6 Definizione di Funzioni

Possiamo creare una funzione che scrive la serie di Fibonacci fino ad un limite arbitrario:

```
>>> def fib(n):    # scrive la serie di Fibonacci fino a n
...     "Stampa una serie di Fibonacci fino a n"
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Adesso si invochi la funzione appena definita:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

La parola chiave `def` introduce una *definizione* di funzione. Dev'essere seguita dal nome della funzione e dalla lista dei parametri formali racchiusa tra parentesi. Le istruzioni che compongono il corpo della funzione iniziano alla linea successiva, indentate da uno stop di tabulazione. La prima istruzione del corpo della funzione può essere facoltativamente una stringa di testo: è la stringa di documentazione della funzione o *docstring*. Ci sono degli strumenti di sviluppo che usano le docstring per produrre automaticamente documentazione stampata, o per permettere all'utente una navigazione interattiva attraverso il codice. È buona abitudine includere le docstring nel proprio codice, si cerchi quindi di farci l'abitudine.

L'*esecuzione* di una funzione introduce una nuova tabella di simboli, usata per le variabili locali della funzione. Più precisamente, tutti gli assegnamenti di variabili in una funzione memorizzano il valore nella tabella dei simboli locale, laddove i riferimenti a variabili cercano prima nella tabella dei simboli locale, poi nella tabella dei simboli globale, e quindi nella tabella dei nomi interni. Di conseguenza ad una variabile globale non può essere assegnato direttamente un valore entro una funzione (a meno che non compaia in un'istruzione `global`), malgrado la variabile possa essere referenziata.

I parametri attuali (argomenti) di una chiamata di funzione sono introdotti nella tabella dei simboli locale della funzione al momento della chiamata; perciò gli argomenti sono passati usando una *chiamata per valore* (dove il *valore* è sempre un *riferimento* a un oggetto, non il valore dell'oggetto).[4.1] Quando una funzione chiama un'altra funzione, viene creata una nuova tabella locale dei simboli per tale chiamata.

Una definizione di funzione introduce il nome della funzione nella tabella dei simboli corrente. Il valore del nome della funzione ha un tipo che è riconosciuto dall'interprete come funzione definita dall'utente. Tale valore può essere assegnato a un altro nome che indi può venire anch'esso usato come una funzione. Questo serve come meccanismo generale di ridenominazione:

```
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Si potrebbe obiettare che `fib` non è una funzione ma una procedura. In Python, come in C, le procedure sono semplicemente funzioni che non restituiscono un valore. In effetti, tecnicamente parlando, le procedure restituiscono comunque un valore, quantunque piuttosto inutile. Tale valore è chiamato `None` (è un nome interno). La scrittura del valore `None` è di norma soppressa dall'interprete nel caso si tratti dell'unico valore scritto. Lo si può vedere se si vuole:

```
>>> print fib(0)
None
```

È facile scrivere una funzione che restituisce una lista dei numeri delle serie di Fibonacci anziché stamparli:

```
>>> def fib2(n): # restituisce la serie di Fibonacci fino a n
...     "Restituisce una lista contenente la serie di Fibonacci fino a n"
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # vedi sotto
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # chiama la funzione
>>> f100                # scrive il risultato
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Questo esempio, come al solito, fornisce un esempio di alcune nuove funzionalità di Python:

- L'istruzione `return` restituisce una funzione con un valore. `return` senza un'espressione argomento è usato per ritornare dall'interno di una procedura (anche finire oltre il suo

estremo è un modo di ritornare da una procedura), nel qual caso viene restituito il valore `None`.

- L'istruzione `result.append(b)` chiama un *metodo* dell'oggetto lista `result`. Un metodo è una funzione che `appartiene' ad un oggetto ed ha per nome `oggetto.metodo`, dove `oggetto` è un qualche oggetto (può essere un'espressione), e `metodo` è il nome di un metodo che è definito secondo il tipo di oggetto. Tipi diversi definiscono metodi diversi. Metodi di tipi diversi possono avere lo stesso nome senza causare ambiguità. (È possibile definire tipi di oggetto e metodi propri, usando le *classi*, come trattato più avanti in questo tutorial). Il metodo `append()` visto nell'esempio è definito per gli oggetti lista; esso aggiunge un nuovo elemento alla fine della lista. Nell'esempio riportato è equivalente a "`result = result + [b]`", ma è più efficiente.

4.7 Di più sulla Definizione di Funzioni

È anche possibile definire funzioni con un numero variabile di argomenti. Ci sono tre forme, che possono essere combinate.

4.7.1 Valori di Default per gli Argomenti

La forma più utile è specificare un valore predefinito per uno o più argomenti. In questo modo si crea una funzione che può essere chiamata con un numero di argomenti minore rispetto alla definizione, p.e.

```
def ask_ok(prompt, retries=4, complaint='Sì o no, grazie!'):
    while 1:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return 1
        if ok in ('n', 'no', 'nop', 'nope'): return 0
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint
```

Questa funzione può essere chiamata così: `ask_ok('Vuoi davvero uscire')` o così: `ask_ok('Devo sovrascrivere il file?', 2)`.

I valori di default sono valutati al momento della definizione della funzione nello `scope' *di definizione*, così che p.e.

```
i = 5
def f(arg = i): print arg
i = 6
f()
```

stamperà `5`.

Avviso importante: Il valore di default viene valutato una volta sola. Ciò fa sì che le cose siano molto diverse quando si tratta di un oggetto mutabile come una lista o un dizionario. A esempio, la seguente funzione accumula gli argomenti ad essa passati in chiamate successive:

```
def f(a, l = []):
    l.append(a)
    return l
print f(1)
print f(2)
print f(3)
```

Che stamperà:

```
[1]
 [1, 2]
 [1, 2, 3]
```

Se si desidera che il valore di default non venga condiviso tra chiamate successive, si può scrivere la funzione in questo modo:

```
def f(a, l = None):
    if l is None:
        l = []
    l.append(a)
    return l
```

4.7.2 Argomenti a Parola Chiave

Le funzioni possono essere chiamate anche usando argomenti a parola chiave nella forma "*parolachiave = valore*". Per esempio la funzione seguente:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "Volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

potrebbe essere chiamata in uno qualsiasi dei seguenti modi:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

invece le chiamate seguenti non sarebbero valide:

```
parrot()                    # manca un argomento necessario
parrot(voltage=5.0, 'dead')  # argomento non a parola chiave di seguito a una parola chiave
```

```
parrot(110, voltage=220)     # valore doppio per un argomento
parrot(actor='John Cleese')  # parola chiave sconosciuta
```

In generale, una lista di argomenti deve avere un numero qualunque di argomenti posizionali seguiti da zero o più argomenti a parola chiave, ove le parole chiave devono essere scelte tra i nomi dei parametri formali. Non è importante se un parametro formale ha un valore di default o meno. Nessun argomento deve ricevere un valore più di una volta --- in una medesima invocazione non possono essere usati come parole chiave nomi di parametri formali corrispondenti ad argomenti posizionali. Ecco un esempio di errore dovuto a tale restrizione:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

Quando è presente un parametro formale finale della forma `**`*nome*, esso riceve un dizionario contenente tutti gli argomenti a parola chiave la cui parola chiave non corrisponde a un parametro formale. Ciò può essere combinato con un parametro formale della forma `*`*nome* (descritto nella prossima sottosezione) che riceve una tupla contenente gli argomenti posizionali in eccesso rispetto alla lista dei parametri formali. (`*`*nome* deve trovarsi prima di `**`*nome*). Per esempio, se si definisce una funzione come la seguente:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, '?'
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print '-'*40
    for kw in keywords.keys(): print kw, ':', keywords[kw]
```

Essa potrà venire invocata così:

```
cheeseshop('Limburger', "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           client='John Cleese',
           shopkeeper='Michael Palin',
           sketch='Cheese Shop Sketch')
```

e naturalmente stamperà:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
```

```
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

4.7.3 Liste di Argomenti ad Arbitrio

Infine, l'opzione usata meno di frequente è specificare che una funzione può essere chiamata con un numero arbitrario di argomenti. Tali argomenti verranno incapsulati in una tupla. Prima degli argomenti in numero variabile possono esserci zero o più argomenti normali.

```
def fprintf(file, format, *args):
    file.write(format % args)
```

4.7.4 Forme Lambda

A seguito di numerose richieste, a Python sono state aggiunte alcune (poche) funzionalità che si trovano comunemente nei linguaggi di programmazione funzionale e in Lisp. Con la parola chiave `lambda` possono essere create piccole funzioni senza nome. Ecco una funzione che ritorna la somma dei suoi due argomenti: "`lambda a, b: a+b`". Le forme lambda possono essere usate ovunque siano richiesti oggetti funzione. Esse sono sintatticamente ristrette ad una singola espressione. Dal punto di vista semantico, sono solo un surrogato di una normale definizione di funzione. Come per le definizioni di funzioni annidate, le forme lambda non possono referenziare variabili dallo scope che le contiene, ma questo può essere aggirato attraverso un uso giudizioso dei valori di default per gli argomenti, p.e.

```
def make_incrementor(n):
    return lambda x, incr=n: x+incr
```

4.7.5 Stringhe di Documentazione

Si stanno formando delle convenzioni sul contenuto e la formattazione delle stringhe di documentazione.

La prima riga dovrebbe essere sempre un sommario, breve e conciso, della finalità dell'oggetto. Per brevità, non vi dovrebbero comparire in forma esplicita il tipo o il nome dell'oggetto, dato che essi sono disponibili attraverso altri mezzi (eccetto nel caso in cui il

nome sia un verbo che descriva un'azione della funzione). La riga dovrebbe iniziare con una lettera maiuscola e terminare con un punto.

Se la stringa di documentazione è composta da più righe, la seconda dovrebbe essere vuota, per separare visivamente il sommario dal resto della descrizione. Le righe successive dovrebbero costituire uno o più paragrafi che descrivono le convenzioni di chiamata dell'oggetto, i suoi effetti collaterali ecc.

L'analizzatore sintattico (parser) di Python non elimina l'indentazione da stringhe di testo di più righe, perciò i programmi utilità che processano la documentazione devono toglierla da sè. Ciò viene fatto usando la seguente convenzione. La prima linea non vuota *dopo* la prima linea della stringa determina il totale dell'indentazione presente nell'intera stringa di documentazione. (Non si può usare la prima linea poiché di solito è adiacente alle virgolette di apertura della stringa, quindi la sua indentazione non è chiara nella stringa di testo). Gli spazi equivalenti a tale indentazione vengono quindi eliminati dall'inizio di tutte le linee della stringa. Non dovrebbero esserci linee indentate in misura minore, ma se ci fossero tutti gli spazi in testa alla riga dovrebbero venir eliminati. L'equivalenza in spazi dovrebbe essere saggiata dopo l'espansione dei tab (normalmente a 8 spazi).

Ecco un esempio di stringa di documentazione su più righe:

```
>>> def my_function():
...     """Non fa nulla, ma lo documenta.
...
...     Davvero, non fa proprio nulla.
...     """
...     pass
...
>>> print my_function.__doc__
Non fa nulla, ma lo documenta.

    Davvero, non fa proprio nulla.
```

5. Strutture Dati

Questo capitolo descrive in maggiore dettaglio alcuni punti che sono già stati affrontati, e ne aggiunge pure alcuni nuovi.

5.1 Di più sulle Liste

Il tipo di dato lista è dotato di ulteriori metodi. Ecco tutti i metodi degli oggetti lista:

`append(x)`
Aggiunge un elemento in fondo alla lista; equivale a `a[len(a):] = [x]`.

`extend(L)`
Estende la lista aggiungendovi in coda tutti gli elementi della lista fornita; equivale a `a[len(a):] = L`.

`insert(i, x)`
Inserisce un elemento nella posizione fornita. Il primo argomento è l'indice dell'elemento davanti al quale va effettuato l'inserimento, quindi `a.insert(0, x)` inserisce `x` in testa alla lista, e `a.insert(len(a), x)` equivale a `a.append(x)`.

`remove(x)`
Rimuove il primo elemento della lista il cui valore è `x`. L'assenza di tale elemento produce un errore.

`pop([i])`
Rimuove l'elemento di indice `i` e lo restituisce come risultato dell'operazione. Se non è specificato alcun indice, `a.pop()` restituisce l'ultimo elemento della lista. L'elemento viene comunque rimosso dalla lista.

`index(x)`
Restituisce l'indice del primo elemento della lista il cui valore è `x`. L'assenza di tale elemento produce un errore.

`count(x)`
Restituisce il numero di occorrenze di `x` nella lista.

`sort()`
Ordina gli elementi della lista, sul posto [l'output è la stessa lista riordinata NdT].

`reverse()`
Inverte gli elementi della lista, sul posto [come sopra NdT].

Un esempio che utilizza buona parte dei metodi delle liste:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
```

```
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 Usare le Liste come Pile

I metodi delle liste rendono assai facile utilizzare una lista come una pila, dove l'ultimo elemento aggiunto è il primo ad essere prelevato (``last-in, first-out''). Per aggiungere un elemento in cima alla pila, si usi `append()`. Per ottenere un elemento dalla sommità della pila si usi `pop()` senza un indice esplicito. Per esempio:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Usare le Liste come Code

Si può anche usare convenientemente una lista come una coda, dove il primo elemento aggiunto è il primo ad essere prelevato (``first-in, first-out''). Per aggiungere un elemento in fondo alla coda, si usi `append()`. Per ottenere l'elemento in testa, si usi `pop()` con indice `0`. per esempio:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Aggiunge Terry
>>> queue.append("Graham")          # Aggiunge Graham
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

5.1.3 Strumenti per la Programmazione Funzionale

Ci sono tre funzioni interne molto utili quando usate con le liste: `filter()`, `map()`, e `reduce()`.

"`filter(`*funzione*`, `*sequenza*`)`" restituisce una sequenza (dello stesso tipo, ove possibile) composta dagli elementi della sequenza originale per i quali è vera *funzione*`(`*elemento*`)`. Per esempio, per calcolare alcuni numeri primi:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

"`map(`*funzione*`, `*sequenza*`)`" invoca *funzione*`(`*elemento*`)` per ciascuno degli elementi della sequenza e restituisce una lista dei valori ottenuti. Per esempio, per calcolare i cubi di alcuni numeri:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Può essere passata più di una sequenza; in questo caso la funzione deve avere tanti argomenti quante sono le sequenze e viene invocata con gli elementi corrispondenti di ogni sequenza (o `None` se qualche sequenza è più breve di altre). Se al posto della funzione viene passato `None`, esso viene sostituito con una funzione che restituisce i propri argomenti.

Combinando questi due casi speciali, si può vedere che "`map(None, `*lista1*`, `*lista2*`)`" è un modo conveniente di trasformare una coppia di liste in una lista di coppie. A esempio:

```
>>> seq = range(8)
>>> def square(x): return x*x
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

"`reduce(`*funzione*`, `*sequenza*`)`" restituisce un singolo valore ottenuto invocando la funzione binaria [a due argomenti NdT] *funzione* sui primi due elementi della sequenza, quindi sul risultato dell'operazione e sull'elemento successivo, e così via. A esempio, per calcolare la somma dei numeri da 1 a 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

Se la sequenza è composta da un unico elemento, viene restituito il suo valore; se la successione è vuota, viene sollevata un'eccezione.
Si Può anche passare un terzo argomento per indicare il valore di partenza. In tal caso è il valore di partenza ad essere ritornato se la sequenza è vuota, e la funzione è applicata prima a tale valore e al primo elemento della sequenza, quindi al risultato di tale operazione e all'elemento successivo, e così via. Per esempio:

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

5.1.4 Descrizioni di Lista

Le descrizioni di lista forniscono un modo conciso per creare liste senza ricorrere all'uso di `map()`, `filter()` e/o `lambda`. La definizione risultante è spesso più comprensibile di ciò che si ottiene con le funzioni di cui sopra. Ogni descrizione di lista consiste di un'espressione fatta seguire da una clausola `for`, quindi zero o più clausole`for` o `if`. Il risultato sarà una lista creata valutando l'espressione nel contesto delle clausole `for` e `if` che la seguono. Se l'espressione valuta una tupla, questa dev'essere racchiusa tra parentesi.

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]            # errore - per le tuple è richiesta la parentesi
```

```
  File "<stdin>", line 1
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

5.2 L'Istruzione `del`

C'è un modo per rimuovere un elemento da una lista secondo il suo indice piuttosto che secondo il suo valore: l'istruzione `del`. È possibile usarla anche per rimuovere fette di una lista (cosa che precedentemente abbiamo ottenuto assegnando una lista vuota alle fette). Per esempio:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

`del` può essere usata anche per eliminare semplici variabili:

```
>>> del a
```

Fare riferimento al nome `a` dopo di ciò costituirà un errore (almeno fino a quando non gli verrà assegnato un altro valore). Si vedranno altri usi di `del` più avanti.

5.3 Tuple e Sequenze

Si è visto come stringhe e liste abbiano molte proprietà in comune, p.e. le operazioni di indicizzazione e affettamento. Si tratta di due esempi di tipi di dati del genere *sequenza*. Dato che Python è un linguaggio in evoluzione, potranno venir aggiunti altri tipi di dati dello stesso genere. C'è anche un altro tipo di dato standard del genere sequenza: la *tupla*.

Una tupla è composta da un certo numero di valori separati da virgole, per esempio:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Le tuple possono essere annidate:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Come si può vedere, una tupla è sempre racchiusa tra parentesi nell'output, cosicché le tuple annidate possono essere interpretate correttamente; possono essere introdotte con o senza parentesi che le racchiudano, sebbene spesso queste siano comunque necessarie (se la tupla è parte di un'espressione più vasta).

Le tuple hanno molti usi, p.e. coppie di coordinate (x, y), record da un database ecc. Le tuple, come le stringhe, sono immutabili: non è possibile effettuare assegnamenti a elementi individuali di una tupla (sebbene si possa imitare quasi lo stesso effetto a mezzo affettamenti e concatenazioni). È anche possibile creare tuple che contengano oggetti mutabili, come liste.

Un problema particolare è la costruzione di tuple contenenti zero o un elementi: la sintassi fornisce alcuni sotterfugi per ottenerle. Le tuple vuote vengono costruite usando una coppia vuota di parentesi; una tupla con un solo elemento è costruita facendo seguire a un singolo valore una virgola (non è infatti sufficiente racchiudere un singolo valore tra parentesi). Brutto, ma efficace. Per esempio:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- si noti la virgola alla fine
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

L'istruzione `t = 12345, 54321, 'hello!'` è un esempio di *impacchettamento in tupla*: i valori `12345`, `54321` e `'hello!'` sono impacchettati assieme in una tupla. È anche possibile l'operazione inversa, p.e.:

```
>>> x, y, z = t
```

È chiamata, in modo piuttosto appropriato, *spacchettamento di sequenza*. Lo spacchettamento di sequenza richiede che la lista di variabili a sinistra abbia un numero di elementi pari alla lunghezza della sequenza. Si noti che l'assegnamento multiplo è in realtà solo una combinazione di impacchettamento in tupla e spacchettamento di sequenza!
C'è una certa asimmetria qui: l'impacchettamento di valori multipli crea sempre una tupla, mentre lo spacchettamento funziona per qualsiasi sequenza.

5.4 Dizionari

Un altro tipo nativo di dato utile è il *dizionario*. I dizionari si trovano a volte in altri linguaggi come `memorie associative' o `array associativi'. A differenza delle sequenze, che sono indicizzate secondo un intervallo numerico, i dizionari sono indicizzati tramite *chiavi*, che possono essere di un qualsiasi tipo immutabile. Stringhe e numeri possono essere usati come chiavi in ogni caso, le tuple possono esserlo se contengono solo stringhe, numeri o tuple; se una tupla contiene un qualsivoglia oggetto mutabile, sia direttamente che indirettamente, non può essere usata come chiave. Non si possono usare come chiavi le liste, dato che possono essere modificate usando i loro metodi `append()` e `extend()`, come pure con assegnamenti su fette o indici.
La cosa migliore è pensare a un dizionario come un insieme non ordinato di coppie *chiave:valore*, con il requisito che ogni chiave dev'essere unica (all'interno di un dizionario). Una coppia di parentesi graffe crea un dizionario vuoto: `{}`. Mettendo tra parentesi graffe una lista di coppie *chiave:valore* separate da virgole si ottengono le coppie iniziali del dizionario; nello stesso modo i dizionari vengono stampati sull'output.
Le operazioni principali su un dizionario sono la memorizzazione di un valore con una qualche chiave e l'estrazione del valore corrispondente a una data chiave. È anche possibile cancellare una coppia *chiave:valore* con `del`. Se si memorizza un valore usando una chiave già in uso, il vecchio valore associato alla chiave viene sovrascritto. Cercare di estrarre un valore usando una chiave non presente nel dizionario produce un errore.
Il metodo `keys()` di un oggetto dizionario restituisce una lista di tutte le chiavi usate nel dizionario in oggetto, in ordine casuale (se la si vuole ordinata, basta applicare il metodo `sort()` alla lista delle chiavi). Per verificare se una data chiave si trova nel dizionario, si può usare il metodo `has_key()`.
Ecco un piccolo esempio di operazioni su un dizionario:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1
```

5.5 Di Più sulle Condizioni

Le condizioni usate nelle istruzioni `while` e `if` possono contenere altri operatori oltre a quelli di confronto classici.

Gli operatori di confronto `in` e `not in` verificano se un valore compare (o non compare) in una sequenza. Gli operatori `is` e `is not` confrontano due oggetti per vedere se siano in realtà lo stesso oggetto; questo ha senso solo per oggetti mutabili, come le liste. Tutti gli operatori di confronto hanno la stessa priorità, che è minore di quella di tutti gli operatori matematici.

I confronti possono essere concatenati: p.e., `a < b == c` verifica se `a` sia minore di `b` e inoltre se `b` eguagli `c`.

Gli operatori di confronto possono essere combinati tramite gli operatori booleani `and` e `or`, e il risultato di un confronto (o di una qualsiasi altra espressione booleana) può essere negato con `not`. Inoltre questi tre operatori hanno priorità ancora minore degli operatori di confronto; tra di essi `not` ha la priorità più alta e `or` la più bassa, per cui `A and not B or C` è equivalente a `(A and (not B)) or C`. Naturalmente per ottenere la composizione desiderata possono essere usate delle parentesi.

Gli operatori booleani `and` e `or` sono dei cosiddetti operatori *shortcut* : i loro argomenti vengono valutati da sinistra a destra e la valutazione si ferma non appena viene determinato il risultato. Per esempio se `A` e `C` sono veri ma `B` è falso, `A and B and C` non

valuta l'espressione C. In generale, il valore restituito da un operatore shortcut, quando usato come valore generico e non come booleano, è l'ultimo argomento valutato.

È possibile assegnare il risultato di un confronto o di un'altra espressione booleana a una variabile. A esempio,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Si noti che in Python, a differenza che in C, all'interno delle espressioni non può comparire un assegnamento. I programmatori C potrebbero lamentarsi di questa scelta, però essa evita un tipo di problema che è facile incontrare nei programmi C: l'introduzione di `=` in un'espressione volendo invece intendere `==`.

5.6 Confrontare Sequenze con altri Tipi di Dati

Gli oggetti Sequenza possono essere confrontati con altri oggetti Sequenza dello stesso tipo. Il confronto utilizza un ordinamento *lessicografico*: innanzitutto vengono confrontati i primi due elementi [delle due sequenze NdT] e, se questi differiscono tra di loro, ciò determina il risultato del confronto; se sono uguali, vengono quindi confrontati tra di loro i due elementi successivi, e così via, fino a che una delle sequenze si esaurisce. Se due elementi che devono essere confrontati sono essi stessi delle sequenze dello stesso tipo, il confronto lessicografico viene effettuato ricorsivamente. Se tutti gli elementi delle due sequenze risultano uguali al confronto, le sequenze sono considerate uguali. Se una sequenza è una sottosequenza iniziale dell'altra, la sequenza più breve è la minore. L'ordinamento lessicografico per le stringhe usa l'ordine ASCII per i singoli caratteri. Ecco alcuni esempi di confronti tra sequenze dello stesso tipo:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)              == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Si noti che confrontare oggetti di tipi diversi è lecito. Il risultato è deterministico ma arbitrario: i tipi vengono ordinati secondo il loro nome. Perciò una lista è sempre minore di

una stringa, una stringa è sempre minore di una tupla, ecc. Tipi numerici eterogenei vengono confrontati secondo il loro valore numerico, così 0 è uguale a 0.0, ecc.[5.1]

6. Moduli

Se si esce dall'interprete Python e poi vi si rientra, le definizioni introdotte (funzioni e variabili) vengono perse. Perciò, se si desidera scrivere un programma un po' più lungo, è meglio usare un editor di testo per preparare un file da usare come input al lancio dell'interprete. Ciò viene chiamato `creazione di uno *script*'.Con l'aumentare delle dimensioni del proprio programma, si potrebbe volerlo suddividere in più file per facilitarne la manutenzione. Si potrebbe anche voler riutilizzare in programmi diversi una funzione utile che si è già scritta senza doverne copiare l'intera definizione in ogni programma.

A tale scopo, Python permette di porre le definizioni in un file e usarle in uno script o in una sessione interattiva dell'interprete. Un file di questo tipo si chiama *modulo*; le definizioni presenti in un modulo possono essere *importate* in altri moduli o entro il modulo *main* (la collezione di variabili cui si ha accesso in uno script eseguito al livello più alto e in modalità calcolatrice).

Un modulo è un file che contiene definizioni e istruzioni Python. Il nome del file è il nome del modulo con il suffisso .py aggiunto. All'interno di un modulo, il nome del modulo è disponibile (sotto forma di una stringa) come valore della variabile globale `__name__`. Per esempio, si usi il proprio editor di testo favorito per creare un file chiamato fibo.py nella directory corrente che contenga quanto segue:

```
# modulo dei numeri di Fibonacci

def fib(n):    # scrive le serie di Fibonacci fino a n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # restituisce le serie di Fibonacci fino a n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Ora si lanci l'inteprete Python e si importi questo modulo con il seguente comando:

```
>>> import fibo
```

Ciò non introduce i nomi delle funzioni definite in `fibo` direttamente nella tabella dei simboli corrente; vi introduce solo il nome del modulo `fibo`. Usando il nome del modulo è possibile accedere alle funzioni:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Se si intende usare spesso una funzione, si può assegnare ad essa un nome locale:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Di più sui Moduli

Un modulo può contenere istruzioni eseguibili oltre che definizioni di funzione. Queste istruzioni servono a inizializzare il modulo. Esse sono eseguite solo la *prima* volta che il modulo viene importato da qualche parte.[6.1]

Ogni modulo ha la sua tabella dei simboli privata, che è usata come tabella dei simboli globale da tutte le funzioni definite nel modulo. Quindi l'autore di un modulo può utilizzare le variabili globali nel modulo senza preoccuparsi di conflitti accidentali con le variabili globali di un utente. D'altro canto, se si sa quel che si sta facendo si può accedere alle variabili globali di un modulo mediante la stessa notazione usata per riferirsi alle sue funzioni, `nome_modulo.nome_elemento`.

I moduli possono importare altri moduli. È uso comune, ma non indispensabile, mettere tutte le istruzioni `import` all'inizio del modulo (o dello script, in questo ambito). I nomi del modulo importato vengono messi nella tabella dei simboli globali del modulo che lo importa.

C'è una variante dell'istruzione `import` che importa nomi da un modulo direttamente nella tabella dei simboli del modulo che li importa. Per esempio:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

In questo modo non si introduce il nome del modulo dal quale vengono importati i nomi nella tabella dei simboli locali (così nell'esempio sopra `fibo` non è definito).

C'è anche una variante per importare tutti i nomi definiti in un modulo:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

In questo modo si importano tutti i nomi tranne quelli che iniziano con un `underscore' (_).

6.1.1 Il Percorso di Ricerca del Modulo

Quando un modulo di nome `spam` viene importato, l'interprete cerca un file chiamato spam.py nella directory corrente, e quindi nella lista di directory specificata dalla variabile d'ambiente $PYTHONPATH. Tale variabile ha la stessa sintassi della variabile di shell $PATH, cioè una lista di nomi di directory. Quando $PYTHONPATH non è configurata, o quando il file non si trova nelle directory ivi menzionate, la ricerca continua su un percorso di default dipendente dall'installazione; su Unix di solito si tratta di .:/usr/local/lib/python.

In effetti i moduli vengono cercati nella lista delle directory contenuta nella variabile `sys.path`, che è inizializzata con la directory che contiene lo script in input (o la directory corrente), la $PYTHONPATH e il valore di default dipendente dall'installazione. Questo permette ai programmi Python scritti con cognizione di causa di modificare o rimpiazzare il percorso di ricerca dei moduli. Si veda la sezione sui Moduli Standard più avanti.

8.1.2 File Python `compilati'

Un'accelerazione rilevante dei tempi di avvio di brevi programmi che usano un sacco di moduli standard si ottiene se nella directory dove si trova spam.py esiste un file spam.pyc, ove si assume che questo contenga una versione già compilata a livello di bytecode del modulo `spam`. L'orario di modifica della versione di spam.py usata per creare spam.pyc è registrata in spam.pyc, e il file .pyc è ignorato se gli orari non corrispondono.

Normalmente, non c'è bisogno di fare nulla per creare il file spam.pyc. Ogni volta che spam.py è compilato con successo, viene fatto un tentativo di scrivere su spam.pyc la versione compilata. Il fallimento di tale tentativo non comporta un errore; se per qualsiasi ragione non viene scritto completamente, il file spam.pyc risultante verrà riconosciuto come non valido e perciò successivamente ignorato. I contenuti di spam.pyc sono indipendenti dalla piattaforma, quindi una directory di moduli Python può essere condivisa da macchine con architetture differenti.

Alcuni consigli per esperti:

- Quando l'interprete Python viene invocato con l'opzione **-O**, viene generato un codice ottimizzato che viene memorizzato in file .pyo. L'ottimizzatore attualmente non è di grande aiuto. Rimuove solamente le istruzioni `assert` e le istruzioni bytecode `SET_LINENO`. Quando viene usato **-O**, *tutto* il bytecode viene ottimizzato; i file `.pyc` vengono ignorati e i file `.py` vengono compilati in bytecode ottimizzato.
- Passando un doppio flag **-O** all'interprete Python (**-OO**), il compilatore bytecode eseguirà delle ottimizzazioni che potrebbero causare in alcuni rari casi il malfunzionamento dei programmi. Attualmente solo le stringhe `__doc__` vengono rimosse dal bytecode, ottenendo così file .pyo più compatti. Dato che alcuni programmi possono fare assegnamento sulla loro disponibilità, si dovrebbe usare questa opzione solo se si sa cosa si sta facendo.
- Un programma non viene eseguito più velocemente quando viene letto da un file .pyc o .pyo di quanto succeda con un file .py; l'unica cosa più rapida nei file .pyc o .pyo è il caricamento.
- Quando uno script viene eseguito da riga di comando, il bytecode ricavato dallo script non viene mai scritto su un .pyc o .pyo. Così il tempo di avvio di uno script può essere ridotto spostando la maggior parte del suo codice in un modulo ed facendo sì da avere un piccolo script di avvio che importi tale modulo. È anche possibile nominare un file .pyc o .pyo direttamente dalla riga di comando.
- È possibile avere un file chiamato spam.pyc (o spam.pyo} quando viene usato **-O**) senza uno spam.py nello stesso modulo. In questo modo si può distribuire una libreria di codice Python in una forma da cui è leggermente difficile risalire al listato originario.
- Il modulo `compileall` può creare i file .pyc (o .pyo quando viene usato **-O**) per tutti i moduli presenti in una directory.

6.2 Moduli Standard

Python viene fornito con una libreria di moduli standard, descritta in un documento separato, il *Python Library Reference* (da qui in avanti verrà indicato come `Library Reference'). Alcuni moduli sono interni all'interprete (`built-in'). Essi forniscono supporto a operazioni che non fanno parte del nucleo del linguaggio ma cionondimeno sono interne, per garantire efficienza o per fornire accesso alle primitive del sistema operativo, come chiamate di sistema. L'insieme di tali moduli è un'opzione di configurazione; p.e., il modulo `amoeba` è fornito solo su sistemi che in qualche modo supportano le primitive Amoeba. Un modulo particolare merita un po' di attenzione: `sys`, che è presente come modulo interno in ogni interprete Python. Le variabili `sys.ps1` e `sys.ps2` definiscono le stringhe usate rispettivamente come prompt primario e secondario:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
```

```
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Queste due variabili sono definite solo se l'interprete è in modalità interattiva.

La variabile `sys.path` è una lista di stringhe che determinano il percorso di ricerca dei moduli dell'interprete. È inizializzata con un percorso di default ottenuto dalla variabile di ambiente $PYTHONPATH, o da un valore predefinito interno se $PYTHONPATH non è configurata. È possibile modificarla usando le operazioni standard delle liste, p.e.:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 La Funzione `dir()`

La funzione interna `dir()` viene usata per ottenere i nomi definiti da un modulo. Essa restituisce una lista ordinata di stringhe:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names', 'copyright', 'exit',
'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile', 'settrace',
'stderr', 'stdin', 'stdout', 'version']
```

Invocata senza argomenti, `dir()` elenca i nomi attualmente definiti:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Si noti che la lista comprende tutti i tipi di nomi: variabili, moduli, funzioni, ecc.

La funzione `dir()` non elenca i nomi delle funzioni e delle variabili interne. Se se ne desidera l'elenco, si possono trovare le definizioni nel modulo standard `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError', 'EOFError', 'IOError',
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'MemoryError', 'NameError', 'None', 'OverflowError', 'RuntimeError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
```

```
'ZeroDivisionError', '__name__', 'abs', 'apply', 'chr', 'cmp', 'coerce',
'compile', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',
'getattr', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'len', 'long',
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'round', 'setattr', 'str', 'type', 'xrange']
```

6.4 I Package

I `package' sono un modo di strutturare lo spazio di nomi di modulo di Python usando nomi di modulo separati da punti. Per esempio, il nome di modulo `A.B` designa un sottomodulo chiamato "`B`" in un package chiamato "`A`". Proprio come l'uso dei moduli permette agli autori di moduli differenti di non doversi preoccupare dei nomi delle rispettive variabili globali, usare i nomi di modulo separati da punti risparmia agli autori di package multi-modulari come NumPy o PIL ogni preoccupazione circa i nomi di modulo.

Si supponga di voler progettare una collezione di moduli (un ``package'') per il trattamento coerente di file e dati audio. Ci sono molti formati diversi per i file audio (di solito riconoscibili dalla loro estensione, p.e. .wav, .aiff, .au), quindi potrebbe essere necessario creare e mantenere una collezione crescente di moduli per la conversione tra i vari formati di file. Ci sono anche molte operazioni differenti che si potrebbe voler effettuare sui dati (p.e. mixing, aggiunta di echi, applicazione di una funzione di equalizzazione, creazione di un effetto stereo artificiale), quindi si scriveranno via via una successione senza fine di moduli che effettuano tali operazioni. Ecco una possibile struttura per il pacchetto (espressa secondo un filesystem gerarchico):

```
Sound/                          Package principale
    __init__.py                 Inizializza il package Sound
    Formats/                    Sottopackage per le conversioni tra formati
            __init__.py
            wavread.py
            wavwrite.py
            aiffread.py
            aiffwrite.py
            auread.py
            auwrite.py
            ...
    Effects/                    Sottopackage per gli effetti sonori
            __init__.py
            echo.py
```

```
        surround.py
        reverse.py
        ...
    Filters/            Sottopackage per i filtri
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

I file __init__.py sono necessari per far sì che Python tratti correttamente le directory come contenitori di package; ciò viene fatto per evitare che directory con un nome comune, come "`string`", nascondano involontariamente moduli validi che le seguano nel percorso di ricerca dei moduli. Nel caso più semplice, __init__.py può essere un file vuoto, ma può anche eseguire codice di inizializzazione per il package oppure configurare la variabile `__all__` descritta più avanti.

Gli utenti del package possono importare singoli moduli del package, per esempio:

```
import Sound.Effects.echo
```

effettua il caricamento del sottomodulo `Sound.Effects.echo`. Dev'essere referenziato con il suo nome completo, p.e.

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Un modo alternativo per importare il sottomodulo è:

```
from Sound.Effects import echo
```

anche questo carica il sottomodulo `echo`, e lo rende disponibile senza il prefisso del suo package, così da poter essere utilizzato nel seguente modo:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Un'altra variazione ancora è importare direttamente la funzione o la variabile desiderata

```
from Sound.Effects.echo import echofilter
```

Di nuovo, questo carica il sottomodulo `echo`, ma questa volta rendendo la sua funzione `echofilter()` direttamente disponibile:

```
echofilter(input, output, delay=0.7, atten=4)
```

Si noti che quando si usa `from` *package* `import` *item*, l'elemento può essere un sottomodulo (o sottopackage) del package o un altro nome definito nel package, come una funzione, una classe o una variabile. L'istruzione `import` per prima cosa si assicura che l'elemento sia definito nel package; se non lo è, assume che si tratti di un modulo e tenta di caricarlo. Se non lo trova, viene sollevata una `ImportError`.

Al contrario, quando si usa una sintassi del tipo `import` *elemento.sottoelemento.sottosottoelemento*, ogni elemento eccetto l'ultimo dev'essere un package; l'ultimo può essere un modulo o un package ma non può essere una classe, una funzione o una variabile definita nell'elemento precedente.

6.4.1 Importare con * da un Package

Ora, che succede quando l'utente scrive `from Sound.Effects import *`? Idealmente si spererebbe che in qualche modo questo si muova sul filesystem, trovi quali sottomoduli sono presenti nel package e li importi tutti. Sfortunatamente questa operazione non funziona molto bene sulle piattaforme Mac e Windows, dove il filesystem non sempre ha informazioni accurate sulle maiuscole e minuscole dei nomi di file! Su queste piattaforme non c'è un modo sicuro di sapere se un file ECHO.PY dovrebbe essere importato come modulo `echo`, `Echo` o `ECHO`. (A esempio, Windows 95 ha la noiosa abitudine di mostrare tutti i nomi di file con la prima lettera maiuscola). La restrizione dei nomi di file a 8+3 caratteristica del DOS aggiunge un altro problema interessante per i nomi di modulo lunghi.

L'unica soluzione è che spetti all'autore provvedere un indice esplicito del package. L'istruzione import adopera la seguente convenzione: se il codice del file __init__.py di un package definisce una lista di nome `__all__`, si assume si tratti della lista dei nomi di modulo che devono essere importati quando viene incontrato un `from` *package* `import *`. È compito dell'autore del package mantenere aggiornata tale lista quando viene rilasciata una nuova versione. Gli autori dei package possono anche decidere di non supportare tale funzionalità se non vedono l'utilità di importare con * dal loro package. Per esempio, il file Sounds/Effects/__init__.py potrebbe contenere il codice seguente:

```
__all__ = ["echo", "surround", "reverse"]
```

Ciò significa che `from Sound.Effects import *` importerebbe i tre sottomoduli sopracitati del package `Sound`.

Se `__all__` non è definito, l'istruzione `from Sound.Effects import *` *non* importa tutti i sottomoduli del package `Sound.Effects` nello spazio di nomi corrente; si assicura solamente che il package `Sound.Effects` sia stato importato (possibilmente eseguendo il suo codice di inizializzazione, __init__.py) e quindi importa qualunque nome sia definito nel package. Ciò include qualsiasi nome definito (e i sottomoduli esplicitamente caricati) in __init__.py. Comprende inoltre tutti i sottomoduli del package che siano stati esplicitamente caricati da precedenti istruzioni `import`, p.e.

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

In questo esempio, i moduli echo e surround sono importati nello spazio di nomi corrente poiché essi sono definiti nel package `Sound.Effects` al momento dell'esecuzione dell'istruzione `from...import`. (Funziona così anche quando è definito `__all__`).

Si noti che in generale l'uso generalizzato di importare con * da un modulo è malvisto, dato che spesso comporta poca leggibilità del codice. Comunque è accettabile usarlo per risparmiare un po' di lavoro sulla tastiera nelle sessioni interattive, e certi moduli sono pensati per esportare solo nomi che seguono certi schemi.

Si tenga presente che non c'è nulla di sbagliato nell'uso di `from Package import Uno_specifico_sottomodulo` ! Infatti questa è la notazione consigliata a meno che il modulo importatore debba usare sottomoduli con lo stesso nome da package differenti.

6.4.2 Riferimenti interni a un package

I sottomoduli spesso devono fare riferimento l'uno all'altro. Per esempio, il modulo `surround` potrebbe dover usare il modulo `echo`. In effetti tali riferimenti sono tanto comuni che l'istruzione `import` per prima cosa considera il package contenitore del modulo prima di effettuare una ricerca nel percorso standard di ricerca dei moduli. Perciò `surround` può semplicemente usare `import echo` o `from echo import echofilter`. Se il modulo importato non viene trovato nel package corrente (il package di cui il modulo corrente è un sottomodulo), l'istruzione `import` ricerca un modulo con quel nome al livello più alto.

Quando i package sono strutturati in sottopackage (come nel package `Sound` di esempio), non ci sono scorciatoie per riferirsi a package imparentati - dev'essere usato il nome completo del sottopackage. Per esempio, se il modulo `Sound.Filters.vocoder` vuole servirsi del modulo `echo` nel package `Sound.Effects`, può usare `from Sound.Effects import echo`.

7. Input e Output

Ci sono parecchi modi per mostrare l'output di un programma; i dati possono essere stampati in una forma leggibile, o scritti in un file per usi futuri. Questo capitolo tratterà alcune delle possibilità.

7.1 Formattazione Avanzata dell'Output

Fino a qui si sono visti due modi di scrivere valori: le *espressioni* e l'istruzione `print`. (Un terzo modo è usare il metodo `write()` degli oggetti file; si può fare riferimento al file di standard output come `sys.stdout`. Si veda la Library Reference per ulteriori informazioni).

Spesso si vorrà avere un controllo sulla formattazione dell'output che vada aldilà dello stampare semplicemente dei valori separati da spazi. Ci sono due modi per formattare l'output; il primo è fare da sè tutto il lavoro di gestione delle stringhe; usando le operazioni di affettamento e concatenamento si possono creare tutti i layout che si vogliono. Il modulo standard `string` contiene alcuni utili operatori di riempimento (`padding') di stringhe ad una colonna di ampiezza data; questi verranno trattati brevemente. Il secondo modo è usare l'operatore `%` con una stringa come argomento a sinistra. `%` interpreta l'argomento di sinistra come una stringa di formato nello stile della funzione C `sprintf()` che dev'essere applicata all'argomento a destra, e restituisce la stringa risultante.

Beninteso, rimane un problema: come si fa a convertire valori in stringhe? Fortunatamente, Python ha un modo per convertire un qualsiasi valore in una stringa: lo si passa alla funzione `repr()`. Altrimenti basta scrivere il valore tra apici inversi (``). Alcuni esempi:

```
>>> x = 10 * 3.14
>>> y = 200*200
>>> s = 'Il valore di x è ' + `x` + ', e y è ' + `y` + '...'
>>> print s
Il valore di x è 31.4, e y è 40000...
>>> # Gli apici inversi funzionano anche su tipi non numerici:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[31.4, 40000]'
>>> # Convertire una stringa aggiunge gli apici di stringa e le barre rovesce:
... hello = 'hello, world\n'
>>> hellos = `hello`
>>> print hellos
'hello, world\012'
>>> # L'argomento racchiuso tra apici inversi può essere una tupla:
... `x, y, ('spam', 'eggs')`
"(31.4, 40000, ('spam', 'eggs'))"
```

Ecco due modi di scrivere una tabella di quadrati e cubi:

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(`x`, 2), string.rjust(`x*x`, 3),
...     # Si noti la virgola in coda sulla riga precedente
...     print string.rjust(`x*x*x`, 4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Si noti che uno spazio fra le colonne è stato aggiunto per il modo in cui lavora `print`: viene sempre inserito uno spazio tra i suoi argomenti).

Questa è una dimostrazione del funzionamento di `string.rjust()`, che giustifica a destra una stringa in un campo di ampiezza data riempiendola di spazi a sinistra. Ovviamente ci sono le analoghe funzioni `string.ljust()` e `string.center()`. Tali funzioni non sovrascrivono nulla, piuttosto restituiscono una nuova stringa. Se la stringa di input è troppo lunga non la troncano ma la restituiscono intatta; questo potrebbe scombussolare l'allineamento delle colonne desiderato, ma questo è di solito preferibile all'alternativa, cioè riportare un valore menzognero. (Se davvero si desidera il troncamento si può sempre aggiungere un'operazione di affettamento, come in "`string.ljust(x, n)[0:n]`").

C'è un'altra funzione, `string.zfill()`, che aggiunge ad una stringa numerica degli zero a sinistra. Tiene conto dei segni più e meno:

```
>>> import string
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'
```

L'uso dell'operatore `%` è il seguente:

```
>>> import math
>>> print 'Il valore di PI è approssimativamente %5.3f.' % math.pi
Il valore di PI è approssimativamente 3.142.
```

Se c'è piu` di un codice di formato nella stringa, si passi una tupla come operando di destra, a esempio:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

La maggior parte dei codici di formato funziona esattamente come in C e richiede che venga passato il tipo appropriato; comunque nel caso non lo sia si ottiene un'eccezione, non un core dump. Il codice di formato `%s` ha un comportamento più rilassato: se l'argomento corrispondente non è un oggetto stringa, viene convertito in stringa tramite la funzione interna `str()`. L'uso di `*` per passare l'ampiezza o precisione come argomento separato (numero intero) è supportato, mentre i codici di formato C `%n` e `%p` non sono supportati.

Nel caso si abbia una stringa di formato davvero lunga che non si vuole suddividere in parti, sarebbe carino poter fare riferimento alle variabili da formattare tramite il nome piuttosto che tramite la posizione. Ciò può essere ottenuto usando un'estensione dei codici di formato C nella forma `%(nome)formato`, p.e.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

È particolarmente utile in combinazione con la nuova funzione interna `vars()`, che restituisce un dizionario contenente tutte le variabili locali.

7.2 Leggere e Scrivere File

`open()` restituisce un oggetto file, ed è perlopiù usata con due argomenti: "`open(`*nomefile*`,` *modo*`)`".

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Il primo argomento è una stringa contenente il nome del file. Il secondo è un'altra stringa contenente pochi caratteri che descrivono il modo nel quale verrà usato il file. *modo* sarà `'r'` (`read') quando il file verrà unicamente letto, `'w'` per la sola scrittura (`write'), in caso esista già un file con lo stesso nome, esso verrà cancellato, mentre `'a'` aprirà il file in "aggiunta" (`append'): qualsiasi dato scritto sul file verrà automaticamente aggiunto alla fine dello stesso. `'r+'` aprirà il file sia in lettura che in scrittura. L'argomento *modo* è opzionale; in caso di omissione verrà assunto essere `'r'`.

Su Windows e Macintosh, `'b'` aggiunto al modo apre il file in modo binario, per cui ci sono ulteriori modi come `'rb'`, `'wb'`, e `'r+b'`. Windows distingue tra file di testo e binari; i caratteri EOF dei file di testo vengono leggermente alterati in automatico quando i dati vengono letti o scritti. Questa modifica che avviene di nascosto ai dati dei file è adatta ai file di testo ASCII, ma corromperà i dati binari presenti a esempio in file JPEG o .EXE. Si raccomanda cautela nell'uso del modo binario quando si sta leggendo o scrivendo su questi tipi di file. (Si noti che l'esatta semantica del modo testo su Macintosh dipende dalla libreria C usata).

7.2.1 Metodi degli Oggetti File

Nel resto degli esempi di questa sezione si assumerà che sia già stato creato un oggetto file chiamato `f`.

Per leggere i contenuti di un file, s'invochi `f.read(`*lunghezza*`)`, che legge una certa quantità di dati e li restituisce come stringa. *lunghezza* è un argomento numerico opzionale. Se omesso o negativo, verrà letto e restituito l'intero contenuto del file. Se il file è troppo grosso rispetto alla memoria della macchina il problema è tutto vostro. Altrimenti viene

letto e restituito al più un numero di byte pari a *lunghezza*. Se è stata raggiunta la fine del file, `f.read()` restituirà una stringa vuota (`""`).

```
>>> f.read()
'Questo è l'intero file.\012'
>>> f.read()
''
```

`f.readline()` legge una singola riga dal file; un carattere di newline (`\n`) viene lasciato alla fine della stringa, ed è omesso solo nell'ultima riga del file in caso esso non finisca con un newline. Ciò rende il valore restituito non ambiguo: se `f.readline()` restituisce una stringa vuota, è stata raggiunta la fine del file, mentre una riga vuota è rappresentata da `'\n'`, stringa che contiene solo un singolo carattere di newline.

```
>>> f.readline()
'Questa è la prima riga del file.\012'
>>> f.readline()
'Seconda riga del file\012'
>>> f.readline()
''
```

`f.readlines()` restituisce una lista contenente tutte le righe di dati presenti nel file. Se le viene passato un parametro opzionale *lunghezza_suggerita*, legge tale numero di byte dal file, poi continua fino alla fine della riga e restituisce le righe. Viene spesso usata per consentire la lettura efficiente per righe di un file, senza dover caricare l'intero file in memoria. Verranno restituite solo righe complete.

```
>>> f.readlines()
['Questa è la prima riga del file.\012', 'Seconda riga del file\012']
```

`f.write(`*stringa*`)` scrive il contenuto di *stringa* nel file, restituendo `None`.

```
>>> f.write('Questo è un test\n')
```

`f.tell()` restituisce un intero che fornisce la posizione nel file dell'oggetto file, misurata in byte dall'inizio del file. Per variare la posizione dell'oggetto file si usi "`f.seek(`*offset*`, `*da_cosa*`)`". La posizione è calcolata aggiungendo *offset* a un punto di riferimento, selezionato tramite l'argomento *da_cosa*. Un valore di *da_cosa* pari a 0 effettua la misura dall'inizio del file, 1 utilizza come punto di riferimento la posizione attuale, 2 usa la fine del file. *da_cosa* può essere omesso e per default è pari a 0, viene quindi usato come punto di riferimento l'inizio del file.

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Va al quinto byte nel file
>>> f.read(1)
```

```
'5'
>>> f.seek(-3, 2) # Va al terzo byte prima della fine del file
>>> f.read(1)
'd'
```

Quando si è terminato di lavorare su un file, si chiami `f.close()` per chiuderlo e liberare tutte le risorse di sistema occupate dal file aperto. Dopo aver invocato `f.close()`, i tentativi di usare `f` falliranno automaticamente.

```
>>> f.close()
>>> f.read()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Gli oggetti file hanno alcuni metodi addizionali, come `isatty()` e `truncate()` che sono usati meno di frequente; si consulti la Library Reference per una guida completa agli oggetti file.

7.2.2 Il Modulo `pickle`

Le stringhe possono essere scritte e lette da un file con facilità. I numeri richiedono uno sforzo un po' maggiore, in quanto il metodo `read()` restituisce solo stringhe, che dovranno essere passate a una funzione tipo `string.atoi()`, che prende una stringa come `'123'` e restituisce il corrispondente valore numerico 123. Comunque quando si desidera salvare tipi di dato più complessi, quali liste, dizionari o istanze di classe, le cose si fanno assai più complicate.

Per non costringere gli utenti a scrivere e correggere in continuazione codice per salvare tipi di dato complessi, Python fornisce un modulo standard chiamato `pickle`. Si tratta di un modulo meraviglioso che può prendere pressoché qualsiasi oggetto Python (persino alcune forme di codice Python!), e convertirlo in una rappresentazione stringa; tale processo è chiamato *pickling* [letteralmente "conservazione sotto aceto", in pratica si tratta di serializzazione NdT]. La ricostruzione dell'oggetto a partire dalla rappresentazione stringa è chiamata *unpickling*. Tra il pickling e l'unpickling, la stringa che rappresenta l'oggetto può essere immagazzinata in un file, o come dato, o inviata a una macchina remota tramite una connessione di rete.

Se si ha un oggetto `x`, e un oggetto file `f` aperto in scrittura, il modo più semplice di fare il pickling dell'oggetto occupa solo una riga di codice:

```
pickle.dump(x, f)
```

Per fare l'unpickling dell'oggetto, se `f` è un oggetto file aperto in scrittura:

```
x = pickle.load(f)
```

(Ci sono altre varianti del procedimento, usate quando si fa il pickling di molti oggetti o quando non si vuole scrivere i dati ottenuti in un file; si consulti la documentazione completa di `pickle` nel Library Reference).

`pickle` è il modo standard per creare oggetti Python che possono essere immagazzinati e riusati da altri programmi o da future esecuzioni dello stesso programma; il termine tecnico è oggetto *persistente*. Poiché `pickle` è così ampiamente usato, molti autori di estensioni Python stanno attenti a garantire che i nuovi tipi di dati quali le matrici possano essere sottoposti senza problemi a pickling e unpickling.

8. Errori ed Eccezioni

Fino ad ora i messaggi di errore sono stati solo nominati, ma se avete provato a eseguire gli esempi ne avrete visto probabilmente qualcuno. Si possono distinguere (come minimo) due tipi di errori: gli *errori di sintassi* e le *eccezioni*.

8.1 Errori di Sintassi

Gli errori di sintassi, noti anche come errori di parsing, sono forse il tipo più comune di messaggio di errore che si riceve mentre si sta ancora imparando Python:

```
>>> while 1 print 'Ciao mondo'
  File "<stdin>", line 1
    while 1 print 'Ciao mondo'
                ^
SyntaxError: invalid syntax
```

L'analizzatore sintattico ("parser") riporta la linea incriminata e mostra una piccola `freccia' che punta al primissimo punto in cui è stato rilevato l'errore nella riga incriminata . L'errore è causato dal token che *precede* la freccia (o quantomeno rilevato presso di esso); nell'esempio l'errore è rilevato alla parola chiave `print`, dato che mancano i duepunti (":") prima di essa. Vengono stampati il nome del file e il numero di linea, in modo che si sappia dove andare a guardare in caso l'input provenga da uno script.

8.2 Le Eccezioni

Anche se un'istruzione, o un'espressione, è sintatticamenta corretta, può causare un errore quando si tenta di eseguirla. Gli errori rilevati durante l'esecuzione sono chiamati *eccezioni* e non sono incondizionatamente fatali: si imparerà presto come gestirli nei programmi Python. La maggior parte delle eccezioni comunque non sono gestite dai programmi e causano dei messaggi di errore, come i seguenti:

```
>>> 10 * (1/0)
Traceback (innermost last):
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (innermost last):
  File "<stdin>", line 1
NameError: spam
>>> '2' + 2
Traceback (innermost last):
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
```

L'ultima riga del messaggio di errore indica cos'è successo. Le eccezioni sono di diversi tipi, ed il loro tipo è stampato come parte del messaggio: i tipi che compaiono nell'esempio sono `ZeroDivisionError`, `NameError` e `TypeError`. La stringa stampata quale tipo dell'eccezione è il nome dell'eccezione occorsa. Ciò è vero per tutte le eccezioni built-in, ma non è necessario che lo sia per le eccezioni definite dall'utente (malgrado si tratti di una convenzione utile). I nomi delle eccezioni standard sono identificatori built-in, non parole chiave riservate.

Il resto della riga è un dettaglio la cui interpretazione dipende dal tipo dell'eccezione; il suo significato dipende dal tipo dell'eccezione.

La parte antecedente del messaggio di errore mostra il contesto in cui è avvenuta l'eccezione, nella forma di una traccia dello stack ("stack backtrace"). In generale contiene una traccia dello stack che riporta linee di codice sorgente; in ogni caso non mostrerà righe lette dallo standard input.

La *Library Reference* elenca le eccezioni built-in e i loro significati.

8.3 Gestire le Eccezioni

È possibile scrivere programmi che gestiscono determinate eccezioni. Si esamini il seguente esempio, che richiede un input fino a quando non viene introdotto un intero valido, ma permette all'utente di interrompere il programma (usando Control-C o qualunque cosa sia equivalente per il sistema operativo); si noti che una interruzione generata dall'utente viene segnalata sollevando un'eccezione `KeyboardInterrupt`.

```
>>> while 1:
...     try:
...         x = int(raw_input("Introduci un numero: "))
...         break
...     except ValueError:
...         print "Occhio! Non era un numero valido. Ritenta..."
...
```

L'istruzione `try` funziona nel modo seguente.

- Per prima cosa viene eseguita la *clausola try* (la/le istruzione/i tra le parole chiave `try` e `except`).
- Se non sopravviene alcuna eccezione, la *clausola except* viene saltata e l'esecuzione dell'istruzione `try` è terminata.
- Se durante l'esecuzione della clausola try ricorre un'eccezione, il resto della clausola viene saltato. Indi se il suo tipo collima con l'eccezione citata dopo la parola chiave `except`, il resto della clausola try viene saltato, viene eseguita la clausola except e infine l'esecuzione continua dopo l'istruzione `try`.
- Se ricorre un'eccezione che non corrisponde a quella citata nella clausola except, essa viene trasmessa a eventuali istruzioni `try` di livello superiore; se non viene trovata una clausola che le gestisca, si tratta di *eccezione non gestita* ["imprevista"] e l'esecuzione si ferma con un messaggio, come mostrato sopra.

Un'istruzione `try` può avere più di una clausola except, per specificare gestori di differenti eccezioni. Al più verrà eseguito un solo gestore. I gestori si occupano solo delle eccezioni che avvengono nella clausola try corrispondente, non in altri gestori della stessa istruzione `try`. Una clausola except può nominare più di un'eccezione in forma di lista tra parentesi, p.e.:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Nell'ultima clausola except si può tralasciare il nome (o i nomi) dell'eccezione, affinché serva da jolly. Si presti estrema cautela, dato che in questo modo è facile mascherare un errore di programmazione vero e proprio! Può anche servire per stampare un messaggio

di errore e quindi risollevare l'eccezione (permettendo pure a un chiamante di gestire l'eccezione):

```
import string, sys

try:
    f = open('miofile.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "Errore I/O(%s): %s" % (errno, strerror)
except ValueError:
    print "Non si può convertire il dato in un intero."
except:
    print "Errore inatteso:", sys.exc_info()[0]
    raise
```

L'istruzione `try` ... `except` ha una *clausola else* opzionale, la quale, ove presente, deve seguire tutte le clausole except. È utile per posizionarvi del codice che debba essere eseguito in caso la clausola try non sollevi un'eccezione. Per esempio:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'non posso aprire', arg
    else:
        print arg, 'ha', len(f.readlines()), 'righe'
        f.close()
```

L'uso della clausola `else` è preferibile all'aggiunta di codice supplementare alla `try` poiché evita la cattura accidentale di un'eccezione che non è emersa dal codice protetto dall'istruzione `try` ... `except`.

Quando sopravviene un'eccezione, essa può avere un valore associato, conosciuto anche come *argomento* dell'eccezione. La presenza e il tipo dell'argomento dipendono dal tipo di eccezione. Per tipi di eccezione che hanno un argomento, la clausola except può specificare, dopo il nome dell'eccezione (o la lista di nomi), una variabile che riceva il valore dell'argomento, come segue:

```
>>> try:
...     spam()
... except NameError, x:
...     print 'nome', x, 'non definito'
```

```
...
nome spam non definito
```

Se un'eccezione ha un argomento, questo viene stampato come ultima parte (`dettaglio') del messaggio per le eccezioni non gestite.

I gestori delle eccezioni non si occupano solo delle eccezioni che avvengono direttamente nella clausola try, ma anche di quelle che ricorrono dall'interno di funzioni chiamate (anche indirettamente) nella clausola try. Per esempio:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detail:
...     print 'Gestione dell'errore a runtime:', detail
...
Gestione dell'errore a runtime: integer division or modulo
```

8.4 Sollevare Eccezioni

L'istruzione `raise` permette al programmatore di forzare una specifica eccezione. Per esempio:

```
>>> raise NameError, 'HiThere'
Traceback (innermost last):
  File "<stdin>", line 1
NameError: HiThere
```

Il primo argomento di `raise` menziona l'eccezione da sollevare. Il secondo argomento, opzionale, specifica l'argomento dell'eccezione..

8.5 Eccezioni Definite dall'Utente

I programmi possono dare un nome a delle proprie eccezioni assegnando una stringa a una variabile. Per esempio:

```
>>> class MyError:
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return `self.value`
...
```

```
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'Occorsa una mia eccezione, valore:', e.value
...
Occorsa una mia eccezione, valore: 4
>>> raise MyError, 1
Traceback (innermost last):
  File "<stdin>", line 1
__main__.MyError: 1
```

Molti moduli standard usano questa tecnica per riportare errori che possono accadere nelle funzioni che definiscono.

Altre informazioni sulle classi sono presentate nel capitolo 9, `Classi'.

8.6 Definire Azioni di Chiusura

L'istruzione `try` ha un'altra clausola opzionale, che serve a definire azioni di chiusura (`clean-up') che devono essere eseguite in tutti i casi. Per esempio:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Ciao, mondo!'
...
Ciao, mondo!
Traceback (innermost last):
  File "<stdin>", line 2
KeyboardInterrupt
```

Una *clausola finally* viene eseguita comunque, che l'eccezione sia occorsa nella clausola try o meno. Nel caso sia sorta un'eccezione, essa viene ri-sollevata dopo che la clausola finally è stata eseguita. La clausola finally viene eseguita anche quando si esce da un'istruzione `try` a mezzo di un `break` o un `return`.

Un'istruzione `try` deve avere una o più clausole except o una clausola **f**inally, ma non entrambe.

9. Classi

Il meccanismo delle classi di Python è in grado di aggiungere classi al linguaggio con un minimo di nuova sintassi e semantica. È un misto dei meccanismi delle classi che si trovano in C++ e Modula-3. Come pure per i moduli, in Python le classi non pongono una barriera invalicabile tra la definizione e l'utente, piuttosto fanno affidamento su una politica-utente di "rispetto della definizione". Le funzionalità più importanti delle classi sono comunque mantenute in tutta la loro potenza: il meccanismo di ereditarietà permette classi base multiple, una classe derivata può sovrascrivere qualsiasi metodo delle sue classi base, un metodo può chiamare il metodo di una classe base con lo stesso nome. Gli oggetti possono contenere una quantità arbitraria di dati privati.

Secondo la terminologia C++, tutti i membri delle classi (inclusi i dati membri) sono *pubblici*, e tutte le funzioni membro sono *virtuali*. Non ci sono speciali costruttori o distruttori. Come in Modula-3, non ci sono scorciatoie per referenziare i membri dell'oggetto dai suoi metodi: la funzione del metodo è dichiarata con un primo argomento esplicito che rappresenta l'oggetto, che è fornito in modo implicito dalla chiamata. Come in Smalltalk, le classi in se stesse sono oggetti, quantunque nel senso più ampio del termine: in Python, tutti i tipi di dati sono oggetti. Ciò fornisce una semantica per l'importazione e la ridenominazione. Ma, proprio come in C++ o Modula-3, i tipi built-in non possono essere usati come classi base per estensioni utente. Inoltre, come in C++ ma diversamente da quanto accade in Modula-3, la maggior parte degli operatori built-in con una sintassi speciale (operatori aritmetici, sottoselezioni ecc.) possono essere ridefiniti per istanze di classe.

9.1 Qualche Parola sulla Terminologia

Data la mancanza di una terminologia universalmente accettata quando si parla di classi, farò occasionalmente uso di termini Smalltalk e C++. (Vorrei usare termini Modula-3, dato che la sua semantica orientata agli oggetti è più vicina a quella di Python di quanto sia quella del C++, ma mi aspetto che pochi dei miei lettori ne abbiano sentito parlare [in effetti io ero rimasto al 2 :-) NdT]).

Devo anche avvisare che per i lettori con conoscenze di programmazione orientata agli oggetti c'è una inghippo terminologico: il termine `oggetto' in Python non significa necessariamente un'istanza di classe. Come in C++ e in Modula-3, e diversamente da

Smalltalk, in Python non tutti i tipi sono classi: i tipi built-in di base come gli interi e le liste non lo sono, e non lo sono neanche alcuni tipi un po' più esotici come i file. Comunque *tutti* i tipi Python condividono un pochetto di semantica comune che trova la sua miglior descrizione nell'uso della parola oggetto.

Gli oggetti sono dotati di individualità, e nomi multipli (in ambiti di visibilità multipli) possono essere associati allo stesso oggetto. In altri linguaggi ciò è noto come `aliasing'. Questo non viene di solito apprezzato in una prima occhiata al linguaggio, e può essere ignorato senza problemi quando si ha a che fare con tipi di base immutabili (numeri, stringhe, tuple). Comunque, l'aliasing ha un effetto (voluto!) sulla semantica del codice Python che riguarda oggetti mutabili come liste, dizionari e la maggior parte dei tipi che rappresentano entità esterne al programma (file, finestre ecc.). Questo è usato a beneficio del programma, dato che gli alias si comportano per certi versi come puntatori. Per esempio passare un oggetto è economico, dato che per implementazione viene passato solo un puntatore. E se una funzione modifica un oggetto passato come argomento, le modifiche saranno visibili al chiamante --- questo ovvia al bisogno di due meccanismi diversi per passare gli argomenti come in Pascal.

9.2 Gli `Scope' di Python e gli Spazi di Nomi

Prima di introdurre le classi, è necessario dire qualcosa circa le regole sugli ambiti di visibilità (`scope') in Python. Le definizioni di classe fanno un paio di graziosi giochetti con gli spazi di nomi, ed è necessario conoscere come funzionano gli scope e gli spazi di nomi per comprendere bene quello che succede. Detto per inciso, la conoscenza di questo argomento è utile ad ogni programmatore Python avanzato.

Iniziamo con alcune definizioni.

Uno *spazio di nomi* è una mappa che collega i nomi agli oggetti. La maggior parte degli spazi di nomi sono attualmente implementati come dizionari Python, ma nell'uso normale ciò non è avvertibile in alcun modo (eccetto che per la velocità di esecuzione), e potrebbe cambiare in futuro. Esempi di spazi di nomi sono: l'insieme dei nomi built-in (funzioni come `abs()`}, e nomi delle eccezioni built-in), i nomi globali in un modulo e i nomi locali in una chiamata di funzione. In un certo senso l'insieme degli attributi di un oggetto costituisce anch'esso uno spazio di nomi. La cosa davvero importante da sapere al riguardo è che non c'è assolutamente nessuna relazione tra nomi uguali in spazi di nomi diversi; a esempio due moduli diversi potrebbero entrambi definire una funzione `massimizza' senza

possibilità di fare confusione --- gli utenti dei moduli dovranno premettere ad essa il nome del modulo.

A proposito, io utilizzo la parola *attributo* per qualsiasi nome che segua un punto --- per esempio, nell'espressione `z.real`, `real` è un attributo dell'oggetto `z`. A rigor di termini, i riferimenti a nomi nei moduli sono riferimenti a attributi: nell'espressione `nomemodulo.nomefunzione`, `nomemodulo` è un oggetto modulo e `nomefunzione` è un suo attributo. In questo caso capita che si tratti di una mappa diretta tra gli attributi del modulo e i nomi globali definiti nel modulo: essi condividono lo stesso spazio di nomi! [9.1]

Gli attributi possono essere a sola lettura o scrivibili. Nel secondo caso, è possibile assegnare un valore all'attributo. Gli attributi dei moduli sono scrivibili: si può digitare "`nomemodulo.la_risposta = 42`".

Gli attributi scrivibili possono anche essere cancellati con l'istruzione `del`, p.e. "`del nomemodulo.la_risposta`".

Gli spazi di nomi sono creati in momenti diversi e hanno tempi di sopravvivenza diversi . Lo spazio di nomi che contiene i nomi built-in è creato all'avvio dell'interprete Python e non viene mai cancellato. Lo spazio di nomi globale di un modulo è creato quando viene letta la definizione del modulo; normalmente anche lo spazio di nomi del modulo dura fino al termine della sessione. Le istruzioni eseguite dall'invocazione a livello più alto dell'interprete, lette da un file di script o interattivamente, sono considerate parte di un modulo chiamato `__main__`, cosicché esse hanno il proprio spazio di nomi globale. (In effetti anche i nomi built-in esistono in un modulo, chiamato `__builtin__`).

Lo spazio di nomi locale di una funzione è creato quando la funzione viene invocata, e cancellato quando la funzione ritorna o solleva un'eccezione che non è gestita al suo l'interno. (In effetti, `oblio' sarebbe un modo migliore di descrivere che cosa accade in realtà). Naturalmente invocazioni ricorsive hanno ciascuna il proprio spazio di nomi locale.

Uno *scope* è una regione del codice di un programma Python dove uno spazio di nomi è accessibile direttamente. `Direttamente accessibile' qui significa che un riferimento non completamente qualificato a un nome cerca di trovare tale nome nello spazio di nomi.

Sebbene gli scope siano determinati staticamente, essi sono usati dinamicamente. In qualunque momento durante l'esecuzione sono in uso esattamente tre scope annidati (cioè, esattamente tre spazi di nomi sono direttamente accessibili): lo scope più interno, in cui viene effettuata per prima la ricerca, contiene i nomi locali, lo scope mediano, esaminato successivamente, contiene i nomi globali del modulo corrente, e lo scope più esterno (esaminato per ultimo) è lo spazio di nomi che contiene i nomi built-in.

Di solito lo scope locale referenzia i nomi locali della funzione corrente (corrente dal punto di vista del codice). All'esterno delle funzioni, lo scope locale referenzia lo stesso spazio di nomi come scope globale: lo spazio di nomi del modulo. La definizione di una classe colloca ancora un'altro spazio di nomi nello scope locale.
È importante capire che gli scope sono determinati letteralmente secondo il codice: lo scope globale di una funzione definita in un modulo è lo spazio di nomi di quel modulo, non importa da dove o con quale alias venga invocata la funzione. D'altro lato, l'effettiva ricerca dei nomi viene fatta dinamicamente in fase di esecuzione. Comunque la definizione del linguaggio si sta evolvendo verso la risoluzione statica dei nomi, al momento della `compilazione' , quindi non si faccia affidamento sulla risoluzione dinamica dei nomi! (Di fatto le variabili locali vengono già determinate staticamente).
Un cavillo particolare di Python è che gli assegnamenti prendono sempre in esame lo scope più interno. Gli assegnamenti non copiano dati, associano solo dei nomi ad oggetti. Lo stesso vale per le cancellazioni: l'istruzione "`del x`" rimuove l'associazione di `x` dallo spazio di nomi referenziato dallo scope locale. In effetti tutte le operazioni che introducono nuovi nomi usano lo scope locale: in particolare, le istruzioni `import` e le definizioni di funzione associano il nome del modulo o della funzione nello scope locale. (L'istruzione `global` può essere usata per indicare che particolari variabili vivono nello scope globale).

9.3 Una Prima Occhiata alle Classi

Le classi introducono un po' di nuova sintassi, tre nuovi tipi di oggetti e nuova semantica.

9.3.1 La Sintassi della Definizione di Classe

La forma più semplice di definizione di una classe è del tipo:

```
class NomeClasse:
    <istruzione-1>
    .
    .
    .
    <istruzione-N>
```

Le definizione di classe, come le definizioni di funzione (istruzioni `def`), devono essere eseguite prima di avere qualunque effetto. (È concepibile che una definizione di classe possa essere collocata in un ramo di un'istruzione `if`, o all'interno di una funzione).

In pratica, le istruzioni dentro una definizione di classe saranno di solito definizioni di funzione, ma è permesso, e talvolta utile, che vi si trovino altre istruzioni, ci ritornemo sopra più avanti. Le definizioni di funzione dentro una classe normalmente hanno una lista di argomenti di aspetto peculiare, dettata dalle convenzioni di chiamata dei metodi. Ancora una volta, questo verrà spiegato più avanti.

Quando viene introdotta una definizione di classe, viene creato un nuovo spazio di nomi, usato come scope locale. Perciò tutti gli assegnamenti a variabili locali finiscono in questo nuovo spazio di nomi. In particolare, le definizione di funzione ivi associano il nome della nuova funzione.

Quando una definizione di classe è terminata normalmente (passando per la sua chiusura), viene creato un *oggetto classe*. Esso è fondamentalmente un involucro (`wrapper') per i contenuti dello spazio di nomi creato dalla definizione della classe; impareremo di più sugli oggetti classe nella sezione seguente. Lo scope locale originale (quello in essere proprio prima che venisse introdotta la definizione di classe) è ripristinato, e l'oggetto classe è quivi associato al nome di classe dato nell'intestazione della definizione (`NomeClasse` nell'esempio).

9.3.2 Oggetti Classe

Gli oggetti classe supportano due tipi di operazioni: riferimenti ad attributo e istanziazione.

I *riferimenti ad attributo* usano la sintassi standard utilizzata per tutti i riferimenti ad attributi in Python: `oggetto.nome`. Nomi di attributi validi sono tutti i nomi che si trovavano nello spazio di nomi della classe al momento della creazione del oggetto classe. Così, se la definizione di classe fosse del tipo:

```
class MiaClasse:
    "Una semplice classe d'esempio"
    i = 12345
    def f(x):
        return 'ciao mondo'
```

allora `MiaClasse.i` e `MiaClasse.f` sarebbero riferimenti validi ad attributi, che restituirebbero rispettivamente un intero e un oggetto funzione. Sugli attributi di una classe è anche possibile effettuare assegnamenti, quindi è possibile cambiare il valore di `MiaClasse.i` con un assegnamento. Anche `__doc__` è un attributo valido, a sola lettura, che restituisce la stringa di documentazione della classe: `"Una semplice classe di esempio"`.

L'*istanziazione* di una classe usa la notazione delle funzioni. Si comporta come se l'oggetto classe sia una funzione senza parametri che ritorna una nuova istanza della classe. A esempio, (usando la classe definita sopra):

```
x = MiaClasse()
```

crea una nuova *istanza* della classe e assegna tale oggetto alla variabile locale `x`.

L'operazione di instanziazione (la `chiamata' di un oggetto classe) crea un oggetto vuoto. In molti casi si preferisce che vengano creati oggetti con uno stato iniziale noto. Perciò una classe può definire un metodo speciale chiamato `__init__()`, come in questo caso:

```
def __init__(self):
    self.data = []
```

Quando una classe definisce un metodo `__init__()`, la sua instanziazione invoca automaticamente `__init__()` per l'istanza di classe appena creata. Così nel nostro esempio una nuova istanza, inizializzata, può essere ottenuta con:

```
x = MiaClasse()
```

Naturalmente il metodo `__init__()` può avere argomenti, per garantire maggior flessibilità. In tal caso, gli argomenti forniti all'istanziazione della classe vengono passati a `__init__()`. Per esempio,

```
>>> class Complesso:
...     def __init__(self, partereale, partimag):
...         self.r = partereale
...         self.i = parteimag
...
>>> x = Complesso(3.0,-4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Oggetti Istanza

Ora, cosa possiamo fare con gli oggetti istanza? Le sole operazioni che essi conoscono sono i riferimenti ad attributo. Ci sono due tipi di nomi di attributo validi.

Chiameremo il primo *attributi dato*. Corrispondono alle `variabili istanza' in Smalltalk, e ai `dati membri' in C++. Gli attributi dato non devono essere dichiarati; come le variabili locali, essi vengono alla luce quando vengono assegnati per la prima volta. Per esempio, se `x` è l'istanza della `MiaClasse` precedentemente creata, il seguente pezzo di codice stamperà il valore `16`, senza lasciare traccia:

```
x.counter = 1
while x.counter < 10:
```

```
    x.counter = x.counter * 2
print x.counter
del x.counter
```

Il secondo tipo di riferimenti ad attributo conosciuti dagli oggetti istanza sono i *metodi*. Un metodo è una funzione che `appartiene a' un oggetto. (In Python, il termine metodo non è prerogativa delle istanze di classi: altri tipi di oggetto possono benissimo essere dotati di metodi; p.e. gli oggetti lista hanno metodi chiamati append, insert, remove, sort, e così via. Comunque più sotto useremo il termine metodo intendendo esclusivamente i metodi degli oggetti istanza di classe, a meno che diversamente specificato).

I nomi di metodo validi per un oggetto istanza dipendono dalla sua classe. Per definizione, tutti gli attributi di una classe che siano oggetti funzione (definiti dall'utente) definiscono metodi corrispondenti della sue istanze. Così nel nostro esempio `x.f` è un riferimento valido ad un metodo, dato che `MiaClasse.f` è una funzione, ma `x.i` non lo è, dato che `MiaClasse.i` non è una funzione. Però `x.f` non è la stessa cosa di `MiaClasse.f`: è un *oggetto metodo*, non un oggetto funzione.

9.3.4 Oggetti Metodo

Di solito un metodo viene invocato direttamente, p.e.:

```
x.f()
```

Nel nostro esempio, questo ritornerà la stringa `'ciao mondo'`. Comunque, non è necessario invocare un metodo in modo immediato: `x.f` è un oggetto metodo, e può essere messo da parte e chiamato in un secondo tempo. A esempio:

```
xf = x.f
while 1:
    print xf()
```

continuerà a stampare "`ciao mondo`" senza fine.

Che cosa succede esattamente quando viene invocato un metodo? Forse si è notato che `x.f()` è stato invocato nell'esempio sopra senza argomenti, anche se la definizione di funzione per `f` specificava un argomento. Che cosa è accaduto all'argomento? Di sicuro Python solleva un'eccezione quando una funzione che richiede un argomento viene invocata senza nessun argomento, anche se poi l'argomento non viene effettivamente utilizzato...

In realtà si potrebbe aver già indovinato la risposta: la particolarità dei metodi è che l'oggetto viene passato come primo argomento della funzione. Nel nostro esempio, la chiamata `x.f()` è esattamente equivalente a `MiaClasse.f(x)`. In generale, invocare un

metodo con una lista di n argumenti è equivalente a invocare la funzione corrispondente con una lista di argomenti creata inserendo l'oggetto cui appartiene il metodo come primo argomento.

Se non fosse ancora chiaro il funzionamento dei metodi, uno sguardo all'implementazione potrebbe forse rendere più chiara la faccenda. Quando viene referenziato un attributo di un'istanza che non è un attributo dato, viene ricercata la sua classe. Se il nome indica un attributo di classe valido che è un oggetto funzione, viene creato un oggetto metodo `impacchettando' insieme in un oggetto astratto (puntatori a) l'oggetto istanza e l'oggetto funzione appena trovato: questo è l'oggetto metodo. Quando l'oggetto metodo viene invocato con una lista di argomenti viene `spacchettato', viene costruita una nuova lista di argomenti dall'oggetto istanza e dalla lista di argomenti originale, e l'oggetto funzione viene invocato con questa nuova lista di argomenti.

9.4 Note Sparse

[Forse dovrebbero essere collocate con maggior cura...]

Gli attributi dato prevalgono sugli attributi metodo con lo stesso nome; per evitare accidentali conflitti di nomi, che potrebbero causare bug difficili da scovare in programmi molto grossi, è saggio usare una qualche convenzione che minimizzi le possibilità di conflitti, p.e. usare le maiuscole per l'iniziale dei nomi di metodi, far precedere i nomi di attributi dato da una piccola stringa particolare (probabilmente basterebbe un trattino basso, `underscore'), o usare verbi per i metodi e sostantivi per gli attributi dato.

Gli attributi dato possono essere referenziati da metodi come pure dagli utenti ordinari (`client') di un oggetto. In altre parole, le classi non sono utilizzabili per implementare tipi di dato puramente astratti. In effetti, in Python non c'è nulla che renda possibile imporre l'occultamento dei dati (`data hiding'), ci si basa unicamente sulle convenzioni. (D'altra parte, l'implementazione di Python, scritta in C, può nascondere completamente i dettagli dell'implementazione e il controllo degli accessi a un oggetto se necessario; questo può essere utilizzato da estensioni a Python scritte in C).

I client dovrebbero usare gli attributi dato con cura, potrebbero danneggiare degli invarianti conservati dai metodi sovrascrivendoli con i loro attributi dato. Si noti che i client possono aggiungere degli attributi dato propri a un oggetto istanza senza intaccare la validità dei metodi, fino quando vengano evitati conflitti di nomi. Ancora una volta, una convenzione sui nomi può evitare un sacco di mal di testa.

Non ci sono scorciatoie per referenziare attributi dato (o altri metodi!) dall'interno dei metodi. Trovo che questo in realtà aumenti la leggibilità dei metodi: non c'è possibilità di confondere le variabili locali e le variabili istanza quando si scorre un metodo.

Convenzionalmente, il primo argomento dei metodi è spesso chiamato `self`. Questa non è niente di più che una convenzione: il nome `self` non ha assolutamente alcun significato speciale in Python. (Si noti comunque che se non si segue tale convenzione il proprio codice può risultare meno leggibile ad altri programmatori Python, ed è anche concepibile venga scritto un programma *browser delle classi* che si basi su tale convenzione).

Qualsiasi oggetto funzione che sia attributo di una classe definisce un metodo per le istanze di tale classe. Non è necessario che il codice della definizione di funzione sia racchiuso nella definizione della classe: va bene anche assegnare un oggetto funzione a una variabile locale nella classe. Per esempio:

```
# Funzione definita all'esterno della classe
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'ciao mondo'
    h = g
```

Ora `f`, `g` e `h` sono tutti attributi della classe `C` che si riferiscono ad oggetti funzione, di conseguenza sono tutti metodi delle istanze della classe `C`, essendo `h` esattamente equivalente a `g`. Si noti che questa pratica di solito serve solo a confondere chi debba leggere un programma.

I metodi possono chiamare altri metodi usando gli attributi metodo dell'argomento `self`, p.e.:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

I metodi possono referenziare nomi globali allo stesso modo delle funzioni ordinarie. Lo scope globale associato a un metodo è il modulo che contiene la definizione della classe.

(La classe in se stessa non è mai usata come scope globale!). Mentre raramente si incontrano buone ragioni per usare dati globali in un metodo, ci sono molti usi legittimi dello scope globale: per dirne uno, funzioni e moduli importati nello scope globale possono essere usati dai metodi, come pure funzioni e classi in esso definite. Di solito la classe che contiene il metodo è essa stessa definita in tale scope globale, e nella sezione seguente saranno esposte alcune ottime ragioni per le quali un metodo potrebbe voler referenziare la sua stessa classe!

9.5 Ereditarietà

Naturalmente le classi non sarebbero degne di tal nome se non supportassero l'ereditarietà. La sintassi per la definizione di una classe derivata ha la forma seguente:

```
class NomeClasseDerivata(NomeClasseBase):
  <istruzione-1>
  .
  .
  .
  <istruzione-N>
```

Il nome `NomeClasseBase` dev'essere definito in uno scope contenente la definizione della classe derivata. Al posto di un nome di classe base è permessa anche un'espressione. Questo è utile quando la classe base è definita in un altro modulo, p.e.,

```
class
  NomeClasseDerivata(nomemodulo.NomeClasseBase):
```

L'esecuzione della definizione di una classe derivata procede nello stesso modo di una classe base. Quando viene costruito l'oggetto classe, la classe base viene ricordata. Questa viene usata per risolvere i riferimenti ad attributi: se un attributo richiesto non è rinvenuto nella classe, esso viene cercato nella classe base. Tale regola viene applicata ricorsivamente se la classe base è a sua volta derivata da una qualche altra classe.

Non c'è nulla di speciale da dire circa l'istanziazione delle classi derivate: `NomeClasseDerivata()` crea una nuova istanza della classe. I riferimenti ai metodi sono risolti come segue: viene ricercato il corrispondente attributo di classe, discendendo lungo la catena delle classi base se necessario, e il riferimento al metodo è valido se questo produce un oggetto funzione.

Le classi derivate possono sovrascrivere i metodi delle loro classi base. Dato che i metodi non godono di alcun privilegio speciale quando chiamano altri metodi dello stesso oggetto,

un metodo di una classe base che chiami un altro metodo definito nella stessa classe base può in effetti finire col chiamare un metodo di una classe derivata che prevale su di esso. (Per i programmatori C++: in Python tutti i metodi sono `virtuali`) .
La sovrascrizione di un metodo di una classe derivata può in effetti voler estendere più che semplicemente rimpiazzare il metodo della classe base con lo stesso nome. C'è un semplice modo per chiamare direttamente il metodo della classe base: basta invocare "`NomeClasseBase.nomemetodo(self, argomenti)`". Questo in alcune occasioni è utile pure ai client. (Si noti che funziona solo se la classe base è definita o importata direttamente nello scope globale).

9.5.1 Ereditarietà Multipla

Python supporta pure una forma limitata di ereditarietà multipla. Una definizione di classe con classi base multiple ha la forma seguente:

```
class NomeClasseDerivata(Base1, Base2, Base3):
    <istruzione-1>
    .
    .
    .
    <istruzione-N>
```

La sola regola necessaria per chiarire la semantica è la regola di risoluzione usata per i riferimenti agli attributi di classe. Essa è `prima in profondità' (`depth-first'), da sinistra a destra. Perciò, se un attributo non viene trovato in `NomeClasseDerivata`, viene cercato in `Base1`, poi (ricorsivamente) nelle classi base di `Base1` e, solo se non vi è stato trovato, viene ricercato in `Base2`, e così via.
(Ad alcuni una regola `prima in larghezza' (`breadth first'), che ricerca in `Base2` e `Base3` prima che nelle classi base di `Base1`, sembra più naturale. Comunque ciò richiederebbe di sapere se un particulare attributo di `Base1` sia in effetti definito in `Base1` o in una delle sue classi base prima che si possano valutare le conseguenze di un conflitto di nomi con un attributo di `Base2`. La regola depth-first non fa alcuna differenza tra attributi direttamente definiti o ereditati di `Base1`).
È chiaro che un uso indiscriminato dell'ereditarietà multipla è un incubo per la manutenzione, visto che Python si affida a convenzioni per evitare conflitti accidentali di nomi. Un problema caratteristico dell'ereditarietà multipla è quello di una classe derivata da due classi che hanno una classe base comune. Mentre è abbastanza semplice

calcolare cosa succede in questo caso (l'istanza avrà una singola copia delle `variabili istanza' o attributi dato usati dalla classe base comune), non c'è evidenza dell'utilità di tali semantiche.

9.6 Variabili Private

C'è un supporto limitato agli identificatori privati a una dasse. Qualsiasi identificatore della forma `__spam` (come minimo due trattini bassi all'inizio, al più un trattino basso in coda) viene ora rimpiazzato a livello di codice eseguito con `_nomeclasse__spam`, dove `nomeclasse` è il nome della classe corrente privato dei trattini bassi in testa. Tale mutilazione (`mangling') viene eseguita senza riguardo verso la posizione sintattica dell'identificatore, quindi può essere usata per definire istanze, metodi e variabili di classe privati, come pure globali, e anche per memorizzare variabili istanza private per questa classe sopra istanze di *altre* classi. Potrebbe capitare un troncamento, nel caso in cui il nome mutilato fosse più lungo di 255 caratteri. Al di fuori delle classi, o quando il nome della classe consiste di soli trattini bassi, non avviene alcuna mutilazione.

La mutilazione dei nomi è intesa a fornire alle classi un modo semplice per definire variabili istanza e metodi `privati', senza doversi preoccupare di variabili istanza definite da classi derivate, o di pasticci con le variabili compiuti da codice esterno alla classe. Si noti che le regole di mutilazione sono pensate principalmente per evitare errori accidentali; è ancora possibile a propria discrezione accedere o modificare una variabile considerata privata. Questo può anche essere utile, p.e. al debugger, e questa è una ragione per cui questa scappatoia non è impedita. (Avviso: la derivazione di una classe con lo stesso nome della classe base rende possibile l'uso delle variabili private della classe base).

Si noti che il codice passato a `exec`, `eval()` o `evalfile()` non considera il nome di classe della classe che le invoca come classe corrente; ciò è simile a quanto succede con la dichiarazione `global`, il cui effetto è ristretto in modo simile al codice che viene byte-compilato assieme. La stessa limitazione si applica a `getattr()`, `setattr()` e `delattr()`, come pure quando si referenzia direttamente `__dict__`.

Ecco un esempio di una classe che implementa i propri metodi `__getattr__()` e `__setattr__()` e memorizza tutti gli attributi in una variabile privata, in un modo che funziona in tutte le versioni di Python, comprese quelle disponibili venisse aggiunta questa funzionalità:

```
class AttributiVirtuali:
```

```
    __vdict = None
    __vdict_name = locals().keys()[0]

    def __init__(self):
        self.__dict__[self.__vdict_name] = {}

    def __getattr__(self, name):
        return self.__vdict[name]

    def __setattr__(self, name, value):
        self.__vdict[name] = value
```

9.7 Rimasugli e Avanzi

A volte è possono essere utili tipi di dati simili al `record' del Pascal o allo `struct' del C, che riuniscono insieme alcuni elementi dato dotati di nome. Lo si otterrà facilmente definendo una classe vuota, p.e.:

```
class Employee:
    pass

john = Employee() # Crea un record vuoto

# Riempie i campi del record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A un pezzo di codice Python che si aspetta un particolare tipo di dato astratto si può invece spesso passare una classe che emula i metodi di tale tipo di dato. Per esempio, se si ha una funzione che effettua la formattazione di alcuni dati provenienti da un oggetto file, si può definire una classe con metodi `read()` e `readline()` che invece prende i dati da un buffer di stringhe, e lo passa come argomento.

Anche gli oggetti istanza di metodo hanno attributi: `m.im_self` è l'oggetto di cui il metodo è un'istanza, e `m.im_func` è l'oggetto funzione corrispondente al metodo.

9.7.1 Le Eccezioni Possono Essere Classi

Le eccezioni definite dall'utente non devono più essere solo oggetti stringa, possono pure essere identificate da classi. Usando tale meccanismo diventa possibile creare gerarchie estendibili di eccezioni.

Ci sono due nuove forme (semantiche) valide per l'istruzione `raise`:

```
raise Classe, istanza

raise instanza
```

Nella prima forma, *istanza* dev'essere un'istanza di `Classe` o di una sua classe derivata. Nel secondo si tratta di un'abbreviazione per:

```
raise istanza.__class__, istanza
```

Una clausola except può elencare classi come pure oggetti stringa. Una classe in una clausola except è compatibile con un'eccezione se è la stessa classe (dell'eccezione) o una sua classe base (non funziona all'inverso, una clausola except che riporti una classe derivata non è compatibile con una classe base). Per esempio, il codice seguente stamperà B, C, D in tale ordine:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Si noti che se le clausole except fossero invertite (con "`except B`" all'inizio), verrebbe stampato B, B, B. Viene infatti attivata la prima clausola except che trova corrispondenza.

Quando viene stampato un messaggio di errore per un'eccezione non gestita e si tratta di una classe, viene stampato il nome della classe, poi un duepunti e uno spazio, infine l'istanza convertita in una stringa tramite la funzione built-in `str()`.

10. E Adesso?

Si auspica che la lettura di questo tutorial abbia rinforzato il vostro interesse per Python. E adesso cosa dovreste fare?

[Nota: stiamo preparando del materiale in italiano da affiancare al tutorial, esempi pratici e note che siano d'aiuto nella fase di "svezzamento" :-) NdT]

Dovreste leggere, o almeno scorrere, il Library Reference, che offre materiale di consultazione completo (sebbene conciso) su tipi, funzioni e moduli che possono farvi risparmiare molto tempo nella stesura di programmi Python. La distribuzione Python standard comprende un *mucchio* di codice sia in C che in Python; ci sono moduli per leggere caselle di posta Unix, recuperare documenti attraverso HTTP, generare numeri casuali, analizzare opzioni da riga di comando, scrivere programmi CGI, comprimere dati e molto altro ancora; scorrere il Library Reference vi darà un'idea di ciò che è disponibile. Il principale sito web Python è http://www.python.org; esso contiene codice, documentazione e puntatori a pagine su Python in tutto il Web. Questo sito web è presente come mirror in molte parti del mondo, come Europa, Giappone e Australia; un sito mirror può essere più veloce del sito principale, in dipendenza della vostra collocazione geografica. Un sito più informale è http://starship.python.net/, che contiene un sacco di home page personali su Python; molta gente rende ivi disponibile del software da scaricare.

Per porre questioni correlate a Python e riportare problemi, si può inviare un messaggio al newsgroup comp.lang.python, o un'email alla lista di discussione presso python-list@python.org. Il newsgroup e la lista di discussione sono collegati, quindi messaggi inviati all'uno saranno inoltrati automaticamente all'altro. Ci sono circa 120 nuovi messaggi al giorno in cui si pongono domande, si ottengono risposte, si propongono nuove funzionalità e si annunciano nuovi moduli. Prima di inviare un messaggio, ci si assicuri di controllare che la soluzione ai vostri quesiti non compaia già nella FAQ presso http://www.python.org/doc/FAQ.html [o alla versione in traduzione su questo sito NdT] o nella directory Misc/ della distribuzione sorgente di Python. Gli archivi della lista di

discussione sono consultabili presso http://www.python.org/pipermail/. Nella FAQ sono contenute le risposte a molte domande che vengono riproposte di frequente, potrebbe contenere anche la soluzione al vostro problema.