Il tutorial di Python

Versione 2.3.4

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

12 dicembre 2004

Python Software Foundation
Email: docs@python.org
Traduzione presso
http://www.zonapython.it
Email: zap@zonapython.it

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 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 *La libreria di riferimento di Python*. *Il manuale di riferimento di Python* 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 fornire un'idea precisa dello stile del linguaggio. Dopo averlo letto si sarà capaci di leggere e scrivere moduli e programmi in Python, e quindi pronti ad imparare di più sui vari moduli della libreria Python descritta nel documento *La libreria di riferimento di Python*.

Traduzione in italiano

Il presente tutorial è stato realizzato da Ferdinando Ferranti zap@zonapython.it, basandosi sulla versione tradotta in italiano del tutorial da Riccardo Fabris, allegata alla documentazione di Python 2.0, del 16 ottobre 2000.

- Traduzione in Italiano del tutorial di Guido van Rossum, Versione 2.3.4, 20 agosto 2004, Ferdinando Ferranti.
- Traduzione in Italiano del tutorial di Guido van Rossum, Versione 2.3, 29 luglio 2003, Ferdinando Ferranti.
- Traduzione in Italiano del tutorial di Guido van Rossum, Versione 2.0, 16 ottobre 2000, Riccardo Fabris.

INDICE

CAPITOLO

UNO

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, 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 C 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" (NdT: nativi) 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 l'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 o 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 e 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 creare link 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 creare link dall'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 deriva 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!

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'interprete. 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.

CAPITOLO

DUE

Usare l'interprete Python

2.1 Invocare l'interprete

L'interprete Python sulle macchine UNIX sulle quali è disponibile di solito è installato in '/usr/local/bin/'; aggiungendo '/usr/local/bin' al 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 Windows) al prompt primario fa sì che 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 echo 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 ad 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 secondo 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, ad esempio chiamate ad `input()` e `raw_input()`, vengono soddisfatte da *file*. Dato che questo file è già 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 allo standard input 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 viene letto dallo standard input, per lo stesso motivo illustrato nel paragrafo precedente.

2.1.1 Passaggio di argomenti

Quando noti all'interprete, il nome dello script e gli 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 né script né argomenti, `sys.argv[0]` è una stringa vuota. Quando il nome dello script è fornito come `'-'` (che identifica lo standard input), `sys.argv[0]` viene impostato a `'-'`. Allorché viene usato **-c** *comando*, `sys.argv[0]` viene impostato a `-c`. Le opzioni trovate dopo **-c** *comando* non vengono 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'interprete è in *modalità interattiva*. 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 predefinito è quello secondario, 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
>>> if 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'interprete 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 nel flusso dello standard error; l'output normale dei comandi eseguiti viene scritto sullo standard output.

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.[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

[1] Un problema con il pacchetto GNU Readline potrebbe impedirlo.

```
#! /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. Su qualche piattaforma, questa prima riga terminerà in stile UNIX ('`\n`'), non in Mac OS ('`\r`') o in Windows ('`\r\n`'). Si noti che il carattere hash '#' (NdT: in italiano cancelletto, diesis) viene usato in Python per iniziare un commento).

Lo script può essere reso eseguibile, modificandone i permessi, usando il comando **chmod**:

```
$ chmod +x myscript.py
```

2.2.3 Codifica dei file sorgenti

È possibile usare una codifica differente dall'ASCII nei file sorgenti Python. La strada maestra consiste nell'inserire una speciale riga di commento che definisca la codifica del file sorgente, appena dopo la riga che inizia con `#!`:

```
# -*- coding: iso-8859-1 -*-
```

Con questa dichiarazione, tutti i caratteri nel sorgente verranno elaborati come `iso-8859-1` e sarà possibile scrivere direttamente stringhe costanti Unicode (NdT: string literals —> stringhe costanti manifeste, abbreviato con stringhe costanti. Nei testi di teoria informatica "costanti manifeste" è la normale espressione italiana dove in quelli in inglese si trova "literals") nella codifica selezionata. L'elenco delle codifiche utilizzabili può essere rinvenuto nella *La libreria di riferimento di Python*, nella sezione `codecs`.

Se si usa un editor che supporta il salvataggio dei file in `UTF-8`, con l'indicazione UTF-8 *byte order mark* (ovvero BOM), è possibile usare questo sistema al posto della dichiarazione. IDLE supporta questa caratteristica se è stato impostato `Options/General/Default Source Encoding/UTF-8`. Da notare che questa indicazione non è ammessa in vecchie versioni di Python (2.2 e precedenti) e anche che non è consentita dal sistema operativo per i file `#!`.

Usando l'UTF-8 (con il metodo dell'indicazione o quello della dichiarazione della codifica), si potranno usare simultaneamente caratteri di molte delle lingue usate nel mondo in stringhe costanti o commenti. Non sono supportati i caratteri non-ASCIInegli identificatori. Per poter visualizzare correttamente tutti questi caratteri, si dovrà usare un editor in grado di riconoscere file UTF-8 e utilizzare font che sappiano rappresentare correttamente i caratteri presenti nel file.

2.2.4 Il file di avvio in modalità interattiva

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 dei 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` tramite questo file.

Se si vuole far leggere all'interprete un file di avvio addizionale dalla directory corrente, è possibile farlo tramite il file di avvio globale, per esempio '`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
nomefile = os.environ.get('PYTHONSTARTUP')
if nomefile and os.path.isfile(nomefile):
    execfile(nomefile)
```

CAPITOLO

TRE

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, dopo degli spazi bianchi o dopo del codice, ma non dentro una stringa costante. Un carattere hash dentro una stringa costante è solamente un carattere hash.

Alcuni esempi:

```
# questo è il primo commento
SPAM = 1                 # e questo è il secondo
                         # ... e ora il terzo!
STRING = "# Questo non è un commento."
```

3.1 Usare Python come una calcolatrice

Proviamo con qualche semplice comando Python. Avviate l’interprete e si attendete 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. Ad 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 ad 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:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Anche i numeri complessi vengono supportati; per contrassegnare i numeri immaginari si usa il suffisso ‘`j`’ o ‘`J`’. I numeri complessi con una componente reale non nulla vengono indicati 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 vengono 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 in 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. Usate `abs(z)` per ottenere la sua grandezza (in virgola mobila) o `z.real` per ottenere la sua parte reale.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

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

```
>>> tassa = 12.5 / 100
>>> prezzo = 100.50
>>> prezzo * tassa
12.5625
>>> prezzo + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Questa variabile dev'essere trattata dall'utente come di 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 ed 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 costanti possono estendersi su più righe in modi diversi. Si possono scrivere lunghe righe usando le barre oblique inverse (NdT: il cosiddetto "escape") come ultimo carattere di riga, questo indicherà che la riga successiva sarà in realtà la logica continuazione della precedente:

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

print ciao
```

Si noti che il carattere di fine riga deve sempre essere inserito in una stringa usando `\n`; il carattere di fine riga successivo, composto dalla barra obliqua inversa viene ignorato. L'esempio verrà quindi stampato così:

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

Se si volesse comporre una stringa letterale "raw", comunque, la sequenza `\n` non verrebbe convertita nel fine riga e, insieme alla barra obliqua inversa alla fine della riga del sorgente, si ritroverebbero entrambe le sequenze come nella stringa data. Quindi, l'esempio:

```
ciao = r"Questa è una stringa abbastanza lunga che contiene\n\
parecchie righe di testo proprio come si farebbe in C."

print ciao
```

stamperebbe:

```
Questa è una stringa abbastanza lunga che contiene\n\
parecchie righe di testo proprio come si farebbe in C.
```

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, questi verranno inclusi nella stringa.

```
print """
Uso: comando [OPZIONI]
   -h                        Visualizza questo messaggio
   -H hostname               Hostname per connettersi a
"""
```

produrrà il seguente output:

```
Uso: comando [OPZIONI]
   -h                        Visualizza questo messaggio
   -H hostname               Hostname per connettersi a
```

L'interprete stampa il risultato delle operazioni sulle stringhe nello stesso modo in cui vengono inserite in input: tra virgolette, con virgolette interne ed altri caratteri particolari protetti da barre oblique inverse (NdT: 'backslash'), per mostrare il loro esatto valore. La stringa viene 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 `*`:

```
>>> parola = 'Aiuto' + 'A'
>>> parola
'AiutoA'
>>> '<' + parola*5 + '>'
'<AiutoAAiutoAAiutoAAiutoAAiutoA>'
```

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

```
>>> 'str' 'ing'                   #  <-  Questo è ok
'string'
>>> 'str'.strip() + 'ing'         #  <-  Questo è ok
'string'
>>> 'str'.strip() 'ing'           #  <-  Questo non è valido
  File "<stdin>", line 1, in ?
    'str'.strip() '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 associato al carattere di separazione; un carattere è semplicemente una stringa di lunghezza uno. Come in Icon, possono essere specificate sottostringhe con la *notazione a fette* (NdT: 'slice'): due indici separati dal carattere due punti.

```
>>> parola[4]
'o'
>>> parola[0:2]
'Ai'
>>> parola[2:4]
'ut'
```

Gli indici della fetta hanno utili comportamenti predefiniti. Il primo indice, se omesso, viene impostato al valore predefinito `0`. Se viene tralasciato il secondo, viene impostato alla dimensione della stringa affettata.

```
>>> parola[:2]    # I primi due caratteri
'Ai'
>>> parola[2:]    # Tutti eccetto i primi due caratteri
'utoA'
```

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:

```
>>> parola[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> parola[:1] = 'Lavagna'
Traceback (most recent call 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' + parola[1:]
'xiutoA'
>>> 'Lavagna' + parola[5]
'LavagnaA'
```

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

```
>>> parola[:2] + parola[2:]
'AiutoA'
>>> parola[:3] + parola[3:]
'AiutoA'
```

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.

```
>>> parola[1:100]
'iutoA'
>>> parola[10:]
''
>>> parola[2:1]
''
```

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

```
>>> parola[-1]     # L'ultimo carattere
'A'
>>> parola[-2]     # Il penultimo carattere
'o'
>>> parola[-2:]    # Gli ultimi due caratteri
'oA'
>>> parola[:-2]    # Tutta la stringa eccetto i due ultimi caratteri
'Aiut'
```

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

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

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

elemento:

```
>>> parola[-100:]
'AiutoA'
>>> parola[-10]    # errore
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Il modo migliore di tenere a mente come lavorano le fette 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:

```
 +---+---+---+---+---+---+
 | A | i | u | t | o | A |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -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, per esempio la lunghezza di `parola[1:3]` è `2`.

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

```
>>> s = 'supercalifragicospiristicalidoso'
>>> len(s)
32
```

Vedete anche:

Tipi sequenza
(../lib/typesseq.html)
: Le stringhe e le stringhe Unicode vengono descritte nella prossima sezione, vi sono esempi di *tipi sequenza* e supporto per comuni operazioni consentite su simili tipi.

Metodi stringa
(../lib/string-methods.html)
: Sia le semplici stringhe che quelle Unicode garantiscono un grande numero di metodi per le trasformazioni e ricerche di base.

Operazioni sulla formattazione delle stringhe
(../lib/typesseq-strings.html)
: Le operazioni di formattazione vengono invocate quando le stringhe, semplici o Unicode, hanno alla loro sinistra l'operatore `%` descritto qui in dettaglio.

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/ [off-site link]) e 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 pagina di codici 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’). Unicode ha risolto tali problemi definendo un’unica pagina di codici per tutti gli script.

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

```
>>> u’Ciao mondo !’
u’Ciao mondo !’
```

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’Ciao\u0020mondo !’
u’Ciao mondo !’
```

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

Gli altri caratteri vengono 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 molto semplificato.

Per gli esperti, c’è anche una modalità raw, proprio come per le stringhe normali. Si deve prefissare una ’ur’ 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’Ciao\u0020mondo !’
u’Ciao mondo !’
>>> ur’Ciao\\u0020mondo !’
u’Ciao\\\\u0020mondo !’
```

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 strumenti 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 codec 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 uno o più byte. La codifica predefinita è normalmente impostata ad ASCII, che permette tutti i caratteri nell’intervallo da 0 a 127 e scarta tutti gli altri emettendo un errore. Quando si stampa una stringa Unicode, scrivendola in un file o convertendola con `str()`, viene usata la codifica predefinita.

```
>>> u"abc"
u’abc’
>>> str(u"abc")
’abc’
>>> u"äöü"
u’\xe4\xf6\xfc’
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: ’ascii’ codec can’t encode characters in position 0-2:
ordinal not in range(128)
```

Per convertire una stringa Unicode in una 8-bit si usa una specifica codifica, gli oggetti Unicode forniscono il metodo `encode()` che prende un argomento, il nome della codifica. Per le codifiche sono preferiti i nomi in

minuscolo.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

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

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

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), ad 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 due più due. Per esempio, possiamo scrivere una sottosuccessione iniziale della serie di *Fibonacci* facendo come segue:

```
>>> # La 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 riga, a dimostrazione che le espressioni sul lato destro dell'assegnamento vengono tutte valutate prima di effettuare qualsiasi assegnamento. Le espressioni sul lato destro vengono valutate da sinistra a destra.
- 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 riga). Osservate che ogni riga 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 il fine riga 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 fine riga prima di stampare il prompt successivo se l'ultima riga non è stata completata.

CAPITOLO

QUATTRO

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 facoltativa. 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. Ad 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, come le 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 ‘step’ (NdT: ‘passo’):

```
>>> 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`. Questo viene 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:
...         # Il ciclo scorre la sequenza senza trovare il fattore
...         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 True:
...       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 ini-

ziano alla riga successiva, e devono essere indentate. 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 built-in. Di conseguenza ad una variabile globale non può essere assegnato direttamente un valore all'interno di una funzione (a meno che non compaia in un'istruzione `global`), malgrado ci si possa riferire alla variabile.

I parametri attuali (argomenti) di una chiamata a funzione vengono 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* ad un oggetto, non il valore dell'oggetto). [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 è di un tipo che viene riconosciuto dall'interprete come funzione definita dall'utente. Tale valore può essere assegnato a un altro nome che quindi 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 built-in). 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:

[1] In realtà, *chiamata per riferimento* ad oggetto sarebbe una descrizione più appropriata, dato che se viene passato un oggetto mutabile, il chiamante vedrà qualsiasi modifica apportata nella chiamata (p.e. elementi inseriti in una lista).

- L'istruzione `return` restituisce una funzione con un valore. `return` senza un'espressione come argomento restituisce `None`. Anche finire oltre la fine della procedura restituisce `None`.
- L'istruzione `result.append(b)` chiama un *metodo* dell'oggetto lista `result`. Un metodo è una funzione che 'appartiene' ad un oggetto e ha per nome `oggetto.metodo`, dove `oggetto` è un qualche oggetto (può essere un'espressione), e `metodo` è il nome di un metodo che viene 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 predefiniti 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 di quanti ne sono stati consentiti. Per esempio:

```
def ask_ok(prompt, retries=4, complaint='Sì o no, grazie!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('s', 'si', 'sì'): return True
        if ok in ('n', 'no', 'nop', 'nope'): return False
        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)`.

Questo esempio introduce anche la parola chiave `in`. Questo esempio verifica se una sequenza contiene o meno un certo valore.

I valori predefiniti sono valutati al momento della definizione della funzione nella *definizione* dello spazio dei nomi, così che per esempio:

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

Stamperà `5`.

Avviso importante: Il valore predefinito viene valutato una volta sola. Ciò fa sì che le cose siano molto diverse quando si tratta di un oggetto mutabile come una lista, un dizionario o istanze di più classi. 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 predefinito 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 seguito
                             #+ da 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 predefinito 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 (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Quando è presente un parametro formale finale nella forma `**`*nome*, esso riceve un dizionario 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
    keys = keywords.keys()
    keys.sort()
    for kw in 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')
```

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
```

Si noti che il metodo `sort()` applicato alla lista dei nomi degli argomenti delle parole chiave viene chiamato prima di stampare il contenuto del dizionario delle `parole chiave`; se questo non fosse fatto, l'ordine con il quale verrebbero stampati gli argomenti non sarebbe definito.

4.7.3 Liste di argomenti arbitrari

Infine, l'opzione usata meno di frequente consiste nello 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 Suddividere gli argomenti di una lista

La situazione inversa capita quando vi sono argomenti presenti nella lista o nella tupla ma è necessario che siano suddivisi perché la chiamata di una funzione richiede argomenti posizionali ben distinti. Per istanza la funzione `range()` si aspetta gli argomenti *start* e *stop* separati. Se non sono disponibili separatamente si può sempre scrivere una funzione con l'operatore `*` per suddividere gli argomenti di una lista o di una tupla:

```
>>> range(3, 6)             #  la canonica chiamata con argomenti
                            #+ separati
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)            #  chiamata con argomenti suddivisi,
                            #+ provenienti da una lista
[3, 4, 5]
```

4.7.5 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 possono riferirsi a variabili dallo scope (NdT: ambito di visibilità) che le contiene:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

4.7.6 Stringhe di documentazione

Si stanno formando delle convenzioni sul contenuto e la formattazione delle stringhe di documentazione, ovvero le "docstring".

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 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 che processano la documentazione devono toglierla da soli, se lo desiderano. Ciò viene fatto usando la seguente convenzione. La prima riga non vuota *dopo* la prima riga della stringa determina l'ammontare dell'indentazione presente nell'intera stringa di documentazione. Non si può usare la prima riga 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 righe della stringa. Non dovrebbero esserci righe indentate in misura minore, ma se ci fossero tutti gli spazi in testa alla riga dovrebbero essere eliminati. L'equivalenza in spazi dovrebbe essere testata 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.
```

CAPITOLO

CINQUE

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. Le parentesi quadrate intorno alla *i*, inserita dopo il metodo, informano che il parametro è facoltativo, nell'uso non dovranno essere scritte le parentesi quadre. Questa notazione è riscontrabile frequentemente nella *Libreria di riferimento di Python*.)

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 (NdT: l'output è la stessa lista riordinata).

reverse()
Inverte gli elementi della lista, sul posto (NdT: come sopra).

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). Per esempio:

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

'`reduce(`*funzione*`, `*sequenza*`)`' restituisce un singolo valore ottenuto invocando la funzione binaria (NdT: a due argomenti) *funzione* sui primi due elementi della *sequenza*, quindi sul risultato dell'operazione e sull'elemento successivo, e così via. Ad 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 quel caso è il valore di partenza ad

essere restituito se la sequenza è vuota, e la funzione viene applicata prima a tale valore e al primo elemento della sequenza, quindi al risultato di tale operazione e l'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
```

Non si usi questo esempio di definizione `sum()`: dato che sommare numeri è piuttosto comune, è già disponibile una funzione built-in `sum(`*sequenza*`)` e funziona esattamente come questa. Nuovo nella versione 2.3

5.1.4 Costruzioni di lista

Le costruzioni 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 i costrutti accennati. Ogni costruzione di lista consiste di un'espressione a cui segue 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]     # errore - per le tuple è richiesta la parentesi
  File "<stdin>", line 1, in ?
    [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]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Le costruzioni di lista sono molto più flessibili di `map()`, vi si possono applicare funzioni con più di un argomento e funzioni annidate:

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

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 intere 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 dato 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, per esempio coppie di coordinate (x, y), record di 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 0 o 1 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 ad 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 Insiemi

Python include anche tipi di dati per *insiemi* (NdT: *sets*). Un insieme è una collezione non ordinata che non contiene elementi duplicati al suo interno. Solitamente viene usato per verificare l'appartenenza dei membri ed eliminare gli elementi duplicati. Gli oggetti insieme supportano anche le operazioni matematiche come l'unione, l'intersezione, la differenza e la differenza simmetrica.

Questa è una breve dimostrazione:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruits = set(basket)               #  crea un insieme con frutti
                                       #+ senza duplicati
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruits                 #  veloce verifica
                                       #+ dell'appartenenza del membro
True
>>> 'crabgrass' in fruits
False

>>> #  Dimostrazione delle operazioni set su un'unica lettera da due
    #+ parole
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  #  unica lettera in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                              #  lettera in a ma non in b
set(['r', 'd', 'b'])
>>> a | b                              #  lettera in a o in b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                              #  lettera comune in a ed in b
set(['a', 'c'])
>>> a ^ b                              #  lettera in a od in b ma non
                                       #+ in comune tra i due
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

5.5 Dizionari

Un altro tipo di dato built-in utile è il Tipo mappa, *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 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')
True
```

Il costruttore `dict()` crea dizionari direttamente dalla lista di coppie *chiave: valore* immagazzinate in tuple. Quando la coppia forma un modello, la costruzione di lista può specificare liste *chiave: valore* in modo compatto.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in vec])     # uso della costruzione di lista
{2: 4, 4: 16, 6: 36}
```

5.6 Tecniche sui cicli

Quando si usano i cicli sui dizionari, la chiave e il valore corrispondente possono essere richiamati contemporaneamente usando il metodo `iteritems()`.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Quando si usa un ciclo su una sequenza, la posizione dell'indice e il valore corrispondente possono essere richiamati contemporaneamente usando la funzione `enumerate()`.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Utilizzando un ciclo su due o più sequenze contemporaneamente, le voci possono essere accoppiate con la funzione `zip()`.

```
>>> domande = ['nome', 'scopo', 'colore preferito']
>>> risposte = ['lancillotto', 'il santo graal', 'il blu']
>>> for q, a in zip(domande, risposte):
...     print 'Qual'e' il tuo %s?  E' il %s.' % (q, a)
...
Qual'e' il tuo nome?  E' lancillotto.
Qual'e' il tuo scopo?  E' il santo graal.
Qual'e' il tuo colore preferito?  E' il blu.
```

Per eseguire un ciclo inverso su di una sequenza, prima si deve specificare la direzione di avanzamento e quindi chiamare la funzione `reversed()`.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
5
3
1
```

Per eseguire un ciclo ordinato su di una sequenza, si deve usare la funzione `sorted()` che restituisce una nuova lista ordinata finché rimane inalterata la sorgente dei dati da elaborare.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

5.7 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` ed `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. Per esempio, `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 rispetto agli 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 *short-circuit*: 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 short-circuit, 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. Ad 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 ==.

5.8 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 (NdT: delle due sequenze) 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.[1]

[1] Sarebbe meglio non fare affidamento sulle regole di confronto di oggetti di tipi diversi, potrebbero cambiare in versioni future del linguaggio.

CAPITOLO

SEI

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 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'interprete 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 ad inizializzare il modulo. Esse sono eseguite solo la *prima* volta che il modulo viene importato da qualche parte.[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 dei moduli importati vengono inseriti nella tabella dei simboli globali del modulo che li 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

[1] In effetti le definizioni di funzione sono anche 'istruzioni' che vengono 'eseguite'; l'esecuzione inserisce il nome della funzione nella tabella dei simboli globale del modulo.

non è configurata, o quando il file non si trova nelle directory ivi menzionate, la ricerca continua su un percorso predefinito 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 predefinito dipendente dall'installazione. Questo permette ai programmi Python scritti con cognizione di causa di modificare o rimpiazzare il percorso di ricerca dei moduli. Si noti che, siccome la directory corrente contiene lo script che deve essere eseguito, è importante che non abbia lo stesso nome di un modulo standard, altrimenti Python cercherà di importare lo script come se fosse un modulo. Questo genererà un errore. Si veda la sezione 6.2, "Moduli Standard" più avanti.

6.1.2 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' viene registrata in 'spam.pyc', ed il file '.pyc' viene 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 stato 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`. 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 e facendo in modo di avere un piccolo script di avvio che importi tale modulo. È anche possibile rinominare un file '.pyc' o '.pyo' direttamente da 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 più difficile risalire al codice 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, la *Libreria di riferimento di Python* (da qui in avanti verrà indicato come "Libreria di riferimento"). Alcuni moduli sono interni all'interprete ("built-in"). 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 che dipende dalla piattaforma sottostante. Per esempio, 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 built-in 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. Viene inizializzata con un percorso predefinito ottenuto dalla variabile di ambiente PYTHONPATH, o da un valore predefinito built-in 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 built-in `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)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',
 'version', 'version_info', 'warnoptions']
```

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, etc..

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

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FloatingPointError', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError', 'OverflowWarning',
 'PendingDeprecationWarning', 'ReferenceError',
 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'True', 'TypeError', 'UnboundLocalError', 'UnicodeError', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '__debug__', '__doc__',
 '__import__', '__name__', 'abs', 'apply', 'bool', 'buffer',
 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
 'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'min',
 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit',
 'range', 'raw_input', 'reduce', 'reload', 'repr', 'round',
 'setattr', 'slice', 'staticmethod', 'str', 'string', 'sum', 'super',
 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 I package

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

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 fra loro, 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 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 package (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
              ...
```

Quando il package viene importato, Python cerca all'interno delle directory in `sys.path` per controllare i package contenuti nelle sottodirectory.

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 un riferimento al suo nome completo.

```
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'ulteriore variazione 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` *elemento*, 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 un'eccezione `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`. Ad 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 nomi lunghi di moduli.

L'unica soluzione è che spetti all'autore fornire 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 che si tratti di una lista di nomi di moduli 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 viene definito, l'istruzione `from Sound.Effects import *` *non* importa tutti i sottomoduli del package `Sound.Effects` nello spazio dei 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. Si consideri questo codice:

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

In questo esempio, i moduli echo e surround sono importati nello spazio dei nomi corrente poiché vengono 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 la pratica di importare con * da un modulo è malvista, 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 ad 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`.

6.4.3 Package in molteplici directory

I package supportano un attributo ancora più speciale, `__path__`. Viene inizializzato per essere una lista al cui interno risulti il nome della directory che contiene il package '__init__.py' prima che il codice in quel file sia eseguito. Questa variabile può essere modificata; in questo modo, avrà effetto su ricerche future dei moduli e dei sottopackage ivi contenuti.

Mentre questa caratteristica non è spesso necessaria, può essere usata per estendere l'insieme dei moduli trovati in un package.

CAPITOLO

SETTE

Input ed 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 Libreria di riferimento 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 sé 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 (NdT: 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()` o `str()`. Gli apici inversi (`‘ ‘`) sono equivalenti a `repr()`, ma l'uso di questa forma è scoraggiato.

La funzione `str()` restituisce la rappresentazione del valore in termini umanamente comprensibili, mentre `repr()` genera la rappresentazione del valore comprensibile dall'interprete (o solleverà un'eccezione del tipo `SyntaxError` se questa non ha una sintassi equivalente). Per oggetti che non hanno una specifica rappresentazione in termini umanamente comprensibili, `str()` restituirà lo stesso valore di `repr()`. Molti valori, simili a quelli numerici o a strutture come liste e dizionari, hanno la stessa rappresentazione usando le due funzioni. Nello specifico, stringhe e numeri in virgola mobile, hanno due distinte rappresentazioni.

Alcuni esempi:

```
>>> s = 'Ciao, mondo.'
>>> str(s)
'Ciao, mondo.'
>>> repr(s)
"'Ciao, mondo.'"
>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'Il valore di x è ' + repr(x) + ' e y è ' + repr(y) + '...'
>>> print s
Il valore di x è 32.5 e y è 40000...
>>> # repr() aggiunge gli apici di stringa e le barre rovesciate:
... ciao = 'ciao, mondo\n'
>>> ciaos = repr(ciao)
>>> print ciaos
'ciao, mondo\n'
>>> # L'argomento di repr() può essere anche un oggetto Python:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
>>> # Gli apici inversi possono essere convenienti in sessioni interattive:
... `x, y, ('spam', 'eggs')`
"(32.5, 40000, ('spam', 'eggs'))"
```

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

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Si noti la virgola in coda sulla riga precedente
...     print repr(x*x*x).rjust(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 metodo `rjust()` di oggetti stringa, che giustifica a destra una stringa in un campo di ampiezza data, riempiendola di spazi a sinistra. Questo metodo non scrive niente ma restituisce una nuova

stringa. Se la stringa di input è troppo lunga non viene troncata ma restituita intatta; questo potrebbe scombussolare l'allineamento delle colonne desiderato, ma di solito questo è preferibile all'alternativa, cioè riportare un valore menzognero. Se davvero si desidera il troncamento si può sempre aggiungere un'operazione di affettamento, come in '`x.ljust( n)[:n]`'.

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

```
>>> '12'.zfill(5)
'00012'
+>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(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 più 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 nome, telefono in table.items():
...     print '%-10s ==> %10d' % (nome, telefono)
...
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 built-in `str()`. L'uso di `*` per passare l'ampiezza o la 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 la 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 built-in `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 solamente 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 facoltativo; 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 ad 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 facoltativo. 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.\n'
>>> f.read()
''
```

`f.readline()` legge una singola riga dal file; un carattere di fine riga (`\n`) viene lasciato alla fine della stringa, e viene omesso solo nell'ultima riga del file nel caso non finisca con un fine riga. 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 fine riga.

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

`f.readlines()` restituisce una lista contenente tutte le righe di dati presenti nel file. Se le viene passato un parametro facoltativo *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.\n', 'Seconda riga del file\n']
```

`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 viene calcolata aggiungendo ad *offset* 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 ed il suo valore predefinito 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 sesto 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 l'oggetto file falliranno automaticamente.

```
>>> f.close()
>>> f.read()
Traceback (most recent call 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 Libreria di riferimento 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 `int()`, che prende una stringa come `'123'` e restituisce il corrispondente valore numerico 123. Comunque quando si desidera salvare tipi di dati 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 dati 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 sotto forma di stringa; tale processo è chiamato *pickling* (NdT: letteralmente conservazione sotto aceto, in pratica si tratta di serializzazione, attenzione a non confonderla con quella del modulo `marshal`). La ricostruzione dell'oggetto a partire dalla rappresentazione sotto forma di stringa è chiamata *unpickling*. Tra la serializzazione e la deserializzazione, 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 la serializzazione dell'oggetto occupa solo una riga di codice:

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

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

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

Ci sono altre varianti del procedimento, usate quando si esegue la serializzazione di molti oggetti o quando non si vuole scrivere i dati ottenuti in un file; si consulti la documentazione completa di `pickle` nella *libreria di riferimento di Python*.

`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 serializzazione ed deserializzazione.

CAPITOLO

OTTO

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

L'analizzatore sintattico ('parser') riporta la riga 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 riga, in modo che si sappia dove andare a guardare, nel caso l'input provenga da uno script.

8.2 Le eccezioni

Anche se un'istruzione, o un'espressione, è sintatticamente 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 (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

L'ultima riga del messaggio di errore indica cos'è successo. Le eccezioni sono di diversi tipi, ed il loro tipo viene stampato come parte del messaggio: i tipi che compaiono nell'esempio sono `ZeroDivisionError`, `NameError` e `TypeError`. La stringa stampa quale tipo d'eccezione è occorsa ed il nome dell'eccezione stessa. 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 d'eccezione; anche il suo significato dipende dal tipo d'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 righe di codice sorgente; in ogni caso non mostrerà righe lette dallo standard input.

La *La libreria Python di riferimento* elenca le eccezioni built-in ed 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 un'interruzione generata dall'utente viene segnalata sollevando un'eccezione `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(raw_input("Introduci un numero: "))
...         break
...     except ValueError:
...         print "Oops! 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 interviene 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 un'*eccezione non gestita* 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 ricorrono nella clausola try corrispondente, non in altri gestori della stessa istruzione `try`. Una clausola except può nominare più di un'eccezione sotto forma di tupla, per esempio:

```
... 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 sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
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* facoltativa, che, 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, 'è di', 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 stata rilevata dal codice protetto dall'istruzione `try` … `except`.

Quando interviene 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.

La clausola except può specificare una variabile dopo il nome dell'eccezione (o una lista di nomi). La variabile è limitata al caso di un'eccezione con argomenti immagazzinati in `instance.args`. Per convenienza, l'eccezione definisce `__getitem__` e `__str__` in modo da poter accedere agli argomenti o stamparli direttamente senza dover riferirsi ad `.args`.

```
>>> try:
...    raise Exception('spam', 'eggs')
... except Exception, inst:
...    print type(inst)    # l'istanza dell'eccezione
...    print inst.args     # argomenti immagazzinati in .args
...    print inst          # __str__ consente di stampare gli argomenti
...    x, y = inst         # __getitem__ consente di scomporre gli argomenti
...    print 'x =', x
...    print 'y =', y
...
<type 'instance'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

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 vengono sollevate 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 (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

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

Se si ha bisogno di sollevare un'eccezione, ma non si vuole gestirla, si può usare una semplice forma dell'istruzione `raise` che consente di risollevare l'eccezione.

```
>>> try:
...     raise NameError, 'HiThere'
... except NameError:
...     print 'Un'eccezione svetta qui intorno!'
...     raise
...
Un'eccezione svetta qui intorno!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

8.5 Eccezioni definite dall'utente

I programmi possono dare un nome a delle proprie eccezioni creando una nuova classe di eccezioni. Le eccezioni dovrebbero, di regola, derivare dalla classe `Exception`, direttamente o indirettamente. Per esempio:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(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, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Una classe di eccezioni si può definire come una qualsiasi altra classe, solitamente piuttosto semplice, in grado di offrire un certo numero di attributi che permettono l'estrazione delle informazioni sull'errore tramite la gestione dell'eccezione. Nel realizzare un modulo capace di sollevare eccezioni per diversi, distinti errori, una pratica comune è creare una classe base per le eccezioni definite nel modulo e una sottoclasse che per ogni specifica condizione d'errore:

```
class Error(Exception):
    """Classe base per le eccezioni definite in questo modulo."""
    pass

class InputError(Error):
    """Eccezione sollevata per errori di immissione.

    Attributi:
        espressione -- dati in ingresso che contengono errori
        messaggio -- spiegazione dell'errore
    """

    def __init__(self, espressione, messaggio):
        self.espressione = espressione
        self.messaggio = messaggio

class TransitionError(Error):
    """Solleva l'eccezione quando in un passaggio si tenta di
    specificare un'operazione non consentita.

    Attributi:
        precedente -- stato all'inizio dell'operazione
        successivo -- cerca il nuovo stato
        messaggio -- spiegazione del perché la specifica operazione non
                     è consentita
    """

    def __init__(self, precedente, successivo, messaggio):
        self.precedente = precedente
        self.successivo = successivo
        self.messaggio = messaggio
```

Molte eccezioni vengono definite con nomi che terminano con "Error," analogamente ai nomi delle eccezioni convenzionali.

Molti moduli standard usano questa tecnica per le proprie eccezioni per riportare errori che possono verificarsi nelle funzioni che definiscono. Ulteriori informazioni sulle classi sono presenti nel capitolo 9, "Classi."

8.6 Definire azioni di chiusura

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

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Addio, mondo!'
...
Addio, mondo!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

Una *clausola finally* viene eseguita comunque, che l'eccezione sia stata sollevata nella clausola try o meno. Nel caso sia stata sollevata un'eccezione, questa viene risollevata dopo che la clausola finally è stata eseguita. La clausola finally viene eseguita anche "quando si esce" da un'istruzione `try` per mezzo di un'istruzione `break` o `return`.

Il codice nella clausola finally è utile per abbandonare risorse esterne (come nel caso di file o connessioni di rete),

senza curarsi del fatto che si sia usata con successo o meno la risorsa esterna.

Un’istruzione `try` deve avere una o più clausole except o una clausola finally, ma non entrambe.

CAPITOLO

NOVE

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 miscuglio 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 riferirsi ai membri dell'oggetto dai suoi metodi: la funzione del metodo viene dichiarata con un primo argomento esplicito che rappresenta l'oggetto, che viene fornito in modo implicito dalla chiamata. Come in Smalltalk, le classi in sé 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, diversamente da da quanto accade 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 mediante istanze di classe.

9.1 Qualche parola sulla terminologia

Data la mancanza di una terminologia universalmente accettata quando si parla di classi, si farà 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.

Si avvisa che per i lettori con conoscenze di programmazione orientata agli oggetti c'è un'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 una parte 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 multipla) possono essere associati allo stesso oggetto. In altri linguaggi ciò è noto come 'aliasing'. Questo non viene di solito apprezzato dando 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 etc.). Questo viene 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 ambiti di visibilità di Python e gli spazi dei nomi

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

Si inizia con alcune definizioni.

Lo *spazio dei nomi* è una mappa che collega i nomi agli oggetti. La maggior parte degli spazi dei nomi vengono 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 dei nomi sono: l'insieme dei nomi built-in (funzioni come `abs()` ed i 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 dei nomi. La cosa davvero importante da sapere al riguardo è che non c'è assolutamente nessuna relazione tra nomi uguali in spazi dei nomi diversi; ad 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, spesso si utilizza 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 ad 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 dei nomi! [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`, per esempio '`del nomemodulo.la_risposta`' rimuoverà l'attributo `la_risposta` dall'oggetto chiamato `nomemodulo`.

Gli spazi dei nomi vengono creati in momenti diversi ed hanno tempi di sopravvivenza diversi. Lo spazio dei nomi che contiene i nomi built-in viene creato all'avvio dell'interprete Python e non viene mai cancellato. Lo spazio dei nomi globale di un modulo viene creato quando viene letta la definizione del modulo; normalmente anche lo spazio dei 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, vengono considerate parte di un modulo chiamato `__main__`, cosicché esse hanno il proprio spazio dei nomi globale. In effetti anche i nomi built-in esistono in un modulo, chiamato `__builtin__`.

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

Uno *scope* è una regione del codice di un programma Python dove uno spazio dei nomi è accessibile direttamente. "Direttamente accessibile" qui significa che un riferimento non completamente qualificato ad un nome cerca di trovare tale nome nello spazio dei 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 dei 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 dei nomi che contiene i nomi built-in.

Se il nome viene dichiarato come globale, allora tutti i riferimenti e gli assegnamenti sono diretti allo scope mediano che contiene i nomi globali del modulo. Altrimenti, tutte le variabili trovate fuori dallo scope più interno saranno in sola lettura.

Di solito lo scope locale si riferisce ai nomi locali della funzione corrente (dal punto di vista del codice). All'esterno delle funzioni, lo scope locale fa riferimento allo stesso spazio dei nomi come scope globale: lo spazio dei nomi del modulo. La definizione di una classe colloca ancora un'altro spazio dei nomi nello scope locale.

[1] Eccetto che per una cosa. Gli oggetti modulo hanno un attributo segreto a sola lettura chiamato `__dict__` che restituisce il dizionario usato per implementare lo spazio dei nomi del modulo; il nome `__dict__` è un attributo ma non un nome globale. Ovviamente usarlo viola l'astrazione dell'implementazione dello spazio dei nomi, e l'uso dovrebbe essere limitato a cose tipo i debugger post-mortem.

È importante capire che gli scope vengono determinati letteralmente secondo il codice: lo scope globale di una funzione definita in un modulo è lo spazio dei 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 solamente nomi ad oggetti. Lo stesso vale per le cancellazioni: l'istruzione '`del x`' rimuove l'associazione di `x` dallo spazio dei nomi in riferimento allo 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 Un primo sguardo 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 plausibile 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 ritorneremo sopra più avanti. Le definizioni di funzione dentro una classe normalmente hanno un elenco di argomenti di aspetto peculiare, dettato da 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 dei nomi, usato come scope locale. Perciò tutti gli assegnamenti a variabili locali finiscono in questo nuovo spazio dei nomi. In particolare, le definizione di funzione vi 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 (NdT: wrapper) per i contenuti dello spazio dei 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) viene ripristinato, e l'oggetto classe è quivi associato al nome della classe indicato 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 dei nomi della classe al momento della creazione dell'oggetto classe. Così, se la definizione di classe fosse del tipo:

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

Quindi `MiaClasse.i` e `MiaClasse.f` sarebbero riferimenti validi ad attributi, che restituirebbero rispettivamente un intero ed un oggetto metodo. 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, in 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 restituisce 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 = partimag
...
>>> 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 per mezzo dell'istanziazione degli oggetti sono i riferimenti ad attributo. Ci sono due tipi di nomi di attributo validi.

Chiameremo il primo *attributo 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 dei metodi 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 alle 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 restituirà 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. Ad esempio:

```
xf = x.f
while True:
    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* argomenti 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 fatto un riferimento ad 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' e costruita una nuova lista di argomenti dall'oggetto istanza e dalla lista di argomenti originale, l'oggetto funzione viene invocato con questa nuova lista di argomenti.

9.4 Note sparse

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, per esempio 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, di sottolineatura), o usare verbi per i metodi e sostantivi per gli attributi dato.

Si può fare riferimento agli attributi dato dai metodi, come pure dagli utenti ordinari (utilizzatori finali) 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.

Gli utilizzatori finali dovrebbero usare gli attributi dato con cura, potrebbero danneggiare degli invarianti conservati dai metodi sovrascrivendoli con i loro attributi dato. Si noti che gli utilizzatori finali possono aggiungere degli attributi dato propri ad 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 plausibile che 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 sé stessa non viene 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 memorizzata. Viene usata per risolvere i riferimenti ad attributi: se un attributo richiesto non viene rinvenuto nella classe, allora 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 vengono risolti così: 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 agli utilizzatori finali. Si noti che funziona solo se la classe base viene 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, 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' (NdT: 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 particolare 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 prima-in-profondità 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 di una classe. Qualsiasi identificatore nella forma `__spam` (come minimo due trattini bassi all'inizio, al più un trattino basso in coda) viene rimpiazzato, a livello di codice eseguito, con `_nomeclasse__spam`, dove `nomeclasse` è il nome della classe corrente, privato dei trattini di sottolineatura all'inizio. Questa mutilazione viene eseguita senza riguardo alla posizione sintattica dell'identificatore, quindi può essere usata per definire istanze, metodi e variabili di classe private, come pure globali, e anche per memorizzare variabili d'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 di sottolineatura, non avviene alcuna mutilazione.

La mutilazione dei nomi fornisce 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, per esempio al debugger, e questa è una ragione per cui questa scappatoia non viene impedita. **Avvertenze:** Derivare 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 fa riferimento direttamente a `__dict__`.

9.7 Rimasugli e avanzi

A volte 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.

9.8 Le eccezioni possono essere classi

Definire le eccezioni da parte dell'utente in classi è giusto. 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. Nella seconda si tratta di un'abbreviazione per:

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

Una classe in una clausola `except` è compatibile con un'eccezione se è della stessa classe o di una sua classe base di questa (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()`.

9.9 Iteratori

Ormai si sarà notato che l'esecuzione della maggior parte degli oggetti può essere replicata ciclicamente mediante un'istruzione `for`:

```
for elemento in [1, 2, 3]:
    print elemento
for elemento in (1, 2, 3):
    print elemento
for chiave in {'uno':1, 'due':2}:
    print chiave
for carattere in "123":
    print carattere
for line in open("myfile.txt"):
    print line
```

Questo stile d'accesso è chiaro, conciso e conveniente. L'uso degli iteratori pervade e unifica Python; dietro le quinte, l'istruzione `for` richiama sull'oggetto contenitore la funzione `iter()`: l'oggetto iteratore da essa restituito definisce il metodo `next()`, il quale introduce degli elementi nel contenitore uno per volta. Quando non ci sono più elementi, `next()` solleva un'eccezione `StopIteration` che termina il ciclo iniziato dal `for`. L'esempio che segue mostra come ciò funzioni:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    it.next()
StopIteration
```

Visti i meccanismi di base del protocollo iteratore, è semplice aggiungere un comportamento iteratore alle proprie classi, basta definire un metodo `__iter__()` che restituisca un oggetto con un metodo `next()`. Se la classe definisce `next()`, `__iter__()` può restituire solo `self`:

```
>>> class Reverse:
    "Iteratore per eseguire un ciclo al contrario su una sequenza"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for carattere in Reverse('spam'):
print carattere

m
a
p
s
```

9.10 Generatori

I generatori sono semplici ma efficaci strumenti per creare iteratori. Sono scritti come funzioni regolari, pur usando l'istruzione `yield` ogni qualvolta restituiscano dei dati. Siccome ricorda tutti i valori dei dati e l'ultima espressione eseguita, alla chiamata a `next()` il generatore riprende da dove s'era fermato. Di seguito si mostra come sia facile crearne uno:

```
>>> def reverse(data):
        for index in range(len(data)-1, -1, -1):
            yield data[index]

>>> for char in reverse('golf'):
        print char

f
l
o
g
```

Ciò che è fattibile con i generatori può essere fatto con iteratori basati su una classe, come visto nella precedente sezione. La creazione automatica dei metodi `__iter__()` e `next()` rende i generatori così compatti.

Un'altra caratteristica chiave è il salvataggio automatico delle variabili locali e dello stato d'esecuzione tra una chiamata e l'altra, cosa che rende la funzione più facile a scriversi e più chiara di un approccio con variabili di classe come `self.index` e `self.data`.

Oltre alla creazione automatica di metodi e salvataggio dello stato del programma, quando terminano, i generatori sollevano l'eccezione `StopIteration`. Insieme, queste caratteristiche facilitano la creazione di iteratori con la semplice scrittura di una normalissima funzione.

CAPITOLO

DIECI

Una breve escursione nella libreria standard

10.1 Interfaccia con il Sistema Operativo

Il modulo `os` fornisce dozzine di funzioni per interagire con il Sistema Operativo:

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()      # Restituisce la directory di lavoro corrente
'C:\\Python24'
>>> os.chdir('/server/accesslogs')
```

Ci si assicuri di usare, per l'importazione, lo stile '`import os`' invece di '`from os import *`'. Questo permetterà di distinguere `os.open()` dalla funzione `open()` che opera in modo del tutto differente.

Le funzioni built-in `dir()` ed `help()` sono utili come aiuto interattivo quando si lavora con moduli di grosse dimensioni come `os`:

```
>>> import os
>>> dir(os)
<restituisce una lista di tutti i moduli delle funzioni>
>>> help(os)
<restituisce una estesa pagina di manuale creata dalle docstring dei moduli>
```

Per sessioni amministrative giornaliere su file e directory, il modulo `shutil` fornisce un'interfaccia di alto livello semplice da usare:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

10.2 File wildcard

Il modulo `glob` fornisce funzioni per creare liste di file che provengono da directory sulle quali è stata effettuata una ricerca con caratteri wildcard (NdT: i caratteri jolly di espansione):

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Argomenti da riga di comando

Molti script spesso invocano argomenti da riga di comando richiedendo che siano processati. Questi argomenti vengono memorizzati nel modulo `sys` ed *argv* è un attributo, come fosse una lista. Per istanza il seguente output fornisce il risultato dall'esecuzione di '`python demo.py one two three`' da riga di comando:

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

Il modulo `getopt` processa *sys.argv* usando le convenzioni della funzione `getopt()` di UNIX. Viene fornita una più potente e flessibile riga di comando dal modulo `optparse`.

10.4 Redirigere gli errori in output e terminare il programma

Il modulo `sys` ha anche attributi per *stdin*, *stdout* e *stderr*. L'ultimo attributo è utile per mostrare messaggi di avvertimento ed errori e renderli visibili perfino quando *stdout* viene rediretto:

```
>>> sys.stderr.write('Attenzione, il file di log non c'è, ne viene\
...  creato uno nuovo')
Attenzione, il file di log non c'è, ne viene creato uno nuovo
```

Il modo più diretto per terminare l'esecuzione dello script è usare '`sys.exit()`'.

10.5 Modello di corrispondenza per le stringhe

Il modulo `re` fornisce gli strumenti per processare le stringhe con le espressioni regolari. Per lavorare su complesse corrispondenze e manipolazioni, le espressioni regolari offrono una succinta, ottimizzata soluzione:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Quando vengono richieste solamente semplici funzionalità, è preferibile usare i metodi delle stringhe perché sono più semplici da leggere e da correggere se dovessero contenere degli errori:

```
>>> 'the per doo'.replace('doo', 'due')
'the per due'
```

10.6 Matematica

Il modulo `math` fornisce l'accesso alle funzioni della sottostante libreria C per i numeri matematici in virgola mobile:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

il modulo `random` fornisce gli strumenti per creare selezioni di numeri casuali:

```
>>> import random
>>> random.choice(['mela', 'pera', 'banana'])
'mela'
>>> random.sample(xrange(100), 10)   # campione senza rimpiazzamento
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # numero in virgola mobile casuale
0.17970987693706186
>>> random.randrange(6)    # intero casuale scelto da range(6)
4
```

10.7 Accesso ad internet

Vi sono numerosi moduli per accedere ad internet e processarne i protocolli. Due moduli semplici sono `urllib2` per recuperare dati da url e `smtplib` per spedire mail:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
... if 'EST' in line:      # vedete Eastern Standard Time
...     print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@tmp.org', 'jceasar@tmp.org',
"""To: jceasar@tmp.org
From: soothsayer@tmp.org

Beware the Ides of March.
""")
>>> server.quit()
```

10.8 Data e tempo

Il modulo `datetime` fornisce classi per manipolare data e tempo in modo semplice ma contemporaneamente complesso. Sebbene la data ed il tempo aritmetico siano supportati, il nucleo dell'implementazione è basato sull'efficienza di estrapolazione dei membri per la formattazione e manipolazione dell'output. Il modulo supporta anche oggetti che abbiano consapevolezza della propria time zone.

```
# le date vengono facilmente costruite e formattate
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y or %d%b %Y is a %A on the %d day of %B")
'12-02-03 or 02Dec 2003 is a Tuesday on the 02 day of December'

# le date supportano i calendari aritmetici
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Compressione dei dati

I comuni sistemi di archiviazione di dati ed i relativi formati di compressione vengono supportati direttamente dai moduli ed includono: `zlib`, `gzip`, `bz2`, `zipfile` e `tarfile`.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(t)
-1438085031
```

10.10 Misura delle prestazioni

Alcuni sviluppatori ed utenti di Python hanno il profondo interesse di conoscere le relative differenze di prestazioni tra due approcci di un identico problema. Python fornisce strumenti di misurazione che rispondono immediatamente a queste richieste.

Per esempio, potreste essere tentati di usare la funzionalità di impacchettamento e spacchettamento delle tuple invece del tradizionale approccio consistente nello scambiare gli argomenti. Il modulo `timeit` dimostra velocemente il modesto incremento di prestazioni:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrasto al sottile livello di granularità di `timeit`, i moduli `profile` e `pstats` forniscono strumenti per identificare sezioni critiche per la tempistica in grossi blocchi di codice.

10.11 Controllo di qualità

Un approccio per sviluppare un programma di alta qualità è scrivere dei test per ogni funzione mentre viene sviluppata ed eseguire quindi i suddetti test frequentemente durante il processo di sviluppo.

Il modulo `doctest` fornisce uno strumento per analizzare e validare test inseriti nelle docstring del programma. La costruzione dei test è semplice come il copia-ed-incolla di una tipica chiamata, compresi i risultati, inserita all'interno della docstring. Questo migliora la documentazione fornita all'utente con un esempio e permetterà al modulo doctest di occuparsi di mantenere la documentazione aggiornata con il codice:

```
def media(valori):
    """Computa la media aritmetica di una lista di numeri.

    >>> print media([20, 30, 70])
    40.0
    """
    return sum(valori, 0.0) / len(valori)

import doctest
doctest.testmod()   # automaticamente valida i test nella docstring
```

Il modulo `unittest` richiede uno sforzo maggiore di comprensione rispetto al modulo `doctest`, ma mette a disposizione un più nutrito insieme di test che potranno essere mantenuti in un file separato.

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_media(self):
        self.assertEqual(media([20, 30, 70]), 40.0)
        self.assertEqual(round(media([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, media, [])
        self.assertRaises(TypeError, media, 20, 30, 70)

unittest.main() # La chiamata da riga di comando invoca tutti i test
```

10.12 Le batterie sono incluse

Python adotta la filosofia "le batterie sono incluse". Questo è il miglior modo di rappresentare le sofisticate e robuste capacità dei suoi generosi package. Per esempio:

* I moduli `xmlrpclib` e `SimpleXMLRPCServer` creano un'implementazione di una chiamata di una procedura remota all'interno di una quasi sempre insignificante sessione. Malgrado i nomi, non è necessario conoscere o saper gestire XML.

* Il package `email` è la libreria per gestire i messaggi email, include MIME ed altri messaggi di documenti basati sulla RFC 2822. Diversamente da `smtplib` e `poplib`, che attualmente spediscono e ricevono messaggi, questo package email ha un completo insieme di strumenti per realizzare o decodificare complesse strutture di messaggi (inclusi gli allegati) e implementa le codifiche internet ed i protocolli delle intestazioni.

I package `xml.dom` e `xml.sax` forniscono un robusto supporto per l'analisi di questo popolare formato di interscambio. Allo stesso modo il modulo `csv` supporta direttamente la lettura e la scrittura in un comune formato database. Insieme, questi moduli ed i package, semplificano magnificamente l'interscambio dei dati tra applicazioni scritte in Python ed altri strumenti.

* L'internazionalizzazione viene suportata da numerosi moduli, inclusi `gettext`, `locale` e dal package `codecs`.

CAPITOLO

UNDICI

E adesso?

Si spera che la lettura di questo tutorial abbia rafforzato il vostro interesse per Python – dovreste essere desiderosi di utilizzare Python per risolvere i vostri problemi reali. E adesso cosa dovreste fare?

Dovreste leggere, o almeno scorrere, la *Libreria di riferimento di Python*, 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 la Libreria di riferimento 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 funzione 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 vi rende disponibile del software da scaricare. Molti altri moduli Python, di utenti-realizzatori possono essere reperiti presso Python Package Index (PyPI).

Per porre questioni correlate a Python e riportare problemi, si può inviare un messaggio al newsgroup comp.lang.python, (NdT: per l'Italia it.comp.lang.python) o un'email alla lista di discussione presso python-list@python.org (NdT: per l'Italia python@zonapython.it). Il newsgroup e la lista di discussione sono collegate, quindi messaggi inviati all'uno saranno inoltrati automaticamente all'altro (NdT: Questo non è vero per l'Italia). 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 nelle FAQ (domande poste più frequentemente) o nella directory 'Misc/' della distribuzione sorgente di Python. Gli archivi della lista di discussione sono consultabili presso http://www.python.org/pipermail/. Nelle FAQ sono contenute le risposte a molte domande che vengono riproposte di frequente, potrebbe contenere anche la soluzione al vostro problema.

APPENDICE

A

Editing interattivo dell'input e sostituzione dallo storico

Alcune versioni dell'interprete Python supportano l'editing della riga di input corrente e la sostituzione dallo storico, in modo simile ai servizi che si trovano nelle shell Korn e Bash GNU. Il tutto è implementato attraverso la libreria *GNU Readline*, che supporta l'editing in stile Emacs e vi. Tale libreria è dotata di documentazione propria che non si desidera duplicare qui; comunque i concetti base si possono chiarire facilmente. L'editing interattivo e lo storico qui descritti sono disponibili quali opzioni sulle versioni UNIX e CygWin dell'interprete.

Questo capitolo *non* documenta i servizi di editing del package PythonWin di Mark Hammond o l'ambiente IDLE, basato su Tk e distribuito con Python. Il richiamo dello storico da riga di comando che funziona nelle finestre DOS e su NT, in alcune altre versioni DOS e Windows è un'altra cosa ancora.

A.1 Editing di riga

Ove supportato, l'editing della riga di input è attivo ogniqualvolta l'interprete stampa un prompt primario o secondario. La riga corrente può essere modificata usando i caratteri di controllo convenzionali di Emacs. I più importanti sono: `C-A` (Control-A) muove il cursore all'inizio della riga, `C-E` alla fine, `C-B` lo muove di una posizione a sinistra, `C-F` a destra. Indietro ('backspace') cancella il carattere alla sinistra del cursore, `C-D` il carattere a destra. `C-K` elimina ('kill') il resto della riga alla destra del cursore,`C-Y` rimette al suo posto ('yanks back') l'ultima stringa cancellata. `C-trattino basso` annulla l'ultima modifica effettuata; può essere ripetuto per ottenere un effetto cumulativo.

A.2 Sostituzione dallo storico

La sostituzione dallo storico funziona nel modo seguente. Il contenuto delle righe di input non vuote viene salvato in un buffer dello storico, quando viene dato un nuovo prompt si viene posizionati su una nuova riga in fondo a tale buffer. `C-P` si muove di una riga verso l'alto (indietro) nel buffer dello storico, `C-N` si muove una riga verso il basso. Qualsiasi riga presente nel buffer dello storico può essere modificata; davanti al prompt appare un asterisco per marcare la riga come modificata. Premendo il tasto `Invio` si passa la riga corrente all'interprete. `C-R` inizia una ricerca incrementale inversa; `C-S` inizia una ricerca in avanti.

A.3 Associazioni dei tasti

Le associazione dei tasti e alcuni altri parametri della libreria *GNU Readline* possono essere personalizzati inserendo i comandi in un file di inizializzazione chiamato '~/.inputrc'. Le associazioni dei tasti hanno questa forma:

```
tasto: nome-funzione
```

o

```
"stringa": nome-funzione
```

e le opzioni possono essere configurate con

```
set nome-opzione valore
```

Per esempio:

```
# Prefererisco l'editing in stile vi:
set editing-mode vi

# Usa una singola riga per l'editing:
set horizontal-scroll-mode On

# Riassocia alcune combinazioni di tasti:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Si noti che l'associazione predefinita di `Tab` in Python è l'inserimento di uno spazio di `Tab`Tabulazione, invece quello di Readline è la funzione di completamento dei nomi di file. Se lo si desidera davvero, si può sostituire tale funzione a quella predefinita con:

```
Tab: complete
```

nel proprio '~/.inputrc'. Naturalmente ciò renderà difficoltoso introdurre le righe di continuazione indentate se vi abituerete ad usare `Tab` a quello scopo.

Il completamento automatico dei nomi di variabile e modulo è disponibile come opzione. Per abilitarla nella modalità interattiva dell'interprete si aggiunga quanto segue al proprio file di avvio:[1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Questo associa il tasto `Tab` alla funzione di completamento, così premendo due volte il tasto `Tab` verranno suggeriti i completamenti, prendendo in esame le istruzioni Python, le variabili locali correnti e i nomi dei moduli disponibili. Per espressioni con punto come `string.a`, verrà valutata l'espressione fino al punto '`.`' finale e quindi suggeriti i completamenti possibili dagli attributi dell'oggetto risultante. Si noti che potrebbe essere eseguito codice dell'applicazione ove nell'espressione comparisse un oggetto con un metodo `__getattr__()`.

Ecco come potrebbe essere un file d'avvio più efficace. Si noti che esso cancella i nomi creati non appena non siano più necessari; la ragione è che il file d'avvio viene eseguito nello stesso spazio dei nomi dei comandi interattivi e la rimozione dei nomi evita effetti secondari negli ambienti interattivi. Può essere conveniente conservare alcuni moduli importati, come `os`, che torna utile nella maggior parte delle sessioni con l'interprete.

[1] Python eseguirà i contenuti di un file identificato dalla variabile di ambiente PYTHONSTARTUP quando si avvierà una sessione interattiva dell'interprete.

```
#  S’aggiunga al proprio interprete interattivo Python
#+ l’autocompletamento e una cronologia dei comandi. Occorre minimo
#+ Python 2.0 e la libreria GNU Readline.  Il tasto Esc per
#+ definizione opera l’autocompletamento (per cambiarlo, vedete la
#+ documentazione di readline).
#
#  Si copi il file in ~/.pystartup e s’imposti una variabile
#+ d’ambiente che punti ad esso:
#+ "export PYTHONSTARTUP=/max/home/itamar/.pystartup" in bash.
#
#  Si noti che PYTHONSTARTUP *non* espande "~": bisogna specificarlo
#+ nel percorso completo della propria cartella home.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

A.4 Commenti

Questo servizio è un enorme passo avanti rispetto alle vecchie versioni dell’interprete. Comunque restano da soddisfare alcuni desideri: sarebbe bello se sulle righe di continuazione venisse suggerita l’indentazione opportuna (il parser sa se è richiesta l’indentazione del prossimo token). Il meccanismo di completamento potrebbe usare la tabella dei simboli dell’interprete. Sarebbe anche utile un comando per controllare (o anche suggerire) parentesi e apici combacianti e altro ancora.

APPENDICE

B

La parte aritmetica in virgola mobile: problemi e limiti

Nella parte meccanica dei computer i numeri in virgola mobile si presentano come frazioni in base 2 (binarie): per esempio, il numero razionale in base 10

```
0.125
```

ha valore 1/10 + 2/100 + 5/1000 e parimenti il numero razionale in base 2

```
0.001
```

ha valore 0/2 + 0/4 + 1/8. Entrambi hanno valore identico, la sola differenza è la base del loro sistema di numerazione. Il primo scritto in base 10 ed il secondo in base 2.

Purtroppo, la maggior parte dei numeri razionali decimali non può essere tradotta in base binaria, con la conseguenza che, in genere, i decimali a virgola mobile inseriti sono solo approssimati nei numeri binari in virgola mobile inclusi nella macchina.

Si può comprendere meglio il problema considerando prima la base 10; si prenda la frazione 1/3: essa può essere approssimata nel numero razionale decimale:

```
0.3
```

o, meglio,

```
0.33
```

o, meglio,

```
0.333
```

e così via. Non si arriverà mai al risultato esatto di 1/3, ma, per ogni cifra aggiuntiva, ogni approssimazione sarà progressivamente migliore.

Allo stesso modo, non si riuscirà a rappresentare esattamente il decimale 0,1 traducendolo in base 2, dacché se ne ottiene il numero periodico infinito

```
0.0001100110011001100110011001100110011001100110011001100110011...
```

L'approssimazione si ottiene fermandosi a qualunque numero finito di cifre e per questo si trovano espressioni come:

```
>>> 0.1
0.10000000000000001
```

Questo si otterrebbe solo sulla maggior parte delle macchine se si scrivesse 0,1 al prompt di Python – non sempre, perché il numero di bit usati dalle componenti fisiche, per immagazzinare valori in virgola mobile, può variare da una macchina all'altra. Inoltre, Python visualizza un'approssimazione decimale del vero valore del numero binario immagazzinato dalla macchina. Per lo più, se dovesse visualizzare il vero valore decimale dell'approssimazione binaria di `0,1`, dovrebbe far apparire, invece:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Per ottenere una versione sotto forma di stringa di ciò che fa apparire, il prompt di Python utilizza (implicitamente) la funzione built-in `repr()`; con virgola mobile `repr(`*float*`)`, arrotonda il vero valore decimale a 17 cifre significative:

```
0.10000000000000001
```

`repr(`*float*`)` produce 17 cifre, significative in quanto sufficienti (sulla maggior parte delle macchine) per far sì che `eval(repr(`*x*`)) ==` *x* sia valido per tutte le variabili finite di *x* – questo non avverrebbe, arrotondando a 16 cifre.

Si noti che questo non è un difetto di Python, né del proprio codice, ma risponde alla vera natura dei binari in virgola mobile: anzi, si ripresenta con tutti i linguaggi che supportino l'aritmetica in virgola mobile dei componenti elettronici (anche se taluni non *visualizzano* in modo predefinito la differenza, o, almeno, in tutte le modalità).

La funzione built-in `str()` produce solo 12 cifre significative, ma la si potrebbe preferire: è raro far riprodurre *x* a `eval(str(`*x*`))`; il risultato potrebbe essere più gradevole nella forma:

```
>>> print str(0.1)
0.1
```

Si badi: propriamente, questa è un'illusione; il vero valore di macchina non è esattamente `1/10`, ma, semplicemente, lo si arrotonda per la *visualizzazione*.

E le sorprese non finiscono qui: per esempio, vedendo

```
>>> 0.1
0.10000000000000001
```

si potrebbe essere tentati di usare la funzione `round()` per ridurre tutto a una singola cifra, ma ciò non apporta alcun cambiamento:

```
>>> round(0.1, 1)
0.10000000000000001
```

Il problema è che il valore binario in virgola mobile immagazzinato per 0,1 è già la migliore approssimazione in base 2 di 1/10 e va già bene così, inutile cercare di arrotondarla.

Un'altra conseguenza: poiché la traduzione di 0,1 non corrisponde esattamente a 1/10, sommare 0,1 per dieci volte non dà esattamente 1.0:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.99999999999999989
```

Sorprese come questa sono numerose nell'aritmetica binaria in virgola mobile. Sul problema con 0,1 cfr. si veda la spiegazione in dettaglio, nella sezione '`Errore di rappresentazione`' e *I rischi della virgola mobile*, resoconto completo sulle altre frequenti singolarità.

Alla fin fine, "non ci sono risposte semplici": ciononostante, inutile preoccuparsi troppo! Gli errori nelle operazioni in virgola mobile con Python sono dovuti a impostazioni strutturali della parte meccanica e, per la maggioranza delle macchine, hanno una frequenza di circa 1 su 2**53 – più che adeguata, dunque. Bisogna ricordare, però, che non è artimetica decimale, per cui ogni operazione in virgola mobile può soffrire di errori di arrotondamento.

Quindi: casi patologici esistono, ma negli usi più comuni dell'aritmetica in virgola mobile si otterranno i risultati desiderati arrotondando semplicemente la visualizzazione del risultato finale a un certo numero di cifre decimali. Di solito basta `str()`, ma, per un controllo più preciso, cfr. la discussione sull'operatore di formato % di Python: i codici di formato `%g`, `%f` ed `%e` offrono sistemi semplici e flessibili per visualizzare risultati arrotondati.

B.1 Errore di rappresentazione

Questa sezione spiega in dettaglio l'esempio di "0,1" e mostra come condurre da sé un'analisi esatta di casi analoghi. Si dà per scontata una certa familiarità con la rappresentazione in virgola mobile in base 2.

L'*errore di rappresentazione* si riferisce all'impossibilità di rappresentare esattamente come numeri razionali binari la maggior parte (almeno per ora) dei numeri razionali decimali. Questa è la ragione principale per cui Python (come Perl, C, C++, Java, Fortran e molti altri) non visualizza il numero che ci si aspetterebbe:

```
>>> 0.1
0.10000000000000001
```

Qual è il motivo? 1/10 non è esattamente traducibile in base 2. Oggi (novembre 2000) quasi tutte le macchine usano un'aritmetica in virgola mobile di tipo IEEE-754 e quasi tutte le piattaforme riportano i numeri in virgola mobile di Python alla doppia precisione di tipo IEEE-754. 754 doppi contengono 53 bit di precisione, per cui a comando il computer tende a convertire 0,1 al numero razionale binario più vicino alla forma *J*/2***N*, dove *J* è un intero contenente esattamente 53 bit. Riscrivendo

```
 1 / 10 ~= J / (2**N)
```

nella forma

```
J ~= 2**N / 10
```

e tenendo sempre presente che *J* ha 53 bit (quindi è `>= 2**52` ma `< 2**53`), il miglior valore di trascrizione per *N* sarà 56:

```
>>> 2L**52
4503599627370496L
>>> 2L**53
9007199254740992L
>>> 2L**56/10
7205759403792793L
```

In altre parole, 56 è il solo valore di *N* ad ammettere un *J* con 53 bit, il cui miglior valore possibile sarà, quindi, il quoziente arrotondato:

```
>>> q, r = divmod(2L**56, 10)
>>> r
6L
```

Dal momento che il resto è maggiore di 10/2, la migliore approssimazione si otterrà arrotondando per eccesso:

```
>>> q+1
7205759403792794L
```

Perciò, la migliore approssimazione di 1/10 nella doppia precisione di tipo 754 è quella maggiore di 2**56, ovvero

```
7205759403792794 / 72057594037927936
```

Si badi: siccome abbiamo arrotondato per eccesso, questo valore è effettivamente un po' più grande di 1/10; in caso contrario sarebbe stato leggermente inferiore, ma mai in ogni caso *uguale* a 1/10!

In tal modo, il computer non "vede" mai 1/10, bensì il numero razionale che sia la migliore approssimazione possibile di tipo 754:

```
>>> .1 * 2L**56
7205759403792794.0
```

Moltiplicando tale numero razionale per 10**30, possiamo vedere il valore (tronco) delle sue 30 cifre decimali più significative:

```
>>> 7205759403792794L * 10L**30 / 2L**56
100000000000000005551115123125L
```

Ciò significa che il numero esatto memorizzato nel computer è uguale approssimativamente al valore decimale 0.100000000000000005551115123125, che, arrotondato alle sue 17 cifre significative, dà il famoso 0.10000000000000001 visualizzato da Python (o meglio, visualizzato su piattaforme conformi al tipo 754 e capaci, nella propria libreria di C, della migliore conversione possibile di dati in entrata e in uscita – non tutte lo sono!)

APPENDICE

C

Storia e licenza

C.1 Storia del software

Python è stato creato agli inizi degli anni 90 da Guido van Rossum al Centro di Matematica di Stichting (CWI, si veda http://www.cwi.nl/) nei Paesi Bassi, come successore di un linguaggio chiamato ABC. Guido rimane l'autore principale di Python, anche se molti altri vi hanno contribuito.

Nel 1995, Guido ha continuato il suo lavoro su Python presso il Centro Nazionale di Ricerca (CNRI, si veda http://www.cnri.reston.va.us/) in Reston, Virginia, dove ha rilasciato parecchie versioni del software.

Nel maggio 2000, Guido e la squadra di sviluppo di Python si spostano in BeOpen.com per formare la squadra PythonLabs di BeOpen. Nell'ottobre dello stesso anno, la squadra di PythonLabs diventa la Digital Creations (ora Zope Corporation; si veda http://www.zope.com/). Nel 2001, viene fondata la Python Software Foundation (PSF, si veda http://www.python.org/psf/), un'organizzazione senza scopo di lucro creata specificamente per detenere la propria proprietà intellettuale di Python. Zope Corporation è un membro sponsorizzato dalla PSF.

Tutti i rilasci di Python sono Open Source (si veda http://www.opensource.org/ per la definizione di "Open Source"). Storicamente la maggior parte, ma non tutti i rilasci di Python, sono stati GPL-compatibili; la tabella seguente ricapitola i vari rilasci.

Versione	Derivata da	Anno	Proprietario	GPL compatibile?
0.9.0 thru 1.2	n/a	1991-1995	CWI	si
1.3 thru 1.5.2	1.2	1995-1999	CNRI	si
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	si
2.1.1	2.1+2.0.1	2001	PSF	si
2.2	2.1.1	2001	PSF	si
2.1.2	2.1.1	2002	PSF	si
2.1.3	2.1.2	2002	PSF	si
2.2.1	2.2	2002	PSF	si
2.2.2	2.2.1	2002	PSF	si
2.2.3	2.2.2	2002-2003	PSF	si
2.3	2.2.2	2002-2003	PSF	si
2.3.1	2.3	2002-2003	PSF	si
2.3.2	2.3.1	2003	PSF	si
2.3.3	2.3.2	2003	PSF	si
2.3.4	2.3.3	2004	PSF	si

Note: GPL-compatibile non significa che viene distribuito Python secondo i termini della GPL. Tutte le licenze di Python, diversamente dalla GPL, permettono di distribuire una versione modificata senza rendere i vostri cambiamenti open source. Le licenze GPL-compatibili permettono di combinare Python con altro software rilasciato sotto licenza GPL; le altre no.

Un ringraziamento ai tanti volontari che, lavorando sotto la direzione di Guido, hanno reso possibile permettere questi rilasci.

C.2 Termini e condizioni per l'accesso o altri usi di Python (licenza d'uso, volutamente non tradotta)

PSF LICENSE AGREEMENT FOR PYTHON 2.3.4

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.3.4 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.3.4 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2004 Python Software Foundation; All Rights Reserved" are retained in Python 2.3.4 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.3.4 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.3.4.

4. PSF is making Python 2.3.4 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.3.4 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3.4 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3.4, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.3.4, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT 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 LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to

create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the “ACCEPT” button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

C.3 Licenze e riconoscimenti per i programmi incorporati

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.keio.ac.jp/ matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
```

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ‘‘AS IS’’ AND
GAI_ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR GAI_ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON GAI_ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI_ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
```

C.3.5 rotor – Enigma-like encryption and decryption

The source code for the `rotor` contains the following notice:

```
Copyright 1994 by Lance Ellinghouse,
Cathedral City, California Republic, United States of America.

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.

LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE BE LIABLE FOR ANY SPECIAL,
INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING
FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.7 Cookie management

The `Cookie` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.8 Profiling

The `profile` and `pstats` modules contain the following notice:

```
Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software
and its associated documentation for any purpose (subject to the
restriction in the following sentence) without fee is hereby granted,
provided that the above copyright notice appears in all copies, and
that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of InfoSeek not be used in
advertising or publicity pertaining to distribution of the software
without specific, written prior permission.  This permission is
explicitly restricted to the copying and modification of the software
to remain in Python, compiled Python, or other languages (such as C)
wherein the modified or derived code is exclusively imported into a
Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY
SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER
RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.9 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.10 UUencode and UUdecode functions

The uu module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with python standard
```

C.3.11 XML Remote Procedure Calls

The xmlrpclib module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

APPENDICE

D

Glossario

>>> Il tipico prompt di Python della shell interattiva. Spesso si vedono esempi di codice che possono mostrare il codice partendo dalla destra dell'interprete.

... Il tipico prompt di Python della shell interattiva quando il codice immesso riguarda blocchi di codice indentato.

BDFL Il "benevolo dittatore per la vita", ovvero Guido van Rossum, il creatore di Python.

byte code La rappresentazione interna di un programma Python vista dall'interprete. Il byte code è anche memorizzato in file `.pyc` e `.pyo` così che l'esecuzione dello stesso file sia più veloce in un secondo tempo (la compilazione del sorgente in byte code può essere salvata). Questo "linguaggio intermedio" viene eseguito su di una "macchina virtuale" che chiama delle corrispondenti subroutine per ogni bytecode.

classe classic Una classe che non eredita da `object`. Vedete le *classi new-style* (NdT: *classi di nuovo stile*).

coercizione L'implicita conversione di un'istanza di un tipo in un'altro, durante un'operazione che coinvolge due argomenti dello stesso tipo. Per esempio, `int(3.15)` converte il numero in virgola mobile in un numero intero, `3`; ma in `3+4.5` ogni argomento è di un tipo differente (uno intero ed uno in virgola mobile) ed entrambi dovranno essere convertiti nello stesso tipo prima che siano sommati o che venga sollevata l'eccezione `TypeError`. La coercizione tra due operandi può essere realizzata con la funzione built-in `coerce`; quindi `3+4.5` è equivalente a chiamare `operator.add(*coerce(3, 4.5))` ed al risultato di `operator.add(3.0, 4.5)`. Senza la coercizione, tutti gli argomenti, anche di tipo compatibile, dovrebbero essere normalizzati allo stesso valore dal programmatore, per esempio `float(3)+4.5` piuttosto che `3+4.5`.

numeri complessi Un'estensione del familiare sistema dei numeri reali in cui tutti i numeri vengono espressi come la somma di una parte reale ed una immaginaria. I numeri immaginari sono reali multipli dell'unità immaginaria (la radice quadrata di `-1`), spesso scritta `i` in matematica o `j` in ingegneria. Python ha il supporto built-in per i numeri complessi, che vengono scritti con quest'ultima notazione; la parte immaginaria viene scritta con il suffisso `j`, ad esempio `3+1j`. Per avere accesso alle complesse equivalenza del modulo `math`, usate `cmath`. L'uso dei numeri complessi è una funzionalità matematica abbastanza avanzata. Se non ne siete consapevoli o non avete necessità di conoscere l'argomento, potrete quasi certamente ignorarli.

descrittore Un oggetto di *nuovo stile* che definisce i metodi `__get__()`, `__set__()` o `__delete__()`. Quando un attributo di classe è un descrittore, esso è uno speciale vincolo comportamentale innescato su di un attributo di consultazione del dizionario. Normalmente, scrive *a.b* per la consultazione su di un oggetto *b* nella classe dizionario per *a*, ma se *b* è un descrittore, viene chiamato il metodo definito. Per capirsi, un descrittore è una chiave di lettura che porta ad una profonda comprensione di Python poiché è la base per molte funzionalità, incluse funzioni, metodi, proprietà, metodi di classe, metodi statici e riferimenti a super classi.

dizionario Un array associativo, dove chiavi arbitrarie vengono mappate in valori. L'uso di `dict` somiglia molto a quello per le liste `list`, ma le chiavi possono essere ogni oggetto con una funzione `__hash__()`, non giusto interi che iniziano da zero. Chiamata hash in Perl.

EAFP È più facile domandare perdono che chiedere il permesso. Questo comune stile di scrivere codice Python assume l'esistenza di chiavi valide o attributi ed eccezioni catturate se si assumono come valide dimostrazioni che poi si dimostrano errate. Questo stile veloce e pulito viene caratterizzato dalla presenza di molte

istruzioni `try` ed `except`. La tecnica contrasta con lo stile *LBYL* che è invece comune in molti altri linguaggi come il C.

__future__ Uno pseudo modulo che i programmatori usano per abilitare nuove funzionalità del linguaggio che non sono compatibili con il corrente interprete. Per esempio l'espressione `11/4` correntemente viene valutata come `2`. Se il modulo in cui viene eseguito ha abilitato *true division* per mezzo dell'esecuzione di:

```
from __future__ import division
```

l'espressione `11/4` verrà valutata come `2.75`. Attraverso l'importazione del modulo `__future__` e la valutazione delle variabili, potrete notare come la nuova funzionalità viene prima aggiunta al linguaggio e quindi diventa predefinita:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

generatore Una funzione che restituisce un iteratore. Viene vista come una normale funzione, eccetto per il fatto che valuta e ritorna al chiamante usando un'istruzione `yield` invece di un'istruzione `return`. La funzione generatore spesso contiene uno o più cicli `for` o `while` che restituiscono al chiamante elementi `yield`. La funzione viene eseguita e fermata dalla parola chiave `yield` (che restituisce il risultato) e viene ripresa da quel punto quando il prossimo elemento viene richiesto da una chiamata del metodo `next()` restituito dall'iteratore.

GIL Vedete *lock globale dell'interprete*.

lock globale dell'interprete Il lock (NdT: blocco) usato da Python nei thread garantisce che solamente un thread possa essere in esecuzione nello stesso momento. Questa semplificazione in Python assicura che non vi siano due processi che accedono contemporaneamente alla medesima area di memoria. L'intero blocco dell'interprete fa sì che sia semplice per l'interprete creare implementazioni multi-thread, al costo di alcuni parallelismi su macchine multiprocessore. In passato è stato compiuto uno sforzo per creare un interprete "free-threaded" (NdT: libero di usare i thread) (un blocco dei dati condivisi con sottile granularità), ma che soffre di prestazioni nei casi comuni dove c'è un singolo processore.

IDLE Un ambiente di sviluppo integrato per Python. IDLE è un editor basilare ed un interprete d'ambiente che viene fornito con la distribuzione standard di Python. Ottimo per i novizi, è anche utile come esempio di codice per coloro che ricercano un'implementazione di un'applicazione GUI moderatamente sofisticata e multi piattaforma.

immutabile Un oggetto con un valore fissato. Oggetti immutabili sono numeri, stringhe o tuple (ed altri). Simili ad un oggetto che non ha subito alterazioni. Deve essere creato un nuovo oggetto se si vuole memorizzare un valore differente. Giocano un ruolo importante in posti dove un valore costante di hash è necessario. Per esempio come nelle chiavi di un dizionario.

divisione di interi La divisione matematica ha rinunciato a qualsiasi resto. Per esempio l'espressione `11/4` viene correntemente valutata come `2`, in contrasto al `2.75` restituito dalla divisione con numeri in virgola mobile; anche chiamata *floor division*. Quando vengono divisi due interi il risultato sarà sempre un altro intero (a meno che non venga adottata la divisione floor). Comunque, se uno degli operandi è un altro numero di tipo diverso (simile al `float`), al risultato verrà applicata una coercizione (vedete *coercizione*) al tipo comune. Per esempio, un intero diviso un numero in virgola mobile, fornirà un risultato in virgola mobile, possibilmente con frazioni decimali. La divisione di interi può essere forzata usando l'operatore `//` invece dell'operatore `/`. Vedete anche *__future__*.

interattivo Phython ha un interprete interattivo, che significa che ai vostri tentativi digitati e visti sullo schermo seguiranno direttamente i risultati. Questo accadrà non appena avrete lanciato l'esecuzione di `python` senza argomenti (possibilmente selezionandolo sul vostro computer dal menu principale). Questo è un modo veramente pratico e potente per verificare ogni nuova idea o ispezionare moduli e package (ricordatevi `help(x)`).

interpretato Python è un linguaggio interpretato, in opposizione a quelli compilati. Questo significa che i file possono essere eseguiti direttamente, senza prima creare un eseguibile che debba essere lanciato. I linguaggi interpretati tipicamente hanno un breve ciclo di sviluppo/correzione rispetto ai compilati, sebbene questi programmi siano un po' più lenti nell'esecuzione. Vedete anche *interattivo*.

iterabile Un oggetto contenitore capace di restituirvi un membro per volta. Esempi di oggetti iterabili includono i tipi sequenza (simili a `list`, `str` e `tuple`) ed alcuni tipi non-sequenza come `dict`, `file` ed oggetti di ogni classe definita con un metodo `__iter__()` o `__getitem__()`. Gli iterabili possono essere usati in cicli `for` ed in molti altri posti dove è necessaria una sequenza (`zip()`, `map()`, ...). Quando un oggetto iterabile viene passato ad un argomento di una funzione built-in `iter()`, restituisce un iteratore per quest'oggetto. Questo iteratore è ottimo per passare insiemi di valori. Quando usate oggetti iterabili, solitamente non è necessaria una chiamata ad `iter()` od occuparsi con un iteratore del vostro stesso oggetto. L'istruzione `for` lo farà automaticamente per voi, creando una variabile temporanea senza nome che manterrà l'iteratore per la durata del ciclo. Vedete anche *iteratori*, *sequenze* e *generatori*.

iteratore Un oggetto rappresenta un flusso di dati. Ripetute chiamate al metodo `next()` di un iteratore restituiranno elementi successivi nel flusso. Invece, quando non vi saranno più dati disponibili verrà sollevata un'eccezione `StopIteration`. A questo punto l'oggetto iteratore ha esaurito il proprio compito ed ogni ulteriore chiamata al metodo `next()` giustamente solleverà nuovamente un'eccezione `StopIteration`. Gli iteratori richiedono che il metodo `__iter__()` restituisca un oggetto iteratore, perciò ciascun iteratore è anche iterabile e può essere usato in molti posti dove altri iterabili vengono accettati. Una notevole eccezione è codice che tenti molteplici passaggi di iterazioni. Un oggetto contenitore (simile a `list`) produce un nuovo iteratore ogni volta che viene passato alla funzione `iter()` o usato in un ciclo `for`. Facendo questi tentativi con un iteratore, verrà retituito proprio lo stesso esausto oggetto iteratore del secondo passaggio dell'iterazione, creando all'apparenza un contenitore vuoto.

costruzione di lista Un modo compatto per processare tutte od un insieme di elementi in una sequenza e restituire una lista con i risultati. `result = [0x%02x %x for x in range(256) if x %2 == 0]` genera una lista di stringhe contenenti i numeri esadecimali (0x..) che sono pari nell'intervallo da 0 a 255. La clausola `if` è facoltativa. Se omessa, tutti gli elementi nell'intervallo `range(256)` vengono, in questo caso, processati.

mappa Un oggetto contenitore (simile a `dict`) che supporta la consultazione con chiave arbitraria usando lo speciale metodo `__getitem__()`.

metaclasse La classe di una classe. La definizione di una classe creata, una classe con nome, una classe dizionario, un elenco delle classi di base. La metaclasse è responsabile per l'accettazione dei tre argomenti e la creazione della classe. Molti linguaggi di programmazione orientati agli oggetti forniscono come predefinita questa implementazione. Ciò che rende Python speciale è la possibilità di creare metaclassi personalizzate. Molti utenti non hanno la necessità di avere questo strumento, ma quando invece si presenta, le metaclassi forniscono un'ottima ed elegante soluzione. Possono essere usate per ottenere attributi di accesso al logging, aggiungere thread-safety, tracciare la creazione di oggetti, implementare singleton ed in molte altre sessioni.

LBYL Guardatelo prima di non considerarlo. Questo stile di scrittura di codice verifica delle pre-condizioni prima di creare chiamate a consultazioni. Questo stile contrasta con l'approccio *EAFP* e viene caratterizzato dalla presenza di parecchie istruzioni `if`.

mutabile Gli oggetti mutabili possono modificare i loro valori ma mantenere i loro `id()`. Vedete anche *immutabile*.

spazio dei nomi Il luogo dove vengono memorizzate le variabili. Lo spazio dei nomi viene implementato come un dizionario. Questo spazio dei nomi può essere locale, globale o built-in ed anche annidato in un oggetto (nei metodi). Lo spazio dei nomi supporta la modularità per mezzo di una preventiva risoluzione dei conflitti nei nomi. Per istanza, le funzioni `__builtin__.open()` e `os.open()` vengono distinte dai loro spazi dei nomi. Gli spazi dei nomi contribuiscono anche ad aumentare la leggibilità e la manutenibilità tramite una realizzazione chiara che viene implementata nei moduli delle funzioni. Per istanza, scrivere `random.seed()` o `itertools.izip()` sgombra dagli equivoci su quale sia lo spazio dei nomi delle funzioni implementato rispettivamente dai moduli `random` e `itertools`.

area annidata L'abilità di riferirsi a variabili in una definizione acclusa. Per istanza, una funzione definisce all'interno un'altra funzione che fa riferimento a variabili in un'altra funzione. Notate che l'area annidata

lavora solamente per riferimento e non per assegnamento, che verrà comunque scritto nell'area più interna. Per contro, le variabili locali leggono e scrivono nell'area più interna. Allo stesso modo, le variabili globali leggono e scrivono nello spazio dei nomi globale.

classe di nuovo stile Ogni classe che eredita da `object`. Questo include tutti i tipi built-in come `list` e `dict`. Solamente le classi di nuovo stile possono usare queste novità di Python, funzionalità versatili come `__slots__`, descrittori, proprietà, `__getattribute__()`, metodi di classe e metodi statici.

Python3000 Una mitica versione di Python, non consentirà la compatibilità con versioni più vecchie dell'interprete, o meglio, la consentirà con un'interfaccia telepatica. ;-)

`__slots__` Una dichiarazione all'interno di *classi di nuovo stile* che salva memoria tramite uno spazio dove viene inserita una pre-dichiarazione per istanze di attributi ed eliminazione di istanze di dizionari. Sebbene sia popolare, questa tecnica viene piuttosto usata per prendere il giusto spazio di memoria e riservarlo per i rari casi dove vi sono un grande numero di istanze in applicazioni critiche per la memoria.

sequenza Un *iterabile* che supporta efficientemente l'accesso ad elementi usando indici interi tramite gli speciali metodi `__getitem__()` e `__len__()`. Alcuni tipi di sequenze built-in sono `list`, `str`, `tuple` ed `unicode`. Notate che `dict` supporta anche `__getitem__()` e `__len__()`, ma è considerato una mappa piuttosto che una sequenza poiché la consultazione usa chiavi arbitrarie *immutabili* piuttosto che interi.

Lo Zen di Python L'elenco principale del disegno di progettazione e della filosofia che può essere utile alla comprensione ed all'uso del linguaggio. L'elenco può essere rinvenuto digitando al prompt interattivo "`import this`".

INDICE ANALITICO

R

S

U