Table of Contents

Dive Into Python **1**

Capitolo 1. Installare Python **2**
1.1. Qual è il Python giusto per te? 2
1.2. Python su Windows 2
1.3. Python su Mac OS X 3
1.4. Python su Mac OS 9 5
1.5. Python su RedHat Linux 5
1.6. Python su Debian GNU/Linux 6
1.7. Installare dai sorgenti 7
1.8. La shell interattiva 7
1.9. Sommario 8

Capitolo 2. Conoscere Python **9**
2.1. Immergersi 9
2.2. Dichiarare le funzioni 9
2.3. Documentare le funzioni 11
2.4. Tutto è un oggetto 11
2.5. Indentare il codice 13
2.6. Testare i moduli 13
2.7. Introduzione ai dizionari 14
2.8. Introduzione alle liste 17
2.9. Introduzione alle tuple 20
2.10. Definire le variabili 22
2.11. Assegnare valori multipli in un colpo solo 23
2.12. Formattare le stringhe 24
2.13. Mappare le liste 25
2.14. Concatenare liste e suddividere stringhe 26
2.15. Sommario 28

Capitolo 3. La potenza dell'introspezione **30**
3.1. Immergersi 30
3.2. Argomenti opzionali ed argomenti con nome 31
3.3. type, str, dir, ed altre funzioni built−in 32
3.4. Ottenere riferimenti agli oggetti usando getattr 35
3.5. Filtrare le liste 36
3.6. Le particolarità degli operatori and e or 37
3.7. Usare le funzioni lambda 39
3.8. Unire il tutto 41
3.9. Sommario 43

Capitolo 4. Una struttura orientata agli oggetti **45**
4.1. Immergersi 45
4.2. Importare i moduli usando from module import 47
4.3. Definire classi 48
4.4. Istanziare classi 51
4.5. UserDict: una classe wrapper 52
4.6. Metodi speciali per le classi 54
4.7. Metodi speciali di classe avanzati 57
4.8. Attributi di classe 58

Table of Contents

Capitolo 4. Una struttura orientata agli oggetti
4.9. Funzioni private....60
4.10. Gestire le eccezioni....61
4.11. Oggetti file....63
4.12. Cicli for....66
4.13. Ancora sui moduli....68
4.14. Il modulo os....70
4.15. Mettere tutto insieme....72
4.16. Sommario....73

Capitolo 5. Elaborare HTML....76
5.1. Immergersi....76
5.2. Introduciamo sgmllib.py....80
5.3. Estrarre informazioni da documenti HTML....82
5.4. Introdurre BaseHTMLProcessor.py....84
5.5. locals e globals....86
5.6. Formattazione di stringhe basata su dizionario....89
5.7. Virgolettare i valori degli attributi....90
5.8. Introduzione al modulo dialect.py....92
5.9. Introduzione alle espressioni regolari....94
5.10. Mettere tutto insieme....96
5.11. Sommario....98

Capitolo 6. Elaborare XML....100
6.1. Immergersi....100
6.2. Package....106
6.3. Analizzare XML....108
6.4. Unicode....110
6.5. Ricercare elementi....114
6.6. Accedere agli attributi di un elemento....116
6.7. Astrarre le sorgenti di ingresso....117
6.8. Standard input, output, ed error....121
6.9. Memorizzare i nodi e ricercarli....124
6.10. Trovare i figli diretti di un nodo....125
6.11. Create gestori separati per tipo di nodo....126
6.12. Gestire gli argomenti da riga di comando....127
6.13. Mettere tutto assieme....131
6.14. Sommario....132

Capitolo 7. Test delle unità di codice....134
7.1. Immergersi....134
7.2. Introduzione al modulo romantest.py....135
7.3. Verificare i casi di successo....138
7.4. Verificare i casi di errore....140
7.5. Verificare la consistenza....142
7.6. roman.py, fase 1....144
7.7. roman.py, fase 2....147
7.8. roman.py, fase 3....150
7.9. roman.py, fase 4....153
7.10. roman.py, fase 5....156

Table of Contents

Capitolo 7. Test delle unità di codice
7.11. Come gestire gli errori di programmazione....160
7.12. Gestire il cambiamento di requisiti....162
7.13. Rifattorizzazione....168
7.14. Postscritto....172
7.15. Sommario....174

Capitolo 8. Programmazione orientata ai dati....176
8.1. Immergersi....176
8.2. Trovare il percorso....177
8.3. Filtrare liste rivisitate....180
8.4. Rivisitazione della mappatura delle liste....181
8.5. Programmazione data−centrica....182
8.6. Importare dinamicamente i moduli....183
8.7. Mettere assieme il tutto (parte 1)....184
8.8. Dentro PyUnit....186

Appendice A. Ulteriori letture....187

Appendice B. Recensione in 5 minuti....191

Appendice C. Consigli e trucchi....202

Appendice D. Elenco degli esempi....210

Appendice E. Storia delle revisioni....219

Appendice F. Circa questo libro....220

Appendice G. GNU Free Documentation License....221
G.0. Preamble....221
G.1. Applicability and definitions....221
G.2. Verbatim copying....222
G.3. Copying in quantity....222
G.4. Modifications....223
G.5. Combining documents....224
G.6. Collections of documents....224
G.7. Aggregation with independent works....224
G.8. Translation....224
G.9. Termination....225
G.10. Future revisions of this license....225
G.11. How to use this License for your documents....225

Appendice H. Python 2.1.1 license....226
H.A. History of the software....226
H.B. Terms and conditions for accessing or otherwise using Python....226

Dive Into Python

30 gennaio 2003

Capitolo 1. Installare Python

1.1. Qual è il Python giusto per te?

Benvenuto in Python. Immergiamoci.

La prima cosa che dovete fare con Python è installarlo. O no? Se state utilizzando un account in un server in hosting, il vostro ISP potrebbe avere già installato Python. Le più importanti distribuzioni Linux installano Python in modo predefinito. Mac OS X 10.2 e successivi includono una versione a riga di comando di Python, tuttavia vorrete probabilmente installare una versione che include un'interfaccia grafica più consona a Mac.

Windows non installa in modo predefinito alcuna versione di Python. Ma non disperate! Ci sono diversi modi per puntare−e−cliccare il vostro Python su Windows.

Come potete già notare, Python gira su molti sistemi operativi. La lista completa comprende Windows, Mac OS, Mac OS X e tutte le varianti dei sistemi liberi compatibili con UNIX, come Linux. Vi sono anche versioni che girano su Sun Solaris AS/400, Amiga, OS/2, BeOS ed una vasta gamma di altre piattaforme che probabilmente non avrete mai sentito nominare.

Inoltre, programmi Python scritti su una piattaforma, con un minimo di accortezza, girano su *ogni* altra piattaforma supportata. Per esempio, sviluppo regolarmente programmi Python su Windows ed in seguito li rilascio su Linux.

Quindi tornando alla domanda che iniziò questa sezione: "qual è il Python giusto per te?" La risposta è "qualunque giri sul computer che avete".

1.2. Python su Windows

Su Windows, avete diverse alternative per installare Python.

ActiveState produce un'installazione Windows per Python che comprende una versione completa di Python, un IDE con un editor per il codice orientato a Python, più alcune estensioni di Windows per Python che consentono un accesso completo ai servizi specifici di Windows, alle sue API ed al suo registro.

ActivePython è liberamente scaricabile, tuttavia non è open source. È l'IDE che ho utilizzato per imparare Python, e vi raccomando di provarlo, a meno che non abbiate una ragione specifica per non farlo. (Una ragione potrebbe essere che ActiveState è generalmente di alcuni mesi indietro nell'aggiornamento del loro installatore ActivePython rispetto ai rilasci delle nuove versioni di Python. Se necessitate assolutamente dell'ultima versione di Python ed ActivePython è rimasta alla versione precedente, dovrete passare alla seconda opzione.)

Procedura 1.1. Opzione 1: Installare ActivePython

1. Scaricate ActivePython da http://www.activestate.com/Products/ActivePython/.
2. Se siete su Windows 95, Windows 98, o Windows ME, dovrete anche scaricare ed installare il Windows Installer 2.0 prima di installare ActivePython.
3. Doppio click sull'installer, `ActivePython−2.2.2−224−win32−ix86.msi`.
4. Seguite le istruzioni dell'installer.
5. Se lo spazio è poco, potete effettuare un'installazione personalizzata ed escludere la documentazione, ma non ve lo raccomando a meno che non disponiate davvero dei 14 megabyte necessari.
6. Una volta completata l'installazione, chiudete l'installer ed aprite Start−>Programs−>ActiveState ActivePython 2.2−>PythonWin IDE.

Esempio 1.1. La IDE di ActivePython

```
PythonWin 2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)] on win32.
Portions Copyright 1994−2001 Mark Hammond (mhammond@skippinet.com.au) −
see 'Help/About PythonWin' for further copyright information.
>>>
```

La seconda opzione è l'installer di Python "ufficiale", distribuito dalle persone che sviluppano Python stesso. È liberamente scaricabile ed open source, ed è sempre aggiornato all'ultima versione di Python.

Procedura 1.2. Opzione 2: Installare Python da Python.org

1. Scaricate l'ultimo `installer` per `Windows` di `Python` da http://www.python.org/ftp/python/2.3.2/.
2. Doppio click sull'installer, `Python−2.3.2.exe`.
3. Seguite le istruzioni dell'installer.
4. Se lo spazio su disco non è sufficiente, potete deselezionare il file HTMLHelp, gli script di utilità (`Tools/`) e/o la raccolta di test (`Lib/test/`).
5. Se non avete i diritti di amministratore sulla vostra macchina, potete selezionare le opzioni avanzate (Advanced Options ...) e selezionare l'installazione per non−amministratori (Non−Admin Install). Questo va ad influire solo sulla posizione delle chiavi nel registro e sulla creazione delle scorciatoie dei menu.
6. Una volta completata l'installazione, chiudete l'installer ed aprite Start−>Programs−>Python 2.3−>IDLE (Python GUI).

Esempio 1.2. IDLE (Python GUI)

```
Python 2.3.2 (#49, Oct  2 2003, 20:02:00) [MSC v.1200 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.0
>>>
```

1.3. Python su Mac OS X

Con Mac OS X, ci sono due possibili alternative per installare Python: o lo si installa, oppure no. Probabilmente, voi volete installarlo.

Mac OS X 10.2 e succesivi includono una versione pre−installata di Python, con interfaccia a riga di comando. Se si è a proprio agio con la riga di comando, è possibile usare questa versione almeno per il primo terzo di questo libro. Tuttavia, la versione pre−installata non include un parser XML, per cui arrivati al capitolo su XML è necessario installare la versione completa.

Procedura 1.3. Eseguire la versione di Python preinstallata su Mac OS X.

1. Aprire la cartella `/Applications`.
2. Aprire la cartella `Utilities`.
3. Cliccare due volte su `Terminal` per aprire un terminale e accedere alla riga di comando.
4. Digitare **`python`** al prompt della riga di comando.

Esempio 1.3. Usare la versione di Python preinstallata su Mac OS X

```
Welcome to Darwin!
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you%
```

Ad ogni modo, voi probabilmente volete installare l'ultima versione, che tra l'altro include una shell grafica interattiva.

Procedura 1.4. Installare su Mac OS X

1. Scaricare l'immagine del disco `MacPython-OSX` dal sito http://homepages.cwi.nl/~jack/macpython/download.html.
2. Se il vostro browser non l'ha già fatto, cliccate due volte su `MacPython-OSX-2.3-1.dmg` per fare il mount dell'immagine del disco sul vostro desktop.
3. Fate doppio click sul programma di installazione, `MacPython-OSX.pkg`.
4. Il programma di installazione vi richiederà il nome utente e la password dell'amministratore.
5. Seguite le istruzioni del programma di installazione.
6. Dopo che l'installazione è stata completata, chiudete il programma di installazione ed aprite la cartella `/Applications`.
7. Aprite la cartella `MacPython-2.3`.
8. Fate doppio click su `PythonIDE` per lanciare Python.

L'IDE MacPython dovrebbe visualizzare una schermata di introduzione e poi rendervi disponibile la shell interattiva. Se quest'ultima non dovesse apparire, selezionate l'opzione Window->Python Interactive (**Cmd-0**).

Esempio 1.4. L'IDE MacPython in Mac OS X

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

Da notare che una volta installata l'ultima versione, la versione preinstallata è ancora presente. Se eseguite uno script da riga di comando, occorre che siate coscienti di quale versione di Python state usando.

Esempio 1.5. Due versioni di Python

```
[localhost:~] you% python
Python 2.2 (#1, 07/14/02, 23:25:09)
[GCC Apple cpp-precomp 6.14] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you% /usr/local/bin/python
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)] on darwin
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
[localhost:~] you%
```

1.4. Python su Mac OS 9

Mac OS 9 non include alcuna versione di Python, ma l'installazione è molto semplice e non ci sono altre alternative.

Procedura 1.5. Installare su Mac OS 9

1. Scaricate il file `MacPython23full.bin` da http://homepages.cwi.nl/~jack/macpython/download.html.
2. Se il vostro browser non decomprime automaticamente il file, cliccate due volte su `MacPython23full.bin` per decomprimerlo con Stuffit Expander.
3. Fate il doppio click sul programma di installazione `MacPython23full`.
4. Seguite i passi indicati dal programma di installazione.
5. Una volta che l'installazione è stata completata, chiudete il programma di installazione ed aprite la cartella `/Applications`.
6. Aprite la cartella `MacPython-OS9 2.3`.
7. Fate doppio click su `Python IDE` per lanciare Python.

L'IDE MacPython dovrebbe visualizzare una schermata di introduzione e poi rendervi disponibile la shell interattiva. Se quest'ultima non dovesse apparire, selezionate l'opzione Window->Python Interactive (**Cmd-0**).

Esempio 1.6. La IDE MacPython su Mac OS 9

```
Python 2.3 (#2, Jul 30 2003, 11:45:28)
[GCC 3.1 20020420 (prerelease)]
Type "copyright", "credits" or "license" for more information.
MacPython IDE 1.0.1
>>>
```

1.5. Python su RedHat Linux

L'installazione su sistemi operativi UNIX-compatibili come Linux è facile se avete intenzione di installare un pacchetto binario. Pacchetti binari precompilati sono disponibili per molte popolari distribuzioni Linux. Potete comunque sempre compilare dai sorgenti.

Per installare su RedHat Linux, avete bisogno di scaricare l'RPM da http://www.python.org/ftp/python/2.3.2/rpms/ ed installarlo con il comando **rpm**.

Esempio 1.7. Installazione su RedHat Linux 9

```
localhost:~$ su -
Password: [enter your root password]
[root@localhost root]# wget http://python.org/ftp/python/2.3/rpms/redhat-9/python2.3-2.3-5pydotorg.i38
Resolving python.org... done.
Connecting to python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7,495,111 [application/octet-stream]
...
[root@localhost root]# rpm -Uvh python2.3-2.3-5pydotorg.i386.rpm
Preparing...                ########################################### [100%]
   1:python2.3              ########################################### [100%]
[root@localhost root]# python          (1)
Python 2.2.2 (#1, Feb 24 2003, 19:13:11)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-4)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
```

```
>>> [press Ctrl+D to exit]
[root@localhost root]# python2.3       (2)
Python 2.3 (#1, Sep 12 2003, 10:53:56)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits", or "license" for more information.
>>> [press Ctrl+D to exit]
[root@localhost root]# which python2.3 (3)
/usr/bin/python2.3
```

(1) Ooops! Digitando semplicemente **`python`** otteniamo la vecchia versione di Python, quella installata in modo predefinito. Non è quella che vogliamo.

(2) La versione più recente si chiama **`python2.3`**. Probabilmente vorrete cambiare il percorso nella prima riga dello script di esempio per farlo puntare alla versione più recente.

(3) Questo è il percorso completo della versione più recente di Python che abbiamo appena installato. Utilizzate questo dopo il `#!` (la prima linea dello script) per essere sicuri che gli script vengano lanciati dall'ultima versione di Python, e assicuratevi di digitare **`python2.3`** per entrare nella shell interattiva.

1.6. Python su Debian GNU/Linux

Se siete abbastanza fortunati da utilizzare Debian GNU/Linux, potete installare con il comando **apt**.

Esempio 1.8. Installare su Debian GNU/Linux

```
localhost:~$ su -
Password: [enter your root password]
localhost:~# apt-get install python
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  python2.3
Suggested packages:
  python-tk python2.3-doc
The following NEW packages will be installed:
  python python2.3
0 upgraded, 2 newly installed, 0 to remove and 3 not upgraded.
Need to get 0B/2880kB of archives.
After unpacking 9351kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Selecting previously deselected package python2.3.
(Reading database ... 22848 files and directories currently installed.)
Unpacking python2.3 (from .../python2.3_2.3.1-1_i386.deb) ...
Selecting previously deselected package python.
Unpacking python (from .../python_2.3.1-1_all.deb) ...
Setting up python (2.3.1-1) ...
Setting up python2.3 (2.3.1-1) ...
Compiling python modules in /usr/lib/python2.3 ...
Compiling optimized python modules in /usr/lib/python2.3 ...
localhost:~# exit
logout
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [press Ctrl+D to exit]
```

1.7. Installare dai sorgenti

Se preferite compilare dai sorgenti, potete scaricare i sorgenti di Python da http://www.python.org/ftp/python/2.3.2/ ed effettuare i soliti **`configure`**, **`make`**, **`make install`**.

Esempio 1.9. Installare dai sorgenti

```
localhost:~$ su -
Password: [enter your root password]
localhost:~# wget http://www.python.org/ftp/python/2.3/Python-2.3.tgz
Resolving www.python.org... done.
Connecting to www.python.org[194.109.137.226]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8,436,880 [application/x-tar]
...
localhost:~# tar xfz Python-2.3.tgz
localhost:~# cd Python-2.3
localhost:~/Python-2.3# ./configure
checking MACHDEP... linux2
checking EXTRAPLATDIR...
checking for --without-gcc... no
...
localhost:~/Python-2.3# make
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Modules/python.o Modules/python.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/acceler.o Parser/acceler.c
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall -Wstrict-prototypes
-I. -I./Include  -DPy_BUILD_CORE -o Parser/grammar1.o Parser/grammar1.c
...
localhost:~/Python-2.3# make install
/usr/bin/install -c python /usr/local/bin/python2.3
...
localhost:~/Python-2.3# exit
logout
localhost:~$ which python
/usr/local/bin/python
localhost:~$ python
Python 2.3.1 (#2, Sep 24 2003, 11:39:14)
[GCC 3.3.2 20030908 (Debian prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> [press Ctrl+D to get back to the command prompt]
localhost:~$
```

1.8. La shell interattiva

Adesso che abbiamo Python, che cos'è questa shell interattiva che abbiamo lanciato?

Ecco cos'è: Python conduce una vita doppia. È sia un interprete per gli script che potete lanciare -- o dalla linea di comando o con un doppio clic sugli script, lanciandoli come applicazioni. Ma è anche una shell interattiva, che può valutare dichiarazioni ed espressioni arbitrarie. Ciò è estremamente utile per il debugging, l'hacking rapido ed il testing. Conosco persino delle persone che utilizzano la shell interattiva di Python come calcolatore!

Lanciate la shell interattiva di Python in qualunque modo funzioni la vostra piattaforma ed immergetevi.

Esempio 1.10. Primi passi nella shell interattiva

```
>>> 1 + 1               (1)
2
>>> print 'hello world' (2)
hello world
>>> x = 1               (3)
>>> y = 2
>>> x + y
3
```

(1) La shell interattiva di Python può valutare espressioni Python arbitrarie, inclusa qualsiasi espressione aritmetica di base.

(2) La shell interattiva può eseguire istruzioni Python arbitrarie, inclusa l'istruzione **print**.

(3) Potete inoltre assegnare valori alle variabili, e i valori saranno ricordati finché la shell rimane aperta (ma non oltre).

1.9. Sommario

Dovreste adesso avere una versione di Python installata che lavora per voi.

A seconda della vostra piattaforma, potreste avere più di una versione. Quindi fate attenzione ai percorsi. Se digitaste semplicemente **python** sulla riga di comando e non otteneste la versione di Python che vorreste usare, potreste avere bisogno di inserire il percorso completo della vostra versione preferita.

Questo a parte, congratulazioni e benvenuti in Python.

Capitolo 2. Conoscere Python

2.1. Immergersi

Ecco un completo e funzionante programma in Python.

Molto probabilmente non avrà alcun senso per voi. Non vi preoccupate; andremo a descriverlo linea per linea. Leggetelo innanzitutto e cercate di capire se potrebbe esservi utile.

Esempio 2.1. `odbchelper.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Ora lanciate questo programma e osservate cosa succede.

> **Suggerimento: Lanciate il modulo (Windows)**
> Nel Python IDE di Windows, potere lanciare un modulo con File−>Run... (Ctrl−R). File−>Run... (**Ctrl−R**). L'output viene mostrato nella finestra interattiva.
>
> **Suggerimento: Lanciate il modulo (Mac OS)**
> Nel Python IDE di Mac OS, potete lanciare un modulo con Python−>Run window... (**Cmd−R**), ma c'è prima un'opzione importante che dovete settare. Aprite il modulo nell'IDE, fate comparire il menu delle opzioni dei moduli cliccando sul triangolo nero posto nell'angolo in alto a destra della finestra e siate sicuri che "Run as __main__" sia settato. Questa impostazione verrà salvata con il modulo, così avrete bisogno di impostarla una volta sola per ogni modulo.
>
> **Suggerimento: Lanciate il modulo (UNIX)**
> Sui sistemi UNIX−compatibili (incluso Mac OS X), potete lanciare un modulo direttamente dalla linea di comando: **`python odbchelper.py`**

Esempio 2.2. Output di `odbchelper.py`

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

2.2. Dichiarare le funzioni

Python supporta le funzioni come molti altri linguaggi, ma non necessita di header file separati come il C++ o sezioni di `interfaccia`/`implementazione` come il Pascal. Quando avete bisogno di una funzione, basta che la dichiariate e la definiate.

Esempio 2.3. Dichiarare la funzione `buildConnectionString`

```
def buildConnectionString(params):
```

Ci sono alcune cose da chiarire qui. Inanzitutto, la parola chiave `def` inizia la dichiarazione della funzione, viene poi seguita dal nome della funzione e dagli argomenti fra parentesi. Argomenti multipli (non mostrati qui) devono essere separati da virgole.

Seconda cosa, la funzione non definisce un valore di ritorno. Le funzioni Python non specificano il tipo dei loro valori di ritorno; esse non specificano neppure se ritornano un valore oppure no. In effetti, ogni funzione Python ritorna un valore; se la funzione esegue un `return` essa ritornerà quel valore, altrimenti restituirà `None`, il valore nullo di Python.

> **Nota: Python contro Visual Basic: ritornare un valore**
> Nel Visual Basic, le funzioni (che ritornano un valore) iniziano con `function`, e le procedure (che non ritornano un valore) iniziano con `sub`. Non esistono procedure in Python. Sono tutte funzioni, ritornano un valore, anche se è `None` ed iniziano tutte con `def`.

Terza cosa, l'argomento, o `parametro`, non specifica alcun tipo. In Python le variabili non sono mai definite con un tipo esplicito. È il Python stesso che si occupa dei tipi delle variabili e li gestisce internamente.

> **Nota: Python contro Java: ritornare un valore**
> In Java, C++, ed in altri linguaggi con tipi di dato statici, dovete specificare il tipo del valore di ritorno della funzione e di ogni suo argomento. In Python non si specifica mai espressamente nessun tipo di dato. A seconda del valore che assegnerete, Python terrà conto del tipo di dato internamente.

Appendice. Un lettore erudito mi ha mandato questa spiegazione di come il Python si confronta con altri linguaggi di programmazione:

linguaggio tipato staticamente
: Un linguaggio nel quale i tipi sono fissati al momento della compilazione. Molti linguaggi staticamente tipati ottengono ciò costringendovi a dichiarare tutte le variabili con il loro tipo di dato prima di usarle. Java e C sono linguaggi staticamente tipati.

linguaggio tipato dinamicamente
: Un linguaggio nel quale i tipi sono scoperti durante l'esecuzione; l'opposto dell'impostazione statica. VBScript e Python sono tipati dinamicamente, dato che essi scoprono il tipo di una variabile nel momento in cui le assegnate un valore.

linguaggio fortemente tipato
: Un linguaggio nel quale i tipi sono sempre imposti. Java e Python sono fortemente tipati. Se avete un intero, non potete trattarlo come una stringa senza convertirlo esplicitamente (molte dritte su come farlo le troverete più avanti in questo capitolo).

linguaggio debolmente tipato
: Un linguaggio nel quale i tipi possono essere ignorati; l'opposto di un linguaggio fortemente tipato. VBScript è debolmente tipato. In VBScript, potete concatenare la stringa `'12'` e l'intero `3` in modo da ottenere la stringa `'123'`, per poi trattarla come l'intero `123`, il tutto senza conversione esplicita.

Perciò Python è sia *dinamicamente tipato* (visto che non utilizza dichiarazione esplicite di tipo) sia *fortemente tipato* (visto che nel momento in cui una variabile ha un suo tipo, questo ha una reale importanza).

2.3. Documentare le funzioni

Potete documentare una funzione Python dandole una `doc string` (stringa di documentazione ndr.).

Esempio 2.4. Definire la `doc string` della funzione `buildConnectionString`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
```

Le triple virgolette indicano un stringa multilinea. Ogni cosa tra le virgolette iniziali e finali fanno parte di una singola stringa, inclusi il carriage returns ed altri caratteri speciali. Le potete usare ovunque, ma le vedrete quasi sempre usate durante la definizione di una `doc string`.

> **Nota: Python contro Perl: citazione**
> Le triple virgolette sono anche un semplice modo per definire una stringa con singole e doppie virgolette, come il `qq/.../` del Perl.

Ogni cosa in mezzo alle triple virgolette rappresenta la `doc string`, della funzione, essa documenta ciò che la funzione è in grado di fare. Una `doc string`, se esiste, deve essere la prima cosa definita in una funzione (*i.e.* la prima cosa dopo i due punti). Non è necessario dotare la vostra funzione di una `doc string`, ma è buona cosa farlo sempre. So che avrete sentito questa frase in ogni lezione di programmazione che avete sentito, ma Python vi dà un incentivo: la `doc string` rimane disponibile durante l'esecuzione del programma come attributo della funzione.

> **Nota: Perché le doc string sono la cosa giusta**
> Molti IDE di Python usano la `doc string` per rendere disponibile una documentazione context−sensitive, così quando scrivete il nome di una funzione, la sua `doc string` appare come tooltip. Questo può essere incredibilmente utile, ma la sua qualità si basa unicamente sulla `doc string` che avete scritto.

Ulteriori letture

- PEP 257 definisce le convenzioni sulle `doc string`.
- *Python Style Guide* discute di come scrivere una buona `doc string`.
- *Python Tutorial* discute delle convenzioni per spaziare le `doc string`.

2.4. Tutto è un oggetto

Nel caso non ve ne foste accorti, ho appena detto che le funzioni in Python hanno attributi e che questi attributi sono disponibili a runtime.

Una funzione, come ogni altra cosa in Python, è un oggetto.

Esempio 2.5. Accedere alla `doc string` della funzione `buildConnectionString`

```
>>> import odbchelper                              (1)
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> print odbchelper.buildConnectionString(params) (2)
server=mpilgrim;uid=sa;database=master;pwd=secret
>>> print odbchelper.buildConnectionString.__doc__ (3)
```

```
Build a connection string from a dictionary

Returns string.
```

(1) La prima linea importa il programma `odbchelper` come modulo. Una volta che avete importato un modulo potete fare riferimento ad ogni sua funzione, classe o attributo pubblico. I moduli possono quindi avere accesso a funzionalità presenti in altri moduli e lo stesso principio rimane valido anche per quanto riguarda l'IDE. Questo è un concetto molto importante e ne riparleremo più approfonditamente dopo.

(2) Quando volete usare funzioni definite in moduli importati, dovete includere il nome del modulo. Così non è possibile dire solo `buildConnectionString`, deve essere `odbchelper.buildConnectionString`. Se avete usato le classi in Java, questo dovrebbe esservi vagamente familiare.

(3) Invece di chiamare la funzione come vi sareste aspettati, abbiamo chiesto uno dei suoi attributi, `__doc__`.

> **Nota: Python contro Perl: import**
> `import` in Python è come `require` in Perl. Una volta che avete importato un modulo Python, accedete alle sue funzioni con *`module.function`*; una volta che avete richiesto un modulo Perl, accedete alle sue funzioni con *`modulo::funzione`*.

Prima di continuare, voglio brevemente fare riferimento al percorso di ricerca delle librerie. Python cerca in diversi luoghi quando tentate di importare un modulo. In particolare, cerca in tutte le directory definite in `sys.path`. Questa è semplicemente una lista e potete facilmente vederla o modificarla con i normali metodi delle liste (impareremo di più sulle liste più avanti in questo capitolo).

Esempio 2.6. Percorso di ricerca per l'importazione

```
>>> import sys                  (1)
>>> sys.path                    (2)
['', '/usr/local/lib/python2.2', '/usr/local/lib/python2.2/plat-linux2',
'/usr/local/lib/python2.2/lib-dynload', '/usr/local/lib/python2.2/site-packages',
'/usr/local/lib/python2.2/site-packages/PIL', '/usr/local/lib/python2.2/site-packages/piddle']
>>> sys                         (3)
<module 'sys' (built-in)>
>>> sys.path.append('/my/new/path') (4)
```

(1) L'importazione del modulo `sys` restituisce tutte le sue funzioni e attributi disponibili.

(2) `sys.path` è una lista di nomi di directory che costituiscono l'attuale percorso di ricerca (il vostro sarà diverso, in base al sistema operativo, la versione di Python in esecuzione e dove è stato installato). Python cercherà in queste directory (e in questo ordine) un file `.py` corrispondente al nome del modulo che state tentando di importare.

(3) A dire la verità, ho mentito; la verità è più complicata di questa, poiché non tutti i moduli sono situati nei file `.py`. Alcuni, come il modulo `sys`, sono "moduli built-in"; sono veramente integrati nell'interprete dello stesso Python. I moduli built-in si comportano proprio come moduli normali, ma il loro sorgente Python non è disponibile, visto che non sono scritti in Python ! (Il modulo `sys` è scritto in C)

(4) Potete aggiungere una nuova directory al percorso di ricerca di Python a runtime aggiungendo il nome della directory a `sys.path`. Dopo Python cercherà anche in quella directory, quando tenterete di importare un modulo. L'effetto dura fino a quando Python rimane in esecuzione (parleremo di più riguardo `append` ed altri metodi delle liste, più avanti in questo capitolo).

Tutto in Python è un oggetto e quasi tutto ha attributi e metodi. [1] Tutte le funzioni hanno l'attributo built-in `__doc__`, che ritorna la `doc string` definita nel codice sorgente della funzione. Il modulo `sys` è un oggetto che ha (fra le altre cose) un attributo chiamato `path` ... e così via.

Questo fatto è così importante che sto per ripetertelo, nel caso non lo avessi ancora colto: *tutto in Python è un oggetto*. Le stringhe sono oggetti. Le liste sono oggetti. Le funzioni sono oggetti. Persino i moduli sono oggetti.

Ulteriori letture

- *Python Reference Manual* spiega esattamente cosa significa che tutto in Python è un oggetto, perché alcune persone sono pedanti e amano discutere lungamente su questo genere di cose.
- eff−bot riassume gli oggetti in Python.

2.5. Indentare il codice

Le funzioni in Python non hanno un `inizio` o `fine` esplicito, nessuna parentesi graffa che indichi dove il codice inizia o finisce. L'unico delimitatore sono i due punti (`:`) e l'indentazione del codice.

Esempio 2.7. Indentazione della funzione `buildConnectionString`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

I blocchi di codice (funzioni, istruzioni `if`, cicli `for`, ecc.) sono definiti dalla loro indentazione. L'indentazione inizia un blocco e la rimozione dell'indentazione lo termina; non ci sono graffe, parentesi o parole chiave. Lo spazio in mezzo è insignificante e non è obbligatorio. In questo esempio il codice (inclusa la `doc string`) è indentato di 4 spazi. Se non vi sono 4 spazi non abbiamo la giusta coerenza. La prima linea non indentata non fa parte della funzione.

Dopo qualche protesta iniziale e diverse sarcastiche analogie con Fortran, farete pace con ciò e inizierete a vederne i benefici. Un importante beneficio è che tutti i programmi Python hanno lo stesso aspetto poiché l'indentazione è un requisito del linguaggio e non una questione di stile. Questo fatto rende più facile ad altre persone leggere e capire il codice Python.

> **Nota: Python contro Java: separazione degli statement**
> Python usa il ritorno a capo per separare le istruzioni e i due punti e l'indentazione per separare i blocchi di codice. C++ e Java usano un punto e virgola per separare le istruzioni e le parentesi graffe per separare i blocchi di codice.

Ulteriori letture

- *Python Reference Manual* discute i problemi dell'indentazione cross−platform e mostra vari errori di indentazione.
- *Python Style Guide* discute sul buon stile di indentazione.

2.6. Testare i moduli

I moduli in Python sono oggetti ed hanno diversi attributi utili. Potete usarli per testare i moduli dopo che li avete scritti.

Esempio 2.8. Il trucco `if __name__`

```
if __name__ == "__main__":
```

Qualche osservazione veloce prima di arrivare al succo del discorso. Prima di tutto le parentesi non sono richieste attorno all'espressione `if`. Secondo, lo statement `if` termina con i due punti ed è seguito da codice indentato.

> **Nota: Python contro C: confronto e assegnamento**
> Come il C, Python usa `==` per i confronti e `=` per gli assegnamenti. Al contrario del C, Python non supporta gli assegnamenti in una singola riga, così non c'è modo di assegnare per errore il valore che pensavate di comparare.

Così perché questo `if` particolare è un trucco? I moduli sono oggetti e tutti i moduli hanno un attributo built-in `__name__`. Il `__name__` di un modulo dipende da come state usando il modulo. Se `importate` il modulo, allora `__name__` è il nome del file del modulo, senza il percorso completo o l'estensione. Ma potete anche eseguire il modulo direttamente come un programma indipendente, in questo caso `__name__` avrà un valore default speciale, `__main__`.

Esempio 2.9. Se importate il modulo `__name__`

```
>>> import odbchelper
>>> odbchelper.__name__
'odbchelper'
```

Sapendo ciò, potete creare una serie di test per il vostro modulo nel modulo stesso, mettendolo nello statement `if`. Quando eseguite il modulo direttamente, `__name__` è `__main__`, così i test vengono eseguiti. Quando il modulo è importato, `__name__` è qualcos'altro, così i test sono ignorati. Questo rende più semplice sviluppare ed eseguire il debug di nuovi moduli prima di integrarli in un programma di dimensioni maggiori.

> **Suggerimento: if __name__ su Mac OS**
> In MacPython c'è un passaggio ulteriore per far funzionare il trucco `if __name__`. Fate apparire le opzioni del modulo cliccando sul triangolo nero, nell'angolo in altro a destra della finestra e assicuratevi che "Esegui come __main__" sia selezionato.

Ulteriori letture

- *Python Reference Manual* discute dettagli di basso livello sull'importazione dei moduli.

2.7. Introduzione ai dizionari

Una piccola digressione è dovuta perché avete bisogno di conoscere dizionari, tuple e liste. Se siete dei programmatori esperti in Perl, probabilmente ne sapete abbastanza sui dizionari e sulle liste, ma dovreste comunque prestare attenzione alle tuple.

Uno dei tipi predefiniti in Python è il dizionario che definisce una relazione uno-a-uno tra chiavi e valori.

> **Nota: Python contro Perl: dizionari**
> Un dizionario in Python è come una hash in Perl. In Perl, le variabili che memorizzano degli hash cominciano sempre con il carattere %; in Python, le variabili possono avere qualsiasi nome, e Python tiene traccia del loro tipo internamente.

> **Nota: Python contro Java: dizionari**
> Un dizionario in Python è come l'istanza di una classe `Hashtable` in Java.

Nota: Python contro Visual Basic: dizionari
Un dizionario Python è come l'istanza di un oggetto `Scripting.Dictionary` in Visual Basic.

Esempio 2.10. Definire un dizionario

```
>>> d = {"server":"mpilgrim", "database":"master"} (1)
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]                                    (2)
'mpilgrim'
>>> d["database"]                                  (3)
'master'
>>> d["mpilgrim"]                                  (4)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```

(1) Per prima cosa, creiamo un nuovo dizionario con due elementi e lo assegnamo alla variabile `d`. Ogni elemento è una coppia chiave−valore, e l'intero insieme di elementi è racchiuso tra parentesi graffe.

(2) `'server'` è una chiave ed il suo valore associato, referenziato da `d["server"]` è `'mpilgrim'`.

(3) `'database'` è una chiave ed il suo valore associato, referenziato da `d["database"]` è `'master'`.

(4) Potete ottenere i valori dalle chiavi, ma non potete ottenere le chiavi dai valori. Così, `d["server"]` è `'mpilgrim'`, ma `d["mpilgrim"]` genera un'eccezione perché `'mpilgrim'` non è una chiave.

Esempio 2.11. Modificare un dizionario

```
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["database"] = "pubs" (1)
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"        (2)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

(1) Non possono esserci duplicazioni di chiavi in un dizionario. Assegnare un valore ad una chiave esistente sovrascriverà il vecchio valore.

(2) Potete aggiungere nuove coppie chiave−valore in ogni momento. È la stessa sintassi della modifica di valori già esistenti. (Sì, questo vi disturberà, a volte, quando pensate di aggiungere nuovi valori mentre state in realtà semplicemente modificando lo stesso valore più e più volte solo perché la chiave non sta cambiando come pensavate.)

Notate che il nuovo elemento (chiave `'uid'`, valore `'sa'`) sembra trovarsi nel mezzo. Infatti è una semplice coincidenza che gli elementi appaiano ordinati nel primo esempio; è comunque una coincidenza anche il fatto che ora appaiano disordinati.

Nota: I dizionari non sono ordinati
I dizionari non hanno il concetto di ordinamento tra elementi. È errato dire che gli elementi sono "disordinati"; sono semplicemente non ordinati. Si tratta di un'importante distinzione che vi darà noia quando vorrete accedere agli elementi di un dizionario in un ordine specifico e ripetibile (ad esempio in ordine alfabetico di chiave). Ci sono dei modi per farlo, semplicemente non sono predefiniti nel dizionario.

Esempio 2.12. Mischiare tipi di dato in un dizionario

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
>>> d["retrycount"] = 3 (1)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d[42] = "douglas"   (2)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 42: 'douglas', 'retrycount': 3}
```

(1) I dizionari non funzionano solo con le stringhe. I valori dei dizionari possono essere di qualunque tipo, incluse stringhe, interi, oggetti ed anche altri dizionari. All'interno di un singolo dizionario i valori non devono tutti essere dello stesso tipo, potete mischiarli secondo l'occorrenza.

(2) Le chiavi dei dizionari sono molto più ristrette, ma possono essere stringhe, interi ed alcuni altri tipi (se ne parlerà diffusamente più avanti). Anche i tipi delle chiavi possono essere mischiati in un dizionario.

Esempio 2.13. Cancellare elementi da un dizionario

```
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 42: 'douglas', 'retrycount': 3}
>>> del d[42] (1)
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'master', 'retrycount': 3}
>>> d.clear() (2)
>>> d
{}
```

(1) `del` vi permette di cancellare singoli elementi da un dizionario in base alla chiave.

(2) `clear` cancella tutti gli elementi da un dizionario. Notate che le parentesi graffe vuote indicano un dizionario privo di elementi.

Esempio 2.14. Le stringhe sono case−sensitive

```
>>> d = {}
>>> d["key"] = "value"
>>> d["key"] = "other value" (1)
>>> d
{'key': 'other value'}
>>> d["Key"] = "third value" (2)
>>> d
{'Key': 'third value', 'key': 'other value'}
```

(1) Assegnare un valore ad una chiave già esistente in un dizionario semplicemente sostituisce un vecchio valore con uno nuovo.

(2) Questo non significa assegnare un valore ad una chiave già esistente perché le stringhe in Python sono case−sensitive, così `'key'` non è la stessa cosa di `'Key'`. Si crea una nuova coppia chiave/valore nel dizionario; potrebbe sembrarvi simile, ma per come è stato progettato Python è completamente diverso.

Ulteriori letture

- *How to Think Like a Computer Scientist* insegna ad usare i dizionari e mostra come usarli per modellare matrici rade.
- Python Knowledge Base contiene molti esempi di codice sull'uso dei dizionari.
- Python Cookbook parla di come ordinare il valori di un dizionario per chiave.
- *Python Library Reference* riassume tutti i metodi dei dizionari.

2.8. Introduzione alle liste

Le liste sono, tra i tipi di dati in Python, la vera forza motrice. Se la vostra sola esperienza con le liste sono gli array in Visual Basic o (Dio ve ne scampi!) il datastore di Powerbuilder, fatevi forza e osservate come si usano in Python.

> **Nota: Python contro Perl: liste**
> Una lista in Python è come un array in Perl. In Perl, le variabili che contengono array hanno un nome che inizia sempre con il carattere @; in Python le variabili possono avere qualunque nome, è il linguaggio che tiene traccia internamente del tipo di dato.

> **Nota: Python contro Java: liste**
> Una lista in Python è molto più di un array in Java (sebbene possa essere usato allo stesso modo, se davvero non volete altro). Un'analogia migliore sarebbe la classe `Vector`, che può contenere oggetti di tipo arbitrario e si espande automaticamente quando vi si aggiungono nuovi elementi.

Esempio 2.15. Definire una lista

```
>>> li = ["a", "b", "mpilgrim", "z", "example"] (1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0]                                       (2)
'a'
>>> li[4]                                       (3)
'example'
```

(1) Per prima cosa definiamo una lista di 5 elementi. Notate che questi mantengono l'ordine d'inserimento, in quanto una lista e' un insieme ordinato di elementi racchiusi tra parentesi quadrate.

(2) A differenza di altri linguaggi, Python non permette di scegliere se gli array cominciano con l'elemento 0 oppure 1. Se la lista non è vuota, il primo elemento si indica sempre con `li[0]`.

(3) In questa lista di 5 elementi, l'ultimo si indica con `li[4]` perché le liste hanno sempre base zero.

Esempio 2.16. Indici di liste negativi

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[-1] (1)
'example'
>>> li[-3] (2)
'mpilgrim'
```

(1) Un indice negativo accede agli elementi a partire dalla fine della lista, contando all'indietro. L'ultimo elemento di una lista non vuota è sempre `li[-1]`.

(2) Se gli indici negativi vi confondono, pensate in questo modo: `li[-n] == li[len(li) - n]`. Così, in questa lista, `li[-3] == li[5 - 3] == li[2]`.

Esempio 2.17. Sezionare una lista.

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[1:3]  (1)
['b', 'mpilgrim']
>>> li[1:-1] (2)
['b', 'mpilgrim', 'z']
```

```
>>> li[0:3]  (3)
['a', 'b', 'mpilgrim']
```

(1) Potete estrarre una sezione di una lista, chiamata "slice", (fetta), specificandone i due estremi. Il valore della sezione è una nuova lista, contenente tutti gli elementi, nello stesso ordine, dal primo indice (in questo caso `li[1]`), fino al secondo indice escluso (in questo caso `li[3]`).

(2) Il sezionamento, o slicing, funziona anche se uno o entrambi gli indici sono negativi. Potete pensarla in questo modo: leggendo la lista da sinistra a destra, il primo indice specifica il primo elemento che volete, mentre il secondo specifica il primo elemento che non volete. Il valore ritornato è una lista degli elementi che stanno in mezzo.

(3) Gli elementi della lista si contano da zero, così `li[0:3]` ritorna i primi tre elementi della lista, partendo da `li[0]` fino a `li[3]` escluso.

Esempio 2.18. Sezionamento abbreviato

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3] (1)
['a', 'b', 'mpilgrim']
>>> li[3:] (2) (3)
['z', 'example']
>>> li[:]  (4)
['a', 'b', 'mpilgrim', 'z', 'example']
```

(1) Se l'indice di sinistra è 0, potete non indicarlo e lo 0 è sottinteso. In questo caso `li[:3]` è identico a `li[0:3]` dell'esempio precedente.

(2) Allo stesso modo, se l'indice destro è pari alla lunghezza della lista, potete lasciarlo sottinteso. Così `li[3:]` è identico a `li[3:5]`, perché la lista ha 5 elementi.

(3) Notate la simmetria. In questa lista di 5 elementi, `li[:3]` ritorna i primi 3, mentre `li[3:]` ritorna gli ultimi 2 elementi. Di fatto, `li[:n]` ritornerà sempre i primi `n` elementi e `li[n:]` ritornerà la parte restante della lista a prescindere dalla lunghezza di quest'ultima.

(4) Se chiedete un sezionamento senza specificare alcun indice, tutti gli elementi della lista saranno inclusi. L'oggetto restituito però non è la lista `li` originale; è una nuova lista che contiene gli stessi elementi. `li[:]` è un'abbreviazione per creare una copia completa di una lista.

Esempio 2.19. Aggiungere elementi ad una lista

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new")               (1)
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new")            (2)
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.extend(["two", "elements"]) (3)
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
```

(1) `append` aggiunge un singolo elemento alla fine della lista.

(2) `insert` inserisce un singolo elemento in una posizione specifica della lista. L'argomento numerico è l'indice del primo elemento che verrà spostato nella posizione successiva. Notate che gli elementi di una lista non devono necessariamente essere unici; adesso ci sono 2 elementi distinti con il valore `'new'`, `li[2]` e `li[6]`.

(3)

`extend` concatena due liste. Notate che non si può chiamare `extend` con argomenti multipli: l'unico argomento accettato è una lista. In questo caso la lista ha due elementi.

Esempio 2.20. Ricerca in una lista

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.index("example") (1)
5
>>> li.index("new")     (2)
2
>>> li.index("c")       (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li           (4)
0
```

(1) `index` cerca la prima presenza di un valore nella lista e ne ritorna l'indice.

(2) `index` cerca la *prima* presenza di un valore nella lista. In questo caso, `'new'` compare due volte nella lista, in `li[2]` e `li[6]`, ma `index` ritorna solamente il primo indice, `2`.

(3) Se il valore non è presente nella lista, Python solleva un'eccezione. Questo comportamento è diverso dalla maggior parte dei linguaggi, in cui viene ritornato qualche valore di indice non valido. Per quanto il lancio di un'eccezione possa sembrare seccante, in realtà è meglio così perché il vostro programma vi indicherà la radice del problema, anziché bloccarsi più tardi quando cercherete di usare l'indice non valido.

(4) Per controllare se un valore è nella lista usate l'operatore `in`, che ritorna `1` se il valore è presente e `0` in caso contrario.

Nota: Cos'è vero in Python?
Prima della versione 2.2.1, Python non aveva un tipo di dato booleano distinto dai numeri interi. Per compensare, Python accettava quasi tutto in un contesto booleano (come un comando `if`), secondo la seguente regola: `0` è falso; tutti gli altri numeri sono veri. Una stringa vuota (`""`) è falsa; tutte le altre stringhe sono vere. Una lista vuota (`[]`) è falsa; tutte le altre liste sono vere. Una tupla vuota (`()`) è falsa; tutte le altre tuple sono vere. Un dizionario vuoto (`{}`) è falso; tutti gli altri dizionari sono veri. Queste regole valgono ancora in Python 2.2.1 e oltre, ma ora si può usare anche un valore booleano vero e proprio, che ha valore `True` o `False`. Notate le maiuscole; questi valori, come tutto il resto in Python, sono case−sensitive.

Esempio 2.21. Togliere elementi da una lista

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("z")   (1)
>>> li
['a', 'b', 'new', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("new") (2)
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two', 'elements']
>>> li.remove("c")   (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()         (4)
'elements'
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
```

(1) `remove` toglie la prima presenza di un valore nella lista.

(2) `remove` toglie *solo la prima ricorrenza* di un valore specificato. In questo caso, `'new'` compare due volte nella lista, ma `li.remove("new")` ha tolto solo il primo elemento che ha trovato.

(3) Se il valore non è presente nella lista, Python solleva un'eccezione. Questo comportamento rispecchia quello del metodo `index`.

(4) `pop` è un animale interessante. Fa due cose: toglie l'ultimo elemento della lista e ne ritorna il valore. Notate che ciò è diverso da `li[-1]`, che ritorna un valore ma non cambia la lista ed è diverso anche da `li.remove(`*`valore`*`)`, che cambia la lista ma non ritorna un valore.

Esempio 2.22. Operatori su liste

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new'] (1)
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two']                (2)
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3              (3)
>>> li
[1, 2, 1, 2, 1, 2]
```

(1) Le liste possono anche essere concatenate con l'operatore `+`. *`lista`* `= ` *`lista`* `+ ` *`altralista`* ha lo stesso risultato di *`lista`*`.extend(`*`altralista`*`)`, ma l'operatore `+` ritorna una nuova lista (concatenata) come valore, mentre `extend` modifica la lista già esistente. Pertanto, `extend` è più veloce, in modo particolare in riferimento a grandi liste.

(2) Python supporta l'operatore `+=`. `li += ['two']` equivale a `li.extend(['two'])`. L'operatore `+=` funziona con liste, stringhe ed interi e può essere aggiunto alle classi create dal programmatore. (Per le classi, vedi il capitolo 3.)

(3) L'operatore `*` ripete gli elementi di una lista. `li = [1, 2] * 3` è uguale a `li = [1, 2] + [1, 2] + [1, 2]`, che concatena le tre liste in una sola.

Ulteriori letture

- *How to Think Like a Computer Scientist* spiega le liste e spiega come funziona il passaggio di liste come argomenti di funzione.
- *Python Tutorial* mostra come usare liste come pile e code.
- Python Knowledge Base risponde a domande comuni sulle liste e contiene esempi di codice che le utilizza le liste.
- *Python Library Reference* riassume tutti i metodi delle liste.

2.9. Introduzione alle tuple

Una tupla è una lista immutabile. Una tupla non può essere modificata in alcun modo una volta che è stata creata.

Esempio 2.23. Definire una tupla

```
>>> t = ("a", "b", "mpilgrim", "z", "example") (1)
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t[0]                                      (2)
'a'
>>> t[-1]                                     (3)
```

```
'example'
>>> t[1:3]                                  (4)
('b', 'mpilgrim')
```

(1) Una tupla è definita allo stesso modo di una lista, eccetto che l'intero gruppo di elementi viene racchiuso fra parentesi tonde invece che quadre.

(2) Gli elementi di una tupla hanno un ordine definito, proprio come le liste. Gli indici delle tuple partono da zero, come visto nelle liste, perciò il primo elemento di una tupla non vuota è sempre `t[0]`.

(3) Gli indici negativi vengono contati dalla fine della tupla, come nelle liste.

(4) Anche l'affettamento (slicing) funziona come nelle liste. Notate che quando affettate una lista, ottenete una nuova lista, quando affettate una tupla ottenete una nuova tupla.

Esempio 2.24. Le tuple non hanno metodi

```
>>> t
('a', 'b', 'mpilgrim', 'z', 'example')
>>> t.append("new")    (1)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> t.remove("z")      (2)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> t.index("example") (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'index'
>>> "z" in t           (4)
1
```

(1) Non potete aggiungere elementi ad una tupla. Le tuple non hanno i metodi `append` ed `extend`.

(2) Non potete rimuovere elementi da una tupla. Le tuple non hanno i metodi `remove` o `pop`.

(3) Non potete cercare elementi in una tupla. Le tuple non hanno il metodo `index`.

(4) Potete, comunque, usare `in` per vedere se un elemento è presente nella tupla.

A cosa servono le tuple?

- Le tuple sono più veloci delle liste. Se state definendo un gruppo costante di valori e l'unica cosa che intendete farci è iterare al suo interno, usate una tupla invece di una lista.
- Potete rendere più sicuro il vostro codice "proteggendo dalla scrittura" i dati che non devono essere modificati. Usare una tupla invece di una lista è come avere un'implicita istruzione `assert` che mantenga il dato costante e che richieda una riflessione (ed una specifica funzione) per modificarlo.
- Ricordate che dissi che le chiavi di un dizionario possono essere interi, stringhe e "pochi altri tipi"? Le tuple fanno parte di questi tipi. Le tuple possono essere usate come chiavi in un dizionario, le liste no. [2]
- Le tuple sono usate nelle formattazione delle stringhe, come vedremo presto.

Nota: Tuple all'interno di liste e di tuple
Le tuple possono essere convertite in liste e viceversa. La funzione built−in `tuple` prende una lista e ritorna una tupla con gli stessi elementi, mentre la funzione `list` prende una tupla e ritorna una lista. In effetti, `tuple` congela una lista e `list` scongela una tupla.

Ulteriori letture

- *How to Think Like a Computer Scientist* parle delle tuple e mostra come concatenarle.
- Python Knowledge Base mostra come ordinare una tupla.
- *Python Tutorial* mostra come definire una tupla con un elemento.

2.10. Definire le variabili

Adesso che pensate di sapere tutto su dizionari, tuple e liste, torniamo sul nostro programma di esempio, `odbchelper.py`.

Python possiede il concetto di variabili locali e globali come molti altri linguaggi, ma non contempla la dichiarazione esplicita delle variabili. Le variabili cominciano ad esistere quando assegnamo loro un valore e vengono automaticamente distrutte quando non servono più.

Esempio 2.25. Definire la variabile `myParams`

```
if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
```

Molte cose sono intessanti in questo esempio. Prima di tutto l'indentazione. Un construtto `if` è un blocco di codice e necessita di essere indentato come una funzione.

Seconda cosa, l'assegnamento della variabile è un'unico comando diviso su molte linee, attraverso un backslash("\") come continuatore di linea.

> **Nota: Comandi multilinea**
> Quando un comando è diviso su più linee attraverso il continuatore di linea ("\"), la linea seguente può essere indentata in ogni maniera; Le rigide regole di indentazione di Python non vengono applicate. Se il vostro Python IDE auto−indenta la linea continuata, probabilmente dovreste accettare il suo default a meno che non abbiate ottime ragioni per non farlo.

> **Nota: Comandi impliciti multilinea**
> A rigor di termini, espressioni fra parentesi, parentesi quadre o graffe (come per la definizione di un dizionario), possono essere divise in linee multiple con o senza il carattere di continuazione linea("\"). Io preferisco includere il backslash anche quando non è richiesto perché credo che renda il codice più leggibile, ma è solo una questione di stile.

Terza cosa, voi non avete mai dichiarato la variabile `myParams`, le avete semplicemente assegnato un valore. Esattamente come VBScript senza l'opzione `option explicit`. Fortunatamente, al contrario di VBScript, Python non vi permette di riferirvi ad una variabile a cui non è mai stato assegnato un valore; provare a farlo solleva un'eccezione.

Esempio 2.26. Riferirsi ad una variabile non assegnata (unbound, "slegata" n.d.T.)

```
>>> x
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'x'
>>> x = 1
```

```
>>> x
1
```

Un giorno ringrazierete Python per questo.

Ulteriori letture

- *Python Reference Manual* mostra esempi su quando potete saltare il carattere di continuazione linea e quando dovete usarlo.

2.11. Assegnare valori multipli in un colpo solo

Una delle più comode scorciatoie di programmazione in Python consiste nell'usare sequenze per assegnare valori multipli in un colpo solo.

Esempio 2.27. Assegnare valori multipli in un colpo solo

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v (1)
>>> x
'a'
>>> y
'b'
>>> z
'e'
```

(1) `v` è una tupla di tre elementi, mentre `(x, y, z)` è una tupla di tre variabili. L'assegnamento del valore di una alle altre assegna ogni valore di `v` ad ognuna delle altre variabili seguendo l'ordine della tupla.

Questo consente molti utilizzi. Mi capita spesso di assegnare nomi ad un range di valori. In C, usereste `enum` e nominereste manualmente ogni costante ed il suo relativo valore, cosa che diventa particolarmente noiosa quando i valori sono consecutivi. In Python, potete usare per gli assegnamenti multi−variabili la funzione built−in `range` per assegnare rapidamente valori consecutivi.

Esempio 2.28. Assegnare valori consecutivi

```
>>> range(7)                                                              (1)
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7) (2)
>>> MONDAY                                                                (3)
0
>>> TUESDAY
1
>>> SUNDAY
6
```

(1) La funzione built−in `range` ritorna una lista di interi. Nella sua forma più semplice, prende un limite superiore e ritorna una lista partente da `0` che continua verso l'alto senza includere il limite superiore. (Se preferite potete passare altri parametri per specificare una base diversa da `0` ed un incremento diverso da `1`. Potete stampare `print range.__doc__` per i dettagli.)

(2) `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` e `SUNDAY` sono le variabili che stiamo definendo. (Questo esempio proviene dal modulo `calendar`, un piccolo e divertente modulo che stampa calendari, come il programma UNIX `cal`. Il modulo `calendar` definisce costanti intere per i giorni della

settimana.)

(3) Ora ogni variabile ha il suo valore: `MONDAY` è `0`, `TUESDAY` è `1` e così via.

Potete anche usare usare l'assegnamento multi−variabile per costruire funzioni che ritornino valori multipli, semplicemente ritornando una tupla di tutti i valori. Chi chiama la funzione può trattarla come una tupla od assegnare i valori a singole variabili. Molte librerie standard di Python lo fanno, incluso il modulo `os`, del quale discuteremo nel capitolo 3.

Ulteriori letture

- *How to Think Like a Computer Scientist* mostra come usare assegnamenti multi−variabile per invertire il valore di due variabili.

2.12. Formattare le stringhe

Python supporta la formattazione dei valori nelle stringhe. Sebbene ciò possa comprendere espressioni molto complicate, il modo più semplice per utilizzarle consiste nell'inserire valori in una stringa attraverso l'istruzione `%s`.

> **Nota: Python contro C: formattazione delle stringhe**
> La formattazione delle stringhe in Python utilizza la stessa sintassi della funzione C `sprintf`.

Esempio 2.29. Introduzione sulla formattazione delle stringhe

```
>>> k = "uid"
>>> v = "sa"
>>> "%s=%s" % (k, v) (1)
'uid=sa'
```

(1) L'intera espressione viene valutata come stringa. Il primo `%s` è rimpiazzato dal valore di `k`; il secondo `%s` è rimpiazzato dal valore di `v`. Tutti i restanti caratteri della stringa (in questo caso il simbolo uguale) rimangono invariati.

Notare che `(k, v)` è una tupla. Vi avevo detto che sarebbero servite a qualcosa.

Potreste pensare che questo sia un lavoro esagerato per realizzare una semplice concatenazione fra stringhe ed avete ragione, eccetto che la formattazione delle stringhe non riguarda solo la concatenazione. Non si tratta solo di formattazione, è anche coercizione di tipo.

Esempio 2.30. Formattazione delle stringhe contro concatenazione

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid      (1)
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid) (2)
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )           (3) (4)
Users connected: 6
>>> print "Users connected: " + userCount                 (5)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot add type "int" to string
```

(1) `+` è l'operatore che effettua la concatenazione fra le stringhe.

(2) In questo banale esempio, la formattazione delle stringhe raggiunge lo stesso risultato della concatenazione.

(3) `(userCount, )` è una tupla con un solo elemento. Ebbene si, la sintassi è un po' anomala, ma esiste una buona ragione per questo: è una tupla che non lascia spazio ad ambiguità. In effetti, potete sempre includere una virgola dopo l'ultimo elemento quando definite liste, tuple o dizionari. Se la virgola non fosse richiesta, Python non potrebbe sapere se `(userCount)` sia una tupla con un solo elemento o semplicemente il valore di `userCount`.

(4) La formattazione delle stringhe lavora con gli interi specificando `%d` al posto di `%s`.

(5) La concatenazione di una stringa con una non−stringa solleva un'eccezione. A differenza della formattazione delle stringhe, la concatenazione delle stesse, funziona unicamente quando tutti i componenti sono già delle stringhe.

Ulteriori letture

- *Python Library Reference* riassume tutti i caratteri di formattazione delle stringhe.
- *Effective AWK Programming* discute tutti i caratteri di formattazione e le tecniche di formattazione avanzate come specificare lunghezza, precisione e il riempimento di zeri.

2.13. Mappare le liste

Una delle più potenti caratteristiche di Python sono le list comprehension (descrizioni di lista), che utilizzano una tecnica compatta per mappare una lista in un'altra, applicando una funzione ad ognuno degli elementi della lista.

Esempio 2.31. Introduzione alle list comprehension

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]      (1)
[2, 18, 16, 8]
>>> li                           (2)
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li] (3)
>>> li
[2, 18, 16, 8]
```

(1) Per fare luce su questo costrutto guardatelo da destra verso sinistra. `li` è la lista che state mappando. Python esegue un ciclo su `li` un'elemento alla volta, assegnando temporaneamente il valore di ogni elemento alla variabile `elem`. Dopodiché Python applica la funzione `elem*2` ed aggiunge il risultato alla lista di ritorno (usando il metodo "append()" n.d.T.).

(2) Osservate che la list comprehension non modifica la lista originale.

(3) È perfettamente sicuro assegnare il risultato di una list comprehension alla variabile che state mappando. Non ci sono particolari condizioni od altre stranezze di cui tener conto; Python crea la nuova lista in memoria e quando la descrizione di lista è completa, assegna il risultato alla nuova variabile.

Esempio 2.32. List comprehension in `buildConnectionString`

```
["%s=%s" % (k, v) for k, v in params.items()]
```

Notate che state chiamando la funzione `items` del dizionario `params`. Questa funzione ritorna una lista di tuple di tutti i dati del dizionario.

Esempio 2.33. `chiavi`, `valori` ed `elementi`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.keys()   (1)
['server', 'uid', 'database', 'pwd']
>>> params.values() (2)
['mpilgrim', 'sa', 'master', 'secret']
>>> params.items()  (3)
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
```

(1) Il metodo `keys` di un dizionario ritorna una lista di tutte le sue chiavi. La lista non è nell'ordine di come il dizionario è stato definito (ricordate, gli elementi di un dizionario non hanno un ordine), ma è una lista.

(2) Il metodo `values` ritorna una lista di tutti i valori. La lista è nello stesso ordine della lista ritornata da `keys`, così che `params.values()[n] == params[params.keys()[n]]` per tutti i valori di `n`.

(3) Il metodo `items` ritorna una lista di tuple nella forma `(key, value)`. La lista contiene tutti i dati del dizionario.

Adesso guardiamo cosa fa la funzione `buildConnectionString`. Prende una lista, `params.items()` e la mappa in una nuova lista applicando la formattazione delle stringhe ad ogni elemento. La nuova lista avrà lo stesso numero di elementi di `params.items()`, ma ogni elemento della nuova lista sarà una stringa che contiene una chiave ed il relativo valore associato, proveniente dal dizionario `params`.

Esempio 2.34. List comprehension in `buildConnectionString`, passo per passo

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> params.items()
[('server', 'mpilgrim'), ('uid', 'sa'), ('database', 'master'), ('pwd', 'secret')]
>>> [k for k, v in params.items()]               (1)
['server', 'uid', 'database', 'pwd']
>>> [v for k, v in params.items()]               (2)
['mpilgrim', 'sa', 'master', 'secret']
>>> ["%s=%s" % (k, v) for k, v in params.items()] (3)
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

(1) Notate che stiamo usando due variabili per iterare attraverso la lista `params.items()`. Questo è un altro uso dell'assegnamento multi−variabile. Il primo elemento di `params.items()` è `('server', 'mpilgrim')`, perciò nella prima iterazione della list comprehension, `k` otterrà il valore `'server'` e `v` otterrà `'mpilgrim'`. In questo caso, ignorando il valore di `v` ed includendo solo il valore di `k` nella lista di ritorno, questa descrizione di lista finirà per essere equivalente a `params.keys()`. (Voi non usereste mai una descrizione di lista come questa in codice reale; è solo un semplice esempio per farvi capire ciò che avviene.)

(2) Qui stiamo facendo la stessa cosa, ma ignorando il valore di `k`, così che questa list comprehension diventi equivalente a `params.values()`.

(3) Combinando i due esempi precedenti con qualche semplice formattazione di stringa, otteniamo una lista di stringhe che includono sia la chiave che il valore di ogni elemento del dizionario. Questo pare sospetto, come l'output del programma; tutto quello che rimane è unire gli elementi di questa lista in una singola stringa.

Ulteriori letture

- *Python Tutorial* discute di un altro modo per mappare le liste usando la funzione built−in `map`.
- *Python Tutorial* mostra come annidare le list comprehensions.

2.14. Concatenare liste e suddividere stringhe

Avete una lista di coppie chiave−valore nella forma *`chiave=valore`* e volete concatenarle in una singola stringa.

Per concatenare qualunque lista di stringhe in una singola stringa, usate il metodo `join` di un oggetto stringa.

Esempio 2.35. Concatenare una lista in `buildConnectionString`

```
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])
```

Una nota interessante prima di continuare. Continuo a ripetere che le funzioni sono oggetti, le stringhe sono oggetti, qualunque cosa è un oggetto. Potreste aver pensato che intendessi dire che le *variabili* di tipo stringa sono oggetti. Ma no, guardando bene questo esempio vedrete che la stessa stringa `";"` è un oggetto e state chiamando il suo metodo `join`.

Ad ogni modo, il metodo `join` concatena gli elementi della lista in una singola stringa, con ogni elemento separato da un punto e virgola. Il separatore non deve essere per forza un punto e virgola, non deve nemmeno essere per forza un singolo carattere. Può essere una stringa qualsiasi.

> **Importante: non potete concatenare oggetti che non siano stringhe.**
> `join` funziona solamente su liste di stringhe, non applica alcuna forzatura sul tipo. Concatenare una lista che contiene uno o più oggetti che non sono di tipo stringa genererà un'eccezione.

Esempio 2.36. Output di `odbchelper.py`

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> ";".join(["%s=%s" % (k, v) for k, v in params.items()])
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

Questa stringa è quindi restituita dalla funzione `help` e stampata dal blocco chiamante, che vi restituisce quell'output che tanto vi ha meravigliato quando avete iniziato a leggere questo capitolo.

Nota storica. Quando ho cominciato ad imparare Python, mi aspettavo che `join` fosse un metodo delle liste che potesse prendere come argomento il separatore. Molte persone la pensano in questo modo e c'è una storia dietro il metodo `join`. Prima di Python 1.6 le stringhe non avevano tutti questi utili metodi. C'era un modulo `string` separato che conteneva tutte le funzioni sulle stringhe; ogni funzione prendeva una stringa come primo parametro. Le funzioni erano sufficientemente importanti da essere inserite direttamente dentro gli oggetti di tipo stringa, cosa che aveva senso per funzioni come `lower`, `upper`, e `split`. Molti programmatori Python navigati obiettarono per la presenza del nuovo metodo `join`, sostenendo che doveva essere un metodo della classe lista, o che non dovesse essere spostato affatto e continuare ad appartenere al modulo `string` (che comunque continua ad avere un sacco di cose utili). Uso esclusivamente il nuovo metodo `join`, ma vedrete del codice scritto nell'altro modo e se la cosa vi dà davvero fastidio, potete continuare ad usare la vecchia funzione `string.join`.

Vi starete probabilmente chiedendo se c'è un metodo analogo per suddividere una stringa in una lista. E naturalmente c'è, è chiamato `split`.

Esempio 2.37. Suddividere una stringa

```
>>> li = ['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s = ";".join(li)
>>> s
'server=mpilgrim;uid=sa;database=master;pwd=secret'
>>> s.split(";")    (1)
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

```
>>> s.split(";", 1) (2)
['server=mpilgrim', 'uid=sa;database=master;pwd=secret']
```

(1) `split` rovescia `join` suddividendo una stringa in una lista a molti elementi. Notate che il limitatore (`";"`) viene completamente rimosso; non appare in nessuno degli elementi ritornati nella lista.

(2) `split` prende un secondo argomento opzionale, che è il numero di volte in cui si vuole dividere la stringa. ("Oooooh, argomenti opzionali..." imparerete come utilizzarli nelle vostre funzioni nel prossimo capitolo.)

Nota: ricercare con split
`anystring``.split (`*`separatore`*`, 1)` è un'utile tecnica quando volete cercare in una stringa la presenza di una particolare sottostringa e quindi utilizzare tutto ciò che c'è prima di detta sottostringa (che va a finire nel primo elemento della lista ritornata) e tutto quello che c'è dopo (che va a finire nel secondo elemento).

Ulteriori letture

- Python Knowledge Base risponde a domande comuni sulle stringhe e molti esempi di codice che usano le stringhe.
- *Python Library Reference* riassume tutti i metodi delle stringhe.
- *Python Library Reference* documenta il modulo `string`.
- *The Whole Python FAQ* spiega perché `join` è un metodo di string invece che di list.

2.15. Sommario

Il programma `odbchelper.py` ed il suo output dovrebbero ora esservi perfettamente chiari.

Esempio 2.38. `odbchelper.py`

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Esempio 2.39. Output di `odbchelper.py`

```
server=mpilgrim;uid=sa;database=master;pwd=secret
```

Prima di immergerci nel prossimo capitolo, assicuratevi di essere a vostro agio nel fare le seguenti cose:

- Usare la IDE di Python per provare le espressioni interattivamente
- Scrivere moduli Python così che possano essere eseguiti anche come programmi autonomi, almeno per fare dei test
- Importare dei moduli e chiamarne le funzioni
- Dichiarare funzioni ed usare le `doc string` , le variabili locali e la corretta indentazione

- Definire dizionari, tuple e liste
- Accedere ad attributi e metodi di ogni oggetto, incluse stringhe, liste, dizionari, funzioni e moduli
- Concatenare valori attraverso la formattazione di stringhe
- Mappare liste in altre liste utilizzando le list comprehension
- Dividere stringhe in liste e concatenare liste in stringhe

---

[1] Diversi linguaggi di programmazione definiscono gli "oggetti" in modi diversi. In alcuni, significa che *tutti* gli oggetti *devono* avere attributi e metodi; in altri, significa che da tutti gli oggetti sono derivabili altre classi. In Python, la definizione è meno ristretta; alcuni oggetti non hanno né attributi né metodi (ne parleremo più avanti in questo capitolo), e non non sono derivabili classi da tutti gli oggetti (altre informazioni su quest'argomento nel capitolo 3). Ma ogni cosa è un oggetto, nel senso che può essere assegnato ad una variabile o passato come argomento ad una funzione (più dettagli nel capitolo 2).

[2] In verità, è più complicato di così. Le chiavi di un dizionario devono essere immutabili. Le stesse tuple sono immutabili, ma se avete una tupla di liste, questa è mutabile e quindi non è sicuro usarla come chiave di un dizionario. Solo tuple di stringhe, numeri o altre tuple sicure possono essere usate come chiavi per un dizionario.

Capitolo 3. La potenza dell'introspezione

3.1. Immergersi

Questo capitolo tratta di uno dei punti di forza di Python: l'introspezione. Come sapete, ogni cosa in Python è un oggetto e l'introspezione è del codice che vede altri moduli e funzioni in memoria come se fossero oggetti, ottenendo informazioni su di loro e manipolandoli. Andando avanti, definiremo delle funzioni senza nome, chiameremo funzioni con gli argomenti nell'ordine sbagliato e referenzieremo funzioni il cui nome non ci sarà mai noto.

Qui abbiamo un programma Python completo e funzionante. Dovreste essere in grado di capire un bel po' di cose solamente guardando l'esempio. Le linee numerate illustrano concetti trattati nella sezione Conoscere Python. Non preoccupatevi se il resto del codice sembra intimidatorio; imparerete tutto nel corso di questo capitolo.

Esempio 3.1. `apihelper.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
def help(object, spacing=10, collapse=1): (1) (2) (3)
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                    for method in methodList])

if __name__ == "__main__":               (4) (5)
    print help.__doc__
```

(1) Questo modulo ha una funzione, `help`. Come descritto nella sua dichiarazione, prende in ingresso tre parametri: `object`, `spacing` e `collapse`. Gli ultimi due sono parametri opzionali, come vedremo brevemente.

(2) La funzione `help` ha una `doc string` multiriga che succintamente descrive lo scopo della funzione. Notate che nessun valore di ritorno è menzionato; questa funzione sarà usata solamente per i suoi effetti, non per il suo valore.

(3) Il codice interno alla funzione è indentato.

(4) Il `if __name__` (trucco) permette al programma di fare qualcosa di utile anche quando viene eseguito da solo, senza comunque interferire con il suo utilizzo come modulo per altri programmi. In questo caso, il programma semplicemente stampa la `doc string` della funzione `help`.

(5) Nelle istruzioni `if` usate `==` per i confronti, le parentesi non sono richieste.

La funzione `help` è realizzata per essere utilizzata da voi, i programmatori, mentre lavorate nella IDE di Python. Prende ogni oggetto che abbia funzioni o metodi (come un modulo, che ha delle funzioni, o una lista, che ha dei metodi) e ne stampa le funzioni con le loro stringhe di documentazione.

Esempio 3.2. Esempio di utilizzo di `apihelper.py`

```
>>> from apihelper import help
>>> li = []
>>> help(li)
```

```
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

Di norma l'output è formattato per essere facilmente leggibile. Le stringhe di documentazione multilinea sono compattate in una singola lunga riga, ma questa opzione può essere cambiata specificando `0` per l'argomento *`collapse`*. Se i nomi delle funzioni sono più lunghi di 10 caratteri, potete specificare un valore più grande per l'argomento *`spacing`* in modo da rendere l'output più semplice da leggere.

Esempio 3.3. Utilizzo avanzato di `apihelper.py`

```
>>> import odbchelper
>>> help(odbchelper)
buildConnectionString Build a connection string from a dictionary Returns string.
>>> help(odbchelper, 30)
buildConnectionString          Build a connection string from a dictionary Returns string.
>>> help(odbchelper, 30, 0)
buildConnectionString          Build a connection string from a dictionary

    Returns string.
```

3.2. Argomenti opzionali ed argomenti con nome

Python permette agli argomenti delle funzioni di avere un valore predefinito; se la funzione è chiamata senza l'argomento, l'argomento prende il suo valore predefinito. Inoltre gli argomenti possono essere specificati in qualunque ordine usando gli argomenti con nome. Le stored procedure in SQL Server Transact/SQL possono farlo; se siete dei guru nella programmazione via script di SQL Server potete saltare questa parte.

Esempio 3.4. `help`, una funzione con due argomenti opzionali

```
def help(object, spacing=10, collapse=1):
```

`spacing` e `collapse` sono opzionali perché hanno un valore predefinito. `object` è richiesto perché non ha alcun valore predefinito. Se `help` viene chiamata con un solo argomento, `spacing` assumerà il valore `10` e `collapse` assumerà valore `1`. Se `help` viene chiamata con due argomenti, `collapse` continuerà a valere `1`.

Diciamo che volete specificare un valore per `collapse`, ma volete accettare il valore predefinito per `spacing`. Nella maggior parte dei linguaggi sareste sfortunati, perché dovreste chiamare la funzione con tre argomenti. Ma in Python gli argomenti possono essere specificati per nome, in qualsiasi ordine.

Esempio 3.5. Chiamate valide per `help`

```
help(odbchelper)                   (1)
help(odbchelper, 12)               (2)
help(odbchelper, collapse=0)       (3)
help(spacing=15, object=odbchelper) (4)
```

(1) Con un solo argomento, `spacing` prende il suo valore predefinito pari a `10` e `collapse` prende il suo valore predefinito che è `1`.

(2) Con due argomenti, `collapse` prende il suo valore predefinito che è `1`.

(3) Qui state nominando l'argomento `collapse` esplicitamente e specificando il suo valore. `spacing` continua ad avere il suo valore predefinito `10`.

(4) Anche gli argomenti richiesti (come `object`, che non ha un valore predefinito) possono essere nominati e gli argomenti con nome possono apparire in qualsiasi ordine.

Può sembrare estremamente forzato fino a che non realizzate che gli argomenti sono semplicemente un dizionario. Il meccanismo "normale", di chiamata a funzione senza nomi di argomenti è solamente una scorciatoia in cui è Python ad abbinare i valori con i nomi degli argomenti, nell'ordine in cui sono stati specificati nella dichiarazione. Nella maggior parte dei casi, chiamerete le funzioni nel modo "normale", ma avrete sempre a disposizione questa flessibilità aggiuntiva se ne avrete bisogno.

> **Nota: La chiamata a funzione è flessibile**
> L'unica cosa che dovete fare per chiamare una funzione è specificare un valore (in qualche modo) per ogni argomento richiesto; il modo e l'ordine in cui lo fate è a vostra discrezione.

Ulteriori letture

- *Python Tutorial* discute esattamente quando e come argomenti predefiniti vengano valutati, che ha importanza quando il valore predefinito è una lista o un'espressione con effetti collaterali.

3.3. `type`, `str`, `dir`, ed altre funzioni built−in

Python ha un piccolo insieme di funzioni built−in estremamente utili. Tutte le altre funzioni sono suddivise in moduli. Si tratta di una decisione coscienziosa, per preservare il nucleo del linguaggio dall'essere soffocato come in moltri altri linguaggi di script (coff coff,Visual Basic).

La funzione `type` restituisce il tipo di dato di ogni oggetto arbitrario. I tipi possibili sono elencati nel modulo `types`. È utile per le funzioni helper che possono gestire diversi tipi di dati.

Esempio 3.6. Introduzione a `type`

```
>>> type(1)           (1)
<type 'int'>
>>> li = []
>>> type(li)          (2)
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)  (3)
<type 'module'>
>>> import types      (4)
>>> type(odbchelper) == types.ModuleType
1
```

(1) `type` prende qualunque cosa in ingresso e ne ritorna il suo tipo. E voglio proprio dire qualunque cosa. Interi, stringhe, liste, dizionari, tuple, funzioni, classi, moduli ed anche tipi.

(2) `type` può prendere una variabile e ritornare il suo tipo.

(3) `type` funziona anche sui moduli.

(4) Puoi usare le costanti nel modulo `types` per confrontare i tipi degli oggetti. Questo è ciò che fa la funzione `help`, come vedremo brevemente.

La funzione `str` forza i dati in una stringa. Qualunque tipo di dato può essere forzato ad essere una stringa.

Esempio 3.7. Introduzione a `str`

```
>>> str(1)          (1)
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen)   (2)
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper) (3)
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)       (4)
'None'
```

(1) Per semplici tipi di dati come gli interi, vi aspettereste che `str` funzioni, perché quasi ogni linguaggio ha una funzione per convertire un intero in una stringa.

(2) Ad ogni modo, `str` funziona su ogni oggetto di ogni tipo. Qui è al lavoro su una lista che abbiamo costruito un po' alla volta.

(3) `str` funziona anche sui moduli. Notate che la rappresentazione in stringa del modulo, include il percorso del modulo su disco, quindi i vostri risultati saranno differenti.

(4) Un sottile ma importante comportamento di `str` è che funziona anche su `None`, il valore nullo in Python. Ritorna la stringa `'None'`. Lo useremo a nostro vantaggio nella funzione `help`, come vedremo brevemente.

Il cuore della nostra funzione `help` è la potente funzione `dir`. `dir` ritorna una lista di attributi e metodi di ogni oggetto: moduli, funzioni, stringhe, liste, dizionari... praticamente qualunque cosa.

Esempio 3.8. Introduzione a `dir`

```
>>> li = []
>>> dir(li)           (1)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d)            (2)
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault', 'update', 'values']
>>> import odbchelper
>>> dir(odbchelper)   (3)
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```

(1) `li` è una lista, quindi `dir(li)` ritorna una lista di tutti i metodi di una lista. Notate che la lista ritornata contiene i nome dei metodi sotto forma di stringhe, non come veri e propri metodi.

(2) `d` è un dizionario, quindi `dir(d)` ritorna una lista dei nomi dei metodi di un dizionario. Come minimo uno di questi, `keys`, dovrebbe suonarvi familiare.

(3) Qui comincia a farsi interessante. `odbchelper` è un modulo, quindi `dir(odbchelper)` ritorna una lista con ogni genere di cose definite nel modulo, inclusi gli attributi built−in, come `__name__` e `__doc__` e qualunque altro attributo e metodo voi definiate. In questo caso, `odbchelper` ha solamente un metodo definito dall'utente, la funzione `buildConnectionString` che abbiamo studiato nella sezione Conoscere Python.

Infine, la funzione `callable` (invocabile ndr) prende ogni tipo di oggetto e ritorna `1` se l'oggetto può essere invocato, `0` in caso contrario. Gli oggetti invocabili includono funzioni, metodi di classi ed anche le classi stesse. (Ulteriori informazioni sulle classi nel capitolo 4).

Esempio 3.9. Introduzione a `callable`

```
>>> import string
>>> string.punctuation           (1)



'!""#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.join                  (2)
<function join at 00C55A7C>
>>> callable(string.punctuation) (3)
0
>>> callable(string.join)        (4)
1
>>> print string.join.__doc__    (5)
join(list [,sep]) -> string

    Return a string composed of the words in list, with
    intervening occurrences of sep.  The default separator is a
    single space.

    (joinfields and join are synonymous)
```

(1) Le funzioni nel modulo `string` sono deprecate (per quanto un sacco di persone continuino ad usare la funzione `join`), ma il modulo contiene molte costanti utili come `string.punctuation`, che contiene tutti i caratteri di punteggiatura standard.

(2) `string.join` è una funzione che concatena una lista di stringhe.

(3) `string.punctuation` non è invocabile; è una stringa. (Una stringa ha dei metodi invocabili, ma la stringa stessa non è affatto invocabile.)

(4) `string.join` è invocabile; è una funzione che prende due argomenti.

(5) Ogni oggetto invocabile può avere una stringa di documentazione. Usando la funzione `callable` su ognuno degli attributi di un oggetto, possiamo determinare quali attributi ci interessano (metodi, funzioni, classi) e quali vogliamo ignorare (costanti, *etc.*), senza sapere nulla in anticipo a proposito dell'oggetto.

`type`, `str`, `dir` e tutto il resto delle funzioni built−in di Python sono raggruppate in un modulo speciale chiamato `__builtin__`. (Cioè con due sottolineature prima e dopo.) Se vi può aiutare, potete pensare che Python automaticamente esegua `from __builtin__ import *` all'avvio, comando che importa tutte le funzioni "built−in" nello spazio dei nomi locale, così che possiate usarle direttamente.

Il vantaggio di vederla in questo modo sta nel fatto che potete accedere a tutte le funzioni ed agli attributi built−in sotto forma di gruppo, ricavando le informazioni dal modulo `__builtin__`. Indovinate, abbiamo una funzione per fare questo, è chiamata `help`. Provate voi stessi e scorrete la lista; ci immergeremo in alcune delle funzioni più importanti più tardi. (Alcune delle classi di errore built−in, come `AttributeError`, dovrebbero già suonarvi familiari.)

Esempio 3.10. Attributi e funzioni built−in

```
>>> from apihelper import help
>>> import __builtin__
>>> help(__builtin__, 20)
ArithmeticError      Base class for arithmetic errors.
AssertionError       Assertion failed.
AttributeError       Attribute not found.
EOFError             Read beyond end of file.
EnvironmentError     Base class for I/O related errors.
```

```
Exception           Common base class for all exceptions.
FloatingPointError  Floating point operation failed.
IOError             I/O operation failed.

[...snip...]
```

> **Nota: Python è auto−documentante**
> Python è dotato di un eccellente manuale di riferimento, che dovreste usare spesso per conoscere tutti i moduli che Python ha da offrire. Ma mentre nella maggior parte dei linguaggi vi ritroverete a dover tornare spesso sul manuale (o pagine man o ... Dio vi aiuti, MSDN) per ricordare come si usano questi moduli, Python è largamente auto−documentante.

Ulteriori letture

- *Python Library Reference* documenta tutte le funzioni built−in e tutte le eccezioni built−in.

3.4. Ottenere riferimenti agli oggetti usando `getattr`

Già sapete che le funzioni in Python sono oggetti. Quello che non sapete è che potete ottenere un riferimento da una funzione senza conoscere il suo nome fino al momento dell'esecuzione, utilizzando la funzione `getattr`.

Esempio 3.11. Introduzione a `getattr`

```
>>> li = ["Larry", "Curly"]
>>> li.pop                      (1)
<built−in method pop of list object at 010DF884>
>>> getattr(li, "pop")          (2)
<built−in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe") (3)
>>> li
["Larry", "Curly", "Moe"]
>>> getattr({}, "clear")        (4)
<built−in method clear of dictionary object at 00F113D4>
>>> getattr((), "pop")          (5)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'pop'
```

(1) Così si ottiene un riferimento al metodo `pop` della lista. Notate che questo non significa chiamare il metodo `pop`; per farlo bisogna scrivere `li.pop()`. Questo è proprio il metodo.

(2) Anche in questo modo si ottiene un riferimento al metodo `pop`, ma questa volta il nome del metodo è specificato come argomento di tipo stringa per la funzione `getattr`. `getattr` è una funzione built−in incredibilmente utile, che ritorna ogni attributo di ogni oggetto. in questo caso, l'oggetto è una lista e l'attributo è il metodo `pop`.

(3) Nel caso non abbiate ancora bene afferrato quanto sia utile, provate questo: il valore di ritorno di `getattr` *è* un metodo, che potete chiamare proprio come se scriveste `li.append("Moe")` direttamente. Ma non avete chiamato la funzione direttamente; avete invece specificato il nome della funzione.

(4) `getattr` funziona anche sui dizionari.

(5) In teoria `getattr` funzionerebbe anche sulle tuple, eccetto per il fatto che le tuple non hanno metodi, così `getattr` genererà un'eccezione a prescindere da quale nome di attributo le passiate.

`getattr` non è fatta solamente per i tipi di dato built−in. Funziona anche con i moduli.

Esempio 3.12. `getattr` in `apihelper.py`

```
>>> import odbchelper
>>> odbchelper.buildConnectionString             (1)
<function buildConnectionString at 00D18DD4>
>>> getattr(odbchelper, "buildConnectionString") (2)
<function buildConnectionString at 00D18DD4>
>>> object = odbchelper
>>> method = "buildConnectionString"
>>> getattr(object, method)                      (3)
<function buildConnectionString at 00D18DD4>
>>> type(getattr(object, method))                (4)
<type 'function'>
>>> import types
>>> type(getattr(object, method)) == types.FunctionType
1
>>> callable(getattr(object, method))            (5)
1
```

(1) Così si ottiene un riferimento alla funzione `buildConnectionString` nel modulo `odbchelper` che abbiamo studiato nella sezione Conoscere Python. (L'indirizzo esadecimale che vedete è specifico della mia macchina, il vostro output sarà diverso.)

(2) Usando `getattr` possiamo ottenere lo stesso riferimento alla stessa funzione. In generale, `getattr(`*`oggetto`*`, "`*`attributo`*`")` è equivalente a *`oggetto.attributo`*. Se *`oggetto`* è un modulo, allora *`attributo`* può essere qualunque cosa definita nel modulo: una funzione, una classe o una variabile globale.

(3) E questo è ciò che usiamo nella funzione `help`. `oggetto` è passato alla funzione come argomento, `method` è una stringa che rappresenta il nome di un metodo o di una funzione.

(4) In questo caso, `method` è il nome di una funzione e lo possiamo dimostrare ottenendone il `type`.

(5) Siccome `method` è una funzione, allora è invocabile.

3.5. Filtrare le liste

Come già si è detto, Python offre potenti strumenti per mappare delle liste in liste corrispondenti, per mezzo delle "list comprehension". Queste possono essere combinate con un meccanismo di filtro, facendo in modo che alcuni elementi delle liste vengano mappati ed altri vengano ignorati completamente.

Esempio 3.13. Sintassi per filtrare una lista

```
[mapping-expression for element in source-list if filter-expression]
```

Questa è un'estensione della "list comprehensions" che tutti conosciamo ed amiamo. I primi due terzi sono uguali; l'ultima parte, a cominciare dalla parola chiave `if`, è l'espressione di filtro. Un'espressione di filtro può avere qualunque valore, purché abbia come risultato un valore vero o falso (il che in Python comprende quasi tutto). Ogni elemento per cui l'espressione di filtro risulti vera viene inclusa nella lista risultante. Tutti gli altri elementi sono ignorati, quindi non sono mai processati dall'espressione di mappatura e non sono inclusi nella lista risultante.

Esempio 3.14. Introduzione alle liste filtrate

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]       (1)
['mpilgrim', 'foo']
>>> [elem for elem in li if elem != "b"]         (2)
['a', 'mpilgrim', 'foo', 'c', 'd', 'd']
```

```
>>> [elem for elem in li if li.count(elem) == 1] (3)
['a', 'mpilgrim', 'foo', 'c']
```

(1) L'espressione di mappatura in questo caso è semplice (restituisce il valore di ogni elemento), perciò concentriamoci sull'espressione di filtro; Python esegue un ciclo sulla lista e ciascun elemento attraversa l'espressione di filtro. Se l'espressione di filtro è vera, l'elemento viene mappato ed il risultato dell'espressione di mappatura è incluso nella lista risultante. In questo caso si vogliono eliminare tutte le stringhe di un carattere, per ottenere solo le stringhe più lunghe.

(2) Qui stiamo filtrando per eliminare uno specifico valore, `b`. Da notare che questo filtra tutte le occorrenze di `b`, dato che ogni volta che appare tale valore l'espressione di filtro risulta falsa.

(3) La funzione `count` è un metodo che restituisce il numero di volte che un valore è presente in una lista. Si potrebbe pensare che un tale filtro elimini i duplicati da una lista, restituendo una lista che contenga solo una copia di ogni valore della lista originale. Ma non è così, perché i valori che appaiono due volte nella lista originale (in questo caso `b` e `d`)vengono esclusi completamente. Ci sono modi per eliminare i duplicati da una lista, ma filtrare la lista non è tra questi.

Esempio 3.15. Filtrare una lista in `apihelper.py`

```
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
```

Questo appare complicato ed in effetti lo è, ma la struttura di base è la stessa. L'intera espressione di filtro restituisce una lista che viene assegnata alla variabile `methodList`. La prima metà dell'espressione è la componente di mappatura. L'espressione di mappatura è un'espressione di uguaglianza; restituisce il valore di ciascun elemento. L'espressione `dir(object)` restituisce una lista di attributi e metodi di `object`; questa è la lista che stiamo mappando. Quindi, l'unica parte veramente nuova è l'espressione di filtro che segue la parola chiave `if`.

L'espressione di filtro spaventa un po', ma solo a prima vista. Si è già detto dei metodi `callable`, `getattr`, e di `in`. Come si è visto nella sezione precedente, l'espressione `getattr(object, method)` restituisce una funzione se `object` è un modulo e `method` è il nome di una funzione del modulo.

Quindi questa espressione prende un oggetto che abbia un nome, ne estrae la lista dei nomi degli attributi, dei metodi, delle funzioni e di qualche altra cosa, quindi filtra la lista per eliminare tutto ciò che non ci interessa. La scrematura della lista viene fatta partendo dal nome di ogni attributo/funzione/metodo ed ottenendo un riferimento all'oggetto corrispondente, attraverso la funzione `getattr`. Quindi si controlla se l'oggetto è invocabile, come è il caso per i metodi e le funzioni, sia quelle built−in (come il metodo `pop` di una lista) sia quelle definite dall'utente (come la funzione `buildConnectionString` del modulo `odbchelper`). Si scartano gli altri attributi, come l'attributo `__name__` che è in ogni modulo.

Ulteriori letture

- *Python Tutorial* tratta di un altro modo di filtrare le liste usando la funzione built−in `filter`.

3.6. Le particolarità degli operatori and e or

In Python, gli operatori `and` e `or` eseguono le operazioni di logica booleane che ci si aspetta dal loro nome, ma non restituiscono valori booleani; essi restituiscono invece il valore di uno degli elementi che si stanno confrontando.

Esempio 3.16. Introduzione all'operatore `and`

```
>>> 'a' and 'b'         (1)
'b'
```

```
>>> '' and 'b'          (2)
''
>>> 'a' and 'b' and 'c' (3)
'c'
```

(1) Quando si usa l'operatore `and`, gli elementi dell'espressione vengono valutati in un contesto booleano da sinistra a destra. I valori `0`, `''`, `[]`, `()`, `{}`, e `None` sono considerati falsi in un contesto booleano; ogni altro valore è considerato vero.[3] Se tutti gli elementi hanno valore vero in contesto booleano, l'operatore `and` restituisce il valore dell'ultimo elemento. In questo caso, l'operatore `and` valuta `'a'`, che è vero, quindi valuta `'b'`, che è vero, ed infine ritorna `'b'`.

(2) Se uno degli elementi ha valore falso in contesto booleano, l'operatore `and` restituisce il valore del primo di tali elementi. In questo caso, `''` è il primo valore falso.

(3) Tutti gli elementi hanno valore vero, per cui l'operatore `and` restituisce il valore dell'ultimo elemento, `'c'`.

Esempio 3.17. Introduzione all'operatore `or`

```
>>> 'a' or 'b'          (1)
'a'
>>> '' or 'b'           (2)
'b'
>>> '' or [] or {}      (3)
{}
>>> def sidefx():
...     print "in sidefx()"
...     return 1
>>> 'a' or sidefx()     (4)
'a'
```

(1) Quando si usa l'operatore `or`, gli elementi sono valutati in contesto booleano da sinistra a destra, come per l'operatore `and`. Se uno degli elementi ha valore vero, l'operarore `or` ritorna immediatamente quel valore. Nel nostro caso, `'a'` è il primo elemento con valore vero.

(2) L'operatore `or` valuta `''`, che è falso, quindi `'b'`, che è vero e restituisce `'b'`.

(3) Se tutti gli elementi hanno valore falso, l'operatore `or` restituisce l'ultimo valore. L'operatore `or` valuta `''`, che è falso, quindi `[]`, che è falso, quindi `{}`, che è falso e restituisce `{}`.

(4) Si noti che l'operatore `or` elabora gli elementi solamente finché non ne trova uno che sia vero in contesto booleano ed ignora i rimanenti. Questa distinzione è importante se alcuni elementi dell'espressione hanno effetti collaterali. Nel nostro caso, la funzione `sidefx` non è mai chiamata, perché l'operatore `or` valuta `'a'`, che è vero e restituisce immediatamente `'a'`.

Gli hakers del linguggio C hanno sicuramente familiarità con il costrutto *`bool ? a : b`*, che restituisce il valore `a` se *`bool`* è vero, altrimenti restituisce `b`. In ragione del modo in cui `and` ed `or` funzionano in Python, è possibile ottenere lo stesso risultato.

Esempio 3.18. Introduzione al trucchetto `and-or`

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b (1)
'first'
>>> 0 and a or b (2)
'second'
```

(1) Questa sintassi sembra molto simile al costrutto *`bool ? a : b`* in C. L'espressione nella sua globalità è valutata da sinistra a destra, quindi l'operatore `and` è valutato per primo. L'espressione `1 and 'first'`

restituisce `'first'`, quindi l'espressione `'first' or 'second'` restituisce `'second'`.

(2) L'espressione `0 and 'first'` restituisce `0`, quindi l'espressione `0 or 'second'` restituisce `'second'`.

Tuttavia, visto che questa espressione in Python è semplice logica booleana, non un costrutto speciale del linguaggio, vi è una differenza molto ma molto ma molto importante tra questo trucchetto `and-or` e l'espressione *`bool ? a : b`* in C. Se il valore di `a` è falso, l'espressione non funzionerà come ci si aspetta. (Si capisce che ci sono cascato? Più di una volta?)

Esempio 3.19. Quando il trucchetto `and-or` fallisce

```
>>> a = ""
>>> b = "second"
>>> 1 and a or b (1)
'second'
```

(1) Dato che `a` è una stringa vuota, che Python considera falsa in un contesto booleano, `1 and ''` restituisce `''`, quindi `'' or 'second'` restituisce `'second'`. Oops! non è quello che si voleva.

Importante: Per usare and-or in modo efficace
Il trucchetto `and-or`, cioè *`bool and a or b`*, non funziona come il costrutto C *`bool ? a : b`* quando `a` ha un valore falso in un contesto booleano.

Dunque la parte veramente complicata del trucchetto `and-or` è essere sicuri che il valore di `a` non sia falso. Un modo comune per constatarlo è trasformare `a` in `[a]` e `b` in `[b]` e poi considerare il primo elemento della lista risultante, che sarà `a` oppure `b`.

Esempio 3.20. Usare il trucchetto `and-or` in modo sicuro

```
>>> a = ""
>>> b = "second"
>>> (1 and [a] or [b])[0] (1)
''
```

(1) Dato che `[a]` non è una lista vuota, il suo valore non è mai falso. Anche se `a` vale `0` oppure `''` o un altro valore falso, la lista `[a]` ha valore vero perché ha un elemento.

A questo punto, questo trucco sembra troppo complicato per quello che vale. Si potrebbe dopotutto ottenere la stessa cosa con un'istruzione `if`, perciò perché darsi tutto questo daffare? Bene, in molti casi si dovrà scegliere tra due valori costanti, per cui si può usare la versione più semplice del trucco senza preoccupazione, perché si sa che un elemento avrà sempre valore vero. Anche nel caso si debba usare la versione più complicata, ci sono buone ragioni per farlo; ci sono alcuni casi in Python in cui le istruzioni `if` non sono consentite, come per esempio in una funzione `lambda`.

Ulteriori letture

- Python Cookbook tratta di alternative al trucchetto `and-or` .

3.7. Usare le funzioni `lambda`

Python supporta un'interessante sintassi che vi permette di definire al volo delle piccole funzioni di una sola riga. Derivate dal Lisp, queste funzioni `lambda` possono essere usate ovunque sia richiesta una funzione.

Esempio 3.21. Introduzione alle funzioni `lambda`

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2  (1)
>>> g(3)
6
>>> (lambda x: x*2)(3) (2)
6
```

(1) Questa è una funzione `lambda` equivalente alla funzione `f()`. Notare la sintassi abbreviata: non ci sono parentesi attorno alla lista degli argomenti e la parola chiave `return` è mancante (implicita, poiché l'intera funzione può essere solamente composta da una espressione). Inoltre, la funzione `lambda` non ha nome, ma può essere chiamata attraverso la variabile a cui è stata assegnata.

(2) Potete usare anche una funzione `lambda` senza assegnarla ad una variabile. Non è la cosa più utile del mondo, ma dimostra che una funzione `lambda` equivale ad una funzione "inline" del linguaggio C.

Per generalizzare, una funzione `lambda` è una funzione che prende qualunque numero di argomenti (inclusi quelli opzionali) e ritorna il valore di una singola espressione. Le funzioni `lambda` non possono contenere comandi e neppure possono contenere più di una espressione. Non cercate di metterci dentro troppa roba: se avete bisogno di uno strumento più complesso, definite una funzione normale e fatela lunga quanto volete.

> **Nota: lambda è opzionale**
> l'utilizzo o meno delle funzioni `lambda` è questione di stile. Usarle non è mai necessario; in ogni caso in cui vengono usate è possibile definire una funzione normale e usarla al posto della funzione `lambda`. Le funzioni `lambda` sono comode, ad esempio, nei casi in cui si voglia incapsulare codice specifico, non riutilizzato, senza riempire il programma di piccolissime funzioni.

Esempio 3.22. funzioni lambda `lambda` in `apihelper.py`

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

Notiamo alcune cose in questo esempio. Primo, stiamo usando la forma più semplice del trucco `and-or`, il che va bene, perché una funzione `lambda` è sempre vera in un contesto booleano. Ciò non significa che una funzione `lambda` non possa ritornare un valore falso. La funzione è sempre vera; è il suo valore di ritorno, il risultato, che può essere qualunque cosa.

Secondo, stiamo usando la funzione `split` senza argomenti. L'avete già vista usare con 1 o 2 argomenti, ma senza argomenti spezza la stringa sui caratteri di spazio.

Esempio 3.23. `split` senza argomenti

```
>>> s = "this   is\na\ttest"  (1)
>>> print s
this   is
a       test
>>> print s.split()           (2)
['this', 'is', 'a', 'test']
>>> print " ".join(s.split()) (3)
'this is a test'
```

(1) Questa è una stringa su più linee, definita da caratteri di escape anziché triple virgolette. `\n` è il ritorno a capo; `\t` è il carattere tab.

(2) `split` senza argomenti spezza negli spazi vuoti. Così tre spazi, un ritorno a capo e un carattere di tabulazione vengono trattati nello stesso modo.

(3) Potete collassare gli spazi dividendo una stringa e riattaccandola con un carattere di spazio come delimitatore. Questo è ciò che fa la funzione `help` per collassare una stringa di più linee in una linea sola.

Cosa fa quindi la funzione `help` con queste funzioni `lambda`, `split` e trucchi `and-or`?

Esempio 3.24. Assegnare una funzione a una variabile

```
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
```

`processFunc` adesso è una funzione, ma quale funzione sia dipende dal valore della variabile `collapse`. Se `collapse` è vera, `processFunc(`*`string`*`)` collasserà gli spazi vuoti, altrimenti `processFunc(`*`string`*`)` ritornerà il suo argomento senza alcuna modifica.

Per fare la stessa cosa in un linguaggio meno robusto, come Visual Basic, dovremmo probabilmente creare una funzione che riceve una stringa e l'argomento *`collapse`*, usa un'istruzione `if` per decidere se collassare gli spazi vuoti o no, quindi ritorna il valore appropriato. Ciò sarebbe inefficiente, perché la funzione dovrebbe gestire ogni possibile caso; ogni volta che viene chiamata, dovrebbe decidere se collassare gli spazi prima di restituire il valore desiderato. In Python, potete prendere questa logica di decisione e portarla fuori dalla funzione, definendo una funzione `lambda` cucita su misura per fare ciò che volete e soltanto quello. Il sistema è più efficiente, più elegante e meno portato al genere di errore dovuto all'errata disposizione degli argomenti di chiamata.

Ulteriori letture

- Python Knowledge Base discute l'utilizzo delle funzioni `lambda` per chiamare le funzioni indirettamente.
- *Python Tutorial* mostra come accedere alle variabili esterne dall'interno di una funzione `lambda` . La PEP 227 spiega come questo cambierà nelle future versioni di Python.
- *The Whole Python FAQ* ha esempi di codice offuscato, composto da una funzione `lambda`.

3.8. Unire il tutto

L'ultima riga di codice, l'unica che ancora non abbiamo analizzato, è quella che svolge l'intero lavoro. Ma adesso il lavoro è semplice, perché ogni cosa di cui abbiamo bisogno è già impostata come serve a noi. Tutte le tessere sono al loro posto; è ora di buttarle giù.

Esempio 3.25. Il succo di `apihelper.py`

```
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                    for method in methodList])
```

Notate che questo è un solo comando, suddiviso su più righe e non usa il carattere di continuazione ("\"). Ricordate quando dicevo che alcune espressioni possono essere suddivise in più righe senza usare un backslash? La list comprehension è una di queste, in quanto l'intera espressione è contenuta tra parentesi quadre.

Adesso cominciamo dalla fine e lavoriamo all'indietro. Il

```
for method in methodList
```

ci mostra che questa è una list comprehension. Come sappiamo, `methodList` è una lista di tutti i metodi di `object` ai quali siamo interessati. Quindi stiamo eseguendo un ciclo lungo questa lista con un certo `metodo`.

Esempio 3.26. Ottenere dinamicamente una `doc string`

```
>>> import odbchelper
>>> object = odbchelper                    (1)
>>> method = 'buildConnectionString'       (2)
>>> getattr(object, method)                (3)
<function buildConnectionString at 010D6D74>
>>> print getattr(object, method).__doc__ (4)
Build a connection string from a dictionary of parameters.

    Returns string.
```

(1) Nella funzione `help`, `object` è l'oggetto dal quale stiamo ottenendo l'aiuto, passato come argomento.

(2) Siccome stiamo eseguendo un ciclo attraverso `methodList`, `method` è il nome del metodo corrente.

(3) Usando la funzione `getattr`, stiamo ottenendo un riferimento alla funzione *`method`* nel modulo *`object`*.

(4) Adesso stampare l'attuale `doc string` del metodo è semplice.

Il prossimo pezzo del puzzle è l'utilizzo di `str` sulla stringa di documentazione. Come forse ricorderete, `str` è una funzione built−in che forza i dati in una stringa. Ma una stringa di documentazione è sempre una stringa dunque perché preoccuparsi di usare la funzione `str`? La risposta è che non tutte le funzioni hanno una stringa di documentazione e se non ce l'hanno, il loro attributo `__doc__` vale `None`.

Esempio 3.27. Perchè usare `str` su una stringa di documentazione?

```
>>> >>> def foo(): print 2
>>> >>> foo()
2
>>> >>> foo.__doc__     (1)
>>> foo.__doc__ == None (2)
1
>>> str(foo.__doc__)    (3)
'None'
```

(1) Possiamo facilmente definire una funzione che non ha una `doc string`, quindi il suo attributo `__doc__` vale `None`. Per fare ancora più confusione, se valutate l'attributo `__doc__` direttamente, la IDE di Python non stampa nulla, che ha senso se ci pensate, ma non è di grande aiuto.

(2) Potete verificare che il valore dell'attributo `__doc__` è `None` confrontandolo direttamente.

(3) Quando la funzione `str` prende in ingresso un valore nullo, ne ritorna la sua rappresentazione in forma di stringa, `'None'`.

> **Nota: Python contro SQL: confrontare valori nulli**
> In SQL, dovete usare `IS NULL` invece di `= NULL` per confrontare un valore nullo. In Python, potete usare indifferentemente `== None` o `is None`, ma `is None` è più efficiente.

Adesso che ci siamo garantiti di avere una stringa, possiamo passare la stringa a `processFunc`, che abbiamo già definito come una funzione che comprime o meno gli spazi. Ora vedrete perché è stato importante usare `str` per convertire un valore `None` in una rappresentazione in stringa. `processFunc` assume di ricevere un argomento di tipo stringa e chiamare il suo metodo `split`, che restituirebbe un errore se passassimo `None`, perché `None` non ha un metodo `split`.

Facendo un ulteriore passo indietro, possiamo notare che stiamo usando nuovamente la formattazione delle stringhe per concatenare il valore di ritorno di `processFunc` con il valore di ritorno del `metodo ljust`. Questo è un nuovo metodo delle stringhe che non abbiamo visto prima.

Esempio 3.28. Introduzione al metodo `ljust`

```
>>> s = 'buildConnectionString'
>>> s.ljust(30) (1)
'buildConnectionString         '
>>> s.ljust(20) (2)
'buildConnectionString'
```

(1) `ljust` riempie la stringa con spazi bianchi fino ad arrivare al valore assegnato alla lunghezza. È ciò che la funzione `help` usa per creare due colonne di output ed allineare tutte le stringhe di documentazione nella seconda colonna.

(2) Se il valore assegnato alla lunghezza è più piccolo dell'attuale lunghezza della stringa, `ljust` semplicemente ritornerà la stringa senza alcuna modifica. Non tronca mai una stringa.

Abbiamo quasi finito. Dato il nome incolonnato dal metodo `ljust` e la (possibilmente priva di spazi bianchi) `doc string` della chiamata a `processFunc`, concateniamo i due ed otteniamo una singola stringa. Siccome stiamo mappando `methodList`, arriveremo ad ottenere una lista di stringhe. Usando il metodo `join` sulla stringa `"\n"`, concateniamo questa lista in una singola stringa, con ogni elemento della lista su una riga a se stante e stampiamo il risultato.

Esempio 3.29. Stampare una lista

```
>>> li = ['a', 'b', 'c']
>>> print "\n".join(li) (1)
a
b
c
```

(1) Si tratta inoltre di un utile trucco per il debugging quando state lavorando con le liste. Ed in Python, lavorerete continuamente con le liste.

Questo è l'ultimo pezzo del puzzle. Ora il codice dovrebbe essere chiaro.

Esempio 3.30. Il succo di `apihelper.py`, rivisitato

```
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                    for method in methodList])
```

3.9. Sommario

Il programma `apihelper.py` ed il suo output ora dovrebbero essere chiari.

Esempio 3.31. `apihelper.py`

```
def help(object, spacing=10, collapse=1):
    """Print methods and doc strings.
```

```
    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object) if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                     for method in methodList])

if __name__ == "__main__":
    print help.__doc__
```

Esempio 3.32. Output di `apihelper.py`

```
>>> from apihelper import help
>>> li = []
>>> help(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```

Prima di immergerci nel prossimo capitolo, assicuratevi di essere a vostro agio con i seguenti argomenti:

- Definire e chiamare funzioni con argomenti opzionali e con nome
- Usare `str` per forzare ogni valore arbitrario in una rappresentazione in stringa
- Usare `getattr` per ottenere riferimenti a funzioni ed altri attributi dinamicamente
- Estendere la sintassi delle list comprehension per filtrare una lista
- Riconoscere il trucchetto `and-or` ed usarlo in maniera sicura
- Definire funzioni `lambda`
- Assegnare funzioni a variabili e chiamare la funzione referenziando la variabile. Non posso sottolinearlo abbastanza: questo modo di pensare è vitale per migliorare la vostra comprensione di Python. Vedrete applicazioni più complesse di questo concetto nel corso del libro.

---

[3] Beh, quasi ogni cosa. In modo predefinito le istanze di una classe corrispondono a true in un contesto booleano, ma potete definire metodi speciali nella vostra classe per far valutare un'istanza in false. Imparerete tutto a proposito delle classi e dei loro metodi speciali nel capitolo 4.

Capitolo 4. Una struttura orientata agli oggetti

4.1. Immergersi

Questo capitolo, e saltuariamente anche i capitoli successivi, si occuperanno della programmazione ad oggetti in Python.

Eccovi un completo e funzionante programma in Python. Leggete la `doc string` del modulo, le classi e le funzioni, in modo da avere un'idea di cosa faccia e come lavori questo programma. Come al solito, non curatevi delle parti che non comprendete; il resto del capitolo è fatto per loro.

Esempio 4.1. `fileinfo.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
"""Framework for getting filetype-specific metadata.

Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key-value pairs for each piece of metadata.
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
```

```
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]): (1)
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

(1) L'output di questo programma dipende dai file presenti sul vostro disco. Per ottenere un output sensato dovrete modificare il percorso della directory per puntarne una sulla vostra macchina in cui siano presenti dei file MP3.

Esempio 4.2. Output di `fileinfo.py`

Questo è l'output che ho ottenuto sulla mia macchina. Il vostro output sarà differente, a meno che, per una qualche allarmante coincidenza, abbiate i miei stessi gusti in fatto di musica.

```
album=
artist=Ghost in the Machine
title=A Time Long Forgotten (Concept
genre=31
name=/music/_singles/a_time_long_forgotten_con.mp3
year=1999
comment=http://mp3.com/ghostmachine

album=Rave Mix
artist=***DJ MARY-JANE***
title=HELLRAISER****Trance from Hell
genre=31
name=/music/_singles/hellraiser.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Rave Mix
artist=***DJ MARY-JANE***
title=KAIRO****THE BEST GOA
```

```
genre=31
name=/music/_singles/kairo.mp3
year=2000
comment=http://mp3.com/DJMARYJANE

album=Journeys
artist=Masters of Balance
title=Long Way Home
genre=31
name=/music/_singles/long_way_home1.mp3
year=2000
comment=http://mp3.com/MastersofBalan

album=
artist=The Cynic Project
title=Sidewinder
genre=18
name=/music/_singles/sidewinder.mp3
year=2000
comment=http://mp3.com/cynicproject

album=Digitosis@128k
artist=VXpanded
title=Spinning
genre=255
name=/music/_singles/spinning.mp3
year=2000
comment=http://mp3.com/artists/95/vxp
```

4.2. Importare i moduli usando `from module import`

Python utilizza due modi per importare i moduli. Entrambi sono utili e dovreste sapere quando usarli. Il primo, `import module`, l'avete già visto nel capitolo 2. Il secondo, raggiunge lo stesso scopo, ma funziona in modo differente.

Esempio 4.3. Sintassi basilare di `from module import`

```
from UserDict import UserDict
```

Questa sintassi è simile a quella di `import module` che conoscete ed amate, ma con una importante differenza: gli attributi ed i metodi del modulo importato `types` vengono importati direttamente nel namespace locale, in modo tale che siano disponibili direttamente, senza qualificare il nome del modulo. Potete importare elementi individualmente od usare `from module import *` per importare qualsiasi cosa.

> **Nota: Python contro Perl: from module import**
> `from module import *` in Python è simile allo `use module` in Perl; `import module` nel Python è come il `require module` del Perl.

> **Nota: Python contro Java: from module import**
> `from module import *` in Python è come `import module.*` in Java; `import module` in Python è simile a `import module` del Java.

Esempio 4.4. `import module` *contro* `from module import`

```
>>> import types
```

```
>>> types.FunctionType            (1)
<type 'function'>
>>> FunctionType                  (2)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
NameError: There is no variable named 'FunctionType'
>>> from types import FunctionType (3)
>>> FunctionType                  (4)
<type 'function'>
```

(1) Il modulo `types` non contiene metodi, solo attributi per ogni oggetto tipo di Python. Notate che l'attributo, `FunctionType`, deve essere qualificato con il nome del modulo, in questo caso, `types`.

(2) `FunctionType` di per sé non è definito in questo namespace; esiste solo nel contesto di `types`.

(3) Questa sintassi importa l'attributo `FunctionType` dal modulo `types` direttamente nel namespace locale.

(4) Adesso `FunctionType` può essere utilizzato direttamente, senza riferirsi a `types`.

Quando dovreste usare `from` *`module`* `import`?

- Se dovete accedere ad attributi e metodi di frequente e non volete digitare il nome del modulo continuamente, usate `from` *`module`* `import`.
- Se volete importare selettivamente alcuni attributi e metodi ma non altri, usate `from` *`module`* `import`.
- Se il modulo contiene attributi e funzioni con lo stesso nome di altri del vostro modulo, dovete usare `import` *`module`* per evitare conflitti di nomi.

Per altri casi, è solo una questione di stile, in futuro vedrete codice Python scritto in entrambi i modi.

Ulteriori letture

- eff−bot mostra altri esempi su `import` *`module`* *contro* `from` *`module`* `import`.
- Nel *Python Tutorial* è possibile leggere di tecniche avanzate d'importazione, tra cui `from` *`module`* `import *`.

4.3. Definire classi

Python è completamente orientato agli oggetti: potete definire le vostre classi, ereditare dalle vostre classi o da quelle built−in ed istanziare le classi che avete definito.

Definire una classe in Python è semplice; come per le funzioni, non c'è una definizione separata per le interfacce. Semplicemente definite la classe e cominciate a programmare. Una classe in Python inizia con la parola riservata `class`, seguita dal nome della classe. Tecnicamente, questo è tutto ciò che è richiesto, visto che una classe non deve necessariamente essere ereditata da un'altra.

Esempio 4.5. La più semplice classe Python

```
class foo: (1)
    pass   (2) (3)
```

(1) Il nome di questa classe è `foo` e non è ereditata da nessun'altra classe.

(2) Questa classe non definisce nessun metodo o attributo ma sintatticamente, c'è bisogno di qualcosa nella definizione, quindi usiamo `pass`. È una parola riservata in Python, che significa semplicemente "va avanti, non c'è nulla da vedere qui". È un'istruzione che non fa niente, ed è un buon segnaposto quando state abbozzando

funzioni o classi.

(3) Probabilmente lo avete già indovinato, ogni cosa in una classe è indentato, proprio come il codice in una funzione, l'istruzione `if`, il ciclo `for` e così via. La prima cosa non indentata non è nella classe.

Nota: Python contro Java: pass
Lo statement `pass` in Python è come un paio di graffe vuote (`{}`) in Java od in C.

Ovvio che, realisticamente, molte classi saranno ereditate da altre classi e definiranno i loro specifici metodi ed attributi. Ma come avete appena visto, non c'è nulla che una classe debba assolutamente avere, salvo il nome. In particolare, i programmatori C++ troveranno bizzarro il fatto che le classi Python non hanno esplicitamente costruttori e distruttori. Le classi Python hanno qualcosa di simile ad un costruttore: il metodo `__init__`.

Esempio 4.6. Definire la classe `FileInfo`

```
from UserDict import UserDict

class FileInfo(UserDict): (1)
```

(1) In Python l'antenato di una classe è semplicemente elencato tra parentesi subito dopo il nome della classe. Così la classe `FileInfo` è ereditata dalla classe `UserDict` (che è stata importata dal `modulo UserDict`). `UserDict` è una classe che agisce come un dizionario, permettendovi essenzialmente di derivare una classe da quel tipo dizionario ed aggiungere il vostro specifico comportamento. (Ci sono similmente le classi `UserList` e `UserString` che vi permettono di derivare classi per liste e stringhe). Dietro questo comportamento c'è un po' di magia nera, la sfateremo più tardi in questo capitolo, quando esploreremo in maggior dettaglio la classe `UserDict`.

Nota: Python contro Java: antenati
In Python, l'antenato di una classe è semplicemente elencato tra parentesi subito dopo il nome della classe. Non c'è alcuna keyword come la `extends` di Java.

Nota: Ereditarietà multipla
Anche se non la discuterò in profondità nel libro, Python supporta l'ereditarietà multipla. Nelle parentesi che seguono il nome della classe, potete elencare quanti antenati volete, separati da virgole.

Esempio 4.7. Inizializzazione della classe `FileInfo`

```
class FileInfo(UserDict):
    "store file metadata"              (1)
    def __init__(self, filename=None): (2) (3) (4)
```

(1) Anche le classi possono (e dovrebbero) avere le `doc string`s, proprio come i moduli e le funzioni.

(2) `__init__` è chiamato immediatamente dopo la creazione dell'istanza di una classe. Si può essere erroneamente tentati nel chiamare questo metodo il costruttore della classe. Tentati, perché sembra proprio il costruttore (per convenzione, `__init__` è il primo metodo definito nella classe), agisce come un costruttore (è il primo pezzo di codice eseguito in una nuova istanza di una classe) e suona come un costruttore ("init" certamente suggerisce una natura costruttiva). Erroneamente, perché l'oggetto è già stato costruito prima che `__init__` venga chiamato ed avete già un riferimento valido alla nuova istanza della classe. Ma `__init__` è la cosa più vicina ad un costruttore che incontrerete in Python e ricopre essenzialmente il medesimo ruolo.

(3) Il primo argomento di ogni metodo di una classe, incluso `__init__`, è sempre un riferimento all'istanza corrente della classe. Per convenzione, questo argomento viene sempre chiamato `self`. Nel metodo `__init__`, `self` si riferisce all'oggetto appena creato; negli altri metodi della classe, si riferisce all'istanza da cui metodo è stato chiamato. Per quanto necessitiate di specificare `self` esplicitamente quando definite un

metodo, *non* lo specificate quando chiamate il metodo; Python lo aggiungerà per voi automaticamente.

(4) Il metodo `__init__` può prendere un numero arbitrario di argomenti e proprio come per le funzioni, gli argomenti possono essere definiti con valori predefiniti, rendendoli opzionali per il chiamante. In questo caso, `filename` ha il valore predefinito `None`, che è il valore nullo di Python.

Nota: Python contro Java: self
Per convenzione, il primo argomento di ogni metodo di una classe (il riferimento all'istanza corrente) viene chiamato `self`. Questo argomento ricopre il ruolo della parola riservata `this` in C++ o Java, ma `self` non è una parola riservata in Python, è semplicemente una convenzione sui nomi. Non di meno, vi prego di non chiamarlo diversamente da `self`; è una convenzione molto forte.

Esempio 4.8. Realizzare la classe `FileInfo`

```
class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)        (1)
        self["name"] = filename        (2)
                                       (3)
```

(1) Alcuni linguaggi pseudo−orientati agli oggetti come Powerbuilder hanno un concetto di "estensione" dei costruttori ed altri eventi, dove il metodo dell'antenato è chiamato automaticamente prima che il metodo del discendente venga eseguito. Python non fa questo; dovete sempre chiamare esplicitamente il metodo appropriato della classe antenata.

(2) Vi ho detto che questa classe agisce come un dizionario e questo ne è il primo segno. Stiamo assegnando l'argomento `filename` come valore della chiave `name` di questo stesso oggetto.

(3) Notate che il metodo `__init__` non ritorna mai un valore.

Nota: Quando usare self
Quando definite i vostri metodi nella classe, *dovete* elencare esplicitamente `self` come il primo argomento di ogni metodo, incluso `__init__`. Quando chiamate il metodo di una classe antenata dall'interno della vostra classe, *dovete* includere l'argomento `self`. Invece, quando chiamate i metodi della vostra classe dall'esterno, non dovete specificare affatto l'argomento `self`, lo saltate per intero e Python automaticamente aggiunge il riferimento all'istanza per voi. Temo che inizialmente possa confondere; non è proprio inconsistente ma può sembrarlo perché fa affidamento su una distinzione (tra metodi bound ed unbound) che ancora non conoscete.

Wow. Capisco che è un bel po' di roba da comprendere, ma ne verrete fuori. Tutte le classi Python funzionano allo stesso modo, così una volta che ne avrete imparata una, avrete imparato tutto. Se vi dimenticate tutto il resto, ricordate questa cosa, perché vi prometto che vi aiuterà:

Nota: i metodi __init__
i metodi `__init__` sono opzionali, ma quando ne definite uno, dovete ricordarvi di chiamare esplicitamente il metodo `__init__` dell'antenato. Questo è generalmente vero; quando un discendente vuole estendere il comportamento di un antenato, il metodo del discendente deve esplicitamente chiamare il metodo dell'antenato nel momento opportuno e con gli opportuni argomenti.

Ulteriori letture

- *Learning to Program* una semplice introduzione alle classi.
- *How to Think Like a Computer Scientist* mostra come utilizzare le classi per modellare specifici tipi di dato.
- *Python Tutorial* dà uno sguardo in profondità alle classi, spazi dei nomi ed ereditarietà.

- Python Knowledge Base risponde a domande comuni sulle classi.

4.4. Istanziare classi

Istanziare una classe in Python è molto semplice. Per istanziare una classe, basta semplicemente chiamare la classe come se fosse una funzione, passandole gli argomenti che il metodo `__init__` definisce. Il valore di ritorno sarà il nuovo oggetto.

Esempio 4.9. Creare un'istanza di `FileInfo`

```
>>> import fileinfo
>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3") (1)
>>> f.__class__                                        (2)
<class fileinfo.FileInfo at 010EC204>
>>> f.__doc__                                          (3)
'base class for file info'
>>> f                                                  (4)
{'name': '/music/_singles/kairo.mp3'}
```

(1) Stiamo creando un'istanza della classe `FileInfo` (definita nel modulo `fileinfo`) ed assegnando l'istanza appena creata alla variabile `f`. Stiamo passando un parametro, `/music/_singles/kairo.mp3`, che andrà a finire nell'argomento `filename` del metodo `__init__` di `FileInfo`.

(2) Ogni istanza di classe ha un attributo built−in, `__class__`, che è la classe dell'oggetto. (Notate che la sua rappresentazione include l'indirizzo fisico dell'istanza nella mia macchina; la vostra rappresentazione sarà diversa.) I programmatori Java potranno già avere familiarità con la classe `Class`, che contiene metodi come `getName` e `getSuperclass` per ottenere le informazioni sui metadati di un oggetto. In Python, questo tipo di metadato è disponibile direttamente nell'oggetto stesso attraverso attributi come `__class__`, `__name__` e `__bases__`.

(3) Potete accedere alla `doc string` di un'istanza proprio come una funzione o un metodo. Tutte le istanze di una classe condividono la medesima `doc string`.

(4) Ricordate quando il metodo `__init__` assegnava il suo argomento `filename` a `self["name"]`? Beh, questo è il risultato. Gli argomenti che noi passiamo quando creiamo l'istanza della classe vengono mandati direttamente al metodo `__init__` (assieme al riferimento all'oggetto, `self`, che Python aggiunge gratuitamente).

> **Nota: Python contro Java: istanziazione di classi**
> In Python, semplicemente chiamate una classe come se fosse una funzione per creare una sua nuova istanza. Non c'è alcun operatore esplicito new come in C++ o in Java.

Se creare nuove istanze è facile, distruggerle lo è ancora di più. In generale, non c'è alcun bisogno di liberare esplicitamente delle istanze, in quanto vengono automaticamente liberate quando la variabile ad esse assegnata esce dallo scope. I memory leaks sono rari in Python.

Esempio 4.10. Proviamo ad implementare un memory leak

```
>>> def leakmem():
...     f = fileinfo.FileInfo('/music/_singles/kairo.mp3') (1)
...
>>> for i in range(100):
...     leakmem()                                          (2)
```

(1) Ogni volta che la funzione `leakmem` viene chiamata, creiamo un'istanza di `FileInfo` e la assegnamo alla variabile `f`, che è una variabile locale alla funzione. Quindi la funzione termina senza liberare `f`, vi aspettereste un memory leak, ma sareste in errore. Quando la funzione termina, la variabile locale `f` esce dal suo scope. A questo punto, non c'è più alcun riferimento alla nuova istanza di `FileInfo` (in quanto non l'abbiamo mai assegnata ad altra variabile, tranne a `f`), così Python distrugge l'istanza per noi.

(2) Non importa quante volte chiamate la funzione `leakmem`, questa non perderà mai della memoria perché ogni volta, Python distruggerà la nuova istanza di `FileInfo` prima di ritornare da `leakmem`.

Il termine tecnico per questo genere di garbage collection è "reference counting". Python tiene una lista dei riferimenti ad ogni istanza creata. Nell'esempio precedente, c'era un solo riferimento all'istanza di `FileInfo`: la variabile locale `f`. Quando la funzione termina, la variabile `f` esce dallo scope, così il contatore dei riferimenti finisce a `0` e Python distrugge l'istanza automaticamente.

Nelle precedenti versioni di Python vi erano situazioni in cui il reference counting falliva e Python non era in grado di ripulire la memoria. Se creavate due istanze che si referenziavano a vicenda (per esempio, una lista doppiamente linkata, dove ogni nodo ha un puntatore al prossimo nodo ed al precedente), nessuna istanza sarebbe stata distrutta automaticamente perché Python (correttamente) credeva che ci fosse sempre un riferimento ad ogni istanza. Python 2.0 ha una forma addizionale di garbage collection chiamata "mark−and−sweep" che è sufficientemente intelligente da notare questo cappio virtuale e quindi, ripulire correttamente i riferimenti circolari.

In qualità di filosofo mi disturba l'idea che le cose scompaiano quando nessuno le sta guardando, ma questo è quanto accade in Python. In generale, potete semplicemente dimenticarvi della gestione della memoria e lasciare che sia Python a ripulire il vostro lavoro.

Ulteriori letture

- *Python Library Reference* riassume gli attributi built−in come `__class__`.
- *Python Library Reference* documenta il modulo `gc`, che vi dà un controllo a basso livello sulla garbage collection di Python.

4.5. `UserDict`: una classe wrapper

Come avete visto, `FileInfo` è una classe che agisce come un dizionario. Per esplorare ulteriormente questo aspetto, diamo uno sguardo alla classe `UserDict` nel modulo `UserDict`, che è l'antenato della nostra classe `FileInfo`. Non è nulla di speciale; la classe è scritta in Python e memorizzata in un file `.py`, proprio come il nostro codice. In particolare, è memorizzata nella directory `lib` della vostra installazione di Python.

> **Suggerimento: Aprire i moduli rapidamente**
> Nella IDE di Python su Windows, potete aprire rapidamente qualunque modulo nel vostro library path usando File−>Locate... (**Ctrl−L**).

Nota storica. Nelle versioni di Python antecedenti la 2.2, non potevate ereditare direttamente dai tipi built−in come stringhe, liste e dizionari. Per compensare a tale mancanza, Python viene rilasciato con delle classi wrapper che mimano il comportamento di questi tipi built−in: `UserString`, `UserList` e `UserDict`. Usando una combinazione di metodi normali e speciali, la classe `UserDict` fa un'eccellente imitazione di un dizionario, ma è semplicemente una classe come qualunque altra, così potete ereditare da essa per creare delle classi personalizzate che si comportano come dizionari, come abbiamo fatto con `FileInfo`. In Python 2.2 o superiore, potreste riscrivere l'esempio di questo capitolo in modo che `FileInfo` erediti direttamente da `dict` invece che da `UserDict`. Comunque dovreste lo stesso leggere come funziona `UserDict`, nel caso abbiate bisogno di implementare questo genere di oggetto wrapper o nel caso abbiate bisogno di supportare versioni di Python antecedenti alla 2.2.

Esempio 4.11. Definire la classe `UserDict`

```
class UserDict:                                 (1)
    def __init__(self, dict=None):              (2)
        self.data = {}                          (3)
        if dict is not None: self.update(dict) (4) (5)
```

(1) Notate che `UserDict` è una classe base, non eredita da nessun'altra classe.

(2) Questo è il metodo `__init__` che andiamo a sovrascrivere nella classe `FileInfo`. Notate che la lista degli argomenti in questa classe antenato è diversa dai suoi discendenti. Va bene; ogni sottoclasse può avere il suo insieme di argomenti, finché chiama l'antenato con gli argomenti esatti. Qui la classe antenato ha un modo per definire i valori iniziali (passando un dizionario nell'argomento `dict`) di cui la nostra `FileInfo` non si avvantaggia.

(3) Python supporta gli attributi (chiamati "variabili d'istanza" in Java ed in Powerbuilder, "variabili membro" in C++), cioè dati mantenuti da una specifica istanza di una classe. In questo caso, ogni istanza di `UserDict` avrà un attributo `data`. Per referenziare questo attributo dal codice esterno alla classe, lo dovrete qualificare usando il suo nome di istanza, *`instance`*`.data`, nello stesso modo in cui qualificate una funzione con il nome del suo modulo. Per referenziare un attributo all'interno della classe, usiamo `self` come qualificatore. Per convenzione, tutti gli attributi sono inizializzati con dei valori ragionevoli nel metodo `__init__`. Ad ogni modo, questo non è richiesto, in quanto gli attributi, così come le variabili locali, cominciano ad esistere nel momento in cui viene loro assegnato un valore.

(4) Il metodo `update` è un duplicatore di dizionario: copia tutte le chiavi ed i valori da un dizionario ad un altro. Questo metodo *non* cancella il dizionario di destinazione, se il dizionario di destinazione ha già alcune delle chiavi, il loro valore sarà sovrascritto, ma gli altri dati rimarranno invariati. Pensate al metodo `update` come ad una funzione di fusione e non di copia.

(5) Questa è una sintassi che potreste non aver ancora visto (non l'ho usata negli esempi di questo libro). Si tratta di una istruzione `if`, ma invece di avere un blocco indentato nella riga successiva, c'è semplicemente una singola istruzione sulla stessa riga, dopo i due punti. È una sintassi perfettamente legale ed è semplicemente una scorciatoia quando avete una sola istruzione in un blocco. (È come specificare una singola istruzione senza graffe in C++). Potete usare questa sintassi o potete usare il codice indentato in righe successive, ma non potete fare entrambe le cose per lo stesso blocco.

Nota: Python contro Java: overload di funzioni
Java e Powerbuilder supportano l'overload di funzioni per lista di argomenti, cioè una classe può avere più metodi con lo stesso nome, ma diverso numero di argomenti o argomenti di tipo diverso. Altri linguaggi (principalmente PL/SQL) supportano l'overload di funzioni per nome di argomento; cioè una classe può avere più metodi con lo stesso nome e lo stesso numero di argomenti dello stesso tipo, ma i nomi degli argomenti sono diversi. Python non supporta nessuno di questi; non ha nessuna forma di overload di funzione. I metodi sono definiti solamente dal loro nome e ci può essere solamente un metodo per ogni classe con un dato nome. Così, se una classe discendente ha un metodo `__init__`, questo sovrascrive *sempre* il metodo `__init__` dell'antenato, anche se il discendente lo definisce con una lista di argomenti diversa. La stessa regola si applica ad ogni altro metodo.

Nota: Guido sulle classi derivate
Guido, l'autore originale di Python, spiega l'override dei metodi in questo modo: "Classi derivate possono sovrascrivere i metodi delle loro classi base. Siccome i metodi non hanno privilegi speciali quando chiamano altri metodi dello stesso oggetto, un metodo di una classe base che chiama un altro metodo definito nella stessa classe base, può infatti finire per chiamare un metodo di una classe derivata che lo sovrascrive. (Per i programmatori C++ tutti i metodi in Python sono effettivamente virtual.)" Se questo per voi non ha senso (confonde un sacco pure me), sentitevi liberi di ignorarlo. Penso che lo capirò più avanti.

Nota: Inizializzare sempre gli attributi
Assegnate sempre un valore iniziale a tutti gli attributi di un'istanza nel metodo `__init__`. Vi risparmierà ore di debugging più tardi, spese a tracciare tutte le eccezioni `AttributeError` che genera il programma perché state referenziando degli attributi non inizializzati (e quindi inesistenti).

Esempio 4.12. Metodi comuni di `UserDict`

```
    def clear(self): self.data.clear()          (1)
    def copy(self):                             (2)
        if self.__class__ is UserDict:          (3)
            return UserDict(self.data)
        import copy                             (4)
        return copy.copy(self)
    def keys(self): return self.data.keys()     (5)
    def items(self): return self.data.items()
    def values(self): return self.data.values()
```

(1) `clear` è un normale metodo della classe; è pubblicamente disponibile per essere chiamato da chiunque in qualunque momento. Notate che `clear`, come tutti gli altri metodi della classe, ha `self` come primo argomento (ricordate, non siete voi ad includere `self` quando chiamate un metodo; è qualcosa che Python fa già per voi). Notate inoltre la tecnica base di questa classe wrapper: memorizza un vero dizionario (`data`) come attributo, definisce tutti i metodi che ha un vero dizionario e redireziona tutti questi metodi verso i loro corrispondenti del vero dizionario (nel caso lo aveste dimenticato, il metodo `clear` di un dizionario cancella tutte le sue chiavi ed i valori ad esse associati).

(2) Il metodo `copy` di un vero dizionario restituisce un nuovo dizionario che è un esatto duplicato di quello originale (tutte le stesse coppie chiave−valore). Ma `UserDict` non può semplicemente redirezionare il metodo a `self.data.copy`, perché quel metodo ritorna un vero dizionario e noi vogliamo che ritorni un'istanza della stessa classe di `self`.

(3) Usiamo l'attributo `__class__` per vedere se `self` è uno `UserDict`; nel caso, siamo fortunati perché sappiamo come copiare uno `UserDict`: basta creare una nuova istanza di `UserDict` e passarle il vero dizionario che abbiamo ricavato da `self.data`.

(4) Se `self.__class__` non è uno `UserDict`, allora `self` deve essere una sottoclasse di `UserDict` (come `FileInfo`), nel quale caso la vita si complica un po'. `UserDict` non sa come fare una copia di uno dei suoi discendenti; ci potrebbero, per esempio, essere altri attributi definiti nella sottoclasse, così dovremmo iterare attraverso ognuno di essi ed essere sicuri di copiarli tutti. Fortunatamente, Python viene rilasciato con un modulo che fa esattamente questo ed è chiamato `copy`. Non scenderò in dettaglio (per quanto si tratti di un modulo davvero interessante, se vi interessa potete esplorarlo più a fondo da soli), è sufficiente dire che `copy`, può copiare oggetti arbitrari Python ed è così che noi lo usiamo.

(5) I rimanenti metodi sono semplici, redirezionano le chiamate ai metodi built−in di `self.data`.

Ulteriori letture

- *Python Library Reference* documenta il modulo `UserDict` ed il modulo `copy`.

4.6. Metodi speciali per le classi

In aggiunta ai normali metodi delle classi, c'è un certo numero di metodi speciali che le classi Python possono definire. Invece di essere chiamati direttamente dal vostro codice (come metodi normali), i metodi speciali sono chiamati per voi da Python in particolari circostanze o quando viene adoperata una certa sintassi.

Come avete visto nella sezione precedente, i metodi normali vanno ben oltre il wrapping di un dizionario in una classe. Ma i metodi normali da soli non sono abbastanza perché ci sono una sacco di cose che potete fare con i

dizionari oltre a chiamare i loro metodi. Per i novizi, potete ottenere ed impostare elementi con una sintassi che non prevede l'esplicita invocazione di un metodo. È qui che saltano fuori i metodi speciali di una classe: mettono a disposizione un modo per mappare la sintassi senza−chiamata−metodo in una vera chiamata a metodo.

Esempio 4.13. Il metodo speciale `__getitem__`

```
    def __getitem__(self, key): return self.data[key]

>>> f = fileinfo.FileInfo("/music/_singles/kairo.mp3")
>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__getitem__("name") (1)
'/music/_singles/kairo.mp3'
>>> f["name"]             (2)
'/music/_singles/kairo.mp3'
```

(1) Il metodo speciale `__getitem__` sembra essere abbastanza semplice. Come i metodi `clear`, `keys` e `values`, semplicemente redireziona la chiamata al dizionario per ritornare il suo valore. Ma come viene chiamato? Beh, potete chiamare `__getitem__` direttamente, ma in pratica non lo farete; io lo faccio giusto per mostrarvi come funziona. Il modo giusto di usare `__getitem__` è di farlo chiamare per voi da Python.

(2) Questa sembra proprio la sintassi che usereste per ottenere un valore da un dizionario ed infatti restituisce il valore che vi aspettereste. Ma qui si vede l'anello mancante: sotto sotto, Python ha convertito questa sintassi in una chiamata a metodo `f.__getitem__("name")`. Ecco perché `__getitem__` è un metodo speciale; non solo lo potete chiamare esplicitamente ma potete fare in modo che Python lo chiami per voi, utilizzando la sintassi corretta.

Esempio 4.14. Il metodo speciale `__setitem__`

```
    def __setitem__(self, key, item): self.data[key] = item

>>> f
{'name':'/music/_singles/kairo.mp3'}
>>> f.__setitem__("genre", 31) (1)
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':31}
>>> f["genre"] = 32            (2)
>>> f
{'name':'/music/_singles/kairo.mp3', 'genre':32}
```

(1) Come il metodo `__getitem__`, `__setitem__` per fare il suo lavoro semplicemente redireziona la chiamata al vero dizionario `self.data` e come per `__getitem__` non lo chiamerete in maniera diretta come qui; Python chiama `__setitem__` per voi quando usate la sintassi corretta.

(2) Questa somiglia alla sintassi regolare dei dizionari, eccetto naturalmente per il fatto che `f` è una classe che cerca di travestirsi da dizionario e `__setitem__` è una parte essenziale del suo travestimento. Questa riga di codice va. sotto sotto, a chiamare `f.__setitem__("genre", 32)`.

`__setitem__` è un metodo speciale di classe perché viene chiamato per voi, ma è comunque un metodo della classe. Tanto semplicemente quanto il metodo `__setitem__` è stato definito in `UserDict`, noi lo possiamo ridefinire nelle nostre classi discendenti per sovrascrivere il metodo antenato. Questo ci permette di definire classi che agiscono come dizionari in alcuni casi, ma definirne il loro comportamento ben oltre quello dei dizionari built−in.

Questo concetto sta alla base dell'intero framework che stiamo studiando in questo capitolo. Ogni tipo di file può

avere una classe handler che sa come ottenere i metadati da un particolare tipo di file. Una volta che alcuni attributi (come il nome del file e la sua locazione) sono noti, la classe handler sa come derivare altri attributi automaticamente. Questo è fatto sovrascrivendo il metodo `__setitem__`, controllando particolari chiavi ed elaborandole nel momento in cui vengono trovate.

Per esempio, `MP3FileInfo` è un discendente di `FileInfo`. Quando il `nome` di un `MP3FileInfo` è impostato, non si limita ad impostare la chiave `name` (come fa l'antenato `FileInfo`); va anche a cercare tag MP3 nel file e popola l'intero insieme di chiavi.

Esempio 4.15. Sovrascrivere `__setitem__` in `MP3FileInfo`

```
    def __setitem__(self, key, item):         (1)
        if key == "name" and item:            (2)
            self.__parse(item)                (3)
        FileInfo.__setitem__(self, key, item) (4)
```

(1) Notate che il nostro metodo `__setitem__` è definito nello stesso modo in cui era definito l'antenato. Questo è importante, in quanto Python dovrà chiamare il metodo per noi e si aspetta che vi siano definiti un certo numero di argomenti (parlando tecnicamente, i nomi degli argomenti non contano, ha importanza solamente il loro numero).

(2) Questo è il punto cruciale dell'intera classe `MP3FileInfo`: se stiamo assegnando un valore alla chiave `name`, vogliamo fare qualcosa in più.

(3) L'elaborazione aggiuntiva che vogliamo fare per i `nomi` è incapsulata nel metodo `__parse`. Si tratta di un altro metodo della classe definito in `MP3FileInfo` e quando lo chiamiamo, lo qualifichiamo con `self`. La semplice chiamata a `__parse` sembrerebbe l'invocazione di una funzione definita fuori dalla classe, che non è ciò che noi vogliamo; chiamando `self.__parse` sembrerà la chiamata ad un metodo definito nella classe. Non si tratta di nulla di nuovo; referenziate gli attributi nella medesima maniera.

(4) Dopo aver effettuato la nostra elaborazione aggiuntiva, vogliamo chiamare il metodo dell'antenato. Ricordate, questo non viene mai fatto per voi da Python; lo dovete fare manualmente. Notate che stiamo chiamando l'antenato immediatamente precedente, `FileInfo`, per quanto non abbia un metodo `__setitem__`. Va bene, perché Python seguirà l'albero degli antenati finché non trova una classe con il metodo che stiamo chiamando, così questa riga di codice, eventualmente, troverà e chiamerà il metodo `__setitem__`, definito in `UserDict`.

Nota: Chiamare altri metodi di classe
Quando accedete agli attributi di una classe, avete bisogno di qualificare il nome dell'attributo: `self.`*`attributo`*. Quando chiamate altri metodi di una classe, dovete qualificare il nome del metodo: `self.`*`metodo`*.

Esempio 4.16. Impostare il `nome` di un `MP3FileInfo`

```
>>> import fileinfo
>>> mp3file = fileinfo.MP3FileInfo()                  (1)
>>> mp3file
{'name':None}
>>> mp3file["name"] = "/music/_singles/kairo.mp3"     (2)
>>> mp3file
{'album': 'Rave Mix', 'artist': '***DJ MARY-JANE***', 'genre': 31,
'title': 'KAIRO****THE BEST GOA', 'name': '/music/_singles/kairo.mp3',
'year': '2000', 'comment': 'http://mp3.com/DJMARYJANE'}
>>> mp3file["name"] = "/music/_singles/sidewinder.mp3" (3)
>>> mp3file
```

```
{'album': '', 'artist': 'The Cynic Project', 'genre': 18, 'title': 'Sidewinder',
'name': '/music/_singles/sidewinder.mp3', 'year': '2000',
'comment': 'http://mp3.com/cynicproject'}
```

(1) Prima, creiamo un'istanza di `MP3FileInfo`, senza passarle alcun nome di file (possiamo evitarlo in quanto l'argomento `filename` del metodo `__init__` è opzionale). Siccome `MP3FileInfo` non ha un suo metodo `__init__`, Python percorre l'albero degli antenati e trova il metodo `__init__` di `FileInfo`. Questo metodo `__init__` chiama manualmente il metodo `__init__` di `UserDict` e quindi imposta la chiave `name` al valore di `filename`, che è `None`, in quanto non abbiamo passato l'argomento filename. Dunque `mp3file` inizialmente sembra un dizionario con una chiave, `name`, il cui valore è `None`.

(2) Ora inizia il vero divertimento. Impostando la chiave `name` di `mp3file` viene invocato il metodo `__setitem__` di `MP3FileInfo` (non di `UserDict`), il quale nota che stiamo impostando la chiave `name` con un vero valore e chiama `self.__parse`. Anche se non abbiamo ancora analizzato a fondo il metodo `__parse`, potete vedere dall'output che va ad impostare altre chiavi: `album`, `artist`, `genre`, `title`, `year`, e `comment`.

(3) Modificare la chiave `name` innescherà nuovamente lo stesso processo: Python chiama `__setitem__`, che chiama `self.__parse`, che imposta tutte le altre chiavi.

4.7. Metodi speciali di classe avanzati

Ci sono molti più metodi speciali dei soli `__getitem__` e `__setitem__`. Alcuni di questi vi permettono di emulare funzionalità che non avreste mai conosciuto.

Esempio 4.17. Altri metodi speciali in `UserDict`

```
    def __repr__(self): return repr(self.data)     (1)
    def __cmp__(self, dict):                       (2)
        if isinstance(dict, UserDict):
            return cmp(self.data, dict.data)
        else:
            return cmp(self.data, dict)
    def __len__(self): return len(self.data)       (3)
    def __delitem__(self, key): del self.data[key] (4)
```

(1) `__repr__` è un metodo speciale che è chiamato quando chiamate `repr(`*`istanza`*`)`. La funzione `repr` è una funzione built−in che ritorna una rappresentazione sotto forma di stringa di un oggetto. Funziona su ogni oggetto, non solo su istanze di classi. Avete già familiarizzato con `repr` ed ancora non lo sapete. Nella finestra interattiva, quando scrivete solamente il nome di una variabile e premete **ENTER**, Python usa `repr` per mostrare il valore della variabile. Create un dizionario `d` con alcuni dati e quindi stampate `repr(d)` per vederlo con i vostri occhi.

(2) `__cmp__` è chiamato quando confrontate istanze di classe. In generale, potete confrontare ogni coppia di oggetti Python, non solamente istanze di classe, usando `==`. Ci sono regole che definiscono quando tipi di dato built−in sono considerati uguali; per esempio, i dizionari sono uguali quando hanno tutte le stesse chiavi e valori, le stringhe sono uguali quando hanno la stessa lunghezza e contengono la stessa sequenza di caratteri. Per le istanze di classe, potete definire il metodo `__cmp__` ed implementarne la logica di confronto, potrete quindi usare `==` per confrontare istanze delle vostre classi e Python chiamerà il metodo speciale `__cmp__` per voi.

(3) `__len__` è chiamata quando chiamate `len(`*`istanza`*`)`. La funzione `len` è una funzione built−in che ritorna la lunghezza di un oggetto. Funziona su ogni oggetto di cui si può ragionevolmente pensare di misurarne la lunghezza. La lunghezza di una stringa è il numero dei suoi caratteri; la lunghezza di un dizionario è il numero delle sue chiavi; la lunghezza di una lista o di una tupla è il numero dei suoi

elementi. Per le istanze di classe, definite il metodo `__len__` ed implementate il calcolo della lughezza, quindi chiamate `len(istanza)` e Python chiamerà il metodo speciale `__len__` per voi.

(4) `__delitem__` è chiamato quando chiamate `del istanza[chiave]`, che potreste ricordare come il modo per cancellare singoli elementi da un dizionario. Quando usate `del` su un'istanza di classe, Python chiama il metodo speciale `__delitem__` per voi.

> **Nota: Python contro Java: uguaglianza ed identità**
> In Java, determinate quando due variabili di tipo stringa referenziano la stessa memoria fisica utilizzando `str1 == str2`. Questa è chiamata *identità di oggetto*, ed è espressa in Python come `str1 is str2`. Per confrontare valori di tipo stringa in Java, dovreste usare `str1.equals(str2)`; in Python, usereste `str1 == str2`. Programmatori Java a cui è stato insegnato a credere che il mondo è un posto migliore in quanto `==` in Java confronta l'identità invece del valore potrebbero avere qualche difficoltà nell'acquisire questo nuovo "concetto" di Python.

A questo punto, potreste pensare, "tutto questo lavoro solo per fare in una classe qualcosa che potrei fare con un tipo built−in". Ed è vero che la vita sarebbe più semplice (e l'intera classe `UserDict` inutile) se poteste ereditare direttamente da un tipo built−in come un dizionario. Ma anche se poteste, i metodi speciali sarebbero comunque utili perché possono essere usati in ogni classe, non solamente nelle classi wrapper come `UserDict`.

Metodi speciali vuol dire che *ogni classe* può memorizzare coppie chiave−valore come un dizionario, solo definendo il metodo `__setitem__`. *Ogni classe* può agire come una sequenza, semplicemente definendo il metodo `__getitem__`. Ogni classe che definisce il metodo `__cmp__` può essere confrontata con `==`. Se invece la vostra classe rappresenta qualcosa che ha una lunghezza, non definite il metodo `GetLength`, ma il metodo `__len__` ed usate `len(istanza)`.

> **Nota: Modelli fisici contro modelli logici**
> Mentre altri linguaggi orientati agli oggetti vi lasciano definire il modello fisico di un oggetto ("questo oggetto ha il metodo `GetLength`"), i metodi speciali di classe Python come `__len__` vi permettono di definire il modello logico di un oggetto ("questo oggetto ha una lunghezza").

Ci sono molti altri metodi speciali. Ce n'è un intero gruppo che permette alle vostre classi di agire come numeri, permettendovi di aggiungere, sottrarre e fare altre operazioni aritmetiche sulle istanze di classe. L'esempio canonico di questo concetto è una classe che rappresenta i numeri complessi, numeri con parte reale ed immaginaria. Il metodo `__call__` permette ad una classe di agire come una funzione, permettendovi di chiamare un'istanza di classe direttamente. Ci sono altri metodi speciali che permettono alle classi di avere attributi in sola lettura ed in sola scrittura; ne parleremo più diffusamente nei capitoli successivi.

Ulteriori letture

- *Python Reference Manual* documenta tutti i metodi speciali per le classi.

4.8. Attributi di classe

Conoscete già gli attributi dato, che sono variabili di una specifica istanza di una classe. Python supporta anche gli attributi di classe, che sono variabili riferite alla classe stessa.

Esempio 4.18. Introdurre gli attributi di classe

```
class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
```

```
                    "album"   : ( 63,  93, stripnulls),
                    "year"    : ( 93,  97, stripnulls),
                    "comment" : ( 97, 126, stripnulls),
                    "genre"   : (127, 128, ord)}

>>> import fileinfo
>>> fileinfo.MP3FileInfo            (1)
<class fileinfo.MP3FileInfo at 01257FDC>
>>> fileinfo.MP3FileInfo.tagDataMap (2)
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
>>> m = fileinfo.MP3FileInfo()      (3)
>>> m.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
```

(1) `MP3FileInfo` è la classe stessa, non una particolare istanza della classe.

(2) `tagDataMap` è un "class attribute": letteralmente, un attributo della classe. È disponibile prima di ogni istanza della classe.

(3) Gli attributi di classe sono disponibili sia attraverso un riferimento diretto alla classe che attraverso qualunque istanza della classe.

Nota: Attributi di classe in Java

In Java, sia le variabili statiche (chiamate attributi di classe in Python) che le variabili di istanza (chiamate attributi dato in Python), sono definite immediatamente dopo la definizione della classe (il primo con la parola chiave `static`, il secondo senza). In Python, solo gli attributi di classe posso essere definiti così; gli attributi dato sono definiti nel metodo `__init__`.

Gli attributi di classe possono essere usati come costanti a livello di classe (cioè il modo in cui li usiamo in `MP3FileInfo`), ma non sono realmente costanti. [4] Potete anche modificarli.

Esempio 4.19. Modificare gli attributi di classe

```
>>> class counter:
...     count = 0                    (1)
...     def __init__(self):
...         self.__class__.count += 1 (2)
...
>>> counter
<class __main__.counter at 010EAECC>
>>> counter.count                    (3)
0
>>> c = counter()
>>> c.count                          (4)
1
>>> counter.count
1
>>> d = counter()                    (5)
>>> d.count
2
```

```
>>> c.count
2
>>> counter.count
2
```

(1) `count` è un attributo di classe della classe `counter`.

(2) `__class__` è un attributo built−in di ogni istanza della classe (di ogni classe). È un riferimento alla classe di cui `self` è un'istanza (in questo caso, la classe `counter`).

(3) Dato che `count` è un attributo di classe, è quindi disponibile attraverso un riferimento diretto, prima di aver creato una qualunque istanza della classe.

(4) Creando un'istanza della classe viene chiamato il metodo `__init__`, che incrementa l'attributo di classe `count` di `1`. Questo ha effetto sulla classe stessa, non solo sull'istanza appena creata.

(5) Creando una seconda istanza della classe verrà incrementato di nuovo l'attributo di classe `count`. Notate come l'attributo di classe venga condiviso dalla classe e da tutte le sue istanze.

4.9. Funzioni private

Come molti linguaggi, anche Python ha il concetto di funzioni private, che non possono essere chiamate dall'esterno del loro modulo; metodi privati di una classe, che non possono essere chiamati dall'esterno della loro classe ed attributi privati, che non possono essere referenziati dall'esterno della loro classe. Al contrario di molti linguaggi, il fatto che una funzione, metodo o attributo in Python sia pubblico o privato viene determinato interamente dal suo nome.

In `MP3FileInfo` vi sono due metodi: `__parse` e `__setitem__`. Come abbiamo già detto, `__setitem__` è un metodo speciale; normalmente, lo chiamereste indirettamente usando la sintassi del dizionario su un'istanza di classe ma è pubblico e potreste chiamarlo direttamente (anche dall'esterno del modulo `fileinfo`) se avete una buona ragione. Comunque, `__parse` è privato, visto che ha due underscore all'inizio del suo nome.

> **Nota: Cos'è privato in Python?**
> Se il nome di una funzione, metodo di classe o attributo in Python inizia con (ma non finisce con) due underscore, è privato; ogni altra cosa è pubblica.

> **Nota: Convenzioni sui nomi dei metodi**
> In Python, tutti i metodi speciali (come `__setitem__`) e gli attributi built−in (come `__doc__`) seguono una convenzione standard sui nomi: iniziano e finiscono con due underscore. Non nominate i vostri metodi e attributi in questa maniera, servirà solo a confondervi. Inoltre, in futuro, potrebbe anche confondere altri che leggeranno il vostro codice.

> **Nota: Metodi non protetti**
> Python non ha il concetto di metodi protetti di classe (accessibili solo nella loro stessa classe ed in quelle discendenti). I metodi di classe sono o privati (accessibili unicamente nella loro stessa classe) o pubblici (accessibili ovunque).

Esempio 4.20. Provare ad invocare un metodo privato

```
>>> import fileinfo
>>> m = fileinfo.MP3FileInfo()
>>> m.__parse("/music/_singles/kairo.mp3") (1)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'MP3FileInfo' instance has no attribute '__parse'
```

(1) Se provate a chiamare un metodo privato, Python solleverà un'eccezione un po' ingannevole, dicendo che il metodo non esiste. Ovviamente esiste, ma è privato, quindi non accessibile al di fuori della classe. [5]

Ulteriori letture

- Nel *Python Tutorial* troverete informazioni difficilmente rintracciabili altrove sulle variabili private.

4.10. Gestire le eccezioni

Come molti linguaggi orientati agli oggetti, Python consente la manipolazione delle eccezioni tramite i blocchi `try...except`.

> **Nota: Python contro Java: gestire le eccezioni**
> Python usa `try...except` per gestire le eccezioni e `raise` per generarle. Java e C++ usano `try...catch` per gestirle e `throw` per generarle.

Se conoscete già tutto riguardo le eccezioni, potete saltare questa sezione. Se siete rimasti bloccati programmando in un linguaggio minore che non ha la gestione delle eccezioni, o avete usato un linguaggio vero, ma senza usare le eccezioni, questa sezione è molto importante.

Le eccezioni sono ovunque in Python; virtualmente ogni modulo nella libreria standard di Python le usa e Python stesso le solleva in molte differenti circostanze. Le avete già viste spesso in questo libro.

- Accedere ad una chiave non esistente di un dizionario solleva una eccezione `KeyError`.
- Ricercare in una lista un valore non esistente solleva un'eccezione `ValueError`.
- Chiamare un metodo non esistente solleva un'eccezione `AttributeError`.
- Riferirsi ad una variabile non esistente solleva un'eccezione `NameError`.
- Mischiare tipi di dato senza coercizione solleva un'eccezione `TypeError`.

In ognuno di questi casi, stavamo semplicemente giocando intorno all'IDE di Python: si incappa in un errore, l'eccezione viene stampata (a seconda del vostro IDE, in un intenzionale e stonante ombra di rosso) e niente più. Questa viene chiamata eccezione *non gestita*; quando l'eccezione veniva sollevata, non c'era codice che la notasse esplicitamente e se ne occupasse, perciò tutto proseguiva secondo il comportamento predefinito di Python, stampando alcune informazioni di debug e terminando il programma. Nell'IDE questo non è un grosso problema, ma se accade mentre il vostro programma Python reale è in esecuzione, l'intero programma va incontro ad un'improvvisa e poco piacevole interruzione. [6]

Un'eccezione non deve tuttavia comportare il crash dell'intero programma. Le eccezioni, quando sollevate, possono essere *gestite*. Qualche volta un'eccezione è autentica perché c'è un difetto nel vostro codice (come l'accesso ad una variabile inesistente), spesso però un'eccezione è qualcosa che avevate previsto. Se, ad esempio, state aprendo un file, questi potrebbe non esistere; se vi state connettendo ad un database, potrebbe non essere disponibile o potreste non avere i necessari requisiti di sicurezza per accedervi. Se sapete che una linea di codice potrebbe sollevare un'eccezione, dovreste gestirla usando un blocco `try...except`.

Esempio 4.21. Aprire un file inesistente

```
>>> fsock = open("/notthere", "r")      (1)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
IOError: [Errno 2] No such file or directory: '/notthere'
>>> try:
...     fsock = open("/notthere")       (2)
... except IOError:                     (3)
```

```
...     print "The file does not exist, exiting gracefully"
... print "This line will always print" (4)
The file does not exist, exiting gracefully
This line will always print
```

(1) Usando la funzione built−in `open`, possiamo provare ad aprire un file in lettura (molte altre cose su `open` nella prossima sezione). Il file non esiste, quindi viene sollevata l'eccezione `IOError`. Dato che non abbiamo provveduto a gestire l'eccezione `IOError`, Python si limita a stampare alcune informazioni di debug di quello che è accaduto ed a terminare il programma.

(2) Stiamo provando ad aprire lo stesso file inesistente ma questa volta lo stiamo facendo all'interno di un blocco `try...except`.

(3) Quando il metodo `open` solleva un'eccezione `IOError`, siamo pronti per gestirla. `except IOError:` questa linea cattura l'eccezione ed esegue il nostro blocco di codice, che in questo caso si limita a stampare un messaggio di errore più cortese.

(4) Una volta che un'eccezione è stata gestita, il processo continua con la linea che segue il blocco `try...except`. Notate che questa linea stamperà sempre se l'eccezione avviene oppure no. Se aveste veramente un file chiamato `notthere` nella vostra root directory, la chiamata ad `open` funzionerebbe, la clausola `except` sarebbe ignorata e questa linea verrebbe ancora eseguita.

Le eccezioni possono sembrare ostiche (dopotutto, se non catturate l'eccezione, il vostro intero programma terminerà), ma considerate l'alternativa. Vorreste veramente ottenere un oggetto file riferito ad un file non esistente? Dovreste in ogni caso controllarne la validità e se ve ne dimenticate, il vostro programma vi darebbe strani errori da qualche parte nelle linee seguenti e voi sareste costretti a ricontrollare l'intero codice, alla ricerca della causa del problema. Sono sicuro che lo avete già fatto; non è divertente. Con le eccezioni, gli errori vengono rilevati immediatamente e potete risolvere il problema alla radice, gestendoli in modo standard.

Ci sono molti altri usi delle eccezioni oltre alla gestione degli errori. Un uso comune nella libreria standard Python consiste nel provare ad importare un modulo e poi provare se funzioni o meno. Importare un modulo che non esiste causa il sollevamento dell'eccezione `ImportError`. Potete usarla per definire molteplici livelli di funzionalità, basati su quali moduli sono disponibili a run−time, o per supportare molteplici piattaforme (dove il codice specifico per ogni piattaforma è suddiviso in moduli differenti).

Potete inoltre definire le vostre stesse eccezioni, creando una classe che eredita dalla classe built−in `Exception` e sollevare le vostre eccezioni tramite il comando `raise`. Questo va al di fuori dallo scopo di questa sezione ma leggete gli ulteriori paragrafi se siete interessati.

Esempio 4.22. Supportare le funzionalità specifiche per piattaforma

Questo codice proviene dal modulo `getpass`, un modulo wrapper per ottenere password dagli utenti. Ottenere una password è significativamente differente da piattaforme UNIX, Windows e Mac OS, ma questo codice incapsula tutte differenze.

```
# Bind the name getpass to the appropriate function
try:
    import termios, TERMIOS                     (1)
except ImportError:
    try:
        import msvcrt                           (2)
    except ImportError:
        try:
            from EasyDialogs import AskPassword (3)
        except ImportError:
            getpass = default_getpass           (4)
        else:                                   (5)
```

```
            getpass = AskPassword
    else:
        getpass = win_getpass
else:
    getpass = unix_getpass
```

(1) `termios` è un modulo specifico UNIX, che rende disponibile un controllo di basso livello sul terminale di input. Se il modulo non è disponibile sul vostro sistema o non è supportato, Python solleverà l'eccezione `ImportError`, che noi cattureremo.

(2) Ok, non abbiamo `termios`, perciò proviamo `msvcrt`, un modulo specifico di Windows, che fornisce un API per molte funzioni utili del Microsoft Visual C++ run−time. Se l'import fallisce, Python solleverà `ImportError` e noi cattureremo l'eccezione.

(3) Se i primi due non funzionano, proviamo ad importare una funzione da `EasyDialogs`, che è un modulo specifico di Mac OS che provvede a funzioni di dialogo popup di vario tipo. Se, ancora una volta, questo import fallisce, Python solleverà l'eccezione `ImportError`, che noi cattureremo.

(4) Nessuno di questi moduli specifici di piattaforma sono disponibili (il che è possibile, visto che Python è stato portato su molte differenti piattaforme), perciò dobbiamo puntare su una funzione standard di input delle password (che è definita da qualche altra parte nel modulo `getpass`). Notate cosa stiamo facendo: stiamo assegnando la funzione `default_getpass` alla variabile `getpass`. Se leggete la documentazione ufficiale di `getpass`, vi spiegherà che il modulo `getpass` definisce una funzione `getpass`. Questo è ciò che fa: passa `getpass` alla giusta funzione per la vostra piattaforma. Quindi, quando chiamate la funzione getpass, state realmente chiamando una funzione specifica per la vostra piattaforma, che questo codice ha fatto per voi. Non avete bisogno di conoscere o di far caso alla piattaforma su cui gira il vostro programma; limitatevi a chiamare `getpass` e questi farà sempre la cosa giusta.

(5) Un blocco `try...except` può avere una clausola `else`, come l'istruzione `if`. Se nessuna eccezione è sollevata durante il blocco `try`, la clausola `else` viene eseguita di seguito. In questo caso, questo significa che `from EasyDialogs import AskPassword` funziona, per cui noi dovremmo legare `getpass` alla funzione `AskPassword`. Ognuno degli altri blocchi `try...except` hanno clausole `else` simili, per legare `getpass` alla funzione appropriata, quando troviamo un `import` che funzioni.

Ulteriori letture

- *Python Tutorial* discute della definizione e del sollevamento delle vostre eccezioni, ed infine della gestione di eccezioni multiple, tutto in una volta.
- *Python Library Reference* riassume tutte le eccezioni built−in.
- *Python Library Reference* documenta il modulo `getpass`.
- *Python Library Reference* documenta il modulo `traceback`, che consente un accesso a basso livello agli attributi delle funzioni dopo che è stata sollevata un'eccezione.
- *Python Reference Manual* discute del funzionamento interno del blocco `try...except`.

4.11. Oggetti file

Python ha una funzione built−in, `open`, per aprire un file su disco. `open` ritorna un oggetto di tipo file, che dispone di metodi e attributi per ottenere informazioni sul file aperto e manipolarlo.

Esempio 4.23. Aprire un file

```
>>> f = open("/music/_singles/kairo.mp3", "rb") (1)
>>> f                                           (2)
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.mode                                      (3)
'rb'
>>> f.name                                      (4)
```

```
'/music/_singles/kairo.mp3'
```

(1) Il metodo `open` può prendere fino a tre parametri: il nome del file, una modalità di apertura e una dimensione di buffer. Solo il primo, il nome, è obbligatorio; gli altri due sono opzionali. Se non è specificato, il file è aperto per la lettura in modo testo. In questo esempio apriamo il file per la lettura in binario (`print open.__doc__` visualizza la spiegazione completa di tutte le modalità possibili).

(2) La funzione `open` ritorna un oggetto (a questo punto non dovrebbe più essere una sorpresa per voi). Un oggetto file ha molti utili attributi.

(3) L'attributo `mode` di un oggetto file ritorna la modalità di apertura del file.

(4) L'attributo `name` di un oggetto file ritorna il nome del file con cui è stato aperto.

Esempio 4.24. Leggere un file

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.tell()              (1)
0
>>> f.seek(-128, 2)       (2)
>>> f.tell()              (3)
7542909
>>> tagData = f.read(128) (4)
>>> tagData
'TAGKAIRO****THE BEST GOA         ***DJ MARY-JANE***            Rave Mix                       2000htt
>>> f.tell()              (5)
7543037
```

(1) Un oggetto file contiene lo stato del file con cui è stato creato. Il metodo `tell` ritorna la posizione corrente all'interno del file. Poiché non abbiamo ancora fatto nulla con questo file, la posizione corrente è `0`, ovvero l'inizio del file.

(2) Il metodo `seek` di un oggetto file sposta la posizione corrente all'interno del file. Il secondo parametro specifica il significato del primo; `0` significa "spostati a una posizione assoluta" (contando dall'inizio del file), `1` significa che il primo parametro è relativo alla posizione attuale, e `2` significa "spostati relativamente dalla fine del file". Poiché i tag MP3 che stiamo cercando sono memorizzati alla fine del file, usiamo `2` e diciamo all'oggetto file di spostare la posizione attuale a `128` byte prima della fine del file.

(3) Il metodo `tell` conferma che la posizione corrente all'interno del file è cambiata.

(4) Il metodo `read` legge un numero specificato di byte dal file aperto e ritorna una stringa contenente i dati che sono stati letti. Il parametro opzionale specifica il massimo numero di byte da leggere. In assenza di questo parametro, il metodo `read` legge il file fino alla fine. (Avremmo potuto scrivere semplicemente `read()` qui, poiché sappiamo esattamente qual'è la posizione corrente nel file e stiamo, in effetti, leggendo gli ultimi 128 byte.) I dati letti sono assegnati alla variabile `tagData` e la posizione corrente viene aggiornata.

(5) Il metodo `tell` conferma che la posizione corrente è stata aggiornata. Se fate i calcoli, vedrete che dopo aver letto 128 byte, la posizione è avanzata di 128.

Esempio 4.25. Chiudere un file

```
>>> f
<open file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed  (1)
0
>>> f.close() (2)
>>> f
<closed file '/music/_singles/kairo.mp3', mode 'rb' at 010E3988>
>>> f.closed
1
```

```
>>> f.seek(0) (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.tell()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.read()
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: I/O operation on closed file
>>> f.close() (4)
```

(1) L'attributo `closed` di un oggetto file indica se l'oggetto ha un file aperto oppure no. In questo caso, il file è ancora aperto (`closed` vale `0`). I file aperti consumano risorse di sistema e a seconda della modalità di apertura, gli altri programmi potrebbero non essere in grado di accedervi. È importante chiudere i file appena avete finito di usarli.

(2) Per chiudere un file, chiamate il metodo `close` dell'oggetto file. Questo metodo libera il lock (se presente) che state tenendo sul file, svuota i buffer di scrittura (se presenti) che il sistema non ha ancora scritto su disco e rilascia le risorse di sistema. L'attributo `closed` conferma che il file è stato chiuso.

(3) Solo perché il file è stato chiuso, non significa che l'oggetto file non esista più. La variabile `f` continua ad esistere finché non esce dallo scope corrente o è cancellata manualmente. In ogni caso, nessuno dei metodi che manipolano un file funzionerà una volta che il file è stato chiuso, ma solleveranno un'eccezione.

(4) Chiamare `close` su un oggetto il cui file è già stato chiuso *non* genera un'eccezione; fallisce silenziosamente.

Esempio 4.26. Oggetti file in `MP3FileInfo`

```
    try:                                (1)
        fsock = open(filename, "rb", 0) (2)
        try:
            fsock.seek(-128, 2)         (3)
            tagdata = fsock.read(128)   (4)
        finally:                        (5)
            fsock.close()
        .
        .
        .
    except IOError:                     (6)
        pass
```

(1) Poiché aprire e leggere un file è un'operazione che potrebbe generare un'eccezione, tutto il codice è racchiuso in un blocco `try...except` (e qui dovreste cominciare ad apprezzare le indentazioni standard di Python...).

(2) La funzione `open` può generare un `IOError` (forse il file non esiste).

(3) Il metodo `seek` può generare un `IOError` (forse il file è più piccolo di 128 byte).

(4) Il metodo `read` può generare un `IOError` (forse il disco ha un settore illeggibile, o è un disco di rete e la rete non è disponibile).

(5) Questo è nuovo: un blocco `try...finally`. Una volta che il file è stato aperto con successo dalla funzione `open`, vogliamo essere assolutamente sicuri che lo chiuderemo, anche se venisse generata un'eccezione dai metodi `seek` o `read`. A questo serve il blocco `try...finally`: il codice nel blocco `finally` verrà *sempre* eseguito, anche se qualcosa nel blocco `try` generasse un'eccezione. Pensate al blocco `finally` come a codice che viene eseguito "all'uscita", indipendentemente da ciò che è successo prima.

(6) Infine, gestiamo la nostra eccezione `IOError` che potrebbe essere generata dalla chiamata a `open`, `seek`, o `read`. Qui non fa differenza, perché ci limitiamo ad ignorarla e continuare. Ricordate, `pass` è un comando Python che non fa nulla. Questo è perfettamente lecito; "gestire" un'eccezione può significare esplicitamente non fare nulla. L'eccezione verrà comunque considerata gestita e il programma continuerà normalmente con la riga di codice successiva al blocco `try...except`.

Ulteriori letture

- *Python Tutorial* discute della lettura e della scrittura dei file, come la lettura una riga per volta in una lista.
- eff−bot discute l'efficienza e le prestazioni di alcuni sistemi di lettura di file.
- Python Knowledge Base risponde alle domande più frequenti sui file.
- *Python Library Reference* riassume tutti i metodi degli oggetti file.

4.12. Cicli `for`

Come molti altri linguaggi, Python ha i cicli `for`. L'unica ragione per cui non si sono visti finora è perché Python è valido per così tante cose che spesso non se ne sente tanto il bisogno.

Molti altri linguaggi non hanno tipi di liste così potenti come quelle di Python, cosicché si finisce per fare un sacco di lavoro in più a mano, dovendo specificare l'inizio, la fine e l'incremento necessario per definire un intervallo di numeri o di caratteri o altre entità su cui è possibile iterare. In Python invece un ciclo `for` itera semplicemente su una lista, nello stesso modo in cui lavorano le list comprehensions.

Esempio 4.27. Introduzione al ciclo `for`

```
>>> li = ['a', 'b', 'e']
>>> for s in li:         (1)
...     print s          (2)
a
b
e
>>> print "\n".join(li)  (3)
a
b
e
```

(1) La sintassi per un ciclo `for` è simile a quella delle list comprehensions. La variabile `li` è una lista e la variabile `s` assume a turno il valore di ogni elemento della lista, a partire dal primo elemento.

(2) Come in una istruzione `if` o in ogni altro blocco indentato, un ciclo `for` può avere un qualsiasi numero di linee al uso interno.

(3) Questa è la ragione per cui non si è ancora visto un ciclo `for`: non ne abbiamo ancora avuto il bisogno. È incredibile quanto spesso si usi il ciclo `for` in altri linguaggi quando tutto quello che si vuole realmente è una `join` o una list comprehension.

Esempio 4.28. Semplici contatori

```
>>> for i in range(5):       (1)
...     print i
0
1
2
3
4
>>> li = ['a', 'b', 'c', 'd', 'e']
```

```
>>> for i in range(len(li)): (2)
...     print li[i]
a
b
c
d
e
```

(1) Eseguire un "normale" (secondo gli standard Visual Basic) ciclo `for` su di un contatore è anche semplice. Come si vede nell'Esempio 2.28, Assegnare valori consecutivi , la funzione `range` produce una lista di interi, attraverso la quale iteriamo. Lo so, sembra un po' strano, ma è occasionalmente (e sottolineo *occasionalmente*) utile avere un ciclo con contatore.

(2) Non fatelo mai. Questo è il modo di pensare secondo lo stile di Visual Basic. Occorre staccarsene. Semplicemente, iterate attraverso una lista, come mostrato nell'esempio precedente.

Esempio 4.29. Iterare sugli elementi di un dizionario

```
>>> for k, v in os.environ.items(): (1) (2)
...     print "%s=%s" % (k, v)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
>>> print "\n".join(["%s=%s" % (k, v) for k, v in os.environ.items()]) (3)
USERPROFILE=C:\Documents and Settings\mpilgrim
OS=Windows_NT
COMPUTERNAME=MPILGRIM
USERNAME=mpilgrim

[...snip...]
```

(1) `os.environ` è il dizionario delle variabili d'ambiente definite dal sistema su cui gira Python. In Windows, queste corrispondono alle variabili utente e di sistema accessibili da MS−DOS. In UNIX, sono le variabili esportate dallo script di inizializzazione della shell usata. In Mac OS, non esiste il concetto di variabile d'ambiente, per cui questo dizionario è vuoto.

(2) `os.environ.items()` restituisce una lista di tuple: `[(`*`chiave1`*`, `*`valore1`*`), (`*`chiave2`*`, `*`valore2`*`), ...]`. Il ciclo `for` itera su questa lista. Al primo giro, si assegna *`chiave1`* alla variabile `k` e *`valore1`* alla variabile `v`, cosicché `k` = `USERPROFILE` e `v` = `C:\Documents and Settings\mpilgrim`. Al secondo giro, la variabile `k` prende il valore della seconda chiave, `OS` e la variabile `v` prende il valore corrispondente, `Windows_NT`.

(3) Con assegnamenti multipli e list comprehensions, è possibile rimpiazzare un intero ciclo `for` con una singola istruzione. Se farlo o no in codice non didattico dipende dal personale stile di codifica di ognuno; Io preferisco questo modo perché rende evidente che quello che si vuole fare è mappare un dizionario in una lista, quindi unire tutti gli elementi di una lista in una singola stringa. Altri programmatori preferiscono scriverlo come un ciclo `for`. Si noti che il risultato è lo stesso in entrambi i casi, sebbene questa versione sia leggermente più veloce perché vi è una sola istruzione `print` invece di molte.

Esempio 4.30. Ciclo `for` in `MP3FileInfo`

```
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
```

```
                "comment" : ( 97, 126, stripnulls),
                "genre"   : (127, 128, ord)} (1)
    .
    .
    .
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items(): (2)
                    self[tag] = parseFunc(tagdata[start:end])                (3)
```

(1) `tagDataMap` è un attributo di classe che definisce i tag che si trovano in un file MP3. Questi tag sono memorizzati in campi a lunghezza fissa; degli ultimi 128 byte letti, i byte da 3 a 32 sono sempre il titolo della canzone, i byte 33−62 sono sempre il nome dell'artista e così via. Si noti che `tagDataMap` è un dizionario che contiene tuple ed ogni tupla contiene due interi ed un riferimento a funzione.

(2) Questo sembra complicato ma non lo è. La struttura delle variabili usate nell'istruzione `for`, corrisponde alla struttura degli elementi della lista, restituita dal metodo `items`. Occorre ricordarsi che il metodo `items` restituisce una lista di tuple nella forma `(`*`chiave, valore`*`)`. Il primo elemento di questa lista è `("title", (3, 33, <function stripnulls>))`, quindi al primo ciclo di iterazione, la variabile `tag` prende il valore `"title"`, la variabile `start` prende il valore `3`, la variabile `end` prende il valore `33` e la variabile `parseFunc` prende il valore `stripnulls`.

(3) Una volta estratti tutti i parametri per un singolo MP3, salvare i valori dei tag estratti è facile. Prima si separano dalla lista dei byte contenenti i `tagdata`, i byte da `start` ad `end`, per ottenere i byte corrispondenti ad un singolo `tag`; poi si chiama la funzione invocata da `parseFunc`, per elaborare i byte estratti ed infine si assegna il valore risultante della chiave `tag` al pseudo−dizionario dell'oggetto `self`. Una volta iterato su tutti gli elementi di `tagDataMap`, l'oggetto `self` contiene i valori di tutti i `tag` e si sa questo cosa vuol dire. (ndt: i `tag` risultano accessibili come normali attributi dell'oggetto).

4.13. Ancora sui moduli

I moduli, come qualunque altra cosa in Python, sono oggetti. Una volta importato, si può sempre ottenere un riferimento ad un modulo attraverso il dizionario globale `sys.modules`.

Esempio 4.31. Introduzione a `sys.modules`

```
>>> import sys                          (1)
>>> print '\n'.join(sys.modules.keys()) (2)
win32api
os.path
os
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
```

(1) Il modulo `sys` contiene informazioni a livello di sistema, quali la versione di Python che si sta usando (`sys.version` oppure `sys.version_info`), nonché opzioni di sistema come la massima profondità di ricorsione permessa (`sys.getrecursionlimit()` e `sys.setrecursionlimit()`).

(2) `sys.modules` è un dizionario contenente tutti i moduli che sono stati importati sin dalla partenza di Python, la chiave è il nome del modulo, il valore è l'oggetto−modulo. Si noti che

questo dizionario non comprende solo i moduli importati esplicitamente dal *vostro* programma. Python carica alcuni moduli alla partenza; se poi si sta usando la IDE di Python, `sys.modules` contiene tutti i moduli importati da tutti i programmi che sono stati lanciati tramite l'IDE.

Esempio 4.32. Usare `sys.modules`

```
>>> import fileinfo         (1)
>>> print '\n'.join(sys.modules.keys())
win32api
os.path
os
fileinfo
exceptions
__main__
ntpath
nt
sys
__builtin__
site
signal
UserDict
stat
>>> fileinfo
<module 'fileinfo' from 'fileinfo.pyc'>
>>> sys.modules["fileinfo"] (2)
<module 'fileinfo' from 'fileinfo.pyc'>
```

(1) Quando vengono importati nuovi moduli, questi vengono aggiunti al dizionario `sys.modules`. Questo spiega perché importare la seconda volta lo stesso modulo è molto veloce: Python lo ha già caricato e memorizzato nel modulo `sys.modules`, per cui importarlo la seconda volta corrisponde semplicemente ad una ricerca nel dizionario.

(2) Dato il nome (come stringa) di ogni modulo precedentemente importato, è possibile ottenere un riferimento al modulo stesso attraverso il dizionario `sys.modules`.

Esempio 4.33. L'attributo di classe `__module__`

```
>>> from fileinfo import MP3FileInfo
>>> MP3FileInfo.__module__              (1)
'fileinfo'
>>> sys.modules[MP3FileInfo.__module__] (2)
<module 'fileinfo' from 'fileinfo.pyc'>
```

(1) Ogni classe Python ha un attributo di classe predefinito di nome `__module__`, che contiene il modulo in cui la classe è definita.

(2) Combinando questo, con il diziorario `sys.modules`, si può ottenere un riferimento al modulo in cui una classe è definita.

Esempio 4.34. Uso di `sys.modules` in `fileinfo.py`

```
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):       (1)
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]        (2)
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo (3)
```

(1)

Questa è una funzione con due argomenti: il `nome del file` è richiesto, ma il parametro `module` è opzionale ed ha come predefinito il modulo che contiene la classe `FileInfo`. Questo sembra inefficiente, perché si potrebbe credere che Python calcoli l'espressione con `sys.modules` ogni volta che la funzione è invocata. In realtà, Python calcola l'espressione usata come valore predefinito solo una volta, la prima volta che il modulo viene importato. Come si vedrà più avanti, questa funzione non viene mai chiamata con l'argomento `module`, cosicché `module` è usato come una costante definita a livello di funzione.

(2) Scaveremo su questa traccia più avanti, dopo che ci saremo immersi nell'analisi del modulo `os`. Per ora fidatevi, la variabile `subclass` finisce per avere come valore il nome di una classe, come `MP3FileInfo`.

(3) Si sa già tutto su `getattr`, che permette di ottenere il riferimento ad un oggetto dal suo nome. `hasattr` è una funzione complementare che controlla se un oggetto ha un particolare attributo; in questo caso, se un modulo ha una classe particolare (sebbene funzioni per ogni oggetto ed ogni attributo, proprio come `getattr`). In Italiano, questa linea di codice recita: "se questo modulo contiene una classe con lo stesso nome di `subclass` allora restituisci tale classe, altrimenti restituisci la classe base `FileInfo`".

Ulteriori letture

- Il *Python Tutorial* tratta esattamente di come e quando gli argomenti predefiniti vengono valutati.
- Il *Python Library Reference* documenta il modulo `sys`.

4.14. Il modulo os

Il modulo `os` ha molte funzioni utili per manipolare file e processi e `os.path` ha molte funzioni per manipolare percorsi di file e directory.

Esempio 4.35. Costruire pathnames

```
>>> import os
>>> os.path.join("c:\\music\\ap\\", "mahadeva.mp3") (1) (2)
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.join("c:\\music\\ap", "mahadeva.mp3")   (3)
'c:\\music\\ap\\mahadeva.mp3'
>>> os.path.expanduser("~")                         (4)
'c:\\Documents and Settings\\mpilgrim\\My Documents'
>>> os.path.join(os.path.expanduser("~"), "Python") (5)
'c:\\Documents and Settings\\mpilgrim\\My Documents\\Python'
```

(1) `os.path` è un riferimento ad un modulo che dipende dalla piattaforma sulla quale lavorate. Proprio come `getpass` considera le differenze fra le piattaforme, impostando `getpass` ad una funzione specifica, anche `os` ne tiene conto impostando `path` come un modulo specifico per piattaforma in uso.

(2) La funzione `join` di `os.path` costruisce un pathname a partire da uno o più pathname parziali. In questo semplice caso, si limita a concatenare delle stringhe. (Notate che occuparsi di pathname su Windows è odioso a causa del fatto che il carattere backslash deve essere escapato.)

(3) Il questo esempio un po' banale, `join` aggiungerà un backslash extra al pathname prima di unirlo al nome del file. Ero contentissimo quando lo scoprii, dato che `addSlashIfNecessary` è una delle piccole e stupide funzioni che devo sempre scrivere, quando costruisco la mia toolbox in un nuovo linguaggio. *Non* scrivete queste piccole e stupide funzioni in Python; qualcun altro se ne è già occupato per voi.

(4)

`expanduser` espanderà un pathname che usa ~ per rappresentare la home directory dell'utente attuale. Questo funziona su ogni piattaforma ove gli utenti hanno una home directory, come Windows, UNIX e Mac OS X, non ha effetto su Mac OS.

(5) Combinando queste tecniche, potete facilmente costruire percorsi per directory e file sotto la home directory dell'utente.

Esempio 4.36. Dividere i pathnames

```
>>> os.path.split("c:\\music\\ap\\mahadeva.mp3")                        (1)
('c:\\music\\ap', 'mahadeva.mp3')
>>> (filepath, filename) = os.path.split("c:\\music\\ap\\mahadeva.mp3") (2)
>>> filepath                                                            (3)
'c:\\music\\ap'
>>> filename                                                            (4)
'mahadeva.mp3'
>>> (shortname, extension) = os.path.splitext(filename)                 (5)
>>> shortname
'mahadeva'
>>> extension
'.mp3'
```

(1) La funzione `split` divide un pathname completo e ritorna una tupla contenente il percorso ed il nome del file. Ricordate quando dissi che potevate usare assegnamenti multi−variabile per ritornare valori multipli da una funzione? Bene, `split` è una di queste funzioni.

(2) Assegnamo il valore di ritorno della funzione `split` ad una tupla di due variabili. Ogni variabile riceve il valore dell'elemento corrispondente dalla tupla ritornata.

(3) La prima variabile, `filepath`, riceve il valore del primo elemento della tupla ritornata da `split`, il percorso del file.

(4) La seconda variabile, `filename`, riceve il valore del secondo elemento della tupla ritornata da `split`, il nome del file.

(5) `os.path` contiene anche una funzione `splitext`, che divide il nome di un file e ritorna una tupla contenente il nome del file e la sua estensione. Usiamo la stessa tecnica per assegnare ognuno di questi a variabili separate.

Esempio 4.37. Elencare directory

```
>>> os.listdir("c:\\music\\_singles\\")                                  (1)
['a_time_long_forgotten_con.mp3', 'hellraiser.mp3', 'kairo.mp3',
'long_way_home1.mp3', 'sidewinder.mp3', 'spinning.mp3']
>>> dirname = "c:\\"
>>> os.listdir(dirname)                                                  (2)
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'cygwin', 'docbook',
'Documents and Settings', 'Incoming', 'Inetpub', 'IO.SYS', 'MSDOS.SYS', 'Music',
'NTDETECT.COM', 'ntldr', 'pagefile.sys', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
>>> [f for f in os.listdir(dirname) if os.path.isfile(os.path.join(dirname, f))] (3)
['AUTOEXEC.BAT', 'boot.ini', 'CONFIG.SYS', 'IO.SYS', 'MSDOS.SYS',
'NTDETECT.COM', 'ntldr', 'pagefile.sys']
>>> [f for f in os.listdir(dirname) if os.path.isdir(os.path.join(dirname, f))]  (4)
['cygwin', 'docbook', 'Documents and Settings', 'Incoming',
'Inetpub', 'Music', 'Program Files', 'Python20', 'RECYCLER',
'System Volume Information', 'TEMP', 'WINNT']
```

(1) La funzione `listdir` prende un pathname e ritorna una lista del contenuto della directory.

(2) `listdir` ritorna sia file che directory, senza indicare quale sia.

(3) Potete usare il list filtering e la funzione `isfile` del modulo `os.path` per separare i file dalle directory. `isfile` prende un pathname e ritorna 1, nel caso in cui il percorso rappresenti un file e 0 altrimenti. Qui stiamo usando `os.path.join` per assicurarci un pathname completo, ma `isfile` funziona anche con un percorso parziale, relativo alla directory corrente. Potete usare `os.getcwd()` per ottenere la directory corrente.

(4) `os.path` ha anche una funzione `isdir` che ritorna 1 se il percorso rappresenta una directory e 0 altrimenti. Potete usarla per ottenere la lista delle sottodirectory di un percorso.

Esempio 4.38. Elencare directory in `fileinfo.py`

```
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]
```

Queste due linee di codice fanno tutto ciò che abbiamo imparato finora sul modulo `os` e anche qualcos'altro.

1. `os.listdir(directory)` ritorna una lista di tutti i file compresi in `directory`.
2. Iterando nella lista con `f`, usiamo `os.path.normcase(f)` per normalizzare il nome del file o della directory, in accordo al comportamento predefinito del sistema operativo in uso. `normcase` è una piccola ed utile funzione che rimedia al fatto che i sistemi operativi case−insensitive pensino che `mahadeva.mp3` e `mahadeva.MP3` siano lo stesso file. Per esempio, su Windows e Mac OS, `normcase` convertirà l'intero filename in caratteri minuscoli; sui sistemi UNIX−compatibili, ritornerà il nome del file invariato.
3. Iterando attraverso la lista normalizzata, ancora con `f`, usiamo `os.path.splitext(f)` per dividere ogni nome di file dalla sua estensione.
4. Per ogni file, vediamo se l'estensione è presente nella lista delle estensioni di cui stiamo tenendo conto (`fileExtList`, che è stata passata alla funzione `listDirectory`).
5. Per ogni file trovato, usiamo `os.path.join(directory, f)` per costruire il percorso completo del file e ritornare una lista dei percorsi completi.

   Nota: Quando usare il modulo os
   Ogniqualvolta sia possibile, dovreste usare le funzioni in `os` e `os.path` per file, directory e manipolazione dei percorsi. Questi moduli sono dei wrapper ai moduli specifici per piattaforma, per cui funzioni come `os.path.split` funzionano su UNIX, Windows, Mac OS ed ogni altra possibile piattaforma supportata da Python.

Ulteriori letture

- Python Knowledge Base risponde a domande sul modulo `os` .
- *Python Library Reference* documenta il modulo `os` ed anche il modulo `os.path`.

4.15. Mettere tutto insieme

Ancora una volta tutte le nostre pedine del domino sono al loro posto. Abbiamo visto come funziona ogni linea di codice, adesso facciamo un passo indietro e vediamo come il tutto venga messo insieme.

Esempio 4.39. `listDirectory`

```
def listDirectory(directory, fileExtList):                              (1)
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
```

```
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]                          (2)
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):       (3)
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]        (4)
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo (5)
    return [getFileInfoClass(f)(f) for f in fileList]                              (6)
```

(1) `listDirectory` è la pricipale attrazione dell'intero modulo. Prende una directory (come `c:\music\_singles\`, nel mio caso) ed una lista di interessanti estensioni di file (come `['.mp3']`) e ritorna una lista di istanze di classe che funzionano come un dizionario. E lo fa in poche schiette linee di codice.

(2) Come abbiamo visto nella precedente sezione, questa linea di codice prende una lista dei pathname di tutti i file in `directory` che hanno un'estensione interessante (come specificato da `fileExtList`).

(3) Ai programmatori Pascal della vecchia scuola potranno essere familiari, ma molte persone hanno sgranato gli occhi quando gli ho detto che Python supporta le *funzioni annidate* −− letteralmente, una funzione dentro un'altra funzione. La funzione annidata `getFileInfoClass` può essere chiamata solo dalla funzione nella quale è definita, `listDirectory`. Come con ogni altra funzione, non avete bisogno di dichiarazione d'interfaccia od altre cose strane; limitatevi a definire la funzione e scriverla.

(4) Adesso che avete visto il modulo `os`, questa linea dovrebbe avere più significato. Di fatto, prende l'estensione del file (`os.path.splitext(filename)[1]`), la forza a caratteri maiuscoli (`.upper()`), affetta via il punto (`[1:]`) e costruisce un nome di classe con la formattazione di stringa. Quindi `c:\music\ap\mahadeva.mp3` diventa `.mp3` che diventa `.MP3` che diventa `MP3` che diventa `MP3FileInfo`.

(5) Avendo costruito il nome del gestore di classe che gestirebbe questo file, controlliamo se quel gestore di classe realmente esiste in questo modulo. Se lo fa, ritorniamo la classe, altrimenti ritorniamo la classe base `FileInfo`. Questo è un punto molto importante: *questa funzione ritorna una classe*. Non un'istanza di una classe, ma la classe stessa.

(6) Per ogni file nella nostra lista "file interessanti" (`fileList`), possiamo chiamare `getFileInfoClass` con il filename (`f`). Chiamando `getFileInfoClass(f)` viene ritornata una classe; noi non sappiamo esattamente quale, ma non ci importa. Creiamo poi un'istanza di questa classe (qualunque essa sia) e passiamo il nome del file (ancora `f`), al metodo di `__init__`. Come abbiamo visto prima in questo capitolo, il metodo `__init__` di `FileInfo` imposta `self["name"]`, che richiama `__setitem__`, che viene sovrascritto nella classe discendente (`MP3FileInfo`), in modo da analizzare correttamente il file, per ottenerne il metadato. Facciamo tutto ciò per ogni file interessante e ritorniamo una lista delle istanze risultanti.

Notate che `listDirectory` è assolutamente generica; non conosce anticipatamente quale tipo di file otterrà o quali classi tra quelle definite potrebbero potenzialmente gestire questi file. Essa analizza la directory, alla ricerca dei file da processare e poi introspetta il suo stesso modulo per vedere quali speciali gestori di classi (come `MP3FileInfo`) sono definiti. Potete estendere questo programma per gestire altri tipi di file, semplicemente definendo una classe col nome appropriato: `HTMLFileInfo` per i file HTML, `DOCFileInfo` per i file `.doc` di Word e così via. `listDirectory` li gestirà tutti, senza modifiche, delegando il lavoro effettivo alle classi appropriate e collazionando i risultati.

4.16. Sommario

Adesso il programma `fileinfo.py` dovrebbe avere un senso compiuto.

Esempio 4.40. `fileinfo.py`

```
"""Framework for getting filetype-specific metadata.

Instantiate appropriate class with filename.  Returned object acts like a
dictionary, with key-value pairs for each piece of metadata.
```

```
    import fileinfo
    info = fileinfo.MP3FileInfo("/music/ap/mahadeva.mp3")
    print "\\n".join(["%s=%s" % (k, v) for k, v in info.items()])

Or use listDirectory function to get info on all files in a directory.
    for info in fileinfo.listDirectory("/music/ap/", [".mp3"]):
        ...

Framework can be extended by adding classes for particular file types, e.g.
HTMLFileInfo, MPGFileInfo, DOCFileInfo.  Each class is completely responsible for
parsing its files appropriately; see MP3FileInfo for example.
"""
import os
import sys
from UserDict import UserDict

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

class FileInfo(UserDict):
    "store file metadata"
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename

class MP3FileInfo(FileInfo):
    "store ID3v1.0 MP3 tags"
    tagDataMap = {"title"   : (  3,  33, stripnulls),
                  "artist"  : ( 33,  63, stripnulls),
                  "album"   : ( 63,  93, stripnulls),
                  "year"    : ( 93,  97, stripnulls),
                  "comment" : ( 97, 126, stripnulls),
                  "genre"   : (127, 128, ord)}

    def __parse(self, filename):
        "parse ID3v1.0 tags from MP3 file"
        self.clear()
        try:
            fsock = open(filename, "rb", 0)
            try:
                fsock.seek(-128, 2)
                tagdata = fsock.read(128)
            finally:
                fsock.close()
            if tagdata[:3] == "TAG":
                for tag, (start, end, parseFunc) in self.tagDataMap.items():
                    self[tag] = parseFunc(tagdata[start:end])
        except IOError:
            pass

    def __setitem__(self, key, item):
        if key == "name" and item:
            self.__parse(item)
        FileInfo.__setitem__(self, key, item)

def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particular extensions"
    fileList = [os.path.normcase(f) for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f) for f in fileList \
                if os.path.splitext(f)[1] in fileExtList]
    def getFileInfoClass(filename, module=sys.modules[FileInfo.__module__]):
        "get file info class from filename extension"
```

```
        subclass = "%sFileInfo" % os.path.splitext(filename)[1].upper()[1:]
        return hasattr(module, subclass) and getattr(module, subclass) or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]

if __name__ == "__main__":
    for info in listDirectory("/music/_singles/", [".mp3"]):
        print "\n".join(["%s=%s" % (k, v) for k, v in info.items()])
        print
```

Prima di immergerci nel prossimo capitolo, accertiamoci che abbiate capito i seguenti concetti:

- Importare moduli usando `import module` o `from module import`
- Definire ed istanziare classi
- Definire metodi `__init__` ed altri metodi speciali di classe e capire quando vengono chiamati
- Derivare da `UserDict` per definire classi che agiscono come dizionari
- Definire attributi dato e attributi di classe e capirne la differenza
- Definire metodi privati
- Catturare le eccezioni con `try...except`
- Proteggere le risorse esterne con `try...finally`
- Leggere da file
- Assegnare valori multipli in un colpo solo in un ciclo `for`
- Usare il modulo `os` per tutte le tue manipolazioni di file multi−piattaforma
- Istanziare dinamicamente classi di tipo sconosciuto trattando le classi come oggetti e passargli intorno

---

[4] Non ci sono costanti in Python. Ogni cosa può essere modificata se provate abbastanza. Questo fa parte di uno dei principali concetti di Python: cattivi comportamenti dovrebbero essere scoraggiati, ma non impediti. Se volete veramente modificare il valore di `None`, potete farlo, ma non venite correndo da me se il vostro codice diventa impossibile da debuggare.

[5] Parlando rigorosamente, i metodi privati sono accessibili al di fuori dalla loro classe, ma non *facilmente*. Nulla in Python è realmente privato, internamente i nomi dei metodi e degli attributi privati vengono cambiati al volo per farli sembrare inaccessibili attraverso i nomi che li identificano. Potete accedere al metodo `__parse` della classe `MP3FileInfo` con il nome `_MP3FileInfo__parse`. Ammesso che sia interessante, promettete di non farlo mai e poi mai nel codice reale. I metodi privati sono tali per una ragione, ma come per altre cose in Python, se sono privati è soprattutto per una questione di convezioni, non di imposizione.

[6] O, come alcuni pignoli farebbero notare, il vostro programma effettuerebbe un'azione illegale. Qualunque.

Capitolo 5. Elaborare HTML

5.1. Immergersi

Spesso in comp.lang.python trovo domande del genere: "Come posso creare una lista di tutti gli [headers | immagini | collegamenti] presenti nel mio documento HTML?" "Come posso [analizzare | tradurre | modificare] il testo del mio documento HTML senza modificare i tag?" "Come posso [aggiungere | rimuovere | mettere tra apici] gli attributi di tutti i tag del mio HTML in una volta sola?" Questo capitolo risponderà a tutte queste domande.

Ecco un completo e funzionante programma in Python diviso in due parti. La prima parte, `BaseHTMLProcessor.py`, è un generico strumento che aiuta ad elaborare file HTML passando attraverso tag e blocchi di testo. La seconda parte, `dialect.py`, è un esempio di come usare `BaseHTMLProcessor.py` per tradurre il testo di un documento HTML lasciando inalterati i tag. Leggete le `doc string` ed i commenti per avere un'idea di cosa succede. La maggior parte sembrerà magia nera, perché non è ovvio come vengano chiamati i vari metodi delle classi. Non vi preoccupate, verrà tutto spiegato a tempo debito.

Esempio 5.1. `BaseHTMLProcessor.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
from sgmllib import SGMLParser
import htmlentitydefs

class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        # extend (called by SGMLParser.__init__)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs):
        # called for each start tag
        # attrs is a list of (attr, value) tuples
        # e.g. for <pre class="screen">, tag="pre", attrs=[("class", "screen")]
        # Ideally we would like to reconstruct original tag and attributes, but
        # we may end up quoting attribute values that weren't quoted in the source
        # document, or we may change the type of quotes around the attribute value
        # (single to double quotes).
        # Note that improperly embedded non-HTML code (like client-side Javascript)
        # may be parsed incorrectly by the ancestor, causing runtime script errors.
        # All non-HTML code must be enclosed in HTML comment tags (<!-- code -->)
        # to ensure that it will pass through this parser unaltered (in handle_comment).
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):
        # called for each end tag, e.g. for </pre>, tag will be "pre"
        # Reconstruct the original end tag.
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):
        # called for each character reference, e.g. for " ", ref will be "160"
        # Reconstruct the original character reference.
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):
        # called for each entity reference, e.g. for "&copy;", ref will be "copy"
        # Reconstruct the original entity reference.
```

```
        self.pieces.append("&%(ref)s" % locals())
        # standard HTML entities are closed with a semicolon; other entities are not
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):
        # called for each block of plain text, i.e. outside of any tag and
        # not containing any character or entity references
        # Store the original text verbatim.
        self.pieces.append(text)

    def handle_comment(self, text):
        # called for each HTML comment, e.g. <!-- insert Javascript code here -->
        # Reconstruct the original comment.
        # It is especially important that the source document enclose client-side
        # code (like Javascript) within comments so it can pass through this
        # processor undisturbed; see comments in unknown_starttag for details.
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):
        # called for each processing instruction, e.g. <?instruction>
        # Reconstruct original processing instruction.
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        # called for the DOCTYPE, if present, e.g.
        # <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
        #     "http://www.w3.org/TR/html4/loose.dtd">
        # Reconstruct original DOCTYPE
        self.pieces.append("<!%(text)s>" % locals())

    def output(self):
        """Return processed HTML as a single string"""
        return "".join(self.pieces)
```

Esempio 5.2. `dialect.py`

```
import re
from BaseHTMLProcessor import BaseHTMLProcessor

class Dialectizer(BaseHTMLProcessor):
    subs = ()

    def reset(self):
        # extend (called from __init__ in ancestor)
        # Reset all data attributes
        self.verbatim = 0
        BaseHTMLProcessor.reset(self)

    def start_pre(self, attrs):
        # called for every <pre> tag in HTML source
        # Increment verbatim mode count, then handle tag like normal
        self.verbatim += 1
        self.unknown_starttag("pre", attrs)

    def end_pre(self):
        # called for every </pre> tag in HTML source
        # Decrement verbatim mode count
        self.unknown_endtag("pre")
        self.verbatim -= 1
```

```
    def handle_data(self, text):
        # override
        # called for every block of text in HTML source
        # If in verbatim mode, save text unaltered;
        # otherwise process the text with a series of substitutions
        self.pieces.append(self.verbatim and text or self.process(text))

    def process(self, text):
        # called from handle_data
        # Process text block by performing series of regular expression
        # substitutions (actual substitions are defined in descendant)
        for fromPattern, toPattern in self.subs:
            text = re.sub(fromPattern, toPattern, text)
        return text

class ChefDialectizer(Dialectizer):
    """convert HTML to Swedish Chef-speak

    based on the classic chef.x, copyright (c) 1992, 1993 John Hagerman
    """
    subs = ((r'a([nu])', r'u\1'),
            (r'A([nu])', r'U\1'),
            (r'a\B', r'e'),
            (r'A\B', r'E'),
            (r'en\b', r'ee'),
            (r'\Bew', r'oo'),
            (r'\Be\b', r'e-a'),
            (r'\be', r'i'),
            (r'\bE', r'I'),
            (r'\Bf', r'ff'),
            (r'\Bir', r'ur'),
            (r'(\w*?)i(\w*?)$', r'\1ee\2'),
            (r'\bow', r'oo'),
            (r'\bo', r'oo'),
            (r'\bO', r'Oo'),
            (r'the', r'zee'),
            (r'The', r'Zee'),
            (r'th\b', r't'),
            (r'\Btion', r'shun'),
            (r'\Bu', r'oo'),
            (r'\BU', r'Oo'),
            (r'v', r'f'),
            (r'V', r'F'),
            (r'w', r'w'),
            (r'W', r'W'),
            (r'([a-z])[.]', r'\1.  Bork Bork Bork!'))

class FuddDialectizer(Dialectizer):
    """convert HTML to Elmer Fudd-speak"""
    subs = ((r'[rl]', r'w'),
            (r'qu', r'qw'),
            (r'th\b', r'f'),
            (r'th', r'd'),
            (r'n[.]', r'n, uh-hah-hah-hah.'))

class OldeDialectizer(Dialectizer):
    """convert HTML to mock Middle English"""
    subs = ((r'i([bcdfghjklmnpqrstvwxyz])e\b', r'y\1'),
            (r'i([bcdfghjklmnpqrstvwxyz])e', r'y\1\1e'),
            (r'ick\b', r'yk'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'e[ea]([bcdfghjklmnpqrstvwxyz])', r'e\1e'),
            (r'([bcdfghjklmnpqrstvwxyz])y', r'\1ee'),
```

```
            (r'([bcdfghjklmnpqrstvwxyz])er', r'\1re'),
            (r'([aeiou])re\b', r'\1r'),
            (r'ia([bcdfghjklmnpqrstvwxyz])', r'i\1e'),
            (r'tion\b', r'cioun'),
            (r'ion\b', r'ioun'),
            (r'aid', r'ayde'),
            (r'ai', r'ey'),
            (r'ay\b', r'y'),
            (r'ay', r'ey'),
            (r'ant', r'aunt'),
            (r'ea', r'ee'),
            (r'oa', r'oo'),
            (r'ue', r'e'),
            (r'oe', r'o'),
            (r'ou', r'ow'),
            (r'ow', r'ou'),
            (r'\bhe', r'hi'),
            (r've\b', r'veth'),
            (r'se\b', r'e'),
            (r"'s\b", r'es'),
            (r'ic\b', r'ick'),
            (r'ics\b', r'icc'),
            (r'ical\b', r'ick'),
            (r'tle\b', r'til'),
            (r'll\b', r'l'),
            (r'ould\b', r'olde'),
            (r'own\b', r'oune'),
            (r'un\b', r'onne'),
            (r'rry\b', r'rye'),
            (r'est\b', r'este'),
            (r'pt\b', r'pte'),
            (r'th\b', r'the'),
            (r'ch\b', r'che'),
            (r'ss\b', r'sse'),
            (r'([wybdp])\b', r'\1e'),
            (r'([rnt])\b', r'\1\1e'),
            (r'from', r'fro'),
            (r'when', r'whan'))

def translate(url, dialectName="chef"):
    """fetch URL and translate using dialect

    dialect in ("chef", "fudd", "olde")"""
    import urllib
    sock = urllib.urlopen(url)
    htmlSource = sock.read()
    sock.close()
    parserName = "%sDialectizer" % dialectName.capitalize()
    parserClass = globals()[parserName]
    parser = parserClass()
    parser.feed(htmlSource)
    parser.close()
    return parser.output()

def test(url):
    """test all dialects against URL"""
    for dialect in ("chef", "fudd", "olde"):
        outfile = "%s.html" % dialect
        fsock = open(outfile, "wb")
        fsock.write(translate(url, dialect))
        fsock.close()
        import webbrowser
        webbrowser.open_new(outfile)
```

```
if __name__ == "__main__":
    test("http://diveintopython.org/odbchelper_list.html")
```

Esempio 5.3. Output di `dialect.py`

L'avvio di questo script tradurrà il paragrafo Introduzione alle liste in pseudo Swedish Chef−speak (dai Muppets), in pseudo Elmer Fudd−speak (dal cartone di Bugs Bunny) e pseudo Middle English (malamente basato su Chaucer's *The Canterbury Tales*). Se guardate nel codice HTML delle pagine in output, noterete che tutti i tag HTML ed i loro attributi non sono stati modificati, mentre il testo tra i tags è stato "tradotto" nello pseudo linguaggio. Se guardate più attentamente vedrete che, di fatto, solo i titoli ed i paragrafi sono stati tradotti; il codice e gli esempi sono rimasti invariati.

5.2. Introduciamo `sgmllib.py`

L'elaborazione dell'HTML è suddivisa in tre passi: spezzare l'HTML nei suoi elementi costitutivi, giocherellare con questi pezzi, infine ricostruire i pezzi nuovamente nell'HTML. Il primo passo è fatto da `sgmllib.py`, che è parte della libreria standard di Python

La chiave di lettura di questo capitolo sta nel capire che l'HTML non è solo testo, ma è testo strutturato. La struttura deriva da quella più o meno gerarchica dei tag di inizio e di fine. Solitamente non si lavora con l'HTML in questa maniera; ci si lavora per quel che riguarda il *testo* con un editor di testo, mentre, per quel che riguarda l'aspetto *visuale*, con un web browser o uno strumento per comporre pagine web. `sgmllib.py` mostra invece l'HTML dal punto di vista della sua *struttura*.

`sgmllib.py` contiene una classe importante: `SGMLParser`. `SGMLParser` suddivide l'HTML in parti utili, come tag di inizio e tag di fine. Non appena riesce a spezzare dei dati in parti utili, chiama un proprio metodo sulla base di ciò che ha trovato. Per poter usare questo parser, dovrete derivare la classe `SGMLParser` e sovrascrivere questi metodi. Questo è ciò che intendevo quando dissi che mostra l'HTML dal punto di vista della sua *struttura*: questa struttura determina la sequenza delle chiamate ai metodi e gli argomenti passati a ciascun metodo.

`SGMLParser` suddivide l'HTML in 8 tipi di dati e chiama un metodo separato per ognuno di loro:

Tag di inizio
: Si tratta di un tag HTML che da inizio ad un blocco, come `<html>`, `<head>`, `<body>` o `<pre>` oppure di un tag solitario come `<br>` o `<img>`. Quando trova il *`nometag`* di inizio, `SGMLParser` cerca un metodo chiamato `start_`*`nometag`* oppure `do_`*`nometag`*. Per esempio, quando trova un tag `<pre>`, cercherà un metodo che si chiami `start_pre` oppure un `do_pre`. Se lo trova, `SGMLParser` chiama questo metodo passandogli una lista degli attributi del tag; altrimenti chiama il metodo `unknown_starttag` passandogli il nome del tag e la lista degli attributi.

Tag di fine
: Si tratta di un tag HTML che chiude un blocco, come `</html>`, `</head>`, `</body>`, o `</pre>`. Quando trova un tag di fine, `SGMLParser` cerca un metodo chiamato `end_`*`nometag`*. Se lo trova, `SGMLParser` chiama questo metodo, altrimenti chiama `unknown_endtag` passandogli il nome del tag.

Riferimento a carattere
: Un carattere di escape indicato dal suo equivalente decimale o esadecimale, come ad esempio ` `. Quando viene trovato, `SGMLParser` chiama `handle_charref` passandogli il testo dell'equivalente carattere decimale o esadecimale.

Riferimento a entità
: Una entità HTML, come `&copy;`. Quando viene trovata, `SGMLParser` chiama `handle_entityref` passandogli il nome dell'entità HTML.

Commento
: Un commento HTML, racchiuso tra `<!-- ... -->`. Quando viene trovato, `SGMLParser` chiama `handle_comment` passandogli il corpo del commento.

Istruzioni di elaborazione
: Un'istruzione di elaborazione HTML, racchiusa tra `<? ... >`. Quando viene trovata, `SGMLParser` chiama `handle_pi` passandogli il corpo dell'istruzione di elaborazione.

Dichiarazione
: Una dichiarazione HTML, come ad esempio un `DOCTYPE`, racchiuso tra `<! ... >`. Quando viene trovata, `SGMLParser` chiama `handle_decl` passandogli il corpo della dichiarazione.

Testo
: Un blocco di testo. Qualunque cosa che non si adatti a una della altre 7 categorie. Quando viene trovato, `SGMLParser` chiama `handle_data` passandogli il testo.

Importante: Evoluzione del linguaggio: DOCTYPE
Python 2.0 ha un bug per il quale `SGMLParser` non riconosce del tutto le dichiarazioni (`handle_decl` non viene mai chiamato), questo implica che i `DOCTYPE` vengano ignorati senza che sia segnalato. Questo è stato corretto in Python 2.1.

`sgmllib.py` è fornito assieme ad una suite di test per chiarirne il funzionamento. Potete eseguire `sgmllib.py` passando il nome di un file HTML tramite la linea di comando, esso vi scriverà come risultato i tag e gli altri elementi che ha esaminato. Ciò viene fatto derivando la classe `SGMLParser` e definendo i metodi `unknown_starttag`, `unknown_endtag`, `handle_data` e gli altri metodi, di modo che, semplicemente, scrivano i loro argomenti.

Suggerimento: Specificare argomenti da linea di comando in Windows
Nella IDE di Python su Windows, potete specificare argomenti da linea di comando nella finestra di dialogo "Run script". Argomenti multipli vanno separati da spazi.

Esempio 5.4. Test di esempio di `sgmllib.py`

Questo è un frammento preso dall'indice della versione HTML di questo libro, `toc.html`.

```
<h1>
  <a name='c40a'></a>
  Dive Into Python
</h1>
<p class='pubdate'>
  28 Feb 2001
</p>
<p class='copyright'>
  Copyright copy 2000, 2001 by
  <a href='mailto:f8dy@diveintopython.org' title='send e-mail to the author'>
    Mark Pilgrim
  </a>
</p>
<p>
  <a name='c40ab2b4'></a>
  <b></b>
</p>
<p>
  This book lives at
  <a href='http://diveintopython.org/'>
    http://diveintopython.org/
  </a>
  .
  If you're reading it somewhere else, you may not have the latest version.
</p>
```

Eseguendo questo frammento di codice, attraverso la suite di test di `sgmllib.py`, si ottiene il seguente risultato:

```
start tag: <h1>
start tag: <a name="c40a" >
end tag: </a>
data: 'Dive Into Python'
end tag: </h1>
start tag: <p class="pubdate" >
data: '28 Feb 2001'
end tag: </p>
start tag: <p class="copyright" >
data: 'Copyright '
*** unknown entity ref: &copy;
data: ' 2000, 2001 by '
start tag: <a href="mailto:f8dy@diveintopython.org" title="send e-mail to the author" >
data: 'Mark Pilgrim'
end tag: </a>
end tag: </p>
start tag: <p>
start tag: <a name="c40ab2b4" >
end tag: </a>
start tag: <b>
end tag: </b>
end tag: </p>
start tag: <p>
data: 'This book lives at '
start tag: <a href="http://diveintopython.org/" >
data: 'http://diveintopython.org/'
end tag: </a>
data: ".\012If you're reading it somewhere else, you may not have the lates"
data: 't version.\012'
end tag: </p>
```

Questo è ciò che troverete nel resto del capitolo:

- Derivare `SGMLParser` per creare classi che estraggano informazioni interessanti da documenti HTML.
- Derivare `SGMLParser` per creare la classe `BaseHTMLProcessor`, che sovrascrive tutti gli otto metodi manipolatori e li usa per ricostruire l'HTML originale partendo dai vari pezzi.
- Derivare `BaseHTMLProcessor` per creare la classe `Dialectizer` che aggiunge alcuni metodi, specialmente per trattare tag specifici dell'HTML; sovrascrive anche il metodo `handle_data` per fornire una struttura adatta a processare i blocchi di testo in mezzo ai tag HTML.
- Derivare `Dialectizer` per creare classi che definiscano regole per l'elaborazione di testo usate da `Dialectizer.handle_data`.
- Scrivere una suite di test che prenda una vera pagina da `http://diveintopython.org/` e la elabori.

5.3. Estrarre informazioni da documenti HTML

Per estrarre informazioni da documenti HTML, create una sottoclasse `SGMLParser` e definite metodi per ogni tag o altro che volete catturare.

Il primo passo per estrarre informazioni da un documento HTML è prendere del codice HTML. Se avete del codice HTML in giro per il vostro hard disk, potete usare delle funzioni per leggerlo, ma il vero divertimento inizia quando prendete l'HTML direttamente da pagine web.

Esempio 5.5. Introdurre `urllib`

```
>>> import urllib                                        (1)
>>> sock = urllib.urlopen("http://diveintopython.org/") (2)
>>> htmlSource = sock.read()                             (3)
>>> sock.close()                                         (4)
>>> print htmlSource                                     (5)
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

      <meta http-equiv='Content-Type' content='text/html; charset=ISO-8859-1'>
   <title>Dive Into Python</title>
<link rel='stylesheet' href='diveintopython.css' type='text/css'>
<link rev='made' href='mailto:f8dy@diveintopython.org'>
<meta name='keywords' content='Python, Dive Into Python, tutorial, object-oriented, programming, docu
<meta name='description' content='a free Python tutorial for experienced programmers'>
</head>
<body bgcolor='white' text='black' link='#0000FF' vlink='#840084' alink='#0000FF'>
<table cellpadding='0' cellspacing='0' border='0' width='100%'>
<tr><td class='header' width='1%' valign='top'>diveintopython.org</td>
<td width='99%' align='right'><hr size='1' noshade></td></tr>
<tr><td class='tagline' colspan='2'>Python for experienced programmers</td></tr>

[...snip...]
```

(1) Il modulo `urllib` è parte della libreria standard di Python. Contiene delle funzioni che consentono di ottenere e scaricare informazioni da URL internet (fondamentalmente pagine web).

(2) Il modo più semplice di utilizzare le `urllib` è scaricare l'intero testo di una pagina web con la funzione `urlopen`. Aprire una URL è simile ad aprire un file. Il valore di ritorno di `urlopen` somiglia ad un oggetto file, in quanto ha alcuni metodi propri degli oggetti file.

(3) La cosa più semplice da fare con l'oggetto ritornato da `urlopen` è `read`, che legge tutto il codice HTML della pagina web in una sola stringa. L'oggetto supporta anche `readlines`, che legge il testo riga per riga inserendolo in una lista.

(4) Quando avete finito di utilizzare l'oggetto, assicuratevi di chiuderlo (`close`), proprio come un normale oggetto file.

(5) Adesso abbiamo tutto il codice HTML della pagina web di `http://diveintopython.org/` in una stringa, e siamo pronti ad analizzarlo.

Esempio 5.6. Introdurre `urllister.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
from sgmllib import SGMLParser

class URLLister(SGMLParser):
    def reset(self):                              (1)
        SGMLParser.reset(self)
        self.urls = []

    def start_a(self, attrs):                     (2)
        href = [v for k, v in attrs if k=='href'] (3) (4)
        if href:
            self.urls.extend(href)
```

(1) `reset` è chiamata dal metodo `__init__` di `SGMLParser`, e può essere anche chiamata manualmente non appena è stata creata un'istanza dell'analizzatore. Quindi, se avete bisogno di fare una qualsiasi inizializzazione, fatela in `reset`, non in `__init__`, così che sarà adeguatamente reinizializzata quando qualcuno riutilizza un'istanza dell'analizzatore.

(2)

`start_a` è chiamata da `SGMLParser` ogni qual volta trova il tag `<a>`. Il tag può contenere un attributo `href`, e/o altri attributi, come `name` o `title`. Il parametro `attrs` è una lista di tuple, `[(attributo, valore), (attributo, valore), ...]`. Nel caso in cui ci fosse solo `<a>`, un valido (ma inutile) tag HTML, `attrs` sarebbe una lista vuota.

(3) Possiamo controllare se questo tag `<a>` ha un attributo `href` tramite una semplice list comprehension di valori multipli.

(4) Confronti tra stringhe quale `k=='href'` sono sempre case−sensitive, ma in questo caso non c'è problema, in quanto `SGMLParser` converte i nomi degli attributi in caratteri minuscoli durante la costruzione di `attrs`.

Esempio 5.7. Utilizzare `urllister.py`

```
>>> import urllib, urllister
>>> usock = urllib.urlopen("http://diveintopython.org/")
>>> parser = urllister.URLLister()
>>> parser.feed(usock.read())         (1)
>>> usock.close()                     (2)
>>> parser.close()                    (3)
>>> for url in parser.urls: print url (4)
toc.html
#download
toc.html
history.html
download/dip_pdf.zip
download/dip_pdf.tgz
download/dip_pdf.hqx
download/diveintopython.pdf
download/diveintopython.zip
download/diveintopython.tgz
download/diveintopython.hqx

[...snip...]
```

(1) Chiama il metodo `feed`, definito in `SGMLParser`, per raggiungere il codice HTML in parser. [7] Prende una stringa, che è ciò che `usock.read()` ritorna.

(2) Come per i file, potete chiudere i vostri oggetti URL non appena avete finito di utilizzarli.

(3) Dovreste anche `chiudere` il vostro oggetto parser, ma per una ragione diversa. Non è sicuro che il metodo `feed` processi tutto il codice HTML che gli viene passato; potrebbe bufferizzarlo, aspettandone altro. Una volta che non ce n'è più, chiamate `close` per svuotare il buffer e forzare l'analisi di ogni cosa.

(4) Una volta che parser è `chiuso`, l'analisi è completa e `parser.urls` conterrà l'elenco di tutti i nostri collegamenti alle URL presenti nel nostro documento HTML.

5.4. Introdurre `BaseHTMLProcessor.py`

`SGMLParser` non produce nulla da solo. Analizza, analizza e analizza, e chiama un metodo per ogni cosa interessante che trova, ma il metodo non fa nulla. `SGMLParser` è un *consuma* HTML: prende il codice HTML e lo divide in piccoli pezzi strutturati. Come avete visto nella sezione precedente, potete creare una sottoclasse di `SGMLParser` per definire classi che catturano tag specifici e producono cose utili, come la lista dei collegamenti delle pagine web. Adesso facciamo un ulteriore passo avanti, per definire la classe che cattura ogni cosa che `SGMLParser` rilascia, e quindi ricostruire l'intero documento HTML. In termini tecnici, questa classe sarà un *produttore* di codice HTML.

`BaseHTMLProcessor` crea una sottoclasse `SGMLParser` e fornisce gli 8 metodi handler essenziali: `unknown_starttag`, `unknown_endtag`, `handle_charref`, `handle_entityref`, `handle_comment`, `handle_pi`, `handle_decl`, e `handle_data`.

Esempio 5.8. Introdurre `BaseHTMLProcessor`

```
class BaseHTMLProcessor(SGMLParser):
    def reset(self):                        (1)
        self.pieces = []
        SGMLParser.reset(self)

    def unknown_starttag(self, tag, attrs): (2)
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())

    def unknown_endtag(self, tag):          (3)
        self.pieces.append("</%(tag)s>" % locals())

    def handle_charref(self, ref):          (4)
        self.pieces.append("&#%(ref)s;" % locals())

    def handle_entityref(self, ref):        (5)
        self.pieces.append("&%(ref)s" % locals())
        if htmlentitydefs.entitydefs.has_key(ref):
            self.pieces.append(";")

    def handle_data(self, text):            (6)
        self.pieces.append(text)

    def handle_comment(self, text):         (7)
        self.pieces.append("<!--%(text)s-->" % locals())

    def handle_pi(self, text):              (8)
        self.pieces.append("<?%(text)s>" % locals())

    def handle_decl(self, text):
        self.pieces.append("<!%(text)s>" % locals())
```

(1) `reset`, chiamata da `SGMLParser.__init__`, inizializza `self.pieces` come una lista vuota prima di chiamare il metodo padre. `self.pieces` è un attributo data che conterrà i pezzi del documento HTML che stiamo costruendo. Ogni metodo handler ricostruirà il codice HTML analizzato da `SGMLParser`, ed ogni metodo aggiungerà quella stringa a `self.pieces`. Notate che `self.pieces` è una lista. Potreste essere tentati di definirla come una stringa e semplicemente aggiungere ogni pezzo alla fine. Funzionerebbe, ma Python è molto più efficiente quando ha a che fare con le liste.[8]

(2) Poiché `BaseHTMLProcessor`BaseHTMLProcessor non definisce alcun metodo per particolari tag (come il metodo `start_a` in `URLLister`), `SGMLParser` chiamerà `unknown_starttag` per ogni inizio di tag. Questo metodo prende il tag (`tag`) e la lista degli attributi con le coppie nome/valore (`attrs`), ricostruisce il codice HTML originale e lo aggiunge a `self.pieces`. La formattazione della stringa qui è un po' strana; la spiegheremo nella prossima sezione.

(3) Ricostruire la fine dei tag è molto più semplice; prendete semplicemente il nome del tag ed inseritelo fra le parentesi `</...>`.

(4) Quando `SGMLParser` trova un riferimento ad un carattere, chiama `handle_charref` con il puro riferimento. Se il documento HTML contiene il riferimento ` `, `ref` varrà `160`. Per ricostruire il riferimento completo al carattere è sufficiente inserire `ref` tra i caratteri `&#...;`.

(5) I riferimenti alle entità sono simili a quelli ai caratteri, ma senza il carattere # (hash ndt). Per ricostruire il riferimento all'entità originale bisogna inserire `ref` fra i caratteri `&...;`. (Effettivamente, come mi ha fatto notare un erudito lettore, è un po' piu' complicato di così. Solo alcune entità HTML standard terminano con una semicolonna; altre entità somiglianti non lo fanno. Per nostra fortuna, il set di entità HTML standard è definito in un dizionario di un modulo Python chiamato `htmlentitydefs`. Da qui l'istruzione `if`.)

(6) I blocchi di testo sono semplicemente aggiunti inalterati in fondo a `self.pieces`.

(7) I commenti HTML sono inseriti fra i caratteri `<!--...-->`.

(8) Le istruzioni da trattare sono inserite fra i caratteri `<?...>`.

Importante: Processare il codice HTML con uno script migliorato
La specifica HTML richiede che tutto ciò che non è HTML (come JavaScript lato client) debba essere racchiuso tra commenti HTML, ma non in tutte le pagine ciò è stato fatto (e tutti i moderni browsers chiudono un occhio in merito). `BaseHTMLProcessor` non perdona; se uno script è inserito in modo improprio, verrà analizzato come se fosse codice HTML. Per esempio, se lo script contiene i simboli di minore e maggiore, `SGMLParser` potrebbe pensare, sbagliando, di aver trovato tag e attributi. `SGMLParser` converte sempre i tag e i nomi degli attributi in lettere minuscole, il che potrebbe bloccare lo script, e `BaseHTMLProcessor` racchiude sempre i valori tra doppi apici (anche se originariamente il documento HTML usava apici singoli o niente del tutto), cosa che di sicuro interromperebbe lo script. Proteggete sempre i vostri script lato client inserendoli dentro commenti HTML.

Esempio 5.9. `BaseHTMLProcessor` output

```
    def output(self):              (1)
        """Return processed HTML as a single string"""
        return "".join(self.pieces) (2)
```

(1) Questo è il metodo in `BaseHTMLProcessor` che non viene mai chiamato dal padre `SGMLParser`. Mentre gli altri metodi handler salvano il loro codice HTML ricostruito in `self.pieces`, questa funzione serve a unire tutti questi pezzi in una stringa. Come è stato fatto notare prima, Python lavora benissimo con le liste, ma meno bene con le stringhe, e quindi creiamo la stringa completa solo quando qualcuno ce lo chiede esplicitamente.

(2) Se preferite, potete utilizzare il metodo `join` del modulo `string`: `string.join(self.pieces, "")`.

Ulteriori letture

- W3C discute dei riferimenti a caratteri ed ad entità.
- *Python Library Reference* conferma i vostri sospetti che il modulo `htmlentitydefs` sia esattamente ciò che sembra.

5.5. `locals` e `globals`

Python dispone di due funzioni built−in, `locals` e `globals` che forniscono un accesso basato sui dizionari alle variabili locali e globali.

Prima però, una parola sui namespace (ndt: spazio dei nomi). È un argomento noioso, ma importante, perciò fate attenzione. Python usa quelli che sono chiamati namespace per tenere traccia delle variabili. Un namespace è come un dizionario nel quale le chiavi sono i nomi delle variabili ed i valori del dizionario sono i valori di quelle variabili. Infatti, potete accedere ad un namespace come ad un dizionario Python, come vedremo fra un minuto.

Ad ogni particolare punto di un programma Python, ci sono alcuni namespace disponibili. Ogni funzione ha il suo namespace, chiamato namespace locale, il quale tiene traccia delle variabili della funzione, inclusi gli argomenti della funzione e le variabili definite localmente. Ogni modulo ha il suo proprio namespace, chiamato namespace globale, che tiene traccia delle variabili del modulo, incluse funzioni, classi, ogni altro modulo importato, variabili e costanti di livello modulo. E c'è anche il namespace built−in, accessibile da ogni modulo, che contiene le funzioni built−in e le eccezioni.

Quando una linea di codice chiede il valore della variabile `x`, Python ricerca quella variabile in tutti i namespace disponibili, nell'ordine:

1. namespace locale − specifico alla funzione o metodo di classe corrente. Se la funzione definisce una variabile locale `x`, o ha un argomento `x`, Python userà questa e interromperà la ricerca.
2. namespace globale − specifico al modulo corrente. Se il modulo ha definito una variabile, funzione, o classe chiamata `x`, Python userà quella ed interromperà la ricerca.
3. namespace built−in − globale per tutti i moduli. Come il punto precedente, Python considererà quella `x` come il nome di una variabile o funzione built−in.

Se Python non trova `x` in nessuno di questi namespace, si fermerà e solleverà un'eccezione `NameError` con il messaggio `There is no variable named 'x'`, come avete visto nel Capitolo 2, ma senza sapere quanto lavoro facesse Python prima di darvi quell'errore.

> **Importante: Evoluzione del linguaggio: gli scope nidificati**
> Python 2.2 introdusse un subdolo ma importante cambiamento che modificò l'ordine di ricerca dei namespace: gli scope nidificati. Nelle versioni di Python precedenti al 2.2, quando vi riferite ad una variabile dentro ad una funzione nidificata o una funzione `lambda`, Python ricercherà quella variabile nel namespace corrente della funzione, e poi nel namespace del modulo. Python 2.2 ricercherà la variabile nel namespace della funzione corrente, *poi nel namespace della funzione genitore*, e poi nel namespace del modulo. Python 2.1 può lavorare in ogni modo; predefinitamente, lavora come Python 2.0, ma potete aggiugere la seguente linea di codice in cima al vostro modulo per farlo funzionare come Python 2.2:
>
> ```
> from __future__ import nested_scopes
> ```

Come molte cose in Python, i namespace sono direttamente accessibili a run−time. Specificatamente, il namespace locale è accessibile dalla funzione built−in `locals`, e quello globale (livello modulo) è accessibile dalla funzione built−in `globals`.

Esempio 5.10. Introdurre `locals`

```
>>> def foo(arg): (1)
...     x = 1
...     print locals()
...
>>> foo(7)        (2)
{'arg': 7, 'x': 1}
>>> foo('bar')    (3)
{'arg': 'bar', 'x': 1}
```

(1) La funzione `foo` ha due variabili nel suo namespace locale: `arg`, il cui valore è passato nella funzione, ed `x`, che è definito nella funzione.

(2) `locals` ritorna un dizionario di coppie nome/valore. Le chiavi di questo dizionario sono nomi delle variabili come fossero stringhe; i valori del dizionario sono gli attuali valori delle variabili. Perciò chiamando `foo` con `7` verrà stampato il dizionario contenente le due variabili della funzione: `arg` (`7`) ed `x` (`1`).

(3) Ricordate, Python ha una tipizzazione dinamica, perciò potreste anche facilmente passare una stringa in `arg`; la funzione (e la chiamata a `locals`) funzionerebbe ancora bene. `locals` funziona con tutte le variabili di tutti i tipi di dato.

Ciò che `locals` fa per il namespace locale, `globals` lo fa per il namespace globale. `globals` è più eccitante quindi, perché il namespace di un modulo è più interessante. [9] Il namespace di un modulo non include solo le

variabili e le costanti di livello modulo, include anche tutte le funzioni e le classi definite nel modulo. In più, include ogni cosa che è stata importata nel modulo.

Ricordate la differenza tra `from *module* import` e `import *module*`? Con `import *module*`, il modulo stesso è importato, ma mantiene il suo proprio namespace, visto che dovete usare il nome del modulo per accedere ad ogni sua funzione od attributo: `*modulo.funzione*`. Ma con `from *module* import`, state realmente importando funzioni ed attributi specifici da un altro modulo nel vostro namespace, perciò potete accedervi senza riferirvi direttamente al modulo da cui provengono. Con la funzione `globals`, potete vedere come ciò avvenga.

Esempio 5.11. Introdurre `globals`

Aggiungete il seguente blocco di codice a `BaseHTMLProcessor.py`:

```
if __name__ == "__main__":
    for k, v in globals().items():             (1)
        print k, "=", v
```

(1) Non fatevi intimidire, ricordate che avete già visto tutto questo in precedenza. La funzione `globals` ritorna un dizionario, e stiamo iterando attraverso il dizionario usando il metodo `items` e l'assegnamento multi−variabile. L'unica cosa nuova qui è la funzione `globals`.

Ora lanciare lo script dalla linea di comando darà il seguente output:

```
c:\docbook\dip\py>python BaseHTMLProcessor.py

SGMLParser = sgmllib.SGMLParser                (1)
htmlentitydefs = <module 'htmlentitydefs' from 'C:\Python21\lib\htmlentitydefs.py'> (2)
BaseHTMLProcessor = __main__.BaseHTMLProcessor (3)
__name__ = __main__                            (4)
[...snip...]
```

(1) `SGMLParser` è stato importato da `sgmllib`, usando `from *module* import`. Questo significa che è stato importato direttamente nel namespace del nostro modulo, e lì lo troviamo.

(2) Al contrario il modulo `htmlentitydefs`, è stato importato usando `import`. Questo significa che il modulo `htmlentitydefs` stesso si trova nel nostro namespace, ma la variabile `entitydefs` definita nel modulo `htmlentitydefs` invece no.

(3) Questo modulo definisce un'unica classe, `BaseHTMLProcessor`. Notate che il valore qui è la classe stessa, non una specifica istanza della classe.

(4) Ricordate il trucco `if __name__`? Quando lanciate un modulo (al contrario di importarlo da un altro modulo), l'attributo built−in `__name__` ottiene un valore speciale, `__main__`. Dato che abbiamo lanciato questo modulo come uno script dalla linea di comando, `__name__` diventa `__main__`, questo giustifica il nostro piccolo script di test a stampare la `globals` appena eseguita.

Nota: Accedere dinamicamente alle variabili
Usando le funzioni `locals` e `globals`, potete ottenere il valore di variabili arbitrarie dinamicamente, rendendo disponibile il nome della variabile come una stringa. Questo rispecchia la funzionalità della funzione `getattr`, che vi permette di accedere a funzioni arbitrarie dinamicamente, rendendo disponibile il nome della funzione come stringa.

C'è un'altra importante differenza fra `locals` e `globals` che dovreste imparare adesso, prima di pagarne le conseguenze. Vi colpirà ugualmente, ma almeno poi la ricorderete, imparandola.

Esempio 5.12. `locals` è in sola lettura, `globals` no

```
def foo(arg):
    x = 1
    print locals()    (1)
    locals()["x"] = 2 (2)
    print "x=",x      (3)

z = 7
print "z=",z
foo(3)
globals()["z"] = 8    (4)
print "z=",z          (5)
```

(1) Dato che `foo` è chiamata con `3`, stamperà `{'arg': 3, 'x': 1}`. Questo non dovrebbe essere una sorpresa.

(2) Potreste pensare che questo cambierebbe il valore della variabile locale `x` a `2`, ma non lo fa. `locals` non ritorna realmente il namespace locale, ritorna una copia. Perciò cambiarla non ha rilevanza sulle variabile del namespace locale.

(3) Questo stampa `x= 1`, non `x= 2`.

(4) Dopo essere rimasti scottati da `locals`, potreste pensare che *questo non* cambierebbe il valore di `z`, invece sì. A causa di differenze interne su come Python è implementato (che raramente spiego, dato che non le ho capite pienamente neppure io), `globals` ritorna l'attuale namespace globale, non una copia: l'esatto opposto del comportamento di `locals`. Perciò ogni cambiamento al dizionario ritornato da `globals` ha effetto sulle variabili globali.

(5) Questo stampa `z= 8`, non `z= 7`.

5.6. Formattazione di stringhe basata su dizionario

La formattazione di stringhe permette di inserire facilmente dei valori all'interno delle stringhe. I valori sono elencati in una tupla ed inseriti ordinatamente nella stringa al posto dei vari marcatori per la formattazione. Per quanto questo approccio sia efficiente, non rende il codice molto semplice da leggere, specialmente se si stanno inserendo molti valori. Non potete semplicemente scorrere la stringa in un passaggio e capire quale sarà il risultato; dovete continuamente passare dalla stringa alla tupla di valori.

Esiste una forma alternativa per la formattazione di stringhe che usa i dizionari al posto delle tuple di valori.

Esempio 5.13. Introduzione alla formattazione di stringhe basata su dizionario

```
>>> params = {"server":"mpilgrim", "database":"master", "uid":"sa", "pwd":"secret"}
>>> "%(pwd)s" % params                                    (1)
'secret'
>>> "%(pwd)s is not a good password for %(uid)s" % params (2)
'secret is not a good password for sa'
>>> "%(database)s of mind, %(database)s of body" % params (3)
'master of mind, master of body'
```

(1) Al posto di una tupla di valori espliciti, questo metodo di formattazione usa un dizionario, `params`. Ed al posto di un semplice marcatore `%s` nella stringa, il marcatore contiene un nome tra parentesi. Questo nome è usato come una chiave nel dizionario `params` e ne sostituisce il corrispondente valore, `secret`, al posto del marcatore `%(pwd)s`.

(2) La formattazione di stringhe basata su dizionario funziona con qualsiasi numero di chiavi. Ogni chiave deve esistere nel dizionario, o la formattazione fallirà con un `KeyError`.

(3) Potete specificare la stessa chiave due volte, ogni occorrenza verrà rimpiazzata con il medesimo valore.

Quindi perché dovreste usare la formattazione basata su dizionario? Beh, sembra una perdita di tempo impostare un dizionario solo per formattare una stringa nella riga successiva; è davvero molto più utile quando già avete un dizionario significativo. Come `locals`.

Esempio 5.14. Formattazione di stringhe basata su dizionario in `BaseHTMLProcessor.py`

```
    def handle_comment(self, text):
        self.pieces.append("<!--%(text)s-->" % locals()) (1)
```

(1) L'utilizzo della funzione built-in `locals` è il metodo più comune di utilizzo della formattazione di stringhe basata su dizionario. Significa che potete usare i nomi di variabili locali all'interno della vostra stringa (in questo caso, `text`, che è stato passato al metodo della classe come argomento) ed ogni variabile menzionata sarà sostituita con il suo valore. Se `text` è `'Begin page footer'`, la formattazione della stringa `"<!--%(text)s-->" % locals()` porterà alla stringa `'<!--Begin page footer-->'`.

```
    def unknown_starttag(self, tag, attrs):
        strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs]) (1)
        self.pieces.append("<%(tag)s%(strattrs)s>" % locals())                  (2)
```

(1) Quando questo metodo è chiamato, `attrs` è una lista di tuple chiave/valore, come gli `items` di un dizionario, il che vuol dire che possiamo usare un assegnamento multi-variabile per scorrerlo. Dovrebbe essere un meccanismo familiare oramai, ma ci sono molte cose da evidenziare, quindi analizziamole singolarmente:

1. Supponiamo che `attrs` sia `[('href', 'index.html'), ('title', 'Go to home page')]`.
2. Nel primo giro di list comprehension, `key` avrà valore `'href'` e `value` avrà valore `'index.html'`.
3. La formattazione della stringa `' %s="%s"' % (key, value)` si risolverà in `' href="index.html"'`. Questa stringa diventa il primo elemento della lista ottenuta dalla list comprehension.
4. Al secondo giro, `key` avrà valore `'title'` e `value` avrà valore `'Go to home page'`.
5. La formattazione della stringa si risolverà in `' title="Go to home page"'`.
6. La list comprehension ritorna una lista di queste due stringhe, e `strattrs` le concatenerà assieme per formare `' href="index.html" title="Go to home page"'`.

(2) Ora, usando una formattazione di stringhe basata su dizionario, inseriamo il valore di `tag` e `strattrs` nella stringa. Così, se `tag` è `'a'`, il risultato finale è `'<a href="index.html" title="Go to home page">'`, ed è ciò che viene aggiunto a `self.pieces`.

Importante: Problemi di performance con locals
Usare una formattazione di stringhe basata su dizionari con `locals` è un modo conveniente per ottenere una formattazione di stringhe complessa in maniera più leggibile, ma ha un prezzo. C'è una minima perdita di performance nell'effettuare una chiamata a `locals`, in quanto `locals` costruisce una copia dello spazio dei nomi locale.

5.7. Virgolettare i valori degli attributi

Una domanda ricorrente su comp.lang.python è: "Io ho un gruppo di documenti HTML con i valori degli attributi espressi senza virgolette e voglio virgolettarli tutti in maniera opportuna. Come posso farlo?" [10] (Questa situazione è generalmente causata da un responsabile di progetto, convertito alla religione "HTML è uno standard", che si

aggiunge ad un grosso progetto e annuncia che tutte le pagine HTML devono essere validate da un verificatore di HTML. Avere i valori degli attributi senza virgolette è una violazione comune dello standard HTML.) Qualunque sia la ragione, è facile rimediare ai valori degli attributi senza virgolette, se si filtra il documento HTML attraverso `BaseHTMLProcessor`.

La classe `BaseHTMLProcessor` consuma codice HTML (giacché discende da `SGMLParser`) e produce codice HTML equivalente, ma il codice di uscita non è identico a quello di entrata. Tag e nomi di attributi sono generati in carattere minuscoli, anche se erano inizialmente in caratteri maiuscoli o misti maiuscolo/minuscolo; i valori degli attributi sono racchiusi tra virgolette, anche se erano originariamente racchiusi tra apici o non avevano affatto delimitatori. È quest'ultimo effetto collaterale che noi possiamo sfruttare.

Esempio 5.15. Valori degli attributi tra virgolette

```
>>> htmlSource = """        (1)
...     <html>
...     <head>
...     <title>Test page</title>
...     </head>
...     <body>
...     <ul>
...     <li><a href=index.html>Home</a></li>
...     <li><a href=toc.html>Table of contents</a></li>
...     <li><a href=history.html>Revision history</a></li>
...     </body>
...     </html>
...     """
>>> from BaseHTMLProcessor import BaseHTMLProcessor
>>> parser = BaseHTMLProcessor()
>>> parser.feed(htmlSource) (2)
>>> print parser.output()   (3)
<html>
<head>
<title>Test page</title>
</head>
<body>
<ul>
<li><a href="index.html">Home</a></li>
<li><a href="toc.html">Table of contents</a></li>
<li><a href="history.html">Revision history</a></li>
</body>
</html>
```

(1) Va notato che i valori degli attributi `href` nei tag `<a>` non sono virgolettati in modo proprio. Va anche notato che stiamo usando le virgolette triple per qualcosa di diverso da una `doc string`, e direttamente nell'interprete interattivo (IDE ndt) per giunta. Le triple virgolette sono utili per molte cose.

(2) Passiamo la stringa al parser.

(3) Usando la funzione di `output` definita in `BaseHTMLProcessor`, otteniamo l'output come una singola stringa, inclusi i valori degli attributi racchiusi tra virgolette. Sebbene tutto ciò possa apparire ben poco sensazionale, si pensi a quante cose sono effettivamente successe a questo punto: `SGMLParser` ha processato l'intero documento HTML e lo ha scomposto in tag, riferimenti, dati, e così via; `BaseHTMLProcessor` ha usato questi elementi per ricostruire pezzi di HTML (che sono ancora memorizzati in `parser.pieces`, nel caso li vogliate vedere); finalmente, è stato chiamato il metodo `parser.output`, che ha ricomposto tutti gli elementi HTML in un'unica stringa.

5.8. Introduzione al modulo `dialect.py`

La classe `Dialectizer` è una semplice (e un po' stupida) specializzazione della classe `BaseHTMLProcessor`. Questa classe sottopone un blocco di testo ad una serie di sostituzioni, ma al contempo fa in modo che tutto ciò che è racchiuso in un blocco `<pre>...</pre>` rimanga inalterato.

Per gestire i blocchi `<pre>`, si definiscono due nuovi metodi in `Dialectizer`: `start_pre` ed `end_pre`.

Esempio 5.16. Gestire tag specifici

```
    def start_pre(self, attrs):             (1)
        self.verbatim += 1                  (2)
        self.unknown_starttag("pre", attrs) (3)

    def end_pre(self):                      (4)
        self.unknown_endtag("pre")          (5)
        self.verbatim -= 1                  (6)
```

(1) Il metodo `start_pre` è chiamato ogni volta che `SGMLParser` trova un tag `<pre>` nel codice HTML di partenza. Più avanti, scopriremo esattamente perché succede questo. Il metodo accetta un singolo parametro, `attrs`, che contiene gli attributi del tag (se ce ne sono). Il parametro `attrs` è una lista di coppie chiave/valore, proprio come quella accettata dal metodo `unknown_starttag`.

(2) Nel metodo `reset`, viene inizializzato un attributo che serve come contatore per i tag `<pre>`. Ogni volta che si incontra un tag `<pre>`, si incrementa il contatore; ogni volta che si incontra un tag `</pre>`, si decrementa il contatore. Si potrebbe usarlo semplicemente come flag, settandolo ad `1` e resettarndolo a `0`, ma è altrettanto facile gestirlo come contatore, ed in questo modo si copre il caso, insolito ma possibile, in cui ci siano tag `<pre>` annidati. Fra poco vedremo come questo contatore torni utile.

(3) Questo è l'unica tipologia di elaborazione specifica che applichiamo ai tag `<pre>`. Ora si passa la lista degli attributi al metodo `unknown_starttag`, così che si possa svolgere l'elaborazione standard.

(4) Il metodo `end_pre` è chiamato ogni volta che la classe `SGMLParser` incontra un tag `</pre>`. Dato che i tag di chiusura non contengono attributi, il metodo non accetta parametri.

(5) Per prima cosa, si vuole eseguire l'elaborazione standard, esattamente come per gli altri tag di chiusura.

(6) Quindi, si decrementa il nostro contatore, per indicare che il blocco `<pre>` è stato chiuso.

A questo punto, vale la pena di tuffarsi un po' più a fondo nel codice di `SGMLParser`. Ho ripetutamente affermato (e finora mi avete creduto sulla parola) che `SGMLParser` cerca i singoli tag e chiama per ognuno di loro il metodo specifico, se esiste. Per esempio, abbiamo appena visto come definire i metodi `start_pre` ed `end_pre` per gestire i tag `<pre>` e `</pre>`. Ma come succede? Bene, non si tratta di magia, ma solo di buon codice Python.

Esempio 5.17. `SGMLParser`

```
    def finish_starttag(self, tag, attrs):               (1)
        try:
            method = getattr(self, 'start_' + tag)       (2)
        except AttributeError:                           (3)
            try:
                method = getattr(self, 'do_' + tag)      (4)
            except AttributeError:
                self.unknown_starttag(tag, attrs)        (5)
                return -1
            else:
                self.handle_starttag(tag, method, attrs) (6)
                return 0
```

```
        else:
            self.stack.append(tag)
            self.handle_starttag(tag, method, attrs)
            return 1                                    (7)

    def handle_starttag(self, tag, method, attrs):
        method(attrs)                                   (8)
```

(1) A questo punto, `SGMLParser` ha già trovato un tag di apertura ed ha analizzato la lista degli attributi. L'unica cosa che rimane da fare è di stabilire se c'è uno specifico metodo per gestire il tipo di tag trovato, oppure se è necessario chiamare il metodo predefinito (`unknown_starttag`).

(2) La "magia" di `SGMLParser` non è altro che la nostra vecchia amica, la funzione `getattr`. Quello che forse non vi è ancora chiaro è che `getattr` è capace di trovare anche i metodi definiti nei "discendenti" di un oggetto, oltre che nell'oggetto stesso. Qui l'oggetto è `self`, l'istanza dell'oggetto corrente. Quindi se la variabile `tag` vale `'pre'`, questa chiamata a `getattr` cercherà un metodo `start_pre` nell'istanza corrente, che è un'istanza della classe `Dialectizer`.

(3) La funzione `getattr` solleva una eccezione `AttributeError` se il metodo che sta cercando non esiste nell'oggetto (inclusi i suoi "discendenti"), ma questo ci sta bene perché abbiamo racchiuso la chiamata a `getattr` in un blocco `try...except` che intercetta esplicitamente l'eccezione `AttributeError`.

(4) Dato che non è stato trovato un metodo `start_xxx`, qui si cerca anche un metodo `do_xxx`, prima di rinunciare. Questo modo alternativo di chiamare i metodi che gestiscono i tag, è di solito adoperato per quei tag che non si usano a coppie ("standalone", ndt), come `<br>`, che non ha nessun corrispondente tag di chiusura. Tuttavia potete usare l'uno o l'altro metodo; così potete osservare `SGMLParser` lavorare contemporaneamente per ciascuno dei tag. Tuttavia, non si dovrebbero definire sia il metodo `start_xxx` che il metodo `do_xxx` per gestire lo stesso tag; in un caso del genere, solo il metodo `start_xxx` sarebbe chiamato.

(5) Un'altra eccezione `AttributeError`, il che significa che la chiamata a `getattr` è fallita anche con `do_xxx`. Dato che non abbiamo trovato né un metodo `start_xxx` né un metodo `do_xxx` corrispondente al tag, qui si intercetta l'eccezione e si chiama il metodo predefinito, `unknown_starttag`.

(6) Ricordate che i blocchi `try...except` possono anche avere una clausola `else`, che è eseguita se nessuna eccezione è sollevata all'interno del blocco `try...except`. Ovviamente, questo significa *è stato* trovato un metodo `do_xxx` corrispondende al tag, che quindi viene invocato.

(7) A proposito, non preoccupatevi dei diversi valori di ritorno; in teoria essi dovrebbero indicare qualcosa, ma in pratica non sono mai usati. Non vi preccupate neanche dell'istruzione `self.stack.append(tag)`; la classe `SGMLParser` tiene traccia internamente del fatto che i vostri tag di apertura e chiusura siano opportunamente bilanciati, ma neanche questa informazione viene in pratica utilizzata. In teoria, potreste usare questo modulo per verificare che tutti i tag siano bilanciati, ma probabilmente non ne vale la pena, e comunque la cosa va oltre gli scopi di questo capitolo. Abbiamo cose più importanti di cui preoccuparci in questo momento.

(8) I metodi `start_xxx` e `do_xxx` non vengono chiamati direttamente; il tag, il metodo e gli attributi dei tag vengono passati a questa funzione, `handle_starttag`, di modo che le classi derivate da questa possano ridefinire *tutti* i metodi, questo e gli altri che gestiscono i tag di apertura. Noi non abbiamo bisogno di un tale livello di controllo, quindi ci limitiamo a lasciare che questo metodo faccia ciò che deve, vale a dire chiamare il metodo (`start_xxx` o `do_xxx`) con la lista degli attributi. Ricordate, un `metodo` è una funzione, ottenuta come valore di ritorno della chiamata a `getattr`, e le funzioni sono oggetti (lo so che vi state

stancando di sentirlo, e prometto che smetterò di dirlo non appena avremo esaurito i modi di usare la cosa a nostro vantaggio). A questo punto, l'oggetto funzione è passato al metodo di attivazione (`handle_starttag`) come argomento, e questi a sua volta chiama la funzione. A questo punto, non è neccessario sapere che funzione è, qual'è il suo nome o dov'è definita; l'unica cosa che occorre sapere sulla funzione è che va invocata con un unico argomento, `attrs`.

Ora possiamo tornare al programma inizialmente previsto: `Dialectizer`. Lo avevamo lasciato al punto in cui stavamo per definire metodi specifici per gestire i tag `<pre>` e `</pre>`. C'è solamente una cosa che rimane da fare, ed è elaborare i blocchi di testo con le nostre sostituzioni predefinite. Per fare ciò dobbiamo ridefinire il metodo `handle_data`.

Esempio 5.18. Ridefinizione del metodo `handle_data`

```
    def handle_data(self, text):                                          (1)
        self.pieces.append(self.verbatim and text or self.process(text)) (2)
```

(1) Il metodo `handle_data` è invocato con un solo argomento, il testo da elaborare.

(2) Nella classe base `BaseHTMLProcessor`, il metodo `handle_data` aggiungeva semplicemente il testo di input al buffer di output, `self.pieces`. Nel nostro caso la logica è solo leggermente più complessa. Se ci si trova all'interno di un blocco `<pre>...</pre>`, allora la variablie `self.verbatim` avrà un valore maggiore di `0`, ed in questo caso si vuole aggiungere il testo al buffer di output senza modificarlo. Altrimenti, viene chiamato un metodo separato per eseguire le sostituzioni e quindi il risultato viene aggiunto al buffer di output. In Python, lo si può fare con una sola linea di codice, usando il trucchetto `and-or`.

Siamo vicini a capire completamente `Dialectizer`. L'unico punto mancante è, nello specifico, che tipo di sostituzioni vengono effettuate. Se conoscete qualcosa del linguaggio Perl, sapete che laddove sono richieste delle complesse sostituzioni di testo, l'unico vero strumento per farlo sono le espressioni regolari.

5.9. Introduzione alle espressioni regolari

Le espressioni regolari sono un strumento potente (e piuttosto standardizzato) per cercare, sostituire o analizzare del testo contenente complessi schemi di caratteri. Se avete già usato espressioni regolari in altri linguaggi (come il Perl), potete saltare questa sezione e leggere direttamente il sommario del modulo `re` per avere una panoramica delle funzioni disponibili e dei loro argomenti.

I tipi stringa hanno i propri metodi di ricerca (`index`, `find` e `count`), di sostituzione (`replace`) e di analisi (`split`), ma questi sono limitati ai casi più semplici di utilizzo. Il metodi di ricerca operano con una singola sotto-stringa dal valore predefinito e sono sempre sensibili alla differenza tra maiuscole e minuscole; per effettuare ricerche in una stringa `s`, volendo ignorare tale differenza, occorre chiamare i metodi `s.lower()` o `s.upper()` ed essere sicuri che la stringa da cercare abbia lo stesso tipo di carattere (maiuscolo/minuscolo) di `s`. I metodi `replace` e `split` hanno le stesse limitazioni. Laddove è possibile conviene usarli (sono veloci e leggibili), ma per cose più complesse è necessario passare alle espressioni regolari.

Esempio 5.19. Cercare corrispondenze alla fine di una stringa

Questa serie di esempi sono ispirati da un problema reale che ho avuto durante il mio lavoro ufficiale, dove dovevo adattare e standardizzare indirizzi stradali estratti da un sistema precedente, prima di inserirli in un nuovo sistema (come vedete non mi invento niente: queste cose sono utili realmente).

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')              (1)
```

```
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')               (2)
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.') (3)
'100 NORTH BROAD RD.'
>>> import re                               (4)
>>> re.sub('ROAD$', 'RD.', s)              (5) (6)
'100 NORTH BROAD RD.'
```

(1) Il mio obiettivo è di standardizzare un indirizzo stradale in modo tale che `'ROAD'` sia sempre abbreviato come `'RD.'`. A primo acchito, avevo considerato la cosa abbastanza semplice da poter usare il metodo `replace`. Dopo tutto, i dati erano già tutti in lettere maiuscole, in modo che la differenza tra maiuscolo e minuscolo non sarebbe stata un problema. E la stringa di ricerca, `'ROAD'`, era una costante. In effetti, in questo caso ingannevolmente semplice, `s.replace` funziona come ci si aspetta.

(2) La vita, sfortunatamente, è piena di controesempi, e scoprii presto quanto ciò sia vero. Il problema qui è che `'ROAD'` appare due volte nell'indirizzo, una volta come parte del nome della strada, `'BROAD'`, ed una volta come parola a sé stante. Il metodo `replace` trova le due occorrenze e ciecamente le sostituisce entrambe; in questo modo, il mio indirizzo viene distrutto.

(3) Per risolvere il problema di indirizzi con più di una sottostringa `'ROAD'`, potremmo ricorrere a qualcosa del genere: cercare e sostituire `'ROAD'` solo negli ultimi 4 caratteri dell'indirizzo (`s[-4:]`), e lasciare intatto il resto della stringa (`s[-4:]`). Ma potete vedere come la cosa stia già diventando poco gestibile. Per esempio, questo schema dipende dalla lunghezza della stringa che vogliamo sostituire (se volessimo sostituire `'STREET'` con `'ST.'`, dovremmo usare `s[:-6]` e `s[-6:].replace(...)`). Vi piacerebbe ritornare su questo codice dopo sei mesi per cercarvi un "baco"?. Io so che preferirei evitarlo.

(4) È tempo di passare alle espressioni regolari. In Python, tutte le funzioni collegate alle espressioni regolari sono contenute nel modulo `re`.

(5) Date un'occhiata al primo parametro `'ROAD$'`. Si tratta di una espressione regolare molto semplice, che corrisponde a `'ROAD'` solo quando questa sottostringa si trova alla fine. Il carattere `$` significa "fine della stringa". Esiste un carattere corrispondente, il carattere `^`, che significa "inizio della stringa".

(6) Usando la funzione `re.sub`, si cerca nella stringa `s` per sottostringhe corrispondenti all'espressione regolare `'ROAD$'` e le si rimpiazza con `'RD.'`. In questo modo si rimpiazza la sottostringa `ROAD` alla fine della stringa `s`, ma *non* quella che è parte della parola BROAD, perché questa è in mezzo alla stringa `s`.

Esempio 5.20. Cercare la corrispondenza con parole complete

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)     (1)
'100 BRD.'
>>> re.sub('\\bROAD$', 'RD.', s)  (2)
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s)  (3)
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s)  (4)
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) (5)
'100 BROAD RD. APT 3'
```

(1) Continuando con la mia storia degli indirizzi da aggiustare, scoprii presto che l'esempio precedente, in cui cercavo corrispondenze con `'ROAD'` alla fine della stringa, non era abbastanza efficiente perché non tutti gli indirizzi includevano una designazione del tipo di strada; alcuni terminavano con il solo nome della strada. Il più delle volte, questo mi andava bene, ma se il nome della strada era `'BROAD'`, allora l'espressione regolare avrebbe corrisposto con `'ROAD'` alla fine della stringa, anche se questi

era parte della parola `'BROAD'`, e questo non era quello che volevo.

(2) Quello che *realmente* volevo era di trovare corrispondenze con `'ROAD'` quando si trovava alla fine della stringa *ed* era una parola a se stante, non una parte di una parola più grande. Per esprimere questo con una espressione regolare, occorre usare il codice `\b`, che significa "in questo punto ci deve essere un limite alla parola". In Python, l'uso di questo codice è complicato dal fatto che per inserire il carattere `'\'` in una stringa, esso deve essere preceduto da un altro carattere `'\'` ("escaped" ndt). A questo fatto si fa qualche volta riferimento come alla "piaga del backslash" (backslash == barra inversa, ndt), e questa è una delle ragioni per cui le espressioni regolari sono più facili in Perl che in Python. D'altra parte, Perl mischia le espressioni regolari con altre notazioni sintattiche, per cui se c'è un "baco" può essere difficile stabilire se esso sia nell'espressione regolare o negli altri componenti sintattici.

(3) Per evitare la piaga del backslash, si possono usare le cosiddette stringe grezze, facendo precedere il `'...'` con la lettera `r` (`raw` cioè grezzo, ndt). Questo dice a Python che niente nella stringa deve essere interpretato come carattere speciale. La stringa `'\t'` indica di solito il carattere di tabulazione, ma la stringa `r'\t'` è proprio il carattere barra inversa `\` seguito dalla lettera `t`. Io di solito raccomando di usare sempre stringe grezze quando si ha a che fare con le espressioni regolari, altrimenti le cose diventano subito troppo confuse (e le espressioni regolari già di per se fanno presto a diventare poco chiare).

(4) **sigh** Sfortunatamente, scoprii presto che molti casi concreti contraddicevano la mia logica. In questo caso, l'indirizzo conteneva la parola `'ROAD'` come parola a se stante, ma non era alla fine, perché l'indirizzo includeva il numero di un appartamento dopo la specifica della strada. Dato che `'ROAD'` non è alla fine della stringa, non corrisponde, e quindi la chiamata a `re.sub` finisce per non sostituire proprio niente, e noi ci ritroviamo con la stringa originale, che non è quello che volevamo.

(5) Per risolvere questo problema, ho rimosso il carattere `$` ed ho aggiunto un'altro `\b`. Ora l'espressione regolare si legge: "corrisponde a `'ROAD'` quando è una parola a se stante in qualunque punto della stringa", sia all'inizio che alla fine che da qualche parte nel mezzo.

Questo rappresenta appena la punta estrema dell'iceberg rispetto a quello che le espressioni regolari possono fare. Si tratta di uno strumento estremamente potente e vi sono interi libri a loro dedicate. Non sono però la soluzione migliore per ogni problema. È importante saperne abbastanza da capire quando sono adeguate e quando invece possono causare più problemi di quanti ne risolvano.

> Alcune persone, quando affrontano un problema, pensano: "Lo so, finirò per usare le espressioni regolari". A questo punto, hanno due problemi.
>
> −− Jamie Zawinski, in comp.lang.emacs

Ulteriori letture

- Regular Expression HOWTO discute delle espressioni regolari e di come usarle in Python.
- *Python Library Reference* riassume il modulo `re` .

5.10. Mettere tutto insieme

È tempo di mettere a frutto tutto quello che abbiamo imparato finora. Spero che abbiate prestato attenzione.

Esempio 5.21. La funzione `translate`, parte prima

```
def translate(url, dialectName="chef"): (1)
    import urllib                       (2)
    sock = urllib.urlopen(url)          (3)
    htmlSource = sock.read()
    sock.close()
```

(1) La funzione `translate` ha un argomento opzionale `dialectName`, che è una stringa indicante il dialetto da usare. Fra poco vedremo come viene usato questo parametro.

(2) Ehi, aspettate un secondo, c'è una istruzione `import` in questa funzione! Questo è perfettamente lecito in Python. Siete abituati a vedere istruzioni `import` all'inizio di un programma, cosa che implica che il modulo importato è disponibile in qualsiasi punto del programma. Ma è anche possibile importare un modulo in una funzione. Se si ha un modulo che è usato solo all'interno di una funzione, questa è una maniera semplice di rendere il vostro codice più modulare. (Quando scoprirete che il vostro esercizio di un fine settimana è diventato un complesso lavoro di 800 linee e deciderete di dividerlo in una dozzina di moduli riusabili, apprezerete questa finezza.)

(3) Ora si prende il sorgente dalla URL specificata.

Esempio 5.22. La funzione `translate`, parte seconda: sempre più curioso

```
    parserName = "%sDialectizer" % dialectName.capitalize() (1)
    parserClass = globals()[parserName]                     (2)
    parser = parserClass()                                  (3)
```

(1) `capitalize` è un metodo del tipo stringa che non abbiamo mai incontrato finora; semplicemente, questo metodo rende maiuscola la prima lettera di una stringa e minuscole tutte le altre. Combinandolo con un po' di formattazione di stringhe, si riesce a prendere il nome di un dialetto ed a trasformarlo nel nome della classe "Dialectizer" corrispondente. Se `dialectName` è uguale alla stringa `'chef'`, allora `parserName` sarà uguale alla stringa `'ChefDialectizer'`.

(2) Abbiamo il nome di una classe in forma di stringa (`parserName`) e abbiamo lo spazio dei nomi globale in forma di dizionario (`globals`()). Combinati, otteniamo un riferimento alla classe nominata dalla stringa. (Ricordate, le classi sono oggetti, e possono essere assegnati a variabili proprio come ogni altro oggetto.) Se `parserName` è uguale alla stringa `'ChefDialectizer'`, allora `parserClass` sarà uguale alla classe `ChefDialectizer`.

(3) Alla fine, ecco che abbiamo l'oggetto classe (`parserClass`), e ci serve una instanza di tale classe. Bene, sappiamo già come fare: basta chiamare la classe come fosse una funzione. Il fatto che la classe è memorizzata in una variabile locale non fa assolutamente alcuna differenza; chiamiamo semplicemente la variabile locale come una funzione e vien fuori un'istanza della classe. Se `parserClass` è uguale alla classe `ChefDialectizer`, allora `parser` sarà uguale ad un instanza della classe `ChefDialectizer`.

Ma perché darsi tanta pena? Dopo tutto, ci sono solo tre classi `Dialectizer`; perché non usare semplicemente una istruzione `case`? (Beh, non c'è l'istruzione `case` in Python, ma perché non usare una serie di istruzioni `if` ?) Una ragione c'è: estensibilità. La funzione `translate` non dipende in nessun modo dal numero di classi "Dialectizer" che sono state definite. Immaginate che domani si definisca una nuova classe `FooDialectizer`; la funzione `translate` funzionerebbe semplicemente passando `'foo'` come valore del parametro `dialectName`.

Ancora meglio, immaginatevi di mettere `FooDialectizer` in un modulo separato, e di importare tale modulo con la sintassi `from` *`module`* `import`. Abbiamo già visto come questo includa il modulo nello spazio dei nomi globali restituito da `globals`(), per cui `translate` funzionerebbe ancora senza bisogno di modifiche, sebbene `FooDialectizer` sia definita in un file separato.

Ora immaginate che il nome del dialetto venga da qualche luogo esterno al programma, magari da un database o da un valore inserito dall'utente in una form. È possibile usare una qualsiasi delle molte architetture di scripting server−side in Python per la generazione dinamica di pagine HTML; questa funzione potrebbe accettare una URL ed il nome di un dialetto (entrambi stringhe) nella stringa di query di una richiesta di pagina web, e restituire come output la pagina web "tradotta".

Infine, immaginate una infrastruttura `Dialectizer` con una architettura che faccia uso di plug−in. Potreste mettere ogni classe `Dialectizer` in un file diverso, lasciando solo la funzione `translate` nel modulo `dialect.py`.

Assumendo una metodologia consistente nell'assegnare i nomi, la funzione `translate` potrebbe importare dinamicamente la classe richiesta dal file corrispondente, partendo da nient'altro che il nome del dialetto. (Non abbiamo ancora visto esempi di import dinamico, ma vi prometto di trattare questo argomento in uno dei prossimi capitoli.) Per aggiungere un nuovo dialetto, potreste semplicemente aggiungere un file con il nome appropriato nella directory dei plug−in (ad esempio un file `foodialect.py` che contenga la classe `FooDialectizer`). Chiamando la funzione `translate` usando `'foo'` come nome del dialetto provocherebbe il ritrovamento del modulo `foodialect.py`, l'import della classe `FooDialectizer`, e così via.

Esempio 5.23. La funzione `translate`, parte terza

```
    parser.feed(htmlSource) (1)
    parser.close()          (2)
    return parser.output()  (3)
```

(1) Dopo tutto quell' immaginare, questo vi potrà sembrare piuttosto noioso, ma è la funzione `feed` che inizializza l'intera trasformazione. L'intero sorgente HTML è memorizzato in una stringa, per cui basta chiamare `feed` solo una volta. Tuttavia, è anche possibile chiamare `feed` quante volte si vuole e il parser continuerà semplicemente ad analizzare. Se vi state preoccupando di usare troppa memoria (o sapete di dover gestire pagine HTML molto grandi), è possibile creare un ciclo in cui si leggano pochi bytes di HTML per volta da passare al parser. Il risultato sarebbe identico.

(2) Poiché `feed` mantiene un buffer interno, è opportuno chiamare sempre il metodo `close` del parser una volta finito (anche se si passa tutto il codice HTML in un colpo solo, come abbiamo fatto noi). Altrimenti, potreste scoprire che al vostro output mancano gli ultimi bytes.

(3) Ricordate, `output` è la funzione che abbiamo definito nella classe `BaseHTMLProcessor` per mettere insieme tutte le parti dell'output che avevamo memorizzato e restituirle in una singola stringa.

Ed ecco fatto, abbiamo "tradotto" una pagina web, passando nient'altro che una URL ed il nome di un dialetto.

Ulteriori letture

- Pensavate che stessi scherzando sull'idea del server−side scripting. Così ho fatto, fino a che non ho trovato un dialettizatore. Non ho idea se sia implementato in Python, ma la home page della mia compagnia è felice. Sfortunatamente, il codice sorgente non sembra essere disponibile.

5.11. Sommario

Python fornisce uno strumento potente, il modulo `sgmllib.py`, per manipolare codice HTML trasformando la sua struttura in un oggetto modello. È possibile usare questo strumento in molti modi diversi.

- analizzare il codice HTML per cercarvi qualcosa di specifico
- produrre da esso un risultato aggregato, come nel caso del programma che elencava le URL
- alterarne la struttura mentre lo si analizza, come per il programma che "virgolettava" gli attributi
- trasformare il codice HTML in qualcos'altro, elaborando il testo ma lasciando inalterati i tag, come nel caso del `Dialectizer`

Oltre a questi questi esempi, dovreste essere a vostro agio nell'eseguire una delle seguenti operazioni:

- Usare le funzioni `locals`() e `globals`() per accedere agli spazi dei nomi
- Formattare stringhe usando sostituzioni basate su dizionari
- Usare le espressioni regolari. Sono una componente importante nella borsa degli attrezzi di ogni programmatore ed avranno un ruolo importante nei prossimi capitoli.

[7] Il termine tecnico per un analizzatore come `SGMLParser` è *consumatore*: questa classe consuma codice HTML e lo decompone. Presumibilmente, il nome `feed` (nutrire, ndt) è stato scelto proprio per adattarsi all'analogia del "consumatore". Personalmente, la cosa mi fa pensare ad un esemplare in uno zoo, in una gabbia scura in cui non si vedano piante o vita di alcun tipo, ma in cui, se restate immobili e guardate molto da vicino, potete scorgere due occhi piccoli e lucenti che vi osservano di rimando dall'angolo più lontano, per convincervi subito dopo che è solo uno scherzo della vostra immaginazione L'unico modo per capire che non si tratta semplicemente di una gabbia vuota è un piccolo cartello sulle sbarre, dall'aspetto innocuo, che dice "Non date da mangiare al parser". Ma forse sono solo io. Ad ogni modo, è un'interessante quadretto mentale.

[8] La ragione per cui Python è più efficiente con le liste che con le stringhe è che le liste sono oggetti modificabili, mentre le stringhe no. Questo significa che eseguire un 'append' ad una lista significa semplicemente aggiungere l'elemento ed aggiornare l'indice. Dato che invece le stringhe non possono essere cambiate, occorre usare codice come `s = s + newpiece`, che crea dapprima una nuova stringa dalla concatenazione della vecchia e del nuovo pezzo, per poi buttar via la vecchia. Questo richiede un sacco di costosa gestione della memoria, e la quantità di lavoro necessario aumenta con il crescere della dimensione della stringa, per cui fare `s = s + newpiece` in un ciclo è micidiale. In termini tecnici, aggiungere `n` elementi ad una lista è un algoritmo di classe `O(n)`, mentre aggiungere `n` elementi ad una stringa è un algoritmo di classe $O(n^2)$.

[9] Non è che io esca poi molto. ;−)

[10] Daccordo, non è poi una domanda così comune. Non allo stesso livello di "Quale editor conviene usare per scrivere codice Python?" (risposta: Emacs) oppure "Python è migliore o peggiore di Perl?" (risposta: "Perl è peggiore di Python perché la gente lo ha voluto peggiore." − Larry Wall, 14/10/1998). Ma comunque domande sull'elaborazione di testo HTML spuntano fuori in una forma o nell'altra circa una volta al mese e tra queste domande una delle più gettonate è quella che ho citato.

Capitolo 6. Elaborare XML

6.1. Immergersi

Questo capitolo affronta l'elaborazione dell'XML in Python. Potrebbe esservi utile sapere come è fatto un documento XML, per esempio sapere che è composto da elementi strutturati in modo da formare una gerarchia e così via. Se qualcosa non vi è chiaro, leggete un tutorial su XML prima di proseguire.

Non è essenziale essere un dotto filosofo, ma se avete avuto la sfortuna di affrontare gli scritti di Immanuel Kant, forse apprezzerete il programma di esempio molto più di come potrebbe uno studioso di qualche materia più utile, tipo l'informatica.

Ci sono due principali metodi per lavorare con l'XML. Il primo è chiamato SAX ("Simple API for XML"), funziona leggendo l'XML un pezzo per volta e chiamando un metodo per ogni elemento trovato. Se avete letto il capitolo Elaborare HTML, dovrebbe risultarvi familiare, perché è così che lavora il modulo `sgmllib`. L'altro è chiamato DOM ("Document Object Model"), funziona leggendo l'intero documento XML in un'unica volta e creando una rappresentazione interna dell'XML basata su classi native Python collegate in una struttura ad albero. Python ha dei moduli standard per entrambi i tipi di parsing, ma questo capitolo affronterà solo l'uso del DOM.

Quello che segue è un completo programma in Python che genera un testo pseudo−casuale basandosi su una grammatica a contenuto libero definita in formato XML. Non preoccupatevi se non capite ancora cosa significhi, esamineremo sia l'input che l'output prodotto dal programma in modo più approfondito andando avanti nel capitolo.

Esempio 6.1. `kgp.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
"""Kant Generator for Python

Generates mock philosophy based on a context−free grammar

Usage: python kgp.py [options] [source]

Options:
  −g ..., −−grammar=...   use specified grammar file or URL
  −h, −−help              show this help
  −d                      show debugging information while parsing

Examples:
  kgp.py                  generates several paragraphs of Kantian philosophy
  kgp.py −g husserl.xml   generates several paragraphs of Husserl
  kpg.py "<xref id='paragraph'/>"  generates a paragraph of Kant
  kgp.py template.xml     reads from template.xml to decide what to generate
"""
from xml.dom import minidom
import random
import toolbox
import sys
import getopt

_debug = 0

class NoSourceError(Exception): pass

class KantGenerator:
```

```
    """generates mock philosophy based on a context-free grammar"""

    def __init__(self, grammar, source=None):
        self.loadGrammar(grammar)
        self.loadSource(source and source or self.getDefaultSource())
        self.refresh()

    def _load(self, source):
        """load XML input source, return parsed XML document

        - a URL of a remote XML file ("http://diveintopython.org/kant.xml")
        - a filename of a local XML file ("~/diveintopython/common/py/kant.xml")
        - standard input ("-")
        - the actual XML document, as a string
        """
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc

    def loadGrammar(self, grammar):
        """load context-free grammar"""
        self.grammar = self._load(grammar)
        self.refs = {}
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref

    def loadSource(self, source):
        """load source"""
        self.source = self._load(source)

    def getDefaultSource(self):
        """guess default source of the current grammar

        The default source will be one of the <ref>s that is not
        cross-referenced.  This sounds complicated but it's not.
        Example: The default source for kant.xml is
        "<xref id='section'/>", because 'section' is the one <ref>
        that is not <xref>'d anywhere in the grammar.
        In most grammars, the default source will produce the
        longest (and most interesting) output.
        """
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        if not standaloneXrefs:
            raise NoSourceError, "can't guess source, and no source specified"
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)

    def reset(self):
        """reset parser"""
        self.pieces = []
        self.capitalizeNextWord = 0

    def refresh(self):
        """reset output buffer, re-parse entire source file, and return output

        Since parsing involves a good deal of randomness, this is an
        easy way to get new output without having to reload a grammar file
        each time.
        """
```

```
        self.reset()
        self.parse(self.source)
        return self.output()

    def output(self):
        """output generated text"""
        return "".join(self.pieces)

    def randomChildElement(self, node):
        """choose a random child element of a node

        This is a utility method used by do_xref and do_choice.
        """
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE]
        chosen = random.choice(choices)
        if _debug:
            sys.stderr.write('%s available choices: %s\n' % \
                (len(choices), [e.toxml() for e in choices]))
            sys.stderr.write('Chosen: %s\n' % chosen.toxml())
        return chosen

    def parse(self, node):
        """parse a single XML node

        A parsed XML document (from minidom.parse) is a tree of nodes
        of various types.  Each node is represented by an instance of the
        corresponding Python class (Element for a tag, Text for
        text data, Document for the top-level document).  The following
        statement constructs the name of a class method based on the type
        of node we're parsing ("parse_Element" for an Element node,
        "parse_Text" for a Text node, etc.) and then calls the method.
        """
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__)
        parseMethod(node)

    def parse_Document(self, node):
        """parse the document node

        The document node by itself isn't interesting (to us), but
        its only child, node.documentElement, is: it's the root node
        of the grammar.
        """
        self.parse(node.documentElement)

    def parse_Text(self, node):
        """parse a text node

        The text of a text node is usually added to the output buffer
        verbatim.  The one exception is that <p class='sentence'> sets
        a flag to capitalize the first letter of the next word.  If
        that flag is set, we capitalize the text and reset the flag.
        """
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Element(self, node):
        """parse an element
```

```
        An XML element corresponds to an actual tag in the source:
        <xref id='...'>, <p chance='...'>, <choice>, etc.
        Each element type is handled in its own method.  Like we did in
        parse(), we construct a method name based on the name of the
        element ("do_xref" for an <xref> tag, etc.) and
        call the method.
        """
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)

    def parse_Comment(self, node):
        """parse a comment

        The grammar can contain XML comments, but we ignore them
        """
        pass

    def do_xref(self, node):
        """handle <xref id='...'> tag

        An <xref id='...'> tag is a cross-reference to a <ref id='...'>
        tag.  <xref id='sentence'/> evaluates to a randomly chosen child of
        <ref id='sentence'>.
        """
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))

    def do_p(self, node):
        """handle <p> tag

        The <p> tag is the core of the grammar.  It can contain almost
        anything: freeform text, <choice> tags, <xref> tags, even other
        <p> tags.  If a "class='sentence'" attribute is found, a flag
        is set and the next word will be capitalized.  If a "chance='X'"
        attribute is found, there is an X% chance that the tag will be
        evaluated (and therefore a (100-X)% chance that it will be
        completely ignored)
        """
        keys = node.attributes.keys()
        if "class" in keys:
            if node.attributes["class"].value == "sentence":
                self.capitalizeNextWord = 1
        if "chance" in keys:
            chance = int(node.attributes["chance"].value)
            doit = (chance > random.randrange(100))
        else:
            doit = 1
        if doit:
            for child in node.childNodes: self.parse(child)

    def do_choice(self, node):
        """handle <choice> tag

        A <choice> tag contains one or more <p> tags.  One <p> tag
        is chosen at random and evaluated; the rest are ignored.
        """
        self.parse(self.randomChildElement(node))

def usage():
    print __doc__

def main(argv):
```

```
    grammar = "kant.xml"
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:
        if opt in ("-h", "--help"):
            usage()
            sys.exit()
        elif opt == '-d':
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"):
            grammar = arg

    source = "".join(args)

    k = KantGenerator(grammar, source)
    print k.output()

if __name__ == "__main__":
    main(sys.argv[1:])
```

Esempio 6.2. `toolbox.py`

```
"""Miscellaneous utility functions"""

def openAnything(source):
    """URI, filename, or string --> stream

    This function lets you define parsers that take any input source
    (URL, pathname to local or network file, or actual data as a string)
    and deal with it in a uniform manner.  Returned object is guaranteed
    to have all the basic stdio read methods (read, readline, readlines).
    Just .close() the object when you're done with it.

    Examples:
    >>> from xml.dom import minidom
    >>> sock = openAnything("http://localhost/kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("c:\\inetpub\\wwwroot\\kant.xml")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    >>> sock = openAnything("<ref id='conjunction'><text>and</text><text>or</text></ref>")
    >>> doc = minidom.parse(sock)
    >>> sock.close()
    """
    if hasattr(source, "read"):
        return source

    if source == '-':
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)
    except (IOError, OSError):
```

```
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)
    except (IOError, OSError):
        pass

    # treat source as string
    return StringIO.StringIO(str(source))
```

Lanciate `kgp.py` senza opzioni, il programma analizzerà la grammatica−XML predefinita in `kant.xml` e stamperà alcuni paragrafi degni della filosofia e dello stile di Immanuel Kant.

Esempio 6.3. Esempio di output di `kgp.py`

```
[f8dy@oliver kgp]$ python kgp.py
     As is shown in the writings of Hume, our a priori concepts, in
reference to ends, abstract from all content of knowledge; in the study
of space, the discipline of human reason, in accordance with the
principles of philosophy, is the clue to the discovery of the
Transcendental Deduction.  The transcendental aesthetic, in all
theoretical sciences, occupies part of the sphere of human reason
concerning the existence of our ideas in general; still, the
never−ending regress in the series of empirical conditions constitutes
the whole content for the transcendental unity of apperception.  What
we have alone been able to show is that, even as this relates to the
architectonic of human reason, the Ideal may not contradict itself, but
it is still possible that it may be in contradictions with the
employment of the pure employment of our hypothetical judgements, but
natural causes (and I assert that this is the case) prove the validity
of the discipline of pure reason.  As we have already seen, time (and
it is obvious that this is true) proves the validity of time, and the
architectonic of human reason, in the full sense of these terms,
abstracts from all content of knowledge.  I assert, in the case of the
discipline of practical reason, that the Antinomies are just as
necessary as natural causes, since knowledge of the phenomena is a
posteriori.
    The discipline of human reason, as I have elsewhere shown, is by
its very nature contradictory, but our ideas exclude the possibility of
the Antinomies.  We can deduce that, on the contrary, the pure
employment of philosophy, on the contrary, is by its very nature
contradictory, but our sense perceptions are a representation of, in
the case of space, metaphysics.  The thing in itself is a
representation of philosophy.  Applied logic is the clue to the
discovery of natural causes.  However, what we have alone been able to
show is that our ideas, in other words, should only be used as a canon
for the Ideal, because of our necessary ignorance of the conditions.

[...snip...]
```

Il risultato è ovviamente un discorso completamente incomprensibile. Insomma, non completamente incomprensibile. È corretto dal punto di vista sintattico e grammaticale −− anche se molto prolisso Kant non era proprio il tipo che andava subito al sodo. Alcune affermazioni potrebbero essere realmente vere, altre vistosamente false, la maggior parte semplicemente incoerenti. Ma comunque nello stile di Immanuel Kant.

Lasciatemi ripetere che tutto questo è molto più divertente se studiate o avete studiato filosofia.

La cosa interessante di questo programma è che non contiene niente di Kant−specifico. Tutto il contenuto dell'esempio precedente è estratto dal file di grammatica `kant.xml`. Se diciamo al programma di usare un altro file di grammatica (cosa che possiamo specificare dalla linea di comando), il risultato sarà completamente diverso.

Esempio 6.4. Semplice output di `kgp.py`

```
[f8dy@oliver kgp]$ python kgp.py −g binary.xml
00101001
[f8dy@oliver kgp]$ python kgp.py −g binary.xml
10110100
```

Più avanti in questo capitolo vedremo con attenzione la struttura del file di grammatica . Per ora, tutto quello che dovete sapere è che la grammatica definisce la struttura di ciò che viene prodotto e che il programma `kgp.py` legge la grammatica e sceglie casualmente che parole prendere e dove inserirle.

6.2. Package

Analizzare un documento XML è estremamente semplice: una sola riga di codice. Comunque, prima di studiare questa riga di codice, dobbiamo fare una piccola deviazione per parlare dei package.

Esempio 6.5. Caricare un documento XML (una rapida occhiata)

```
>>> from xml.dom import minidom (1)
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')
```

(1) Questa è una sintassi che non abbiamo visto in precedenza. Assomiglia molto alla `from` *`module`* `import` che conosciamo ed amiamo, ma il `"."` la fa sembrare qualche cosa che va oltre una semplice importazione di un modulo. Infatti, `xml` è quella cosa nota come package, `dom` è un package annidato in `xml` e `minidom` è un modulo all'interno di `xml.dom`.

Suona complicato, ma in realtà non lo è. Uno sguardo all'implementazione può aiutare. I package sono qualcosa di più che directory di moduli; i package annidati sono sottodirectory. I moduli in un package (o in un package annidato) sono semplicemente dei file `.py`, come sempre, salvo il fatto che sono in una sottodirectory invece che nella directory principale `lib/` della vostra installazione Python.

Esempio 6.6. Disposizione dei file di un package

```
Python21/           root Python installation (home of the executable)
|
+−−lib/             library directory (home of the standard library modules)
   |
   +−− xml/         xml package (really just a directory with other stuff in it)
       |
       +−−sax/      xml.sax package (again, just a directory)
       |
       +−−dom/      xml.dom package (contains minidom.py)
       |
       +−−parsers/  xml.parsers package (used internally)
```

Quindi, quando diciamo `from xml.dom import minidom`, Python capisce che questo significa "cerca una directory `dom` nella directory `xml`, cerca *lì* dentro il modulo `minidom` ed importalo come `minidom`". Ma Python è ancora più intelligente di così; non solo potete importare interi moduli contenuti in un package, potete selettivamente importare classi specifiche o funzioni da un modulo contenuto in un package. Potete anche importare l'intero package

come un modulo. La sintassi è la stessa; Python capisce che cosa volete in base alla disposizione dei file nel package ed automaticamente fa la cosa giusta.

Esempio 6.7. I package sono anche dei moduli

```
>>> from xml.dom import minidom         (1)
>>> minidom
<module 'xml.dom.minidom' from 'C:\Python21\lib\xml\dom\minidom.pyc'>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml.dom.minidom import Element (2)
>>> Element
<class xml.dom.minidom.Element at 01095744>
>>> minidom.Element
<class xml.dom.minidom.Element at 01095744>
>>> from xml import dom                 (3)
>>> dom
<module 'xml.dom' from 'C:\Python21\lib\xml\dom\__init__.pyc'>
>>> import xml                          (4)
>>> xml
<module 'xml' from 'C:\Python21\lib\xml\__init__.pyc'>
```

(1) Qui stiamo importando un modulo (`minidom`) da un package annidato (`xml.dom`). Il risultato è che `minidom` è importato nel nostro namespace e per ottenere un riferimento alle classi interne al modulo `minidom` (come `Element`), dobbiamo anteporre al loro nome il nome del modulo.

(2) Qui stiamo importando una classe (`Element`) da un modulo (`minidom`) contenuto in un package annidato (`xml.dom`). Il risultato è che `Element` è importata direttamente nel nostro namespace. Notate che questo non interferisce con l'import precedente; ora si può fare riferimento alla classe `Element` in due modi, ma è sempre la medesima classe.

(3) Qui stiamo importando il package `dom` (un package annidato di `xml`) come se fosse un modulo a sé stante. Ogni livello di un package può essere trattato come un modulo, come vedremo tra breve. Può anche avere i propri attributi e metodi, come i moduli che abbiamo già visto prima.

(4) Qui stiamo importando come un modulo il livello radice del package `xml`.

Come può dunque un package (che è semplicemente una directory nel disco) essere importato e trattato come un modulo (che è sempre un file in un disco)? La risposta sta nella magia del file `__init__.py`. Vedete, i package non sono semplicemente directory, sono directory con un file specifico, `__init__.py`, al loro interno. Tale file definisce gli attributi ed i metodi del package. Per esempio, `xml.dom` contiene una classe `Node` che è definita in `xml/dom/__init__.py`. Quando importate un package come un modulo (come `dom` da `xml`), state in realtà importando il suo file `__init__.py`.

> **Nota: Di che cosa è fatto un package**
> Un package è una directory con il file speciale `__init__.py` dentro. Il file `__init__.py` definisce gli attributi ed i metodi del package. Non deve obbligatoriamente definire nulla; può essere un file vuoto, ma deve esistere. Ma se `__init__.py` non esiste, la directory è soltanto una directory, non un package e non può essere importata, contenere moduli o package annidati.

Allora perché preoccuparsi dei package? Beh, mettono a disposizione un modo per ragruppare logicamente moduli tra loro correlati. Invece di avere un package `xml` con i package `sax` e `dom` al suo interno, gli autori avrebbero dovuto scegliere di inserire tutte le funzionalità `sax` in `xmlsax.py` e tutte le funzionalità `dom` in `xmldom.py`, oppure metterle tutte quante in un solo modulo. Ma questo sarebbe stato molto ingombrante (al momento della scrittura, il package XML conta più di 3000 righe di codice) e difficile da gestire (sorgenti separati significa che più persone possono lavorare contemporaneamente su aree diverse).

Se vi ritrovate a scrivere un vasto sottosistema in Python (o più probabilmente, quando realizzate che il vostro piccolo sottosistema è diventato grande), investite un po' di tempo nel disegnare una buona architettura del package. È una delle molte cose in cui Python eccelle, quindi avvantaggiatevene.

6.3. Analizzare XML

Come stavo dicendo, analizzare un documento XML è molto semplice: una riga di codice. Dove andare poi, dipende da voi.

Esempio 6.8. Caricare un documento XML (questa volta per davvero)

```
>>> from xml.dom import minidom                                          (1)
>>> xmldoc = minidom.parse('~/diveintopython/common/py/kgp/binary.xml')  (2)
>>> xmldoc                                                               (3)
<xml.dom.minidom.Document instance at 010BE87C>
>>> print xmldoc.toxml()                                                 (4)
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

(1) Come abbiamo visto nella sezione precedente, questo importa il modulo `minidom` dal package `xml.dom`.

(2) Questa è la riga di codice che fa tutto il lavoro: `minidom.parse` prende un argomento e ritorna la rappresentazione analizzata del documento XML. L'argomento può essere di vario tipo; in questo caso, è semplicemente un nome di file di un documento XML nel mio disco (per seguirmi, occorre che cambiate il percorso del file in modo che punti alla directory dove avete scaricato gli esempi). Ma potete anche passare un oggetto di tipo file o anche un oggetto che si comporta come un file. Ci avvantaggeremo di questa flessibilità più avanti nel capitolo.

(3) L'oggetto ritornato da `minidom.parse` è un oggetto di tipo `Document`, un discendente della classe `Node`. Questo oggetto `Document` è la radice di una struttura ad albero piuttosto complessa di oggetti Python interconnessi, che rappresenta per intero il documento XML che abbiamo passato a `minidom.parse`.

(4) `toxml` è un metodo della classe `Node` (ed è dunque disponibile nell'oggetto `Document` ottenuto da `minidom.parse`). `toxml` stampa l'XML rappresentato da questo `Node`. Per il nodo `Document`, viene stampato l'intero documento XML.

Ora che abbiamo un documento XML in memoria, possiamo iniziare ad elaborarlo.

Esempio 6.9. Ottenere i nodi figli

```
>>> xmldoc.childNodes    (1)
[<DOM Element: grammar at 17538908>]
>>> xmldoc.childNodes[0] (2)
<DOM Element: grammar at 17538908>
>>> xmldoc.firstChild    (3)
<DOM Element: grammar at 17538908>
```

(1) Ogni `Node` ha un attributo `childNodes`, che è una lista di oggetti `Node`. Un `Document` ha sempre solo un nodo figlio, l'elemento radice del documento XML (in questo caso, l'elemento `grammar`).

(2) Per ottenere il primo (ed in questo caso, il solo) nodo figlio, usate la solita sintassi vista per le liste. Ricordate, non c'è nulla di speciale qui; è soltanto una normale lista Python di normali oggetti Python.

(3) Siccome ottenere il primo nodo figlio di un nodo è un'attività utile e comune, la classe `Node` ha un attributo `firstChild`, che è sinonimo di `childNodes[0]`. C'è anche l'attributo `lastChild`, che è sinonimo di `childNodes[-1]`.

Esempio 6.10. `toxml` funziona su ogni nodo

```
>>> grammarNode = xmldoc.firstChild
>>> print grammarNode.toxml() (1)
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

(1) Siccome il metodo `toxml` è definito nella classe `Node`, è disponibile in ogni nodo XML, non solo nell'elemento `Document`.

Esempio 6.11. I nodi figli possono essere di testo

```
>>> grammarNode.childNodes                  (1)
[<DOM Text node "\n">, <DOM Element: ref at 17533332>, \
<DOM Text node "\n">, <DOM Element: ref at 17549660>, <DOM Text node "\n">]
>>> print grammarNode.firstChild.toxml()    (2)



>>> print grammarNode.childNodes[1].toxml() (3)
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print grammarNode.childNodes[3].toxml() (4)
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
>>> print grammarNode.lastChild.toxml()     (5)
```

(1) Guardando l'XML in `binary.xml`, potreste pensare che il nodo `grammar` ha solo due nodi figli, i due elementi `ref`. Ma vi state perdendo qualcosa: le andate a capo! Dopo il `'<grammar>'` e prima del primo `'<ref>'` c'è un'andata a capo e questo testo viene considerato come nodo figlio dell'elemento `grammar`. Analogamente, ci sono delle andate a capo dopo ogni `'</ref>'`; anche queste vengono considerate nodi. Dunque `grammar.childNodes` è una lista di 5 oggetti: 3 oggetti di tipo `Text` e 2 oggetti di tipo

`Element`.

(2) Il primo figlio è un oggetto di tipo `Text` rappresentante l'andata a capo dopo il tag `'<grammar>'` e prima del primo tag `'<ref>'`.

(3) Il secondo figlio è un oggetto di tipo `Element` rappresentante il primo elemento `ref`.

(4) Il quarto figlio è un oggetto di tipo `Element` rappresentante il secondo elemento `ref`.

(5) L'ultimo figlio è un oggetto di tipo `Text` rappresentante l'andata a capo dopo il tag di chiusura `'</ref>'` e prima del tag di chiusura `'</grammar>'`.

Esempio 6.12. Tirare fuori tutti i nodi di testo

```
>>> grammarNode
<DOM Element: grammar at 19167148>
>>> refNode = grammarNode.childNodes[1] (1)
>>> refNode
<DOM Element: ref at 17987740>
>>> refNode.childNodes                  (2)
[<DOM Text node "\n">, <DOM Text node "  ">, <DOM Element: p at 19315844>, \
<DOM Text node "\n">, <DOM Text node "  ">, \
<DOM Element: p at 19462036>, <DOM Text node "\n">]
>>> pNode = refNode.childNodes[2]
>>> pNode
<DOM Element: p at 19315844>
>>> print pNode.toxml()                 (3)
<p>0</p>
>>> pNode.firstChild                    (4)
<DOM Text node "0">
>>> pNode.firstChild.data               (5)
u'0'
```

(1) Come abbiamo visto nell'esempio precedente, il primo elemento `ref` è `grammarNode.childNodes[1]`, in quanto childNodes[0] è un nodo di tipo `Text` per l'andata a capo.

(2) L'elemento `ref` ha il suo insieme di nodi figli, uno per l'andata a capo, un altro per gli spazi, uno per l'elemento `p`, e così via.

(3) Puoi sempre usare il metodo `toxml` qui, profondamente annidato nel documento.

(4) L'elemento `p` ha solo un nodo figlio (non potete dirlo da questo esempio, ma guardate `pNode.childNodes` se non mi credete) ed è un nodo di tipo `Text` contenente il solo carattere `'0'`.

(5) L'attributo `.data` di un nodo di tipo `Text` restituisce la stringa rappresentata dal nodo. Ma che cos'è la `'u'` davanti la stringa? La risposta merita di avere una propria sezione.

6.4. Unicode

Unicode è un sistema per rappresentare i caratteri di tutti i differenti linguaggi del mondo. Quando Python analizza un documento XML, tutti i dati sono immagazzinati in memoria sottoforma di unicode.

Sarà spiegato tutto fra un minuto, ma prima, una piccola introduzione.

Nota storica. Prima dell'unicode, esistevano sistemi di codifica dei caratteri separati per ogni linguaggio, ognuno usava gli stessi numeri (0−255) per rappresentare i caratteri di quel linguaggio. Alcuni (come il russo) avevano multipli standard, in conflitto sul modo di rappresentare lo stesso carattere; altri linguaggi (come il giapponese) avevano così tanti caratteri che richiedevano numerosi set di caratteri. Lo scambio di documenti fra i sistemi era difficile perché non c'era modo per un computer di dire quale schema di codifica dei caratteri avesse usato l'autore del documento; il computer vedeva solo numeri ed i numeri potevano significare cose differenti. Perciò pensate di porre

questi documenti nel medesimo luogo (ad esempio la stessa tabella di un database); avreste bisogno di registrare la codica dei caratteri per ogni parte del testo ed essere sicuri di fornirla insieme allo stesso. Quindi pensate a documenti multilingue, con caratteri provenienti da vari linguaggi nello stesso documento. Solitamente questi utilizzano caratteri di escape per cambiare il modo di codifica; usiamo il modo russo koi8−r, perciò il carattere 241 indica questo; adesso usiamo il Greco del Mac, perciò il carattere 241 indica qualcos'altro. E così via. Questi sono i problemi per i quali l'unicode è stato ideato.

Per risolvere questi problemi, l'unicode rappresenta ogni carattere come un numero di 2 byte, da 0 a 65535. [11] Ogni numero di 2 byte rappresenta un unico carattere usato in almeno uno dei linguaggi del mondo. Caratteri che sono usati in molteplici linguaggi hanno lo stesso codice numerico. C'è esattamente 1 numero per carattere ed esattamente un carattere per numero. I dati unicode non sono mai ambigui.

Ovviamente c'è ancora il problema di tutte queste eredità dei sistemi di codifica. L'ASCII a 7 bit, per esempio, registra i caratteri inglesi come numeri da 0 a 127. (65 è la "A" maiuscola, 97 è la "a minuscola" e così via.) L'inglese ha un alfabeto molto semplice, perciò può essere espresso completamente tramite l'ASCII a 7 bit. I linguaggi dell'ovest europeo come il francese, lo spagnolo e il tedesco (e l'italiano N.d.T.) usano tutti una codifica dei caratteri chiamata ISO−8859−1 ("latin−1"), per i numeri da 0 al 127 ma vengono estesi nel gruppo di caratteri dal 128 al 255 come la "n con la tilde sopra"(241), o la "u con due punti sopra"(252). L'unicode utilizza gli stessi caratteri dell'ASCII a 7 bit per i numeri dallo 0 al 127 e gli stessi caratteri dell'ISO−8859−1 per i numeri da 128 a 255, estende da quel punto in poi, tutti i caratteri per gli altri linguaggi con i numeri rimanenti, da 256 a 65535.

Quando vi occupate dei dati unicode, potreste aver bisogno, in alcuni punti, di convertire i dati in una delle altre codifiche dei caratteri descritte. Per esempio, per integrarli con qualche altro sistema che si aspetta dei dati in uno schema di codifica specifico ad 1 byte o stamparlo da un terminale o da una stampante non unicode. Oppure immagazzinarli in un documento XML che esplicita specificatamente lo schema di codifica.

E dopo questa nota, torniamo al Python.

Python supporta l'unicode nativamente sin dalla versione 2.0. [12] Il pacchetto XML usa l'unicode per registrare tutti i dati XML analizzati, ma potete usare l'unicode ovunque.

Esempio 6.13. Introduzione sull'unicode

```
>>> s = u'Dive in'           (1)
>>> s
u'Dive in'
>>> print s                  (2)
Dive in
```

(1) Per creare una stringa unicode, invece di una regolare stringa ASCII, aggiungete la lettera "u" prima della stringa. Notate che questa particolare stringa non contiene nessun carattere non−ASCII. Questo è bene; unicode è un superset dell'ASCII (un superset molto grande in realtà), così che ogni stringa regolare ASCII possa essere registrata come unicode.

(2) Quando stampiamo una stringa, Python si preoccupa di convertirla nel vostra codifica di default, che solitamente è l'ASCII (vi dirò di più fra un minuto). Dato che questa stringa unicode è formata da caratteri che sono anche caratteri ASCII, stamparla ha lo stesso risultato di stampare una normale stringa ASCII; la conversione è senza cuciture e se non sapeste che s era una stringa unicode, non notereste mai la differenza.

Esempio 6.14. immagazzinare caratteri non−ASCII

```
>>> s = u'La Pe\xf1a'        (1)
>>> print s                  (2)
```

```
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> print s.encode('latin-1') (3)
La Peña
```

(1) Il vero vantaggio dell'unicode è, ovviamente, la sua abilità di immagazzinare caratteri non-ASCII, come la spagnola "ñ" (`n` con una tilde sopra). Il carattere unicode per la tilde-n è `0xf1` in esadecimale (241 in decimale), perciò potete digitarla in questo modo: `\xf1`.

(2) Ricordate che dissi che la funzione `print` tenta di convertire una stringa unicode in ASCII in modo da poterla stampare? Bene, non funzionerà qui, visto che la nostra stringa unicode contiene caratteri non-ASCII, perciò Python solleva un errore `UnicodeError`.

(3) Ecco dove avviene la conversione dall'unicode ad altri schemi di codifica. `s` è una stringa unicode ma `print` può stampare solo stringhe regolari. Per risolvere questo problema, chiamiamo il metodo `encode`, disponibile in ogni stringa unicode, per convertire una stringa unicode in una stringa regolare nello schema di codifica dato, che passiamo come parametro. In questo caso, stiamo usando `latin-1` (anche conosciuto come `iso-8859-1`), che include la tilde-n (che invece l'ASCII non include, dato che include unicamente i caratteri numerati da 0 a 127).

Ricordate quando dissi che Python solitamente convertiva unicode in ASCII, ovunque occorresse creare una stringa regolare da una unicode? Bene, questo schema di codifica predefinito è un'opzione che potete personalizzare.

Esempio 6.15. `sitecustomize.py`

```
# sitecustomize.py                    (1)
# this file can be anywhere in your Python path,
# but it usually goes in ${pythondir}/lib/site-packages/

import sys

sys.setdefaultencoding('iso-8859-1') (2)
```

(1) `sitecustomize.py` è uno script speciale; Python tenterà di importarlo all'avvio, così ogni codice al suo interno partirà automaticamente. Come menzionato dal commento, può essere posto ovunque (fin dove `import` lo possa trovare), ma solitamente viene messo nella directory `lib/site-packages` della vostra distribuzione Python.

(2) La funzione `setdefaultencoding` imposta, quindi, la codifica predefinita. Questo è lo schema di codifica che Python proverà ad usare, ogniqualvolta occorresse automatizzare la conversione da una stringa unicode ad una stringa regolare.

Esempio 6.16. Effetti dell'impostazione di una codifica predefinita

```
>>> import sys
>>> sys.getdefaultencoding() (1)
'iso-8859-1'
>>> s = u'La Pe\xf1a'
>>> print s                  (2)
La Peña
```

(1) Questo esempio richiede che voi abbiate fatto le modifiche elencate nell'esempio precedente al vostro file `sitecustomize.py` e che abbiate riavviato Python. Se la vostra codifica predefinita è ancora `'ascii'`, vuol dire che non avete impostato correttamente il vostro `sitecustomize.py` o che non avete riavviato Python. La codifica predefinita può essere modificata unicamente durante l'avvio di Python; non potete farlo dopo. Grazie ad alcuni trucchi di programmazione che non voglio mostrare adesso, potete chiamare anche

`sys.setdefaultencoding` dopo che Python è stato avviato. Scavate in `site.py` e cercate "`setdefaultencoding`" per maggiori approfondimenti.

(2) Adesso che lo schema di codifica predefinito include tutti i caratteri usati nella nostra stringa, Python non ha problemi ad auto−convertire la stringa ed a stamparla.

E riguardo l'XML? Bene, ogni documento XML contiene una codifica specifica. ISO−8859−1 è una popolare codifica per dati nei linguaggi dell'ovest europeo. KOI8−R è comune nei testi russi. La codifica, se specificata, si trova nell'header del documento XML.

Esempio 6.17. `russiansample.xml`

```
<?xml version="1.0" encoding="koi8-r"?>       (1)
<preface>
<title> @548A;>285</title>                    (2)
</preface>
```

(1) Questo esempio è estratto da un reale documento XML in russo; È parte della traduzione di questo libro, notate che la codifica, `koi8-r`, è specificata nell'header.

(2) Questi sono i caratteri cirillici che, per quanto ne so, esprimono in russo la parola "Prefazione". Se aprite questo file con un normale editor di testi, i caratteri sembreranno incomprensibili, visto che sono codificati usando lo schema di codifica `koi8-r`, ma vengono mostrati in `iso-8859-1`.

Esempio 6.18. Analizzare `russiansample.xml`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('russiansample.xml') (1)
>>> title = xmldoc.getElementsByTagName('title')[0].firstChild.data
>>> title                                       (2)
u'\u041f\u0440\u0435\u0434\u0438\u0441\u043b\u043e\u0432\u0438\u0435'
>>> print title                                 (3)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
UnicodeError: ASCII encoding error: ordinal not in range(128)
>>> convertedtitle = title.encode('koi8-r')     (4)
>>> convertedtitle
'\xf0\xd2\xc5\xc4\xc9\xd3\xcc\xcf\xd7\xc9\xc5'
>>> print convertedtitle                        (5)
 @548A;>285
```

(1) Assunto che abbiate salvato l'esempio precedente come `russiansample.xml` nella directory corrente ed inoltre, per completezza, che abbiate riportato la vostra codifica predefinita di nuovo in `'ascii'`, rimuovendo il file `sitecustomize.py` o almeno commentando la linea `setdefaultencoding`.

(2) Notate i dati testo della etichetta del `titolo` ... ora nella variabile `title`, grazie a quella lunga concatenazione di funzioni Python che ho precipitosamente saltato e, fastidiosamente, non spiegherò fino alla prossima sezione ... i dati di testo nel `titolo` del documento XML sono registrati come unicode.

(3) Stampare il titolo non è possible, perché questa stringa unicode contiene caratteri non−ASCII, e Python non può convertirli in ASCII perché non hanno significato.

(4) Possiamo, comunque, convertirlo esplicitamente in `koi8-r` ed otterremmo una (regolare, non unicode) stringa di caratteri ad un byte (`f0`, `d2`, `c5` e così via) che sono la vesione `koi8-r`−encoded dei caratteri nella stringa unicode originale.

(5) Stampare la stringa codificata in `koi8-r` probabilmente mostrerà insensatezze sul vostro schermo, dato che il vostro IDE Python sta interpretando questi caratteri come `iso-8859-1`, non `koi8-r`. Ma almeno le stampa. Se osservate attentamente, è la stessa roba che avete visto quando avete aperto il documento originale XML in

un editor non unicode. Python lo convertì da `koi8-r` in unicode quando analizzò il documento XML ed ora lo abbiamo appena riconvertito.

Per riassumere, unicode stesso è un pò intimidente se non lo avete mai visto prima, ma i dati unicode sono veramente facili da gestire in Python. Se i vostri documenti XML sono tutti in ASCII a 7 bit (come gli esempi in questo capitolo) non avrete mai bisogno di pensare all'unicode. Python convertirà i dati ASCII nel documento XML in unicode, durante il parsing, ed auto−convertirà in ASCII ogniqualvolta sarà necessario, non ve ne accorgerete neanche. Ma se avete bisogno di occuparvi di dati in altre lingue, Python è pronto.

Ulteriori letture

- Unicode.org è la home page dello standard unicode, inclusa una breve introduzione tecnica.
- Unicode Tutorial ha alcuni esempi su come usare le funzioni unicode di Python, incluso come forzare Python a coercizzare l'unicode in ASCII anche quando non lo vuole.
- Unicode Proposal è l'originale specifica tecnica per le funzionalità unicode di Python. Solo per per programmatori esperti dell'unicode.

6.5. Ricercare elementi

Attraversare documenti XML passando da un nodo all'altro può essere noioso. Se state cercando qualcosa in particolare, bene in profondità nel vostro documento XML, c'è una scorciatoia che potete usare per trovarlo più velocemente: `getElementsByTagName`.

Per questa sezione, useremo il file di grammatica `binary.xml`, che si presenta come segue:

Esempio 6.19. `binary.xml`

```
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "−//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

Ha due `ref`, `'bit'` e `'byte'`. Un `bit` può essere uno `'0'` od un `'1'` ed un `byte` è composto da 8 `bit`.

Esempio 6.20. Introduzione a `getElementsByTagName`

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref') (1)
>>> reflist
[<DOM Element: ref at 136138108>, <DOM Element: ref at 136144292>]
>>> print reflist[0].toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> print reflist[1].toxml()
```

```
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
```

(1) `getElementsByTagName` prende un argomento, il nome dell'elemento che volete trovare. Ritorna una lista di oggetti di tipo `Element`, corrispondenti agli elementi XML che hanno quel nome. In questo caso, troviamo due elementi `ref`.

Esempio 6.21. Ogni elemento è ricercabile

```
>>> firstref = reflist[0]                      (1)
>>> print firstref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> plist = firstref.getElementsByTagName("p") (2)
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>]
>>> print plist[0].toxml()                      (3)
<p>0</p>
>>> print plist[1].toxml()
<p>1</p>
```

(1) Continuando dall'esempio precedente, il primo oggetto nella nostra `lista di ref` è il `ref` associato a `'bit'`.

(2) Possiamo usare lo stesso metodo `getElementsByTagName` su questo oggetto `Element` per trovare tutti gli elementi `<p>` all'interno dell'elemento `ref` associato a `'bit'`.

(3) Proprio come prima, il metodo `getElementsByTagName` restituisce una lista con tutti gli elementi trovati. In questo caso, ne abbiamo due, uno per ogni bit.

Esempio 6.22. La ricerca è ricorsiva

```
>>> plist = xmldoc.getElementsByTagName("p") (1)
>>> plist
[<DOM Element: p at 136140116>, <DOM Element: p at 136142172>, <DOM Element: p at 136146124>]
>>> plist[0].toxml()                         (2)
'<p>0</p>'
>>> plist[1].toxml()
'<p>1</p>'
>>> plist[2].toxml()                         (3)
'<p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>'
```

(1) Notate attentamente la differenza tra questo esempio ed il precedente. Precedentemente, stavamo cercando gli elementi `p` in `firstref`, qui stiamo cercando gli elementi `p` in `xmldoc`, cioè l'oggetto radice che rappresenta l'intero documento XML. In *questo modo* trova tutti gli elementi `p` annidati negli elementi `ref` all'interno del nodo radice `grammar`.

(2) I primi due elementi `p` sono all'interno del primo `ref` (il `ref` associato a `'bit'`).

(3) L'ultimo elemento `p` è quello all'interno del secondo `ref` (il `ref` associato a `'byte'`).

6.6. Accedere agli attributi di un elemento

Gli elementi XML possono avere uno o più attributi ed è incredibilmente semplice accedervi una volta che avete analizzato il documento XML.

Per questa sezione, useremo il file di grammatica `binary.xml` che abbiamo visto nella sezione precedente.

> **Nota: Attributi XML ed attributi Python**
> Questa sezione potrebbe risultare un po' confusa a causa della sovrapposizione nella terminologia. Gli elementi in un documento XML hanno degli attributi ed anche gli oggetti Python hanno degli attributi. Quando analizziamo un documento XML, otteniamo un gruppo di oggetti Python che rappresentano tutti i pezzi del documento XML ed alcuni di questi oggetti rappresentano gli attributi degli elementi XML. Ma anche gli oggetti (Python) che rappresentano gli attributi (XML) hanno attributi, che vengono utilizzati per accedere alle varie parti degli attributi (XML) che l'oggetto rappresenta. Ve l'ho detto che vi avrebbe confuso. Sono aperto a suggerimenti su come distinguere le due cose in maniera più chiara.

Esempio 6.23. Accedere agli attributi di un elemento

```
>>> xmldoc = minidom.parse('binary.xml')
>>> reflist = xmldoc.getElementsByTagName('ref')
>>> bitref = reflist[0]
>>> print bitref.toxml()
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
>>> bitref.attributes          (1)
<xml.dom.minidom.NamedNodeMap instance at 0x81e0c9c>
>>> bitref.attributes.keys()   (2) (3)
[u'id']
>>> bitref.attributes.values() (4)
[<xml.dom.minidom.Attr instance at 0x81d5044>]
>>> bitref.attributes["id"]    (5)
<xml.dom.minidom.Attr instance at 0x81d5044>
```

(1) Ogni oggetto di tipo `Element` ha un attributo chiamato `attributes`, che è un oggetto `NamedNodeMap`. Suona spaventoso, ma non lo è, perché una `NamedNodeMap` è un oggetto che si comporta come un dizionario, dunque già sapete come usarla.

(2) Trattando la `NamedNodeMap` come un dizionario, possiamo ottenere una lista dei nomi degli attributi di questo elemento usando `attributes.keys()`. Questo elemento ha un solo attributo, `'id'`.

(3) I nomi degli attributi, come tutti gli altri testi in un documento XML, sono memorizzati in unicode.

(4) Ancora, trattando la `NamedNodeMap` come un dizionario, possiamo ottenere una lista dei valori degli attributi utilizzando `attributes.values()`. I valori sono essi stessi degli oggetti, di tipo `Attr`. Vedremo come ottenere utili informazioni da questi oggetti nel prossimo esempio.

(5) Continuando a trattare la `NamedNodeMap` come un dizionario, possiamo accedere ad un singolo attributo tramite il nome, usando la normale sintassi dei dizionari. I lettori che hanno prestato estrema attenzione già sapranno come la classe `NamedNodeMap` riesce a fare questo trucco: definendo il metodo speciale `__getitem__`. Gli altri lettori si possono tranquillizzare perché non hanno bisogno di sapere come funzioni per usarla.

Esempio 6.24. Accedere agli attributi individuali

```
>>> a = bitref.attributes["id"]
>>> a
<xml.dom.minidom.Attr instance at 0x81d5044>
>>> a.name  (1)
u'id'
>>> a.value (2)
u'bit'
```

(1) L'oggetto `Attr` rappresenta completamente un singolo attributo XML di un singolo elemento XML. Il nome dell'attributo (lo stesso nome che abbiamo usato per trovare questo oggetto nello pseudo−dizionario `NamedNodeMap` `bitref.attributes`) è memorizzato in `a.name`.

(2) Il valore di testo dell'attributo XML è memorizzato in `a.value`.

Nota: Gli attributi non hanno ordinamento
Come un dizionario, gli attributi di un elemento XML non hanno ordinamento. Gli attributi *possono essere* elencati in un certo ordine nel documento XML originale e gli oggetti `Attr` *possono essere* elencati in un certo ordine quando il documento XML viene interpretato in oggetti Python, ma questi ordinamenti sono arbitrari e non dovrebbero avere alcun significato particolare. Dovreste sempre accedere agli attributi in base al nome, come nel caso delle chiavi di un dizionario.

6.7. Astrarre le sorgenti di ingresso

Uno dei punti di forza di Python è il suo binding dinamico, ed uno degli usi più potenti del binding dinamico sono gli *oggetti file* (ovvero, che si comportano come dei file).

Molte funzioni che richiedono una sorgente di ingresso potrebbero semplicemente prendere un nome di un file, aprirle il file in lettura, leggerlo e chiuderlo quando hanno finito. Ma non lo fanno. Prendono, invece, degli *oggetti file*.

Nel caso più semplice, un *oggetto file* è un oggetto dotato di un metodo `read` con un parametro opzionale `size`, che ritorna una stringa. Quando viene chiamato senza il parametro `size`, legge tutto ciò che c'è da leggere dalla sorgente di ingresso e ritorna tutti i dati come una sola stringa. Quando viene chiamato con il parametro `size`, legge quella quantità dalla sorgente di ingresso e ritorna quella quantità di dati; quando viene chiamato nuovamente, riprende da dove aveva lasciato e ritorna il prossimo spezzone di dati.

È così che funziona la lettura dai veri file; la differenza è che non ci stiamo limitando ai veri file. La sorgente di ingresso potrebbe essere qualunque cosa: un file su disco, una pagina web, anche una stringa hard−coded. Fino a quando passiamo un oggetto file−like alla funzione e la funzione chiama semplicemente il metodo `read` dell'oggetto, la funzione può gestire ogni genere di sorgente di ingresso senza la necessità di codice specifico per ogni tipo.

Nel caso vi stiate chiedendo cosa ha a che vedere con l'analisi dell'XML, `minidom.parse` è una di quelle funzioni che può prendere un *oggetto file*.

Esempio 6.25. Analizzare XML da file

```
>>> from xml.dom import minidom
>>> fsock = open('binary.xml')    (1)
>>> xmldoc = minidom.parse(fsock) (2)
>>> fsock.close()                 (3)
>>> print xmldoc
<?xml version="1.0" ?>
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
```

```
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
```

(1) Per prima cosa, apriamo il file su disco. Questo ci restituisce un oggetto di tipo file.

(2) Passiamo l'oggetto file a `minidom.parse`, che chiama il metodo `read` di `fsock` e legge il documento XML dal file su disco.

(3) Siate sicuri di chiamare il metodo `close` sull'oggetto file quando avete finito. `minidom.parse` non lo farà per voi.

Bene, tutto questo sembra una colossale perdita di tempo. Dopo tutto, abbiamo già visto che `minidom.parse` può semplicemente prendere il nome del file ed effettuare tutte le operazioni di apertura e chiusura autonomamente. Ed è vero che se sapete di dover analizzare un file locale, potete passargli il nome del file e `minidom.parse` è sufficientemente intelligente da Fare La Cosa Giusta (tm). Ma notate quanto sia simile, e facile, analizzare un documento XML direttamente da Internet.

Esempio 6.26. Analizzare XML da una URL

```
>>> import urllib
>>> usock = urllib.urlopen('http://slashdot.org/slashdot.rdf') (1)
>>> xmldoc = minidom.parse(usock)                              (2)
>>> usock.close()                                              (3)
>>> print xmldoc.toxml()                                       (4)
<?xml version="1.0" ?>
<rdf:RDF xmlns="http://my.netscape.com/rdf/simple/0.9/"
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<channel>
<title>Slashdot</title>
<link>http://slashdot.org/</link>
<description>News for nerds, stuff that matters</description>
</channel>

<image>
<title>Slashdot</title>
<url>http://images.slashdot.org/topics/topicslashdot.gif</url>
<link>http://slashdot.org/</link>
</image>

<item>
<title>To HDTV or Not to HDTV?</title>
<link>http://slashdot.org/article.pl?sid=01/12/28/0421241</link>
</item>

[...snip...]
```

(1) Come abbiamo visto nel capitolo precedente, `urlopen` prende una pagina web URL e restituisce un *oggetto file*. Più importante, questo oggetto ha un metodo `read` che restituisce il sorgente HTML della pagina web.

(2) Ora passiamo l'*oggetto file* a `minidom.parse`, che obbedientemente chiama il metodo `read` dell'oggetto ed analizza i dati XML che il metodo `read` restituisce. Il fatto che questi dati XML stiano arrivando da una pagina web è completamente irrilevante. `minidom.parse` non sa niente di pagine web e non gli importa nulla delle pagine web; sa solo come usare gli *oggetti file*.

(3) Non appena avete finito, siate certi di chiudere l'oggetto file−like che `urlopen` restituisce.

(4) A proposito, questa URL è reale e contiene davvero XML. È una rappresentazione XML delle attuali intestazioni su Slashdot, un sito di tecnica e pettegolezzi.

Esempio 6.27. Analizzare XML da una stringa (il metodo facile ma inflessibile)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> xmldoc = minidom.parseString(contents) (1)
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

(1) `minidom` ha un metodo, `parseString`, che prende un intero documento XML come stringa e lo analizza. Potete usare questo metodo al posto di `minidom.parse` se sapete già di avere il vostro documento XML tutto in una stringa.

Ok, allora possiamo usare la funzione `minidom.parse` per analizzare sia file locali che URL remote, ma per analizzare stringhe, usiamo ... un'altra funzione. Significa che se vogliamo essere in grado di leggere un file, una URL o una stringa, abbiamo bisogno di una logica speciale che controlli se è una stringa, e chiamare la funzione `parseString` al suo posto. Davvero insoddisfacente.

Se ci fosse un modo per trasformare una stringa in un oggetto file−like, allora potremmo semplicemente passare tale oggetto a `minidom.parse`. Ed infatti, c'è un modulo specificamente disegnato per svolgere questo compito: `StringIO`.

Esempio 6.28. Introduzione a `StringIO`

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> import StringIO
>>> ssock = StringIO.StringIO(contents)   (1)
>>> ssock.read()                          (2)
"<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock.read()                          (3)
''
>>> ssock.seek(0)                         (4)
>>> ssock.read(15)                        (5)
'<grammar><ref i'
>>> ssock.read(15)
"d='bit'><p>0</p"
>>> ssock.read()
'><p>1</p></ref></grammar>'
>>> ssock.close()                         (6)
```

(1) Il modulo `StringIO` contiene una sola classe, anch'essa chiamata `StringIO`, che vi permette di trasformare una stringa in un oggetto file−like. La classe `StringIO` prende una stringa come parametro quando crea un'istanza.

(2) Ora abbiamo un oggetto file−like e possiamo fare ogni genere di cose, come con i file, con esso. Come invocare `read`, che ritorna la stringa originale.

(3) Chiamando nuovamente `read` restituisce una stringa vuota. È così che i veri oggetti di tipo file funzionano; una volta letto l'intero file, non potete leggere di più senza spostarvi esplicitamente all'inizio del file. L'oggetto `StringIO` funziona nello stesso modo.

(4) Potete esplicitamente spostarvi all'inizio della stringa, proprio come fareste con un file, utilizzando il metodo `seek` dell'oggetto `StringIO`.

(5) Potete anche leggere la stringa in spezzoni, passando un parametro `size` al metodo `read`.

(6)

In qualunque momento, `read` ritornerà il resto della stringa che non avete ancora letto. Tutto questo rispecchia il modo di funzionamento dei file; da cui il termine *oggetti file−like*.

Esempio 6.29. Analizzare XML da una stringa (alla maniera degli oggetti file)

```
>>> contents = "<grammar><ref id='bit'><p>0</p><p>1</p></ref></grammar>"
>>> ssock = StringIO.StringIO(contents)
>>> xmldoc = minidom.parse(ssock) (1)
>>> print xmldoc.toxml()
<?xml version="1.0" ?>
<grammar><ref id="bit"><p>0</p><p>1</p></ref></grammar>
```

(1) Ora possiamo passare l'oggetto file (in realtà una `StringIO`) a `minidom.parse`, che chiamerà il metodo `read` dell'oggetto e lo analizzerà felicemente, senza mai sapere che il suo input arrivava da una stringa hard−coded.

Così ora sappiamo come usare una singola funzione, `minidom.parse`, per analizzare un documento XML memorizzato in una pagina web, in un file locale o in una stringa hard−coded. Per una pagina web, usiamo `urlopen` per ottenere un oggetto file; per un file locale, usiamo `open`; per una stringa, usiamo `StringIO`. Proseguiamo ora di un ulteriore passo per generalizzare meglio *questa* differenza.

Esempio 6.30. `openAnything`

```
def openAnything(source):                  (1)
    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:
        return urllib.urlopen(source)      (2)
    except (IOError, OSError):
        pass

    # try to open with native open function (if source is pathname)
    try:
        return open(source)                (3)
    except (IOError, OSError):
        pass

    # treat source as string
    import StringIO
    return StringIO.StringIO(str(source))  (4)
```

(1) La funzione `openAnything` prende un solo parametro, `source`, e ritorna un oggetto file. `source` è una stringa di qualche tipo; può essere una URL (come `'http://slashdot.org/slashdot.rdf'`), un percorso completo o parziale verso un file locale (come `'binary.xml'`), o una stringa che contiene i dati XML da analizzare.

(2) Per prima cosa, vediamo se `source` è una URL. Lo facciamo usando la forza bruta: proviamo ad aprirla come una URL e silenziosamente ignoriamo gli errori causati cercando di aprire qualcosa che non è una URL. È elegante nel senso che, se `urllib` mai supporterà nuovi tipi di URL in futuro, li supporteremo anche noi senza problemi.

(3) Se `urllib` ci sgrida sostenendo che `source` non è una URL valida, assumiamo che sia il percorso di un file su disco e proviamo ad aprirlo. Ancora, non facciamo nulla per controllare se `source` è un nome di file valido o meno (le regole sulla validità del nome di un file variano molto tra diverse piattaforme, dunque le sbaglieremmo comunque). Invece, proviamo ciecamente ad aprire il file e silenziosamente catturiamo gli errori.

(4) A questo punto, dobbiamo assumere che `source` sia una stringa con dei dati precodificati all'interno (visto che nient'altro ha funzionato), allora usiamo `StringIO` per creare un oggetto file da essa e lo ritorniamo. (Infatti,

siccome stiamo usando la funzione `str`, `source` non deve necessariamente essere una stringa; potrebbe essere qualunque oggetto e noi useremo la sua rappresentazione sotto forma di stringa, come definita dal metodo speciale `__str__`.)

Ora possiamo usare la funzione `openAnything` in congiunzione con `minidom.parse` per creare una funzione, che prende un parametro `source` facente riferimento in qualche modo ad un documento XML (sia una URL, od il nome di un file locale o una stringa contenente un documento XML) e lo analizza.

Esempio 6.31. Usare `openAnything` in `kgp.py`

```
class KantGenerator:
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
        return xmldoc
```

6.8. Standard input, output, ed error

Gli utenti UNIX hanno già familiarità con i concetti di standard input, standard output e standard error. Questa sezione è per il resto di voi.

Lo standard output e lo standard error (comunemente abbreviati come `stdout` e `stderr`) sono pipe integrate in ogni sistema UNIX. Quando stampate qualcosa, essa va alla pipe `stdout`; quando il vostro programma crasha e stampa informazioni di debug (come un traceback in Python), esse vanno alla pipe `stderr`. Entrambe queste pipe sono solitamente connesse al terminale video dove state lavorando, così quando un programma stampa, vedete l'output e quando un programma crasha, vedete informazioni di debug. Se state lavorando su un sistema con un IDE Python basato su finestra, `stdout` e `stderr` vanno direttamente alla vostra "Finestra interattiva".

Esempio 6.32. Introduzione a `stdout` e `stderr`

```
>>> for i in range(3):
...     print 'Dive in'             (1)
Dive in
Dive in
Dive in
>>> import sys
>>> for i in range(3):
...     sys.stdout.write('Dive in') (2)
Dive inDive inDive in
>>> for i in range(3):
...     sys.stderr.write('Dive in') (3)
Dive inDive inDive in
```

(1) Come abbiamo visto nell'Esempio 4.28, Semplici contatori , possiamo usare la funzione built−in `range` di Python per costruire semplici cicli contatori che ripetono qualcosa un certo numero di volte.

(2) `stdout` è un oggetto simile a quello di un file; chiamando la sua funzione `write` stamperà qualsiasi stringa che voi gli diate. In effetti, questo è ciò che la funzione `print` fa realmente; aggiunge un ritorno di carrello (`\r`) alla fine della stringa che state stampando e chiama `sys.stdout.write`.

(3) Nel caso più semplice, `stdout` ed `stderr` mandano il loro output nello stesso posto: l'IDE Python (se la state usando) o ad un terminale (se state usando Python dalla linea di comando). Come `stdout`, `stderr` non aggiunge i ritorni di carrello per voi; se li volete, dovete aggiungerli da soli.

`stdout` e `stderr` sono entrambi oggetti di tipo file, come quelli discussi nella sezione Astrarre le sorgenti di ingresso, ma sono entrambi di sola scrittura. Non hanno un metodo `read` ma solo quello `write`. Inoltre, sono oggetti di tipo file, potete assegnargli qualunque altro oggetto file o simile a file per redirigere i loro output.

Esempio 6.33. Redigere l'output

```
[f8dy@oliver kgp]$ python stdout.py
Dive in
[f8dy@oliver kgp]$ cat out.log
This message will be logged instead of displayed
```

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
#stdout.py
import sys

print 'Dive in'                                          (1)
saveout = sys.stdout                                    (2)
fsock = open('out.log', 'w')                            (3)
sys.stdout = fsock                                      (4)
print 'This message will be logged instead of displayed' (5)
sys.stdout = saveout                                    (6)
fsock.close()                                           (7)
```

(1) Questo stamperà nella "finestra interattiva" dell'IDE (o del terminale, se lo script gira dalla linea di comando).

(2) Salvate sempre l'`stdout` prima di redirigerlo, così da poter reimpostarlo alla normalità più tardi.

(3) Aprite un nuovo file in scrittura.

(4) Redirigete tutto l'ulteriore output al nuovo file appena aperto.

(5) Questo sarà "stampato" solo nel file di log; non sarà visibile nella finestra dell'IDE o sullo schermo.

(6) Reimpostate `stdout` come si trovava prima che lo modificassimo.

(7) Chiudete il file di log.

Redirigere `stderr` funziona esattamente allo stesso modo, usando `sys.stderr` invece di `sys.stdout`.

Esempio 6.34. Redirigere le informazioni di errore

```
[f8dy@oliver kgp]$ python stderr.py
[f8dy@oliver kgp]$ cat error.log
Traceback (most recent line last):
  File "stderr.py", line 5, in ?
    raise Exception, 'this error will be logged'
Exception: this error will be logged
```

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
#stderr.py
import sys

fsock = open('error.log', 'w')               (1)
sys.stderr = fsock                           (2)
raise Exception, 'this error will be logged' (3) (4)
```

(1) Aprite il file di log dove volevamo immagazzinare le informazioni di debug.

(2) Redirigete lo standar error assegnando l'oggetto file del nostro file di log appena aperto ad `stderr`.

(3) Sollevate un'eccezione. Notate dall'output dello schermo che tutto ciò *non* stampa nulla sullo schermo. Tutte le usuali informazioni di traceback sono state scritte in `error.log`.

(4) Notate anche che non stiamo chiudendo esplicitamente il nostro file di log, e nemmeno settando `stderr` al suo valore originale. Questo va bene, dato che una volta che il programmma è crashato (a causa della nostra eccezione), Python cancellerà e chiuderà il file per noi, non fa alcuna differenza che `stderr` non venga mai reimpostato, visto che, come ho detto, il programma crasha e Python termina. Reimpostare l'originale è molto importante per `stdout` se prevedete di fare altre modifiche con lo stesso script più tardi.

Standard input, per altri versi, è un oggetto file di sola lettura, rappresenta i dati che scorrono nel programma da qualche altro programma precedente. Questo non avrà molto senso per gli utenti del classico Mac OS, o anche agli utenti Windows che non hanno mai avuto a che fare con la linea di comando dell'MS−DOS. Il suo funzionamento consiste nel costruire una catena di comandi in una singola linea, così che l'output di un programma diventi l'input del prossimo programma della catena. Il primo programma si limita a scrivere sullo standard output (senza fare alcuna redirezione, solo normali istruzioni `print` o altro), il secondo programma legge dallo standard input ed il sistema operativo si prende cura di connettere l'output di un programma con l'input del seguente.

Esempio 6.35. Concatenare comandi

```
[f8dy@oliver kgp]$ python kgp.py −g binary.xml         (1)
01100111
[f8dy@oliver kgp]$ cat binary.xml                      (2)
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "−//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
[f8dy@oliver kgp]$ cat binary.xml | python kgp.py −g − (3) (4)
10110001
```

(1) Come abbiamo visto nella sezione Immergersi, in questo capitolo, questo codice stamperà una stringa di otto bit casuali, `0` o `1`.

(2) Questo codice stampa semplicemente l'intero contenuto di `binary.xml`. Gli utenti Windows dovrebbero usare `type` invece di `cat`.

(3) Questo codice stampa il contenuto di `binary.xml`, ma il carattere "`|`", chiamato carattere "pipe", significa che il contenuto non verrà stampato sullo schermo. Al contrario diventerà input per il comando seguente, che in questo caso rappresenta il nostro script Python.

(4) Invece di specificare un modulo (come `binary.xml`), specifichiamo "`−`", che farà caricare al nostro script la grammatica dallo standard input invece che da un file sul disco (di più su come accade nel prossimo esempio). Così l'effetto è lo stesso della prima sintassi, dove specificammo il nome del file grammar direttamente, ma pensate alle possibile espansioni. Invece di fare semplicemente `cat binary.xml`, potremmo lanciare uno script che generi dinamicamente la grammatica, per poi redirigerlo nel nostro script. Può provenire da ovunque: un database, da meta−script generatori di grammatica o altro ancora. Il punto è che non dobbiamo cambiare il nostro script `kgp.py` completamente, per incorporare ognuna di queste funzionalità. Tutto quello che dobbiamo fare è essere capaci di prendere i file di grammatica dallo standard input, e separare tutta l'altra logica in un altro programma.

Perciò come fa il nostro script "know" per leggere dallo standard input quando il file di grammatica è "−"? Non è magia, è solo codice.

Esempio 6.36. Leggere dallo standard input in `kgp.py`

```
def openAnything(source):
    if source == "-":    (1)
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:

[... snip ...]
```

(1) Questa è la funzione `openAnything` da `toolbox.py`, che abbiamo esaminato nella sezione Astrarre le sorgenti di ingresso. Tutto quello che abbiamo fatto è stato aggiungere tre linee di codice all'inizio della funzione per controllare se il sorgente è "-"; se si, ritorniamo `sys.stdin`. Veramente, avviene proprio questo! Ricordate, `stdin` è un oggetto simile ad un file con un metodo `read`, così il resto del nostro codice (in `kgp.py`, dove abbiamo chiamato `openAnything`) non viene modificato di un bit.

6.9. Memorizzare i nodi e ricercarli

`kgp.py` impiega alcuni trucchi che potrebbero esservi utili nella elaborazione XML. Il primo trucco consiste nel trarre vantaggio dalla consistente struttura dei documenti in input per costruire una cache di nodi.

Un file grammar definisce una serie di elementi `ref`. Ogni `ref` contiene uno o più elementi `p`, che possono contenere molte cose differenti, inclusi gli `xref`s. Ogni volta che incontriamo un `xref`, cerchiamo un elemento `ref` corrispondente con lo stesso attributo `id`, scegliamo uno degli elementi `ref` figli e lo analizziamo. Vedremo come questa scelta venga fatta nella prossima sezione.

Ecco come costruiamo il nostro file grammar: definiamo elementi `ref` per il più piccolo pezzo, poi definiamo elementi `ref` che "includono" il primi elementi `ref` usando `xref` e così via. Quindi analizziamo il più "largo" riferimento e seguiamo ogni `xref` ed eventualmente stampiamo il testo reale. Il testo che stampiamo dipende dalla decisione (casuale) che facciamo ogni volta che riempiamo un `xref`, perciò l'output è differente ogni volta.

Tutto ciò è molto flessibile, ma c'è una contropartita: le performance. Quando troviamo un `xref` dobbiamo trovare il corrispondente elemento `ref`, ed incappiamo in un problema. `xref` ha un attributo `id`, noi vogliamo trovare l'elemento `ref` che ha lo stesso attributo `id`, ma non esiste un modo facile per farlo. La via lenta consiste nel considerare l'intera lista degli elementi `ref` ogni volta, e analizzare l'attributo `id` di ognuno di essi manualmente attraverso un ciclo. La via veloce consiste nel fare ciò una sola volta e costruire una cache, nella forma di un dizionario.

Esempio 6.37. `loadGrammar`

```
    def loadGrammar(self, grammar):
        self.grammar = self._load(grammar)
        self.refs = {}                                          (1)
        for ref in self.grammar.getElementsByTagName("ref"): (2)
            self.refs[ref.attributes["id"].value] = ref      (3) (4)
```

(1) Iniziate a creare un dizionario vuoto, `self.refs`.

(2) Come abbiamo visto nella sezione Ricercare elementi, `getElementsByTagName` ritorna una lista di tutti gli elementi di un particolare nome. Possiamo facilmente ottenere una lista di tutti gli elementi `ref` e poi semplicemente eseguire un ciclo in quella lista.

(3)

Come abbiamo visto nella sezione Accedere agli attributi di un elemento, possiamo accedere ad attributi individuali di un elemento per nome, usando la sintassi standard dei dizionari. Così le chiavi del nostro dizionario `self.refs` saranno i valori dell'attributo `id` di ogni elemento `ref`.

(4) I valori del nostro dizionario `self.refs` saranno gli stessi elementi `ref`. Come abbiamo visto nell'analisi effettuata nella sezione Analizzare XML, ogni elemento, ogni nodo, ogni commento, ogni pezzo di testo in un documento XML analizzato è un oggetto.

Una volta che abbiamo costruito questa cache, ogni volta che passiamo su un `xref` e cerchiamo di trovare l'elemento `ref` con lo stesso attributo `id`, possiamo semplicemente leggerlo in `self.refs`.

Esempio 6.38. Usare gli elementi `ref` in memoria

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Esploreremo la funzione `randomChildElement` nella prossima sezione.

6.10. Trovare i figli diretti di un nodo

Un'altra utile tecnica quando analiziamo documenti XML consiste nel trovare tutti gli elementi figli diretti di un particolare elemento. Per esempio, nel nostro file grammar, un elemento `ref` può avere alcuni elementi `p`, ognuno dei quali può contenere molte cose, inclusi altri elementi `p`. Vogliamo trovare gli elementi `p` che sono figli del `ref`, non gli elementi `p` che sono figli degli altri elementi `p`.

Potreste pensare che potremmo semplicemente usare `getElementsByTagName` per questo, ma non possiamo. `getElementsByTagName` ricerca ricorsivamente e ritorna una singola lista per tutti gli elementi che trova. Dato che gli elementi `p` possono contenere altri elementi `p`, non possiamo usare `getElementsByTagName`, visto che ritornerebbe elementi `p` annidati che non vogliamo. Per trovare unicamente elementi figli diretti, dobbiamo farlo da soli.

Esempio 6.39. Trovare elementi figli diretti

```
    def randomChildElement(self, node):
        choices = [e for e in node.childNodes
                   if e.nodeType == e.ELEMENT_NODE] (1) (2) (3)
        chosen = random.choice(choices)             (4)
        return chosen
```

(1) Come abbiamo visto nell'Esempio 6.9, Ottenere i nodi figli , l'attributo `childNodes` ritorna una lista di tutti i nodi figli di un elemento.

(2) Comunque, abbiamo visto nell'Esempio 6.11, I nodi figli possono essere di testo che la lista ritornata da `childNodes` contiene tutti i differenti tipi di nodi, inclusi i nodi di testo. Non è quello che stiamo cercando, noi vogliamo unicamente i figli che sono elementi.

(3) Ogni nodo ha un attributo `nodeType`, che può essere `ELEMENT_NODE`, `TEXT_NODE`, `COMMENT_NODE`, od ogni numero di altri valori. La lista completa di possibili valori è nel file `__init__.py` del package `xml.dom`. Leggete la sezione Package di questo capitolo per ulteriori informazioni sui package. Ma noi siamo interessati ai nodi che sono elementi, così possiamo filtrare la lista per includere unicamente quei nodi il cui `nodeType` è `ELEMENT_NODE`.

(4)

Una volta che abbiamo una lista di elementi reali, scegliendone uno casuale è facile. Python contiene un modulo chiamato `random` che include alcune utili funzioni. La funzione `random.choice` prende una lista di qualsiasi numero di oggetti e ritorna un oggetto random. In questo caso la lista contiene elementi `p`, così `chosen` è adesso un elemento `p` selezionato casualmente dai figli dell'elemento `ref` che abbiato fornito.

6.11. Create gestori separati per tipo di nodo

Il terzo suggerimento utile per processare gli XML comporta la separazione del vostro codice in funzioni logiche, basate sui tipi di nodi e nomi di elementi. I documenti XML analizzati sono fatti di vari tipi di nodi, di cui ognuno rappresenta un oggetto Python. Il livello base del documento stesso è rappresentato da un oggetto `Document`. `Document` contiene uno o più oggetti `Element` (per i tags XML reali), ognuno dei quali può contenere altri oggetti `Element`, oggetti `Text` (per parti di testo), od oggetti `Comment` (per commenti incorporati). Python rende semplice scrivere uno smistatore per separare la logica per ciascun tipo di nodo.

Esempio 6.40. Nomi di classi di oggetti XML analizzati

```
>>> from xml.dom import minidom
>>> xmldoc = minidom.parse('kant.xml') (1)
>>> xmldoc
<xml.dom.minidom.Document instance at 0x01359DE8>
>>> xmldoc.__class__                    (2)
<class xml.dom.minidom.Document at 0x01105D40>
>>> xmldoc.__class__.__name__           (3)
'Document'
```

(1) Pensate per un momento che `kant.xml` si trovi nella directory corrente.

(2) Come abbiamo visto nella sezione Package, l'oggetto ritornato dall'analisi di un documento XML è un oggetto `Document`, come definito in `minidom.py` del package `xml.dom`. Come abbiamo visto nella sezione Istanziare classi, `__class__` è un attributo built−in di ogni oggetto Python.

(3) Inoltre, `__name__` è un attributo built−in di ogni classe Python, ed è una stringa. Questa stringa non è misteriosa; è la stessa del nome della classe che inserite quando definite una classe da voi. (Ritornate eventualmente alla sezione Definire classi.)

Bene, così adesso possiamo ottenere il nome della classe di ogni particolare nodo XML (dato che ogni nodo XML viene rappresentato come un oggetto Python). Come possiamo utilizzare ciò a nostro vantaggio per separare la logica dell'analisi di ogni tipo di nodo? La risposta è `getattr`, che abbiamo visto precedentemente nella sezione Ottenere riferimenti agli oggetti usando `getattr`.

Esempio 6.41. `analizzare`, un generico smistatore di nodi XML

```
    def parse(self, node):
        parseMethod = getattr(self, "parse_%s" % node.__class__.__name__) (1) (2)
        parseMethod(node) (3)
```

(1) Prima cosa, notate che stiamo costruendo una larga stringa basata sul nome della classe del nodo che abbiamo passato (nell'argomento `node`). Così, se passiamo un nodo `Document`, stiamo costruendo la stringa `'parse_Document'`, e così via.

(2) Adesso possiamo trattare la stringa come un nome di funzione ed ottenere un riferimento alla funzione stessa usando `getattr`.

(3) Infine, possiamo chiamare quella funzione e passare il nodo stesso come un argomento. Il prossimo esempio mostra le definizioni di ognuna di queste funzioni.

Esempio 6.42. Funzioni chiamate dall'`analizzatore` di smistamento

```
    def parse_Document(self, node): (1)
        self.parse(node.documentElement)

    def parse_Text(self, node):    (2)
        text = node.data
        if self.capitalizeNextWord:
            self.pieces.append(text[0].upper())
            self.pieces.append(text[1:])
            self.capitalizeNextWord = 0
        else:
            self.pieces.append(text)

    def parse_Comment(self, node): (3)
        pass

    def parse_Element(self, node): (4)
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

(1) `parse_Document` è chiamato solo una volta, dato che c'è solo un nodo `Document` in un documento XML e solo un oggetto `Document` nella rappresentazione XML analizzata. Semplicemente il programma gira intorno ed analizza l'elemento radice del file grammar.

(2) `parse_Text` è chiamato sui nodi che rappresentano parti di testo. La funzione stessa fa alcune speciali operazioni per gestire automaticamente la conversione in maiuscolo delle prime lettere di una frase, ed aggiunge il testo rappresentato ad una lista.

(3) `parse_Comment` è solo un `pass`, dato che non teniamo conto dei commenti incorporati nei nostri file grammar. Notate, comunque, che necessitiamo ancora di definire la funzione e non fargli fare esplicitamente nulla. Se la funzione non esistesse, la nostra generica funzione `parse` fallirebbe nel momento in cui giungerebbe su un commento, perché cercherebbe di trovare la funzione inesistente `parse_Comment`. Definire una funzione separata per ogni tipo di nodo, anche quelli che non usiamo, permette alla funzione generica `parse` di essere semplice e muta.

(4) L'elemento `parse_Element` è realmente esso stesso uno smistatore, basato sul nome dell'etichetta dell'elemento. L'idea di base è la stessa: prendere ciò che distingue gli elementi uno dall'altro (i nomi delle loro etichette) e smistare in funzioni separate per ognuno di loro. Costruire una stringa come `'do_xref'` (per una etichetta `<xref>`), trovare una funzione di quel nome e chiamarla. E così via per ognuno degli altri nomi delle etichette che potrebbero essere trovate nel corso dell'analisi di un file grammar (etichetta `<p>`, etichetta `<choice>`).

In questo esempio, le funzioni di smistamento `parse` e `parse_Element` trovano semplicemente altri metodi nella stessa classe. Se il vostro processo è molto complesso (o avete nomi di etichette differenti), potreste spezzare il vostro codice in moduli separati ed usare l'importazione dinamica per importare ogni modulo e chiamare qualsiasi funzione di cui abbiate bisogno. L'import dinamico sarà discusso nel capitolo sulla Programmazione orientata ai dati.

6.12. Gestire gli argomenti da riga di comando

Python supporta pienamente la creazione di programmi che possono essere eseguiti da riga di comando, completi di argomenti, sia con flag nel formato breve che nel formato esteso per specificare le varie opzioni. Nessuno di questi è specifico di XML, ma questo script fa un buon uso dell'interpretazione da riga di comando, dunque pare un buon momento per menzionarla.

È difficile parlare dell'elaborazione da riga di comando senza prima capire come gli argomenti da riga di comando vengono presentati al programma Python, dunque scriviamo un semplice programma per vederli.

Esempio 6.43. Introduzione a `sys.argv`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
#argecho.py
import sys

for arg in sys.argv: (1)
    print arg
```

(1) Ogni argomento da riga di comando passato al programma sarà in `sys.argv`, che è semplicemente una lista. Qui stiamo stampando ogni argomento su una riga diversa.

Esempio 6.44. Il contenuto di `sys.argv`

```
[f8dy@oliver py]$ python argecho.py             (1)
argecho.py
[f8dy@oliver py]$ python argecho.py abc def     (2)
argecho.py
abc
def
[f8dy@oliver py]$ python argecho.py --help      (3)
argecho.py
--help
[f8dy@oliver py]$ python argecho.py -m kant.xml (4)
argecho.py
-m
kant.xml
```

(1) La prima cosa da sapere su `sys.argv` è che contiene il nome dello script che stiamo chiamando. Utilizzeremo questa conoscenza a nostro vantaggio più tardi, nel capitolo Programmazione orientata ai dati. Non preoccupatevi di questo per ora.

(2) Gli argomenti da riga di comando sono separati da spazi ed ognuno di essi viene rappresentato da un elemento nella lista `sys.argv`.

(3) Anche i flag da riga di comando, come `--help`, vengono rappresentati come elementi della lista `sys.argv`.

(4) Per rendere le cose ancor più interessanti, alcuni flag da riga di comando possono a loro volta prendere degli argomenti. Per esempio, qui abbiamo il flag (`-m`) che prende un argomento (`kant.xml`). Entrambi, sia il flag che il suo argomento, sono semplicemente degli elementi sequenziali nella lista `sys.argv`. Non viene fatto alcun tentativo di associare l'uno con l'altro; tutto quello che ottenete è una lista.

Come possiamo vedere, tutte le informazioni vengono passate da riga di comando ma non sembra che sia poi tutto così semplice da usare. Per programmi semplici che prendono un solo argomento e non hanno flag, potete semplicemente usare `sys.argv[1]` per leggere l'argomento. Non c'è vergogna in questo, io lo faccio sempre. Per programmi più complessi, avete bisogno del modulo `getopt`.

Esempio 6.45. Introduzione a `getopt`

```
def main(argv):
    grammar = "kant.xml"                (1)
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="]) (2)
    except getopt.GetoptError:          (3)
        usage()                         (4)
        sys.exit(2)

...
```

```
if __name__ == "__main__":
    main(sys.argv[1:])
```

(1) Per prima cosa, date un'occhiata alla fine dell'esempio e notate che stiamo chiamando la funzione `main` con `sys.argv[1:]`. Ricordate, `sys.argv[0]` è il nome dello script che stiamo eseguendo; non ne siamo interessati per l'elaborazione da riga di comando, così lo possiamo togliere e passare il resto della lista.

(2) È qui che avvengono le elaborazioni interessanti. La funzione `getopt` del modulo `getopt` prende tre parametri: la lista degli argomenti (che abbiamo ottenuto da `sys.argv[1:]`), una stringa contenente tutti i possibili flag composti da un solo carattere che questo programma può accettare, ed una lista di comandi più lunghi che sono equivalenti alla loro versione da un solo carattere. È abbastanza intricato a prima vista e sarà spiegato in maggior dettaglio in seguito.

(3) Se qualcosa dovesse andare male provando ad analizzare questi flag da riga di comando, `getopt` solleverà un'eccezione, che noi cattureremo. Abbiamo detto a `getopt` tutti i flag che conosciamo, dunque questo significa che probabilmente l'utente ha passato qualche flag che non siamo in grado di interpretare.

(4) Come di consueto nel mondo UNIX, quando al nostro script vengono passati dei flag che non può capire, stampiamo un riassunto sull'utilizzo corretto del programma ed usciamo. Notate che non ho mostrato la funzione `usage` qui. Dovremmo scriverla da qualche parte e farle stampare un riassunto appropriato; non è automatico.

Cosa sono dunque tutti questi parametri che passiamo alla funzione `getopt`? Beh, il primo è semplicemente una lista grezza dei flag e degli argomenti da riga di comando (escluso il primo elemento, il nome dello script, che abbiamo rimosso prima di chiamare la funzione `main`). Il secondo è la lista dei flag brevi che il nostro script accetta.

`"hg:d"`

`-h`
: print usage summary

`-g ...`
: use specified grammar file or URL

`-d`
: show debugging information while parsing

Il primo ed il terzo flag sono semplicemente dei flag autonomi; li potete specificare oppure no, fanno comunque qualcosa (stampa un aiuto) o cambiano uno stato (abilita il debugging). Comunque, il secondo flag (`-g`) *deve* essere seguito da un argomento, che è il nome del file di grammatica da cui leggere. Infatti può essere il nome di un file o un indirizzo web, e non sappiamo ancora quale (lo vedremo dopo), ma sappiamo che deve essere *qualcosa*. Dunque lo diciamo a `getopt` inserendo i due punti dopo `g` nel secondo parametro della funzione `getopt`.

Per complicare ulteriormente le cose, il nostro script accetta sia flag brevi (come `-h`) o lunghi (come --help), e vogliamo che facciano la stessa cosa. Questo è lo scopo del terzo parametro di `getopt`, specificare una lista di flag lunghi che corrispondono a quelli brevi specificati nel secondo parametro.

`["help", "grammar="]`

`--help`
: print usage summary

`--grammar ...`
: use specified grammar file or URL

Ci sono tre cose da notare qui:

1. Tutti i flag lunghi sono preceduti da due trattini sulla riga di comando, ma noi non li includiamo nella chiamata a `getopt`. Sono compresi automaticamente.
2. Il flag `--grammar` deve sempre essere seguito da un argomento addizionale, proprio come il flag `-g`. È evidenziato dal segno uguale, `"grammar="`.
3. La lista di flag lunghi è più corta della lista di flag corti, perché il flag `-d` non ha una corrispondente versione lunga. Questo va bene, solamente `-d` imposterà il debugging. Ma l'ordine di flag lunghi e corti deve essere il medesimo, dunque dovete specificare per primi i flag brevi che *hanno* un corrispondente flag lungo, seguiti dal resto dei flag brevi.

Ancora confusi? Diamo un'occhiata al codice e vediamo se ora ha più senso.

Esempio 6.46. Gestire gli argomenti da riga di comando in `kgp.py`

```
def main(argv):                          (1)
    grammar = "kant.xml"
    try:
    except getopt.GetoptError:
        usage()
        sys.exit(2)
    for opt, arg in opts:                (2)
        if opt in ("-h", "--help"):      (3)
            usage()
            sys.exit()
        elif opt == '-d':                (4)
            global _debug
            _debug = 1
        elif opt in ("-g", "--grammar"): (5)
            grammar = arg

    source = "".join(args)               (6)

    k = KantGenerator(grammar, source)
    print k.output()
```

(1) La variabile `grammar` terrà traccia del file di grammatica che stiamo usando. La inizializziamo qui nel caso non sia specificata da riga di comando (utilizzando il flag `-g` o il flag `--grammar`).

(2) La variabile `opts` che otteniamo da `getopt` contiene una lista di tuple: flag ed argomento. Se il flag non prende un argomento, allora `arg` sarà semplicemente `None`. Questo semplifica la creazione di un ciclo che percorre i flag.

(3) `getopt` valida che i flag da riga di comando siano accettabili, ma non fa alcun genere di conversione tra flag lunghi e brevi. Se specificate il flag `-h`, `opt` conterrà `"-h"`; se specificate il flag `--help`, `opt` conterrà `"--help"`. Dunque dobbiamo controllarli entrambi.

(4) Ricordate, il flag `-d` non ha un corrispondente flag lungo, quindi dobbiamo solamente controllarne la forma breve. Se lo troviamo, impostiamo una variabile globale a cui ci riferiremo più avanti per stampare delle informazioni di debug. (L'ho usato durante lo sviluppo dello script. Pensavate che tutti questi esempi fossero funzionati al primo colpo?)

(5) Se troviamo un file di grammatica, sia con il flag `-g` che con il flag `--grammar`, salviamo l'argomento che lo segue (memorizzato in `arg`) nella nostra variabile `grammar`, sovrascrivendo il valore predefinito che abbiamo inizializzato all'inizio della nostra funzione `main`.

(6) Ecco fatto. Abbiamo scorso tutti i flag da riga di comando e ci siamo occupati di ognuno di essi. Significa che tutto ciò che è rimasto devono essere argomenti da riga di comando. Questi tornano indietro dalla funzione `getopt` nella variabile `args`. In questo caso, li stiamo trattando come materiale da passare al nostro parser. Se non vi sono argomenti specificati nella riga di comando, `args`

sarà una lista vuota, e `source` sarà trattato come una stringa vuota.

6.13. Mettere tutto assieme

Abbiamo coperto una buona distanza. Facciamo un passo indietro e vediamo come si mettono assieme tutti i pezzi.

Per cominciare, questo è uno script che prende i suoi argomenti da riga di comando, usando il modulo `getopt`.

```
def main(argv):
...
    try:
        opts, args = getopt.getopt(argv, "hg:d", ["help", "grammar="])
    except getopt.GetoptError:
...
    for opt, arg in opts:
...
```

Creiamo una nuova istanza della classe `KantGenerator` e passiamole il file di grammatica ed il sorgente, che può, o meno, essere stato specificato nella riga di comando.

```
    k = KantGenerator(grammar, source)
```

L'istanza di `KantGenerator` carica automaticamente la grammatica, che è un file XML. Utilizziamo la nostra funzione personalizzata `openAnything` per aprire il file (che potrebbe essere in un file locale o in un web server remoto), quindi usiamo le funzioni di analisi built−in del modulo `minidom` per inserire l'XML in un albero di oggetti Python.

```
    def _load(self, source):
        sock = toolbox.openAnything(source)
        xmldoc = minidom.parse(sock).documentElement
        sock.close()
```

Già che ci siamo, avvantaggiamoci della nostra conoscenza della struttura di un documento XML per impostare una piccola cache di riferimenti, che sono semplicemente gli elementi nel documento XML.

```
    def loadGrammar(self, grammar):
        for ref in self.grammar.getElementsByTagName("ref"):
            self.refs[ref.attributes["id"].value] = ref
```

Se avessimo specificato del sorgente nella riga di comando, avremmo usato questo; in caso contrario dovremmo cercare nella grammatica il riferimento al "top−level" (che non è puntato da nient'altro) ed usarlo come punto di partenza.

```
    def getDefaultSource(self):
        xrefs = {}
        for xref in self.grammar.getElementsByTagName("xref"):
            xrefs[xref.attributes["id"].value] = 1
        xrefs = xrefs.keys()
        standaloneXrefs = [e for e in self.refs.keys() if e not in xrefs]
        return '<xref id="%s"/>' % random.choice(standaloneXrefs)
```

Ora attraversiamo il sorgente. Anche il sorgente è in XML, e lo analizziamo un nodo per volta. Per mantenere il nostro codice separato e maggiormente manutenibile, usiamo diversi gestori per ogni tipo di nodo.

```
    def parse_Element(self, node):
        handlerMethod = getattr(self, "do_%s" % node.tagName)
        handlerMethod(node)
```

Percorriamo tutta la grammatica, analizzando tutti i figli di ogni elemento `p`,

```
    def do_p(self, node):
...
        if doit:
            for child in node.childNodes: self.parse(child)
```

sostituendo gli elementi `choice` con un figlio a caso,

```
    def do_choice(self, node):
        self.parse(self.randomChildElement(node))
```

e sostituendo gli elementi `xref` con un figlio a caso del corrispondente elemento `ref`, che abbiamo precedentemente messo nella cache.

```
    def do_xref(self, node):
        id = node.attributes["id"].value
        self.parse(self.randomChildElement(self.refs[id]))
```

Nel caso, trasformiamo il tutto in testo semplice,

```
    def parse_Text(self, node):
        text = node.data
...
            self.pieces.append(text)
```

che poi stampiamo.

```
def main(argv):
...
    k = KantGenerator(grammar, source)
    print k.output()
```

6.14. Sommario

Python viene rilasciato con delle potenti librerie per l'analisi e la manipolazione dei documenti XML. `minidom` prende un file XML e lo analizza in un albero di oggetti Python, permettendo l'accesso casuale ad elementi arbitrari. Inoltre, questo capitolo mostra come Python può essere usato per creare dei "veri" script da riga di comando autonomi, completi di flag da riga di comando, argomenti da riga di comando, gestione degli errori ed anche la possibilità di acquisire come input il risultato di un programma precedente.

Prima di passare al prossimo capitolo, dovreste trovarvi a vostro agio con i seguenti argomenti:

- Analisi di documenti XML usando `minidom`, ricerca nell'albero generato, accesso agli attributi di elementi arbitrari ed agli elementi figli
- Organizzazione di librerie complesse in package
- Convertire stringhe unicode nelle diverse codifiche dei caratteri
- Concatenare programmi tramite gli standard input ed output
- Definire flag da riga di comando e validarli con `getopt`

[11] Questo, tristemente, *è ancora* una semplificazione. Unicode ora è stato esteso per gestire testi in Cinese antico, in Coreano ed in Giapponese, che hanno talmente tanti caratteri diversi che il sistema unicode a 2 byte non li potrebbe rappresentare tutti. Ma Python attualmente non supporta tutto questo, e non so se ci sia qualche progetto che permetta

di estenderlo in tale senso. Spiacente, avete raggiunto i limiti della mia conoscenza.

[12] Python ha il supporto per unicode dalla versione 1.6, ma la versione 1.6 era un obbligo contrattuale di cui nessuno ama parlare, un figliastro da dimenticare. Anche la documentazione ufficiale di Python dichiara che unicode è "nuovo nella versione 2.0". È una bugia ma, come le bugie dei presidenti che dicono di aver fumato ma che non gli è piaciuto, scegliamo di crederci perché ricordiamo la nostra giovinezza mal spesa in maniera fin troppo vivida.

Capitolo 7. Test delle unità di codice

7.1. Immergersi

Nei capitoli precedenti, ci si è "tuffati" immediatamente nell'analisi del codice, cercando poi di capirlo il più velocemente possibile. Ora che avete immagazzinato un po' di Python, faremo un passo indietro e analizzeremo ciò che si fa *prima* di scrivere il codice.

In questo capitolo scriveremo una serie di funzioni utili per convertire da numeri arabi a numeri romani e viceversa. Molto probabilmente avete già visto dei numeri romani, anche se spesso non li avete riconosciuti. Potete averli visti nella nota di copyright di vecchi film e spettacoli televisivi ("Copyright `MCMXLVI`" invece di "Copyright `1946`"), oppure nelle targhe di dedica di biblioteche e università ("fondata `MDCCCLXXXVIII`" invece di "fondata `1888`"). Potete averli visti in sommari e riferimenti bibliografici. È un sistema di rappresentare numeri che effettivamente risale all'antico Impero Romano (da cui il nome).

Nei numeri romani, ci sono sette caratteri che sono ripetuti e combinati in vari modi per rappresentare i numeri:

1. `I` = `1`
2. `V` = `5`
3. `X` = `10`
4. `L` = `50`
5. `C` = `100`
6. `D` = `500`
7. `M` = `1000`

Ecco alcune regole generali per costruire numeri romani:

1. Il valore del numero è la somma dei valori dei caratteri. `I` è `1`, `II` è `2`, e `III` è `3`. `VI` è `6` (letteralmente, "`5` e `1`"), `VII` è `7` e `VIII` è `8`.
2. I "caratteri di decina" (`I`, `X`, `C`, e `M`) possono esere ripetuti fino a tre volte. Alla quarta, si deve sottrarre uno dal piu vicino "carattere di quintina" (`V`, `L`, `D`). Non si può rappresentare `4` come `IIII`;, lo si deve rappresentare con `IV` ("`1` in meno di `5`"). `40` è scritto come `XL`, `41` come `XLI`, `42` come `XLII`, `43` come `XLIII` ed infine `44` come `XLIV` ("`10` in meno di `50`, più uno in meno di `5`").
3. Similmente, arrivati al `9`, si deve sottrarre dal carattere di decina immediatamente superiore: `8` è `VIII`, ma `9` è `IX` ("uno in meno di dieci"), non `VIIII` (giacché il carattere `I` non può essere ripetuto quattro volte). `90` è `XC`, `900` è `CM`.
4. I caratteri di quintina non possono essere ripetuti. `10` è sempre rappresentato come `X`, mai come `VV`. `100` è sempre `C`, mai `LL`.
5. I numeri romani sono sempre scritti dal più grande al più piccolo e letti da sinistra a destra. per cui l'ordine dei caratteri è molto importante. `DC` è `600`; `CD` è un numero completamente diverso (`400`, "`100` meno di `500`"). `CI` è `101`; `IC` non è neppure un numero romano valido (perché non si può sottrarre `1` direttamente da `100`; `99` si deve scrivere `XCIX`, "`10` in meno di `100` e poi `1` in meno di `10`").

Queste regole conducono ad alcune interessanti osservazioni:

1. C'è solo un modo corretto di rappresentare una quantità come numero romano.
2. Il viceversa è anche vero: se una sequenza di caratteri è un valido numero romano, essa rappresenta una quantità univoca (*i.e.* può essere letto in una sola maniera)
3. C'è un numero finito di numeri arabi che possono essere espressi come numeri romani, specificatamente da `1` a `3999`. I romani avevano diversi modi di esprimere quantità più grandi, per esempio mettendo una barra su un numero per indicare che la sua quantità doveva essere moltiplicata per `1000`, ma non tratteremo questi

casi. Per lo scopo di questo capitolo, i numerali romani vanno da `1` a `3999`.
4. Non c'è modo di rappresentare lo `0` in numeri romani (Incredibilmente, gli antichi romani non avevano il concetto di `0` come numero. I numeri servivano a contare quello che si aveva; come si fa a contare quello che non si ha?).
5. Non c'è modo di rappresentare quantità negative in numeri romani.
6. Non c'è modo di rappresentare decimali o frazioni con i numeri romani.

Dato tutto questo, cosa si può chiedere ad una libreria di funzioni per convertire da numeri romani a numeri arabi e viceversa?

`roman.py` requisiti

1. La funzione `toRoman` dovrebbe restituire la rappresentazione in numeri romani di qualsiasi numero arabo da `1` a `3999`.
2. La funzione `toRoman` dovrebbe andare in errore per un intero fuori dall'intervallo da `1` a `3999`.
3. La funzione `toRoman` dovrebbe andare in errore con un numero non intero.
4. La funzione `fromRoman` dovrebbe accettare in input un valido numero romano e restituire il numero arabo corrispondente.
5. La funzione `fromRoman` dovrebbe andare in errore con un numero romano non valido.
6. Se si prende un numero arabo, lo si converte in numero romano e poi lo si riconverte in numero romano, si dovrebbe riavere il numero di partenza. Vale a dire `fromRoman(toRoman(n)) == n` per tutti i numeri da `1` a `3999`.
7. la funzione `toRoman` dovrebbe sempre restituire un numero romano formato da lettere maiuscole.
8. La funzione `fromRoman` dovrebbe accettare solamente numeri romani (*i.e.* dovrebbe andare in errore con un input espresso a lettere minuscole).

Ulteriori letture

- Questo sito fornisce ulteriori informazioni sui numeri romani, incluso un affascinante resoconto su come i Romani ed altre civiltà li usavano nella vita reale (breve resoconto: in modo approssimativo ed inconsistente).

7.2. Introduzione al modulo `romantest.py`

Ora che abbiamo completamente definito il comportamento che ci aspettiamo dalle nostre funzioni di conversione, faremo qualcosa di inaspettato: scriveremo un modulo di test che faccia eseguire a queste funzioni il proprio codice e verifichi che esse si comportino come vogliamo. Avete letto bene: andremo a scrivere del codice per verificare altro codice che non abbiamo ancora scritto.

Questo si chiama "test delle unità di codice", visto che l'insieme delle due funzioni di conversione può essere verificato come un'unità indipendente, separato da qualsiasi programma di cui esse potranno fare parte in seguito. Python ha una infrastruttura per effettuare il test delle unità di codice, appropriatamente chiamato modulo `unittest`.

> **Nota: Avete unittest?**
> `unittest` è incluso con Python 2.1 e successivi. Gli utilizzatori di Python 2.0 possono scaricarlo da `pyunit.sourceforge.net`.

Validare le unità di codice è una parte importante di una strategia di sviluppo largamente basata sui test. Se decidete di scrivere i test delle unità di codice, fatelo dall'inizio (preferibilmente prima di scrivere il codice da verificare) e manteneteli aggiornati rispetto ai cambi dei requisiti e del codice. Il test delle unità di codice non è un'alternativa per i test funzionali o di sistema, di più alto livello di quelli delle unità, ma è comunque importante in tutte le fasi dello

sviluppo:

- Prima di scrivere codice, obbliga a dettagliare utilmente i requisiti.
- Mentre si scrive codice, impedisce di esagerare nell'implementazione. Quando tutti i test sono eseguiti con successo, la funzione è completa.
- Mentre si riorganizza il codice, assicura che la nuova versione si comporti come la vecchia.
- Durante la manutenzione del codice, aiuta a pararsi il sedere quando qualcuno arriva gridando che la vostra ultima implementazione causa problemi al loro vecchio codice ("Scusi, ma tutti i test delle unità di codice hanno avuto successo quando ho registrato la nuova versione ...")

Questo è l'insieme completo dei test per le nostre funzioni di conversione dei numeri romani, che non abbiamo ancora scritto ma che alla fine saranno nel modulo `roman.py`. Non è immediatamente ovvio come le varie parti siano collegate; nessuna di queste classi o metodi fa riferimento alle altre. Ci sono buone ragioni per questo comportamento, tra breve saranno chiarite.

Esempio 7.1. Il modulo `romantest.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
"""Unit test for roman.py"""

import roman
import unittest

class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
```

```
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'))

    def testToRomanKnownValues(self):
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testDecimal(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
```

```
    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

Ulteriori letture

- La pagina web di PyUnit contiene un'approfondita descrizione di come usare l'infrastruttura del modulo `unittest`, incluse speciali caratteristiche non discusse in questo capitolo.
- La FAQ di PyUnit spiega perché il codice di test è salvato separatamente dal codice da validare.
- Il *Python Library Reference* riassume le caratteristiche del modulo `unittest`.
- ExtremeProgramming.org tratta del perché è oppportuno scrivere i test delle unità di codice.
- Il Portland Pattern Repository contiene una discussione in continuo aggiornamento sui test di unità, inclusa una loro definizione standard, ragioni per cui si dovrebbe cominciare con lo scrivere i test delle unità di codice e molti altri studi approfonditi.

7.3. Verificare i casi di successo

La parte fondamentale della verifica delle unità di codice è costruire i singoli test. Un test risponde ad una singola domanda sul codice che si sta verificando.

Un singolo test dovrebbe essere capace di ...

- ... essere eseguito in modo autonomo, senza input dall'operatore. La verifica delle unità di codice deve essere un processo automatico.
- ... stabilire in modo autonomo se la funzione sotto verifica ha passato o meno il test, senza che l'operatore debba interpretarne il risultato.
- ... essere eseguito in modo isolato, indipendente da altri test (anche se essi verificano la stessa funzione). Ogni test è un'isola.

Detto tutto questo, costruiamo il nostro primo test. Abbiamo il seguente requisito:

1. La funzione `toRoman` dovrebbe restituire la rappresentazione in numeri romani di qualsiasi numero arabo da `1` a `3999`.

Esempio 7.2. `testToRomanKnownValues`

```
class KnownValues(unittest.TestCase):                          (1)
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
```

```
                (3888, 'MMMDCCCLXXXVIII'),
                (3940, 'MMMCMXL'),
                (3999, 'MMMCMXCIX'))                        (2)

    def testToRomanKnownValues(self):                       (3)
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman.toRoman(integer)                 (4) (5)
            self.assertEqual(numeral, result)               (6)
```

(1) Per scrivere un test, occorre per prima cosa specializzare la classe `TestCase` del modulo `unittest`. Questa classe fornisce molti utili metodi che si possono usare nei test per verificare condizioni specifiche.

(2) Questa è una lista di coppie numeri di interi/numeri romani che ho verificato manualmente. Comprende i dieci numeri più bassi, i dieci numeri più alti, tutti i numeri arabi che si traducono in un numero romano con singolo carattere. Il concetto di un test delle unità di codice non è quello di verificare ogni possibile input, ma di verificare un campione rappresentativo.

(3) Ogni test individuale consiste in un metodo dedicato, che non deve accettare parametri né restituire valori. Se il metodo termina senza sollevare eccezioni, il test ha avuto successo; se il metodo solleva un'eccezione, il test è fallito.

(4) Qui viene chiamata la funzione `toRoman` (beh, la funzione non è stata ancora scritta, ma una volta pronta, questa sarà la linea che la chiama). Da notare che ora abbiamo definito l'interfaccia di programma (API) della funzione `toRoman`: deve prendere in input un intero (il numero da convertire) e deve restituire una stringa (la rappresentazione in numeri romani). Se l'interfaccia della funzione risulta diversa da questa, il test è considerato fallito.

(5) Da notare anche che non si cerca di intercettare eventuali eccezioni sollevate dalla chiamata, alla funzione `toRoman`. Questo è intenzionale: `toRoman` non dovrebbe sollevare eccezioni quando è chiamata con input valido, ed i valori di input usati sono validi. Se la funzione `toRoman` solleva un eccezione, il test è considerato fallito.

(6) Assumendo che la funzione `toRoman` sia stata definita correttamente, chiamata correttamente, abbia completato con successo la sua esecuzione ed abbia restituito un valore, l'ultimo passo è controllare se ha restituito il valore *giusto*. Questa è un caso comune, per cui la classe `TestCase` fornisce un metodo, `assertEqual`, per controllare se due valori sono uguali. Se il valore restituito dalla funzione `toRoman` (`result`) non corrisponde al valore noto che ci si aspetta (`numeral`), il metodo `assertEqual` solleva un'eccezione ed il test fallisce. Se i due valori sono uguali, `assertEqual` non fa nulla. Se ogni valore restituito da `toRoman` corrisponde al valore atteso, `assertEqual` non solleva mai eccezioni, per cui `testToRomanKnownValues` alla fine termina normalmente, il che significa che `toRoman` ha superato questo test.

7.4. Verificare i casi di errore

Non è abbastanza verificare che la nostra funzione abbia successo quando gli input sono validi; occorre anche verificare che la funzione vada in errore quando riceve input non validi. E non basta che vada in errore: deve farlo nel modo che ci si aspetta.

Ricordiamoci di due altri requisiti per `toRoman`:

2. La funzione `toRoman` dovrebbe andare in errore con un intero fuori dall'intervallo da `1` a `3999`.
3. La funzione `toRoman` dovrebbe andare in errore con un numero non decimale.

In Python, le funzioni indicano gli errori sollevando eccezioni, ed il modulo `unittest` fornisce metodi per verificare se una funzione solleva una particolare eccezione quando riceve un input non valido.

Esempio 7.3. Verificare la funzione `toRoman` con input non validi

```
class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 4000) (1)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, 0)    (2)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman.OutOfRangeError, roman.toRoman, -1)

    def testDecimal(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman.NotIntegerError, roman.toRoman, 0.5)  (3)
```

(1) La classe `TestCase` del modulo `unittest` fornisce il metodo `assertRaises`, che accetta i seguenti argomenti: l'eccezione attesa, la funzione sotto test e gli argomenti da passare alla funzione. Se la funzione sotto test richiede più di un argomento, passateli nell'ordine giusto al metodo `assertRaises` e questi li passerà a sua volta nello stesso ordine alla funzione da verificare. Fate attenzione a quello che si sta facendo a questo punto: invece di chiamare direttamente `toRoman` e verificare manualmente che sollevi una particolare eccezione (incapsulandola in un blocco `try...except`), il metodo `assertRaises` si incarica al posto nostro di chiamare `toRoman` con il suo argomento (4000) e si assicura che sollevi l'eccezione `roman.OutOfRangeError`. Ho ricordato di recente com'è comodo che tutto in Python sia un oggetto, incluse funzioni ed eccezioni?

(2) Oltre ad effettuare test con numeri troppo grandi, abbiamo bisogno di fare test con numeri troppo piccoli. Ricordiamoci che i numeri romani non possono esprimere lo zero o quantità negative, per cui è opportuno predisporre dei test per questi casi (`testZero` e `testNegative`). In `testZero`, si sta verificando che `toRoman` sollevi un'eccezione `roman.OutOfRangeError` quando è chiamata con `0`; se *non* solleva tale eccezione (sia che restituisca un valore o che sollevi un'eccezione diversa), il test è considerato fallito.

(3) Il requisito #3 specifica che la funzione `toRoman` non può accettare un numero decimale, perciò in questo punto si sta verificando che la funzione `toRoman` sollevi un'eccezione `roman.NotIntegerError` quando chiamata con un parametro decimale (`0.5`). Se la funzione `toRoman` non solleva un eccezione `roman.NotIntegerError`, questo test è considerato fallito.

I prossimi due requisiti sono simili ai primi tre, fatta eccezione per il fatto che si applicano a `fromRoman` invece che a `toRoman`.

4. La funzione `fromRoman` dovrebbe accettare un valido numero romano e restituire il numero arabo corrispondente.
5. La funzione `fromRoman` dovrebbe andare in errore quando è chiamata con un numero romano non valido.

IL requisito #4 è gestito nello stesso modo del requisito #1, iterando attraverso un campione di valori noti ed effettuando una verifica con ciascuno di essi. Il requisito #5 è gestito alla stessa maniera dei requisiti #2 e #3, verificando la funzione con una serie di input non validi e controllando che sollevi le opportune eccezioni.

Esempio 7.4. Verificare `fromRoman` con input non validi

```
class FromRomanBadInput(unittest.TestCase):
```

```
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s) (1)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, s)
```

(1) Non c'è molto da dire su queste linee di codice; lo schema è esattamente lo stesso che è stato usato per verificare `toRoman` con input non validi. Faccio brevemente notare che si è definita una nuova eccezione: `roman.InvalidRomanNumeralError`. Questo fa un totale di tre eccezioni specifiche che bisognerà definire in `roman.py` (contando anche `roman.OutOfRangeError` e `roman.NotIntegerError`). Vedremo come definire queste eccezioni specifiche quando cominceremo effettivamente a scrivere `roman.py`, più avanti in questo capitolo.

7.5. Verificare la consistenza

Spesso vi capiterà di scoprire che un'unità di codice contiene un insieme di funzioni reciproche, di solito in forma di funzioni di conversione, laddove una converte A in B e l'altra converte B in A. In questi casi, è utile creare un "test di consistenza" per essere sicuri che si possa converire A in B e poi riconvertire B in A senza perdere precisione, incorrere in errori di arrotondamento o in qualche altro malfunzionamento.

Si consideri questo requisito:

6. Prendendo un numero arabo, convertendolo in numero romano e poi riconvertendolo in numero arabo, ci si dovrebbe ritrovare con il numero da cui si era partiti. Quindi `fromRoman(toRoman(n)) == n` per tutti i numeri da `1` a `3999`.

Esempio 7.5. Verificare `toRoman` in confronto con `fromRoman`

```
class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 4000):        (1) (2)
            numeral = roman.toRoman(integer)
            result = roman.fromRoman(numeral)
            self.assertEqual(integer, result) (3)
```

(1) Abbiamo già incontrato la funzione `range` in precedenza, ma qui è usata con due argomenti, per cui otteniamo una lista di interi a partire dal primo argomento (`1`) e poi contando successivamente fino al secondo argomento (`4000`), *escludendo quest'ultimo*. Cioe l'intervallo da `1` a `3999`, che è l'insieme dei numeri arabi convertibili in numeri romani.

(2) Vorrei semplicemente citare di passaggio il fatto che `integer` non è una parola chiave di Python. Qui è giusto un nome di variabile come gli altri.

(3) La struttura logica del test è lineare: si prende un numero arabo (`integer`), lo si converte in numero Romano (`numeral`), poi lo si riconverte in numero arabo (`result`) e si controlla che sia uguale al numero di partenza. In caso negativo, `assertEqual` solleverà un'eccezione ed il test sarà immediatamente considerato fallito. Se

tutti i numeri corrispondono, `assertEqual` terminerà sempre in modo normale, l'intero metodo `testSanity` terminerà quindi in modo normale ed il test sarà considerato passato con successo.

Gli ultimi due requisiti sono diversi dagli altri perché entrambi danno l'apparenza di essere tanto arbitrari quanto banali:

7. La funzione `toRoman` deve sempre restituire un numero romano composto di lettere maiuscole.
8. La funzione `fromRoman` dovrebbe accettare in input solo numeri romani composti di lettere maiuscole (*i.e.* dovrebbe andare in errore con un input in lettere minuscole).

In effetti, questi due requisiti sono un tantino arbitrari. Avremmo per esempio potuto decidere che `fromRoman` poteva accettare input composti di lettere minuscole oppure sia maiuscole che minuscole. Tuttavia questi requisiti non sono completamente arbitrari; se `toRoman` restituisce sempre numeri in lettere maiuscole,allora `fromRoman` deve come minimo accettare input in lettere maiuscole, oppure il nostro "controllo di consistenza" (requisito #6) fallirebbe. Il fatto che `fromRoman` accetti *solo* lettere maiuscole è arbitrario, ma come ogni sistemista potrebbe confermare, usare lettere maiuscole o minuscole fa differenza, per cui vale la pena specificarlo sin dall'inizio. E se vale la pena specificarlo, allora vale la pena verificarlo.

Esempio 7.6. Verificare rispetto a maiuscolo/minuscolo

```
class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            self.assertEqual(numeral, numeral.upper()) (1)

    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 4000):
            numeral = roman.toRoman(integer)
            roman.fromRoman(numeral.upper())           (2) (3)
            self.assertRaises(roman.InvalidRomanNumeralError,
                              roman.fromRoman, numeral.lower())
```

(1) La cosa più interessante di questo test è tutto quello che non verifica. Non verifica che il valore restituito da `toRoman` sia giusto o perlomeno consistente; a queste domande rispondono altri test. Questo intero test esiste solo per verificare l'uso delle maiuscole. Si potrebbe essere tentati di combinare questo test con quello di consistenza, visto che entrambi ciclano attraverso l'intero intervallo di valori ed entrambi chiamano `toRoman`.[13] Ma questo violerebbe una delle nostre regole fondamentali: ogni test dovrebbe rispondere ad una sola domanda. Immaginate di combinare questo test con quello di consistenza e che il test fallisca. Dovreste fare ulteriori analisi per scoprire quale parte del test è fallita e così determinare qual'è il problema. Se vi tocca analizzare l'output di un test per capire cosa esso significhi, questo è un sicuro segno che avete disegnato male i vostri test.

(2) Qui c'è da apprendere una lezione simile a quella appena esposta: anche se "sappiamo" che la funzione `toRoman` restituisce sempre un numero romano in lettere maiuscole, convertiamo lo stesso in modo esplicito il risultato di `toRoman` in lettere maiuscole, per verificare che `fromRoman` lo accetti. Perché? Perché il fatto che `toRoman` restituisca sempre lettere maiuscole è un requisito indipendente. Se cambiassimo questo requisito, in modo che, ad esempio, `toRoman` restituisse sempre numeri romani in lettere minuscole, allora il test `testToRomanCase` dovrebbe essere cambiato, ma questo test funzionerebbe ancora. Questa era un'altra delle nostre regole fondamentali: ogni test deve poter funzionare isolato dagli altri. Ogni test è un isola.

(3) Si noti che non assegnamo ad alcunché il valore di ritorno di `fromRoman`. Questo è sintatticamente corretto in Python; se una funzione restituisce un valore ma nessuno lo recepisce, Python semplicemente lo butta via. In questo caso, è quello che vogliamo. Questo test non verifica niente circa il valore di ritorno; esso verifica

semplicemente che `fromRoman` accetti un input in lettere maiuscole senza sollevare eccezioni.

7.6. `roman.py`, fase 1

Ora che i nostri test delle unità di codice sono pronti, è tempo di cominciare a scrivere il codice che stiamo cercando di verificare con i nostri test. Faremo questo in più fasi, in modo che si possa osservare dapprima come tutti i test falliscano, e poi come a poco a poco abbiano successo man mano che riempiamo gli spazi vuoti all'interno del modulo `roman.py`.

Esempio 7.7. Il modulo `roman1.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass               (1)
class OutOfRangeError(RomanError): pass         (2)
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass (3)

def toRoman(n):
    """convert integer to Roman numeral"""
    pass                                        (4)

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

(1) Questo è il modo di definire le proprie eccezioni specifiche in Python. Le eccezioni sono classi, e voi create le vostre specializzando le eccezioni già esistenti. È fortemente raccomandato (ma non richiesto) di specializzare la classe `Exception`, che è la classe base da cui ereditano tutte le eccezioni predefinite. Qui io sto definendo `RomanError` (derivato da `Exception`), che funzionerà da classe base per tutte le mie eccezioni specifiche che saranno definite in seguito. Questa è una questione di stile; avrei potrei altrettanto facilmente derivare ogni singola eccezione direttamente dalla classe `Exception`.

(2) Le eccezioni `OutOfRangeError` e `NotIntegerError` saranno in futuro usate da `toRoman` per segnalare varie forme di input non valido, come specificato in `ToRomanBadInput`.

(3) L'eccezione `InvalidRomanNumeralError` sarà in futuro usata da `fromRoman` per segnalare input non valido, come specificato in `FromRomanBadInput`.

(4) In questa fase, noi vogliamo definire le API di ciascuna delle nostre funzioni, ma non vogliamo ancora scrivere il codice, cosi creiamo delle funzioni vuote usando la parola chiave di Python `pass`.

Siamo arrivati al grande momento (rullo di tamburi, prego): stiamo finalmente per eseguire i nostri test su questo piccolo modulo ancora poco "formato". A questo punto, tutti i test dovrebbe fallire. In effetti, se un test ha successo nella prima fase, vuol dire che dovremmo ritornare sul modulo `romantest.py` e riconsiderare il perché abbiamo scritto un test così inutile da avere successo anche con queste funzioni nullafacenti.

Eseguite `romantest1.py` con l'opzione di linea di comando `-v`, che vi darà un output più prolisso, cosicché si possa vedere esattamente che cosa succede in ogni test. Con un po di fortuna, il vostro output dovrebbe somigliare a questo:

Esempio 7.8. Output del modulo `romantest1.py` eseguito su `roman1.py`

```
fromRoman should only accept uppercase input ... ERROR
toRoman should always return uppercase ... ERROR
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... FAIL
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL

======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 154, in testFromRomanCase
    roman1.fromRoman(numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 148, in testToRomanCase
    self.assertEqual(numeral, numeral.upper())
AttributeError: 'None' object has no attribute 'upper'
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 127, in testRepeatedPairs
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman1.InvalidRomanNumeralError, roman1.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
```

```
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 93, in testToRomanKnownValues
    self.assertEqual(numeral, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: I != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 116, in testDecimal
    self.assertRaises(roman1.NotIntegerError, roman1.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 112, in testNegative
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 104, in testTooLarge
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input                                (1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage1\romantest1.py", line 108, in testZero
    self.assertRaises(roman1.OutOfRangeError, roman1.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError                                       (2)
----------------------------------------------------------------------
Ran 12 tests in 0.040s                                                (3)

FAILED (failures=10, errors=2)                                        (4)
```

(1)

Eseguendo lo script, si lancia `unittest.main()`, che esegue ciascuno dei test, vale a dire ciascuno dei metodi definiti in ciascuna classe presente in `romantest.py`. Per ogni test, viene stampata la `doc string` del metodo comunque, sia se il test ha avuto successo sia se fallisce. Come ci si aspettava, nessuno dei nostri test ha avuto successo.

(2) Per ciascun test fallito, `unittest` visualizza la traccia di esecuzione che mostra esattamente cos'è successo. In questo caso, la nostra chiamata al metodo `assertRaises` (che ha anche il nome di `failUnlessRaises`) solleva una eccezione `AssertionError`, perché si aspettava che `toRoman` sollevasse un'eccezione `OutOfRangeError` e questo non è successo.

(3) Dopo il dettaglio delle esecuzioni dei test, `unittest` visualizza un sommario di quanti test sono stati eseguiti e quanto tempo è stato impiegato.

(4) In termini generali, il test dell'unità di codice è fallito perché almeno un test non ha avuto sucesso, Quando un test fallisce, `unittest` distingue tra fallimenti ed errori. Un fallimento è una chiamata ad un metodo del tipo `assertXYZ`, come `assertEqual` o `assertRaises`, che fallisce perché la condizione verificata non è vera o l'eccezione attesa non è stata sollevata. Un errore è ogni altro tipo di eccezione sollevata nel codice che si sta verificando o nello stesso codice di test. Per esempio, il metodo `testFromRomanCase` ("il metodo `fromRoman` dovrebbe accettare solo input in lettere maiuscole") ha dato come risultato un errore, perché la chiamata a `numeral.upper()` ha sollevato un'eccezione `AttributeError`, dovuta al fatto che `toRoman` avrebbe dovuto restituire una stringa, ma non lo ha fatto. Mentre la chiamata al metodo `testZero` ("`toRoman` non dovrebbe accettare come dato in ingresso lo 0") è fallita perché la chiamata al metodo `fromRoman` non solleva l'eccezione `InvalidRomanNumeral` alla quale `assertRaises` è preposta.

7.7. `roman.py`, fase 2

Ora che abbiamo delineato l'infrastruttura del modulo `roman`, è tempo di cominciare a scrivere il codice e passare con successo qualche test.

Esempio 7.9. `roman2.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000), (1)
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
```

```
    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:      (2)
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

(1) La variabile `romanNumeralMap` è una tupla che definisce tre cose:

1. La rappresentazione in caratteri dei numeri romani di base. Si noti che non si tratta solo dei numeri romani a singolo carattere; sono state anche incluse coppie di caratteri come `CM` ("cento in meno di mille"); come vedremo in seguito, questo rende il codice della funzione `toRoman` più semplice.
2. L'ordine dei numeri romani. I numeri romani vengono elencati per valore discendente, da `M` in poi fino ad `I`.
3. Il valore di ciascuno dei numeri romani elencati. Ciascuna delle tuple interne è una coppia del tipo (*`numero romano, valore corrispondente`*).

(2) Qui è dove la nostra dettagliata struttura di dati mostra la sua utilità, dato che grazie ad essa non abbiamo bisogno di alcuna logica per gestire la "regola della sottrazione". Per convertire in numeri romani, iteriamo semplicemente su `romanNumeralMap`, alla ricerca del più grande valore minore od uguale al nostro input. Una volta trovato, aggiungiamo la corrispondente rappresentazione in numero romano all'output, sottraiamo il valore trovato dall'input e ricominciamo da capo.

Esempio 7.10. Come funziona `toRoman`

Se non vi è chiaro come funziona `toRoman`, potete aggiungere una istruzione di stampa (`print`) alla fine del ciclo `while`:

```
        while n >= integer:
            result += numeral
            n -= integer
            print 'subtracting', integer, 'from input, adding', numeral, 'to output'
```

```
>>> import roman2
>>> roman2.toRoman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

Quindi `toRoman` sembra funzionare, almeno nel nostro piccolo controllo fatto a mano. Ma passerà i test delle unità di codice? Beh, no, non completamente.

Esempio 7.11. Output di `romantest2.py` a fronte di `roman2.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok                  (1)
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
```

```
toRoman should give known result with known input ... ok       (2)
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... FAIL           (3)
toRoman should fail with negative input ... FAIL
toRoman should fail with large input ... FAIL
toRoman should fail with 0 input ... FAIL
```

(1) La funzione `toRoman` restituisce effettivamente un numero romano in lettere maiuscole, perché la variabile `romanNumeralMap` definisce le rappresentazioni dei numeri romani di base in lettere maiuscole. Perciò questo test è già superato con successo.

(2) Questa è la grande novità: questa versione di `toRoman` supera con successo il test dei valori noti. Va ricordato che non è un test comprensivo di tutti i valori, anche se esercita il codice della funzione con una varietà di input validi, includendo input che producono tutti i numeri romani di un solo carattere, il più grande numero che è possibile convertire (`3999`), ed infine il numero arabo che produce il più lungo numero romano (`3888`). A questo punto, siamo ragionevolmente sicuri che la funzione lavora correttamente per qualunque numero valido che gli possiamo passare.

(3) Tuttavia, la funzione non "lavora" correttamente per valori di input non validi; fallisce infatti tutti i test in tal senso. Questo è spiegabile, dato che la funzione non include alcun controllo per valori di input non validi. I test di input non validi si aspettano che vengano sollevate specifiche eccezioni (controllandole con `assertRaises`) e non c'è niente che le possa sollevare. Questo sarà fatto nella prossima fase.

Qui c'è il resto dell'output del test delle unità di codice, che elenca i dettagli di tutti i test falliti. Siamo sotto di 10.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 156, in testFromRomanCase
    roman2.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 127, in testRepeatedPairs
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman2.InvalidRomanNumeralError, roman2.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
```

```
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: toRoman should fail with non-integer input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 116, in testDecimal
    self.assertRaises(roman2.NotIntegerError, roman2.toRoman, 0.5)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: NotIntegerError
======================================================================
FAIL: toRoman should fail with negative input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 112, in testNegative
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, -1)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with large input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 104, in testTooLarge
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 4000)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
======================================================================
FAIL: toRoman should fail with 0 input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage2\romantest2.py", line 108, in testZero
    self.assertRaises(roman2.OutOfRangeError, roman2.toRoman, 0)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: OutOfRangeError
----------------------------------------------------------------------
Ran 12 tests in 0.320s

FAILED (failures=10)
```

7.8. `roman.py`, fase 3

Adesso che la funzione `toRoman` si comporta correttamente con input validi (numeri da `1` a `3999`), è tempo di fare

in modo che lo faccia anche con input non validi (qualsiasi altra cosa).

Esempio 7.12. `roman3.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):                                          (1)
        raise OutOfRangeError, "number out of range (must be 1..3999)" (2)
    if int(n) <> n:                                                 (3)
        raise NotIntegerError, "decimals can not be converted"

    result = ""                                                     (4)
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

def fromRoman(s):
    """convert Roman numeral to integer"""
    pass
```

(1) Questa è una simpatica scorciatoia in stile Python: confronti multipli in un solo colpo. L'espressione è equivalente a `if not ((0 < n) and (n < 4000))`, ma è molto più facile da leggere. Questo è il nostro controllo sull'intervallo di ammissibilità e dovrebbe filtrare gli input che sono troppo grandi, negativi o uguali a zero.

(2) Con l'istruzione `raise` si possono sollevare le proprie eccezioni. Si può sollevare una qualsiasi delle eccezioni predefinite, oppure le proprie eccezioni specifiche definite in precedenza. Il secondo parametro, il messaggio di errore, è opzionale; se viene specificato, è visualizzato nel traceback, che viene stampato se l'eccezione non è gestita dal codice.

(3) Questo è il controllo sui numeri decimali. I decimali non possono essere convertiti in numeri romani.

(4) Il resto della funzione non è cambiata.

Esempio 7.13. Osservare `toRoman` gestire input non corretti

```
>>> import roman3
>>> roman3.toRoman(4000)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 27, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
>>> roman3.toRoman(1.5)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "roman3.py", line 29, in toRoman
    raise NotIntegerError, "decimals can not be converted"
NotIntegerError: decimals can not be converted
```

Esempio 7.14. Output di `romantest3.py` a fronte di `roman3.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... FAIL
toRoman should give known result with known input ... ok (1)
fromRoman(toRoman(n))==n for all n ... FAIL
toRoman should fail with non-integer input ... ok        (2)
toRoman should fail with negative input ... ok           (3)
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

(1) `toRoman` passa ancora il test sui valori noti, il che è confortante. Tutti i test che erano stati superati nella fase 2 sono superati anche ora, indicando che le ultime modifiche non hanno scombinato niente.

(2) Un fatto più eccitante è che tutti i nostri test di input non validi sono stati superati. Questo test, `testDecimal`, è passato con successo per via del controllo `int(n) <> n`. Quando un numero decimale è pasato a `toRoman`, il controllo `int(n) <> n` lo individua e solleva l'eccezione `NotIntegerError`, che è quello che si aspetta `testDecimal`.

(3) Questo test, `testNegative`, è superato con successo per via del controllo `not (0 < n < 4000)`, che solleva una eccezione `OutOfRangeError`, che è ciò che il test `testNegative` si aspetta.

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 156, in testFromRomanCase
    roman3.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
```

```
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 127, in testRepeatedPairs
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman3.InvalidRomanNumeralError, roman3.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 99, in testFromRomanKnownValues
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
======================================================================
FAIL: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage3\romantest3.py", line 141, in testSanity
    self.assertEqual(integer, result)
  File "c:\python21\lib\unittest.py", line 273, in failUnlessEqual
    raise self.failureException, (msg or '%s != %s' % (first, second))
AssertionError: 1 != None
----------------------------------------------------------------------
Ran 12 tests in 0.401s

FAILED (failures=6) (1)
```

(1) Siamo sotto di 6 test falliti e tutti riguardano `fromRoman`: il test sui valori noti, i tre differenti test sugli input non validi, il test sulle maiuscole/minuscole ed il test di consistenza. Questo significa che `toRoman` ha passato tutti i test che poteva superare da sola (è coinvolta nel test di consistenza, ma questo test richiede anche che `fromRoman` sia scritta, e questo non è stato ancora fatto.) Questo significa che dobbiamo smettere di codificare `toRoman` ora. Niente aggiustamenti, niente ritocchi, niente controlli addizionali "giusto per sicurezza". Basta. Togliete le mani dalla tastiera.

Nota: Sapere quando si deve smettere di scrivere il programma
La cosa più importante che test delle unità di codice, condotti in modo esaustivo, possano comunicarci è il momento in cui si deve smettere di scrivere un programma. Quando tutti i test di un modulo hanno successo, è il momento di smettere di scrivere quel modulo.

7.9. `roman.py`, fase 4

Ora che la funzione `toRoman` è completa, è tempo di cominciare a scrivere il codice di `fromRoman`. Grazie alla nostra struttura dati dettagliata che mappa i singoli numeri romani elementari negli interi corrispondenti, la cosa non è più difficile dell'aver scritto la funzione `toRoman`.

Esempio 7.15. `roman4.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
"""Convert to and from Roman numerals"""

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

# toRoman function omitted for clarity (it hasn't changed)

def fromRoman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral: (1)
            result += integer
            index += len(numeral)
    return result
```

(1) Lo schema qui è lo stesso che in `toRoman`. Si itera sulla struttura dati che definisce i numeri romani (una tupla di tuple) ed invece che cercare il valore intero più alto, fino a che ciò risulta possibile, cerchiamo il numero romano "più alto", sempre fino a che ciò risulta possibile.

Esempio 7.16. Come funziona `fromRoman`

Se non è chiaro come funziona `fromRoman`, potete aggiungere un'istruzione di stampa alla fine del ciclo `while`:

```
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print 'found', numeral, ', adding', integer

>>> import roman4
>>> roman4.fromRoman('MCMLXXII')
found M , adding 1000
found CM , adding 900
found L , adding 50
found X , adding 10
found X , adding 10
found I , adding 1
```

```
found I , adding 1
1972
```

Esempio 7.17. Output di `romantest4.py` a fronte di `roman4.py`

```
fromRoman should only accept uppercase input ... FAIL
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... FAIL
fromRoman should fail with repeated pairs of numerals ... FAIL
fromRoman should fail with too many repeated numerals ... FAIL
fromRoman should give known result with known input ... ok (1)
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok                  (2)
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

(1) Due notizie importanti qui. La prima è che `fromRoman` funziona per input validi, almeno per i valori noti che usiamo per il test.

(2) La seconda è che il nostro test di consistenza è anch'esso superato con successo. Combinato con il test sui valori noti, possiamo essere ragionevolmente sicuri che sia `toRoman` che `fromRoman` funzionino in modo corretto con tutti i valori noti. (La cosa non è garantita: è teoricamente possibile che `toRoman` abbia un baco che produca il numero romano sbagliato per qualche particolare insieme di input *e* che `fromRoman` abbia un baco reciproco rispetto al primo che produca lo stesso insieme di valori interi per i quali `toRoman` produce i numeri romani sbagliati. A seconda dell'uso previsto delle funzioni e dei loro requisiti, questo può essere o meno una preoccupazione reale; se lo è, potete scrivete un test sufficientemente comprensivo da escludere tale preoccupazione.)

```
======================================================================
FAIL: fromRoman should only accept uppercase input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 156, in testFromRomanCase
    roman4.fromRoman, numeral.lower())
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with malformed antecedents
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 133, in testMalformedAntecedent
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with repeated pairs of numerals
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 127, in testRepeatedPairs
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
======================================================================
FAIL: fromRoman should fail with too many repeated numerals
----------------------------------------------------------------------
Traceback (most recent call last):
```

```
  File "C:\docbook\dip\py\roman\stage4\romantest4.py", line 122, in testTooManyRepeatedNumerals
    self.assertRaises(roman4.InvalidRomanNumeralError, roman4.fromRoman, s)
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 12 tests in 1.222s

FAILED (failures=4)
```

7.10. `roman.py`, fase 5

Ora che `fromRoman` funziona correttamente con input validi, è tempo di far combaciare l'ultimo pezzo del puzzle: farla funzionare con input non validi. Senza troppi giri di parole cerchiamo di osservare una stringa e di determinare se è un numero romano valido. Questo è ancor più difficoltoso se paragonato agli input validi in `toRoman`, ma noi abbiamo a disposizione un potente strumento a disposizione: le espressioni regolari

Se non avete familiarità con le espressioni regolari e non avete letto il capitolo sulle Espressioni regolari, questo è il momento opportuno per farlo.

Come abbiamo visto all'inizio di questo capitolo, ci sono diverse semplici regole per comporre un numero romano. La prima di tali regole è che le migliaia, ammesso che ve ne siano, sono rappresentate da una serie di caratteri `M`.

Esempio 7.18. Controllare la presenza delle migliaia

```
>>> import re
>>> pattern = '^M?M?M?$'       (1)
>>> re.search(pattern, 'M')    (2)
<SRE_Match object at 0106FB58>
>>> re.search(pattern, 'MM')   (3)
<SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM')  (4)
<SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM') (5)
>>> re.search(pattern, '')     (6)
<SRE_Match object at 0106F4A8>
```

(1) Questo pattern ha tre parti:

1. `^` − che fa in modo che quello che segue corrisponda solo se è all'inizio della stringa. Se dopo non venisse specificato nient'altro, qualunque stringa corrisponderebbe con questo pattern, non importa dove si trovino i caratteri `M`, che non è quello che vogliamo. Noi vogliamo essere sicuri che i caratteri `M`, se ci sono, siano all'inizio della stringa.
2. `M?` − che corrisponde ad un singolo carattere `M`, opzionale. Dato che `M?` viene ripetuto tre volte, abbiamo un pattern che corrisponde a qualunque stringa contenente da 0 a 3 caratteri `M` di seguito.
3. `$` − che corrisponde solamente con ciò che precede la fine di una stringa. Quando combinato con il carattere `^` all'inizio, fa in modo che il pattern specificato debba corrispondere con l'intera stringa, senza altri caratteri prima o dopo i caratteri `M`.

(2) L'essenza del modulo `re` è la funzione di ricerca `search` che prende una espresione regolare (`pattern`) e una stringa (`'M'`) per cercare di confrontarle l'una con l'altra. Se si trova una corrispondenza, la funzione `search` restituisce un oggetto che ha vari metodi che danno i dettagli sulla corrispondenza; se non c'è una corrispondenza, la funzione `search` restituisce `None`, il valore nullo in Python. Non entreremo nei dettagli sull'oggetto restituito dalla funzione (sebbene sia un

argomento interessante), perché tutto ciò che ci importa al momento è sapere se c'è corrispondenza, cosa che possiamo stabilire semplicemente dal valore restituito dalla funzione `search`. La stringa `'M'` corrisponde alla nostra espressione regolare, perché il primo carattere opzionale `M` corrisponde ed il secondo ed il terzo carattere opzionale `M` vengono ignorati.

(3) La stringa `'MM'` corrisponde perché il primo ed il secondo carattere `M` opzionali corrispondono ed il terzo è ignorato.

(4) `'MMM'` corrisponde perché tutti e tre i caratteri `M` corrispondono.

(5) `'MMMM'` non corrisponde. Tutti e tre i caratteri `M` corrispondono, ma poi l'espressione regolare richiede che la stringa finisca (a causa del caratterer `$`) mentre la stringa non è ancora finita (a causa della quarta `M`) . Quindi la funzione `search` restituisce `None`.

(6) È interessante notare che la stringa vuota ha corrispondenza con la nostra espressione regolare perché tutti i caratteri `M` richiesti sono opzionali. Tenetelo a mente, risulterà importante nella prossima sezione.

Trattare le centinaia è più difficile che trattare le migliaia, perché ci sono diversi modi mutualmente esclusivi in cui possono essere espresse, a seconda del valore da considerare.

- `100` = `C`
- `200` = `CC`
- `300` = `CCC`
- `400` = `CD`
- `500` = `D`
- `600` = `DC`
- `700` = `DCC`
- `800` = `DCCC`
- `900` = `CM`

Dunque ci sono quattro possibili schemi:

1. `CM`
2. `CD`
3. da 0 a 3 caratteri `C` (0 se non ci sono centinaia)
4. `D`, seguito da 0 a 3 caratteri `C`

Gli ultimi due schemi possone essere riassunti in uno:

- una `D`, opzionale, seguita da 0 a 3 caratteri `C`

Esempio 7.19. Controllare la presenza delle centinaia

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$' (1)
>>> re.search(pattern, 'MCM')            (2)
<SRE_Match object at 01070390>
>>> re.search(pattern, 'MD')             (3)
<SRE_Match object at 01073A50>
>>> re.search(pattern, 'MMMCCC')         (4)
<SRE_Match object at 010748A8>
>>> re.search(pattern, 'MCMC')           (5)
>>> re.search(pattern, '')               (6)
<SRE_Match object at 01071D98>
```

(1)

Questo pattern comincia come il precedente, controllando che dall'inizio della stringa (^) ci siano fino a tre caratteri di migliaia (M?M?M?). Quindi c'è la parte nuova, tra parentesi, che definisce un insieme di tre pattern mutualmente esclusivi, separati da barre verticali: CM, CD, e D?C?C?C? (quest'ultimo è una D opzionale seguita da 0 a 3 caratteri C opzionali). Il parser delle espressioni regolari cerca una corrispondenza per ciascuno di questi tre pattern, nell'ordine (da sinistra a destra), prendendo il primo pattern che corrisponde ed ignorando il resto.

(2) 'MCM' corrisponde, perché il primo pattern M corrisponde, il secondo ed il terzo M sono ignorati, il pattern CM corrisponde (cosicché i patterns alternativi CD e D?C?C?C? non sono mai nemmeno considerati). MCM è la rappresentazione in numeri romani di 1900.

(3) 'MD' corrisponde perché il primo pattern M corrisponde, il secondo ed il terzo M sono ignorati ed il pattern D?C?C?C? corrisponde con la stringa D (ciascuno dei 3 caratteri C nel pattern è opzionale ed in questo caso sono ignorati). MD è la rappresentazione in numeri romani di 1500.

(4) 'MMMCCC' corrisponde perché tutti e 3 i pattern M corrispondono, ed il pattern D?C?C?C? corrisponde con CCC (la D è opzionale ed è ignorata). MMMCCC è la rappresentazione in numeri romani di 3300.

(5) 'MCMC' non corrisponde. Il primo pattern M corrisponde, il secondo ed il terzo M sono ignorati, il pattern CM corrisponde, ma poi $ non corrisponde perché non siamo ancora alla fine della stringa (abbiamo ancora un carattere C che non corrisponde a niente). Il carattere C *non* non corrisponde come parte del pattern D?C?C?C? perché è già stata trovata una corrispondenza con il pattern CM, che è mutualmente esclusivo rispetto al pattern D?C?C?C?

(6) È interessante notare che la stringa vuota continua a corrispondere a questo pattern, perché tutti i caratteri M sono opzionali ed ignorati, e la stringa vuota corrisponde al pattern D?C?C?C? in cui tutti i caratteri sono opzionali e possono essere ignorati.

Wow! Vedete quanto rapidamente un'espressione regolare può diventare maligna? Ed abbiamo solo coperto le migliaia e le centinaia. (Più tardi, in questa sezione, vedremo una sintassi leggermente diversa per scrivere le espressioni regolari; questa sintassi, pur essendo altrettanto complicata, perlomeno ci permette di documentare sul posto le differenti componenti delle espressioni.) Fortunatamente, se avete seguito fino ad ora, le decine e le unità sono facili, perché seguono esattamente lo stesso pattern.

Esempio 7.20. `roman5.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
```

```
                ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError, "number out of range (must be 1..3999)"
    if int(n) <> n:
        raise NotIntegerError, "decimals can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' (1)

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not re.search(romanNumeralPattern, s):                                    (2)
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

(1) Questa è solo la continuazione del pattern che abbiamo costruito per gestire le migliaia e le centinaia. Le decine sono rappresentate o da `XC` (`90`), `XL` (`40`), oppure da un opzionale `L` seguito da zero a tre opzionali `X`. Le unità sono rappresentate o da `IX` (`9`), `IV` (`4`) o da un opzionale `V` seguito da zero a tre opzionali `I`.

(2) Avendo codificato tutta questa logica nella nostra espressione regolare, il codice per controllare numeri romani non validi diventa banale. Se `re.search` restituisce un oggetto, vuol dire che è stata trovata una corrispondenza con la nostra espressione regolare ed il nostro input è valido; altrimenti, il nostro input non è valido.

A questo punto, siete autorizzati ad essere scettici sul fatto che quella orribile espressione regolare possa filtrare tutti i tipi di numeri romani non validi. Ma non dovete prendere per buona la mia parola; osservate i risultati:

Esempio 7.21. Output di `romantest5.py` a confronto con `roman5.py`

```
fromRoman should only accept uppercase input ... ok          (1)
toRoman should always return uppercase ... ok
fromRoman should fail with malformed antecedents ... ok      (2)
fromRoman should fail with repeated pairs of numerals ... ok (3)
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 12 tests in 2.864s
```

```
OK                                                                  (4)
```

(1) Una cosa che non ho detto a proposito delle espressioni regolari è che, di norma, trattano diversamente maiuscole e minuscole. Dato che la nostra espressione regolare `romanNumeralPattern` era espressa in caratteri maiuscoli, il controllo fatto con `re.search` rigetta ogni input che non sia completamente in lettere maiuscole. Quindi il nostro test sugli input in lettere maiuscole passa con succcesso.

(2) Cosa ancora più importante, tutti i nostri test con valori non validi passano con successo. Per esempio, il test sull'ordine erroneo delle cifre controlla casi come `MCMC`. Come abbiamo visto, questa stringa non corrisponde alla nostra espressione regolare, per cui la funzione `fromRoman` solleva una eccezione `InvalidRomanNumeralError`, che è quello che il nostro test si aspetta, per cui il test ha successo.

(3) In effetti, tutti i nostri test con input non validi hanno successo. Questa nostra espressione regolare individua tutti i casi a cui avevamo pensato quando abbiamo preparato i nostri test.

(4) Il premio dell'anno per il termine più eccitante va alla parola "`OK`", che è stampata dal modulo `unittest` quando tutti i test hanno successo.

Nota: Cosa fare quando tutti i test passano con successo
Quando tutti i test passano con successo, smettete di scrivere codice.

7.11. Come gestire gli errori di programmazione

A dispetto dei nostri migliori sforzi per scrivere test completi per le unità di codice, capita di fare degli errori di programmazione, in gergo chiamati bachi ("bug"). Un baco corrisponde ad un test che non è stato ancora scritto.

Esempio 7.22. Il baco

```
>>> import roman5
>>> roman5.fromRoman("") (1)
0
```

(1) Ricordate nella sezione precedente quando continuavamo a dire che una stringa vuota corrispondeva con l'espressione regolare che usavamo per controllare un numero romano valido? Bene, capita che questo sia ancora valido per la versione finale dell'espressione regolare. Questo è un baco; noi vogliamo che una stringa vuota sollevi un'eccezione `InvalidRomanNumeralError` esattamente come ogni altra sequenza di caratteri che non rappresenta un numero romano valido.

Dopo aver ricostruito il baco, e prima di porvi rimedio, è opportuno scrivere un test che fallisca a causa del baco, illustrandone così le caratteristiche.

Esempio 7.23. Verificare la presenza del baco (`romantest61.py`)

```
class FromRomanBadInput(unittest.TestCase):

    # previous test cases omitted for clarity (they haven't changed)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman.InvalidRomanNumeralError, roman.fromRoman, "") (1)
```

(1)

La cosa è piuttosto semplice. Basta chiamare la funzione `fromRoman` con una stringa vuota e assicurarsi che sollevi un eccezione `InvalidRomanNumeralError`. La parte difficile è stata individuare il baco, adesso che ne conosciamo la presenza, scrivere un test per verificarlo è semplice.

Dato che il nostro codice ha un baco, e noi abbiamo un test che verifica questo baco, il test fallirà.

Esempio 7.24. Output di `romantest61.py` a fronte di `roman61.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... FAIL
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

======================================================================
FAIL: fromRoman should fail with blank string
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage6\romantest61.py", line 137, in testBlank
    self.assertRaises(roman61.InvalidRomanNumeralError, roman61.fromRoman, "")
  File "c:\python21\lib\unittest.py", line 266, in failUnlessRaises
    raise self.failureException, excName
AssertionError: InvalidRomanNumeralError
----------------------------------------------------------------------
Ran 13 tests in 2.864s

FAILED (failures=1)
```

Ora possiamo risolvere il problema.

Esempio 7.25. Eliminazione del baco (`roman62.py`)

```
def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s: (1)
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

(1) Solo due linee di codice sono richieste: un controllo esplicito per la striga vuota ed un' istruzione `raise`.

Esempio 7.26. Output di `romantest62.py` a fronte di `roman62.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok (1)
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 2.834s

OK (2)
```

(1) Il test per la stringa vuota ora passa, quindi il baco è stato eliminato.

(2) Tutti gli altri test continuano a passare con successo e questo significa che l'eliminazione del baco non ha scombinato nient'altro. Smettete di scrivere codice.

Scrivere codice in questo modo non serve a rendere più facile l'eliminazione dei bachi. Bachi semplici (come questo) richiedono test semplici; casi più complessi richiedono test più complessi. In un ambiente orientato ai test, può *sembrare* che ci voglia più tempo per eliminare un baco, giacché è necessario dettagliare in forma di codice esattamente qual'è il baco (per scrivere il test), e poi eliminare il problema. Quindi, se il test non passa subito con successo, occorre scoprire se ad essere sbagliata sia la correzione del problema o il test. Tuttavia, alla distanza, questo andare avanti e indietro tra il codice da verificare ed il codice di test ripaga, perché rende più probabile che un baco sia eliminato correttamente al primo colpo. Inoltre, dato che è possibile eseguire i vecchi test *insieme* ai i nuovi, è molto meno probabile che si danneggi il codice esistente quando si cerca di eliminare il baco. I test delle unità di codice di oggi saranno i test di regressione di domani.

7.12. Gestire il cambiamento di requisiti

A dispetto dei vostri migliori sforzi di bloccare i vostri clienti in un angolo per tirargli fuori gli esatti requisiti del software da sviluppare, sotto la minaccia di sottoporli ad orribili operazioni con forbici e cera bollente, i requisiti cambieranno lo stesso. Molti clienti non sanno cosa vogliono fino a quando non lo vedono, ed anche allora, non sono così bravi a dettagliare esattamente il progetto, a tal punto da fornire indicazioni che possano risultare utili. Ed anche nel caso lo siano, chiederanno sicuramente di più per la prossima versione del software. Siate quindi preparati ad aggiornare i vostri test quando i requisiti cambieranno.

Supponiamo per esempio di voler espandere l'intervallo delle nostre funzioni di conversione dei numeri romani. Ricordate la regola che diceva che nessun carattere dovrebbe essere ripetuto più di tre volte? Bene, i Romani erano disposti a fare un'eccezione a questa regola, rappresentando `4000` con quattro caratteri `M` di seguito. Se facciamo questo cambiamento, saremo capaci di espandere il nostro intervallo di numeri convertibili da `1..3999` a `1..4999`. Ma prima, abbiamo bisogno di fare qualche cambiamento al codice dei nostri test.

Esempio 7.27. Modificare i test per tener conto di nuovi requisiti (`romantest71.py`)

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
import roman71
import unittest
```

```
class KnownValues(unittest.TestCase):
    knownValues = ( (1, 'I'),
                    (2, 'II'),
                    (3, 'III'),
                    (4, 'IV'),
                    (5, 'V'),
                    (6, 'VI'),
                    (7, 'VII'),
                    (8, 'VIII'),
                    (9, 'IX'),
                    (10, 'X'),
                    (50, 'L'),
                    (100, 'C'),
                    (500, 'D'),
                    (1000, 'M'),
                    (31, 'XXXI'),
                    (148, 'CXLVIII'),
                    (294, 'CCXCIV'),
                    (312, 'CCCXII'),
                    (421, 'CDXXI'),
                    (528, 'DXXVIII'),
                    (621, 'DCXXI'),
                    (782, 'DCCLXXXII'),
                    (870, 'DCCCLXX'),
                    (941, 'CMXLI'),
                    (1043, 'MXLIII'),
                    (1110, 'MCX'),
                    (1226, 'MCCXXVI'),
                    (1301, 'MCCCI'),
                    (1485, 'MCDLXXXV'),
                    (1509, 'MDIX'),
                    (1607, 'MDCVII'),
                    (1754, 'MDCCLIV'),
                    (1832, 'MDCCCXXXII'),
                    (1993, 'MCMXCIII'),
                    (2074, 'MMLXXIV'),
                    (2152, 'MMCLII'),
                    (2212, 'MMCCXII'),
                    (2343, 'MMCCCXLIII'),
                    (2499, 'MMCDXCIX'),
                    (2574, 'MMDLXXIV'),
                    (2646, 'MMDCXLVI'),
                    (2723, 'MMDCCXXIII'),
                    (2892, 'MMDCCCXCII'),
                    (2975, 'MMCMLXXV'),
                    (3051, 'MMMLI'),
                    (3185, 'MMMCLXXXV'),
                    (3250, 'MMMCCL'),
                    (3313, 'MMMCCCXIII'),
                    (3408, 'MMMCDVIII'),
                    (3501, 'MMMDI'),
                    (3610, 'MMMDCX'),
                    (3743, 'MMMDCCXLIII'),
                    (3844, 'MMMDCCCXLIV'),
                    (3888, 'MMMDCCCLXXXVIII'),
                    (3940, 'MMMCMXL'),
                    (3999, 'MMMCMXCIX'),
                    (4000, 'MMMM'),                                    (1)
                    (4500, 'MMMMD'),
                    (4888, 'MMMMDCCCLXXXVIII'),
                    (4999, 'MMMMCMXCIX'))

    def testToRomanKnownValues(self):
```

```
        """toRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.toRoman(integer)
            self.assertEqual(numeral, result)

    def testFromRomanKnownValues(self):
        """fromRoman should give known result with known input"""
        for integer, numeral in self.knownValues:
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class ToRomanBadInput(unittest.TestCase):
    def testTooLarge(self):
        """toRoman should fail with large input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 5000) (2)

    def testZero(self):
        """toRoman should fail with 0 input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, 0)

    def testNegative(self):
        """toRoman should fail with negative input"""
        self.assertRaises(roman71.OutOfRangeError, roman71.toRoman, -1)

    def testDecimal(self):
        """toRoman should fail with non-integer input"""
        self.assertRaises(roman71.NotIntegerError, roman71.toRoman, 0.5)

class FromRomanBadInput(unittest.TestCase):
    def testTooManyRepeatedNumerals(self):
        """fromRoman should fail with too many repeated numerals"""
        for s in ('MMMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):     (3)
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testRepeatedPairs(self):
        """fromRoman should fail with repeated pairs of numerals"""
        for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testMalformedAntecedent(self):
        """fromRoman should fail with malformed antecedents"""
        for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV',
                  'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
            self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, s)

    def testBlank(self):
        """fromRoman should fail with blank string"""
        self.assertRaises(roman71.InvalidRomanNumeralError, roman71.fromRoman, "")

class SanityCheck(unittest.TestCase):
    def testSanity(self):
        """fromRoman(toRoman(n))==n for all n"""
        for integer in range(1, 5000):                                  (4)
            numeral = roman71.toRoman(integer)
            result = roman71.fromRoman(numeral)
            self.assertEqual(integer, result)

class CaseCheck(unittest.TestCase):
    def testToRomanCase(self):
        """toRoman should always return uppercase"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            self.assertEqual(numeral, numeral.upper())
```

```
    def testFromRomanCase(self):
        """fromRoman should only accept uppercase input"""
        for integer in range(1, 5000):
            numeral = roman71.toRoman(integer)
            roman71.fromRoman(numeral.upper())
            self.assertRaises(roman71.InvalidRomanNumeralError,
                              roman71.fromRoman, numeral.lower())

if __name__ == "__main__":
    unittest.main()
```

(1) I valori noti precedenti non cambiano (costituiscono sempre dei valori significativi da verificare), ma abbiamo bisogno di aggiungerne qualcun altro nella zona dei quattromila. Qui abbiamo incluso `4000` (il più corto), `4500` (il secondo più corto), `4888` (il più lungo) e `4999` (il più grande).

(2) La definizione di "input troppo grande" è cambiata. Questo test chiamava la funzione `toRoman` con il valore `4000` e si aspettava un errore; ora che i valori da `4000` a `4999` sono validi, dobbiamo innalzare il valore del test a `5000`.

(3) La definizione di "troppe cifre romane ripetute di seguito" è anch'essa cambiata. Questo test chiamava `fromRoman` con `'MMMM'` e si aspettava un errore; ora che `'MMMM'` è considerato un numero romano valido, dobbiamo portare il valore di test a `'MMMM'`.

(4) Il test di consistenza ed il test sulle maiuscole/minuscole iterano su ogni numero nell'intervallo, da `1` a `3999`. Dato che l'intervallo si è espanso, questi cicli `for` hanno bisogno di essere aggiornati per arrivare fino a `4999`.

Ora i nostri test sono aggiornati in accordo con i nuovi requisiti, ma non il nostro codice, per cui c'è da aspettarsi che alcuni dei test falliscano.

Esempio 7.28. Output di `romantest71.py` a fronte di `roman71.py`

```
fromRoman should only accept uppercase input ... ERROR        (1)
toRoman should always return uppercase ... ERROR
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ERROR (2)
toRoman should give known result with known input ... ERROR   (3)
fromRoman(toRoman(n))==n for all n ... ERROR                  (4)
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
```

(1) Il nostro controllo sulle maiuscole/minuscole ora fallisce perché si cicla da `1` a `4999`, ma la funzione `toRoman` accetta solo numeri da `1` a `3999`, quindi va in errore non appena il test raggiunge `4000`.

(2) Il test sui valori noti per `fromRoman` fallisce anch'esso non appena si raggiunge `'MMMM'`, perché `fromRoman` lo considera ancora un numero romano non valido.

(3) Il test sui valori noti per `toRoman` fallisce non appena si raggiunge `4000`, perché `toRoman` lo considera ancora fuori dall'intervallo dei valori validi.

(4) Il test di consistenza fallisce anch'esso quando si arriva a `4000`, perché `toRoman` lo considera ancora fuori dall'intervallo.

```
======================================================================
ERROR: fromRoman should only accept uppercase input
----------------------------------------------------------------------
```

```
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 161, in testFromRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: toRoman should always return uppercase
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 155, in testToRomanCase
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 102, in testFromRomanKnownValues
    result = roman71.fromRoman(numeral)
  File "roman71.py", line 47, in fromRoman
    raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s
InvalidRomanNumeralError: Invalid Roman numeral: MMMM
======================================================================
ERROR: toRoman should give known result with known input
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 96, in testToRomanKnownValues
    result = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
======================================================================
ERROR: fromRoman(toRoman(n))==n for all n
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\docbook\dip\py\roman\stage7\romantest71.py", line 147, in testSanity
    numeral = roman71.toRoman(integer)
  File "roman71.py", line 28, in toRoman
    raise OutOfRangeError, "number out of range (must be 1..3999)"
OutOfRangeError: number out of range (must be 1..3999)
----------------------------------------------------------------------
Ran 13 tests in 2.213s

FAILED (errors=5)
```

Ora che abbiamo dei test che falliscono perché i nuovi requisiti non sono ancora stati implementati, possiamo cominciare a pensare di modificare il codice per allinearlo ai nuovi test. (Una cosa a cui occore un po' di tempo per abituarsi, quando si programma usando i test delle unità di codice, è il fatto che il codice sotto verifica non è mai più "avanti" del codice di test. Di solito rimane indietro e ciò significa che avete ancora del lavoro da fare. Non appena si mette in pari, si smette di programmare.)

Esempio 7.29. Trasformare in codice i nuovi requisiti (`roman72.py`)

```
"""Convert to and from Roman numerals"""
import re

#Define exceptions
class RomanError(Exception): pass
```

```
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
                   ('CD', 400),
                   ('C',  100),
                   ('XC', 90),
                   ('L',  50),
                   ('XL', 40),
                   ('X',  10),
                   ('IX', 9),
                   ('V',  5),
                   ('IV', 4),
                   ('I',  1))

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 5000):                                                    (1)
        raise OutOfRangeError, "number out of range (must be 1..4999)"
    if int(n) <> n:
        raise NotIntegerError, "decimals can not be converted"

    result = ""
    for numeral, integer in romanNumeralMap:
        while n >= integer:
            result += numeral
            n -= integer
    return result

#Define pattern to detect valid Roman numerals
romanNumeralPattern = '^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$' (2)

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not re.search(romanNumeralPattern, s):
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
    return result
```

(1) `toRoman` ha solo bisogno di un piccolo cambio, nel controllo dell'intervallo. Dove prima controllavamo che `0 < n < 4000` ora controlliamo che `0 < n < 5000`. Inoltre cambiamo il messaggio di errore dell'eccezione sollevata per indicare il nuovo intervallo accettabile (`1..4999` invece di `1..3999`). Non abbiamo bisogno di fare alcun cambio per il resto della funzione; i nuovi casi vengono già gestiti correttamente. La funzione aggiunge tranquillamente una `'M'` per ogni migliaia; datole in input `4000`, tira fuori `'MMMM'`. L'unica ragione per cui prima non si comportava così era dovuta al fatto che veniva bloccata esplicitamente dal controllo di intervallo.

(2) Non abbiamo bisogno di fare alcun cambio nella funzione `fromRoman`. L'unica modifica riguarda la variabile `romanNumeralPattern`; se la osservate da vicino, vi accorgerete che abbiamo aggiunto un ulteriore `M`

opzionale nella prima parte della espressione regolare. Questo consentirà fino a quattro caratteri `M` invece che fino a tre; di conseguenza verranno accettati i numeri romani fino all'equivalente di `4999` invece che solo fino all'equivalente di `3999`. La funzione `fromRoman` in sé è assolutamente generica; semplicemente, cerca cifre di numeri romani ripetute e ne somma i rispettivi valori senza preoccuparsi di quante volte essi si ripetano. L'unica ragione per cui la funzione non trattava correttamente `'MMMM'` in precedenza era perché ne era esplicitamente impedita dal confronto con l'espressione regolare.

A questo punto potreste essere scettici sul fatto che due piccole modifiche siano tutto ciò di cui abbiamo bisogno. Ehi, non dovete fidarvi della mia parola. Osservate da soli:

Esempio 7.30. Output di `romantest72.py` a fronte di `roman72.py`

```
fromRoman should only accept uppercase input ... ok
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok

----------------------------------------------------------------------
Ran 13 tests in 3.685s

OK (1)
```

(1) Tutti i test hanno avuto successo. Smettete di scrivere codice.

Progettare test esaustivi significa non dover mai dipendere da un programmatore che dice "Fidati di me!".

7.13. Rifattorizzazione

La cosa migliore nel fare test esaustivi delle unità di codice non è la sensazione piacevole che si ha quando tutti i test hanno finalmente successo, e neanche la soddisfazione di quando qualcun altro ti rimprovera di aver scombinato il loro codice e tu puoi effettivamente *provare* che non è vero. La cosa migliore nell'effettuare i test delle unità di codice è la sensazione che ti lascia la libertà di rifattorizzare senza provare rimorsi.

La rifattorizzazione è il procedimento con il quale si prende del codice funzionante e lo si modifica in modo che funzioni meglio. Di solito, "meglio" significa "più velocemente", sebbene possa anche significare "che usa meno memoria" oppure "che usa meno spazio su disco" o semplicemente "in modo più elegante". Qualunque sia il significato per voi, per il vostro progetto e per il vostro ambiente, la rifattorizzazione è importante per la salute a lungo termine di ogni programma.

Nel nostro caso, "meglio" significa "più veloce". In particolar modo, la funzione `fromRoman` è più lenta del necessario, a causa di quella lunga e ostica espressione regolare che viene usata per verificare i numeri romani. Probabilmente non vale la pena di cercare di fare a meno completamente delle espressioni regolari (sarebbe difficile, e potrebbe risultare niente affatto più veloce), ma possiamo velocizzare la funzione precompilando l'espressione regolare.

Esempio 7.31. Compilare le espressioni regolari

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M')              (1)
<SRE_Match object at 01090490>
>>> compiledPattern = re.compile(pattern) (2)
>>> compiledPattern
<SRE_Pattern object at 00F06E28>
>>> dir(compiledPattern)                (3)
['findall', 'match', 'scanner', 'search', 'split', 'sub', 'subn']
>>> compiledPattern.search('M')         (4)
<SRE_Match object at 01104928>
```

(1) Questa è la sintassi che abbiamo visto in precedenza: `re.search` prende un'espressione regolare in forma di stringa (`pattern`) e una stringa con cui confrontarla (`'M'`). Se il pattern corrisponde, la funzione restituisce un oggetto corrispondenza, che può essere analizzato per capire essattamente cosa corrisponde a cosa ed in che modo.

(2) Questa è la nuova sintassi: `re.compile` prende un'espressione regolare come stringa e restituisce un oggetto di tipo pattern. Si noti che non c'è nessuna stringa da confrontare in questo caso. Compilare un'espressione regolare non ha niente a che vedere con il fare il confronto tra l'espressione ed una stringa specifica (come `'M'`); la compilazione coinvolge solamente l'espressione regolare.

(3) L'oggetto pattern compilato, restituito da `re.compile`, ha diverse funzioni che sembrano utili, incluse alcune (come `search` e `sub`) che sono disponibili direttamente nel modulo `re`.

(4) Chiamare la funzione `search` dell'oggetto pattern compilato, passandogli la stringa `'M'` produce lo stesso risultato di una chiamata a `re.search` passandogli come argomenti sia l'espressione regolare che la stringa `'M'`. Solo che è molto, molto più veloce. (In effetti, la funzione `re.search` semplicemente compila l'espressione regolare di input e chiama al posto nostro il metodo `search` del risultante oggetto pattern compilato.)

> **Nota: Compilare le espressioni regolari**
> Ogni qualvolta si ha intenzione di usare più di una volta una espressione regolare, la si dovrebbe prima compilare per ottenerne il corrispondente oggetto pattern, e quindi chiamare direttamente i metodi di tale oggetto.

Esempio 7.32. Uso di un espressione regolare compilata in `roman81.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
# toRoman and rest of module omitted for clarity

romanNumeralPattern = \
    re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$') (1)

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, 'Input can not be blank'
    if not romanNumeralPattern.search(s):                                (2)
        raise InvalidRomanNumeralError, 'Invalid Roman numeral: %s' % s

    result = 0
    index = 0
    for numeral, integer in romanNumeralMap:
        while s[index:index+len(numeral)] == numeral:
            result += integer
```

```
            index += len(numeral)
    return result
```

(1) Questa sembra molto simile alla precedente, ma in effetti sono cambiate molte cose. `romanNumeralPattern` non è più una stringa; è l'oggetto patter restituito da `re.compile`.

(2) Questo significa che possiamo chiamare direttamente i metodi di `romanNumeralPattern`. Questo risulterà molto, molto più veloce che chiamare ogni volta `re.search`. L'espressione regolare è compilata una volta e poi memorizzata in `romanNumeralPattern` quando il modulo è importato per la prima volta; poi, ogni volta che viene chiamata `fromRoman`, è immediatamente possibile confrontare la stringa di input con l'espressione regolare, senza che nessun passaggio intermedio sia eseguito in modo implicito.

E allora, quanto abbiamo guadagnato in velocità compilando l'espressione regolare? Osservate voi stessi:

Esempio 7.33. Output di `romantest81.py` a fronte di `roman81.py`

```
.............          (1)
----------------------------------------------------------------------
Ran 13 tests in 3.385s (2)

OK                     (3)
```

(1) Solo una nota di passaggio: questa volta, ho eseguito i test *senza* l'opzione `-v`, cosicché invece della completa stringa di documentazione, per ogni test che ha successo è stampato solo un carattere punto. (Se un test fallisce, è stampata una `F`, se ha un errore, è stampata una `E`. Viene sempre stampato il traceback completo per ogni fallimento od errore, cosicché possiamo risalire a qual'era il problema.)

(2) Abbiamo eseguito `13` test in `3.385` secondi, a fronte dei `3.685` secondi ottenuti senza precompilare l'espressione regolare. Questo è un miglioramento complessivo dell'`8%`, senza contare che che la maggior parte del tempo speso durante i test è stato utilizzato facendo altre cose. (Separatamente, ho misurato il tempo occorso all'elaborazione delle espressioni regolari a sé stanti, separate dal resto dei test, e ho trovato che compilare l'espressione regolare accellera l'esecuzione della ricerca di una media del `54%`.) Non male, per una modifica così semplice.

(3) Oh, e nel caso ve lo stesse chiedendo, precompilare la nostra espressione regolare non ha scombinato niente, lo abbiamo appena dimostrato.

C'è un'altra ottimizzazione di prestazioni che voglio provare. Data la complessità della sintassi delle espressioni regolari, non dovrebbe sorprendere il fatto che spesso ci sia più di un modo per scrivere la stessa espressione. Dopo qualche discussione circa questo modulo nel newsgroup comp.lang.python, qualcuno mi ha suggerito di provare ad usare la sintassi `{m,n}` per i caratteri opzionali ripetuti.

Esempio 7.34. `roman82.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#   re.compile('^M?M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$')

#new version
romanNumeralPattern = \
    re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$') (1)
```

(1)

Abbiamo rimpiazzato `M?M?M?M?` con `M{0,4}`. Entrambi significano la stessa cosa: "corrisponde con una stringa formata da 0 a 4 caratteri `M`". Allo stesso modo, `C?C?C?` diventa `C{0,3}` ("corrisponde con una stringa formata da 0 a 3 caratteri `C`") e lo stesso è fatto per le `X` e le `I`.

Questa forma dell'espressione regolare è un po' più corta (sebbene non certo più leggibile). Veniamo alla grande domanda: è più veloce?

Esempio 7.35. Output di `romantest82.py` a fronte di `roman82.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s (1)

OK                     (2)
```

(1) Nel complesso, i test girano il 2% più veloci con questa forma della nostra espressione regolare. Questo non suona particolarmente eccitante, ma tenete presente che la funzione `search` è una piccola parte di tutto il test; la maggior parte del tempo è spesa facendo altre cose. (Separatamente, ho misurato il tempo impiegato solo dalle espressioni regolari, ed ho trovato che la funzione di ricerca è l'11% più veloce con questa sintassi.) Precompilando l'espressione regolare e riscrivendo parte di essa per usare questa nuova sintassi, abbiamo migliorato le prestazioni dell'espressione regolare di più del 60% e le prestazioni generali di tutto il test di oltre il 10%.

(2) Più importante di ogni aumento di prestazioni è il fatto che il modulo continui a funzionare perfettamente. Questa è la libertà di cui vi parlavo in precedenza: la liberta di ritoccare, cambiare o riscrivere qualunque parte del codice e poter poi verificare di non aver combinato guai nel farlo. Questa non è una licenza a ritoccare all'infinito il vostro codice giusto per il piacere di farlo; noi avevamo un obbiettivo molto specifico ("rendere `fromRoman` più veloce"), e siamo stati capaci di raggiungerlo senza l'angoscia latente di avere introdotto nuovi bachi con le modifiche fatte.

C'è un altro ritocco che vorrei fare, e quindi prometto che poi smetterò di rifattorizzare questo modulo e lo metterò a nanna. Come abbiamo visto più volte, le espressioni regolari possono rapidamente diventare piuttosto rognose ed illeggibili. Non mi piacerebbe tornare su questo modulo fra sei mesi e cercare di fargli delle modifiche. Certo, i test hanno successo, per cui so che il modulo funziona, ma se non sono in grado di capire *come* funziona non sarò capace di aggiungere nuove caratteristiche, eliminare nuovi bachi, oppure manutenerlo in altro modo. La documentazione è un fattore critico per la manutenibilità del codice, e Python fornisce un modo di documentare in modo esteso le espressioni regolari.

Esempio 7.36. `roman83.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
# rest of program omitted for clarity

#old version
#romanNumeralPattern = \
#   re.compile('^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$')

#new version
romanNumeralPattern = re.compile('''
    ^                   # beginning of string
    M{0,4}              # thousands - 0 to 4 M's
    (CM|CD|D?C{0,3})    # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 C's),
                        #            or 500-800 (D, followed by 0 to 3 C's)
    (XC|XL|L?X{0,3})    # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 X's),
                        #        or 50-80 (L, followed by 0 to 3 X's)
```

```
    (IX|IV|V?I{0,3})    # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 I's),
                        #        or 5-8 (V, followed by 0 to 3 I's)
    $                   # end of string
    ''', re.VERBOSE) (1)
```

(1) La funzione `re.compile` può prendere un secondo argomento opzionale, che è un insieme di uno o più flag che controllano varie opzioni sull'espressione regolare compilata. Qui stiamo specificando il flag `re.VERBOSE`, che dice a Python che all'interno dell'espressione regolare vi sono commenti. I commenti e i caratteri vuoti prima e dopo di essi *non* vengono considerati come parte dell'espressione regolare; la funzione `re.compile` si limita a rimuoverli prima di compilare l'espressione. Questa nuova, "prolissa" (ndt: "verbose") versione dell'espressione regolare è identica a quella vecchia, ma è infinitamente più leggibile.

Esempio 7.37. Output di `romantest83.py` a fronte di `roman83.py`

```
.............
----------------------------------------------------------------------
Ran 13 tests in 3.315s (1)

OK                     (2)
```

(1) Questa nuova versione "prolissa" gira esattamente alla stessa velocità della vecchia. In effetti, gli oggetti pattern compilati sono identici, visto che la funzione `re.compile` rimuove tutta la roba che abbiamo aggiunto.

(2) Questa nuova versione "prolissa" supera gli stessi test della vecchia versione. Niente è cambiato, eccetto che il programmatore che ritorna su questo modulo dopo sei mesi ha più possibilità di capire come lavora la funzione.

7.14. Postscritto

Un lettore intelligente dopo aver letto la sezione precedente ha subito fatto il passo successivo. Il più grosso grattacapo (e peggioratore delle prestazioni) nel programma com'è scritto al momento è l'espressione regolare, che è necessaria perché non abbiamo altro modo di decomporre un numero romano. Ma ci sono solo 5000 numeri romani: perché non ci costruiamo all'inizio una tavola di riferimento e non usiamo quella? Questa idea è ancora migliore una volta capito che è possibile rimuovere del tutto l'uso delle espressioni regolari. Una volta costruita la tavola di riferimento per convertire interi in numeri romani, è possibile costruire la tavola inversa per convertire i numeri romani in interi.

Ma la cosa migliore è che questo lettore aveva già un set completo di test delle unità di codice. Anche se ha cambiato metà del codice nel modulo `roman.py`, i test delle unità di codice rimanevano gli stessi, cosicché ha potuto verificare che il nuovo codice funzionasse altrettando bene dell'originale.

Esempio 7.38. `roman9.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
#Define exceptions
class RomanError(Exception): pass
class OutOfRangeError(RomanError): pass
class NotIntegerError(RomanError): pass
class InvalidRomanNumeralError(RomanError): pass

#Roman numerals must be less than 5000
MAX_ROMAN_NUMERAL = 4999

#Define digit mapping
romanNumeralMap = (('M',  1000),
                   ('CM', 900),
                   ('D',  500),
```

```
                    ('CD', 400),
                    ('C',  100),
                    ('XC', 90),
                    ('L',  50),
                    ('XL', 40),
                    ('X',  10),
                    ('IX', 9),
                    ('V',  5),
                    ('IV', 4),
                    ('I',  1))

#Create tables for fast conversion of roman numerals.
#See fillLookupTables() below.
toRomanTable = [ None ]  # Skip an index since Roman numerals have no zero
fromRomanTable = {}

def toRoman(n):
    """convert integer to Roman numeral"""
    if not (0 < n <= MAX_ROMAN_NUMERAL):
        raise OutOfRangeError, "number out of range (must be 1..%s)" % MAX_ROMAN_NUMERAL
    if int(n) <> n:
        raise NotIntegerError, "decimals can not be converted"
    return toRomanTable[n]

def fromRoman(s):
    """convert Roman numeral to integer"""
    if not s:
        raise InvalidRomanNumeralError, "Input can not be blank"
    if not fromRomanTable.has_key(s):
        raise InvalidRomanNumeralError, "Invalid Roman numeral: %s" % s
    return fromRomanTable[s]

def toRomanDynamic(n):
    """convert integer to Roman numeral using dynamic programming"""
    result = ""
    for numeral, integer in romanNumeralMap:
        if n >= integer:
            result = numeral
            n -= integer
            break
    if n > 0:
        result += toRomanTable[n]
    return result

def fillLookupTables():
    """compute all the possible roman numerals"""
    #Save the values in two global tables to convert to and from integers.
    for integer in range(1, MAX_ROMAN_NUMERAL + 1):
        romanNumber = toRomanDynamic(integer)
        toRomanTable.append(romanNumber)
        fromRomanTable[romanNumber] = integer

fillLookupTables()
```

E allora, quanto è più veloce?

Esempio 7.39. Output di `romantest9.py` a fronte di `roman9.py`

```
.............
----------------------------------------------------------------------
```

```
Ran 13 tests in 0.791s

OK
```

Ricordate, le migliori prestazioni mai ottenute nella versione originale erano 13 test in 3.315 secondi. Ovviamente, questo non è un paragone onesto, perché questa versione richiederà più tempo per essere importata (dato che è allora che vengono riempite le tavole di riferimento). Ma dato che il modulo è importato una sola volta, questo tempo è trascurabile in confronto a tutta l'esecuzione del programma.

La morale della favola?

- La semplicità è una virtù.
- Specialmente quando ci sono di mezzo le espressioni regolari.
- I test delle unità di codice possono darvi la confidenza necessaria per effettuare rifattorizzazioni su larga scala ... anche se non siete stati voi a scrivere il codice originale.

7.15. Sommario

Il test delle unità di codice è un concetto forte che, se opportunamente implementato, può ridurre sia i costi di manutenzione che aumentare la flessibilità in ogni progetto a lungo termine. È però anche importante capire che i test delle unità di codice non sono una panacea, una bacchetta magica che risolve tutti i problemi, o una pallottola d'argento. Scrivere dei buoni test è difficile, e mantenerli aggiornati richiede disciplina (specialmente quando i clienti vi stanno chiedendo a gran voce la soluzione di qualche problema critico nel software). I test delle unità di codice non sono un sostituto per le altre forme di verifica del codice, tra cui i test funzionali, i test di integrazione e i test di accettazione. Rappresentano tuttavia una pratica realizzabile, e funzionano, ed una volta che li avrete provati vi chiederete come avete fatto ad andare avanti senza di loro.

Questo capitolo ha trattato di un mucchio di cose e molte di esse non erano specifiche di Python. Ci sono infrastrutture per i test delle unità di codice in molti linguaggi, ciascuna delle quali richiede che si capiscano gli stessi concetti di base:

- Progettare test che siano specifici, automatici ed indipendenti
- Scrivere i test *prima* di scrivere il codice da verificare
- Scrivere test che usano input validi e controllano la correttezza del risultato
- Scrivere test che usano input non validi e controllano che il programma fallisca nel modo atteso
- Scrivere nuovi test e tenere aggiornati quelli esistenti per documentare i bachi trovati o per verificare l'implementazione di nuovi requisiti
- Rifattorizzare senza rimorsi per migliorare prestazioni, scalabilità, leggibilità, manutenibilità, o qualunque altra "−ità" sia necessaria

In aggiunta, dovreste ora essere a vostro agio nell'eseguire una delle seguenti attività specifiche di Python:

- Derivare una nuova classe, `unittest.TestCase` e scrivere metodi per i singoli test
- Usare `assertEqual` per verificare che una funzione restituisca il valore atteso
- Usare `assertRaises` per verificare che una funzione sollevi l'eccezione attesa
- Chiamare `unittest.main()` nella vostra clausola `if __name__` per eseguire tutti i test in una sola volta
- Eseguire i test sia in modalità prolissa che in modalità normale

Ulteriori letture

- XProgramming.com include riferimenti utili a scaricare infrastrutture per i test delle unità di codice per molti differenti linguaggi.

---

[13] "Posso resistere a tutto, tranne che alle tentazioni." −−Oscar Wilde

Capitolo 8. Programmazione orientata ai dati

8.1. Immergersi

Nel capitolo relativo al Test delle unità di codice, ne abbiamo discusso la filosofia e ne abbiamo vista l'implementazione in Python. Questo capitolo si dedicherà a tecniche specifiche di Python, più avanzate, basate sul modulo `unittest`. Se non avete letto il capitolo Test delle unità di codice, vi perderete a metà lettura. Siete stati avvertiti.

Quello che segue è un programma Python completo che si comporta come una semplice e grossolano infrastruttura di test di regressione. Prende i test per le unità che avete scritto per i singoli moduli, li riunisce tutti in un grande insieme di test e li esegue tutti insieme. Io uso questo script come parte del processo di creazione di questo libro; ho test di unità di codice per numerosi programmi di esempio (non solo il modulo `roman.py` contenuto nel capitolo Test delle unità di codice) e la prima cosa che i miei script per creare il libro fanno è di eseguire questo programma per assicurarsi che tutti gli esempi lavorino ancora. Se questo test di regressione fallisce, la creazione si blocca immediatamente. Non voglio rilasciare esempi non funzionanti più di quanto vogliate scaricarli e spremervi le meningi urlando contro il vostro monitor domandandovi perché non funzionano.

Esempio 8.1. `regression.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
"""Regression testing framework

This module will search for scripts in the same directory named
XYZtest.py.  Each such script should be a test suite that tests a
module through PyUnit.  (As of Python 2.1, PyUnit is included in
the standard library as "unittest".)  This script will aggregate all
found test suites into one big test suite and run them all at once.
"""

import sys, os, re, unittest

def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
    load = unittest.defaultTestLoader.loadTestsFromModule
    return unittest.TestSuite(map(load, modules))

if __name__ == "__main__":
    unittest.main(defaultTest="regressionTest")
```

Questo script, se eseguito nella stessa directory degli altri script di esempio forniti con questo libro, troverà tutti i test di unità, chiamati *`module`*`test.py`, li eseguirà come un test singolo e li farà superare o fallire tutti assieme.

Esempio 8.2. Semplice output di `regression.py`

```
[f8dy@oliver py]$ python regression.py -v
help should fail with no object ... ok                          (1)
```

```
help should return known result for apihelper ... ok
help should honor collapse argument ... ok
help should honor spacing argument ... ok
buildConnectionString should fail with list input ... ok          (2)
buildConnectionString should fail with string input ... ok
buildConnectionString should fail with tuple input ... ok
buildConnectionString handles empty dictionary ... ok
buildConnectionString returns known result with known input ... ok
fromRoman should only accept uppercase input ... ok               (3)
toRoman should always return uppercase ... ok
fromRoman should fail with blank string ... ok
fromRoman should fail with malformed antecedents ... ok
fromRoman should fail with repeated pairs of numerals ... ok
fromRoman should fail with too many repeated numerals ... ok
fromRoman should give known result with known input ... ok
toRoman should give known result with known input ... ok
fromRoman(toRoman(n))==n for all n ... ok
toRoman should fail with non-integer input ... ok
toRoman should fail with negative input ... ok
toRoman should fail with large input ... ok
toRoman should fail with 0 input ... ok
kgp a ref test ... ok
kgp b ref test ... ok
kgp c ref test ... ok
kgp d ref test ... ok
kgp e ref test ... ok
kgp f ref test ... ok
kgp g ref test ... ok

----------------------------------------------------------------------
Ran 29 tests in 2.799s

OK
```

(1) I primi cinque test sono contenuti in `apihelpertest.py`, che esegue i test sugli script di esempio provenienti dal capitolo La potenza dell'introspezione.

(2) I cinque test successivi provengono da `odbchelpertest.py`, che esegue i test sugli script visti nel capitolo Conoscere Python.

(3) I test rimanenti provengono da `romantest.py` e sono esaminati approfonditamente nel capitolo Test delle unità di codice.

8.2. Trovare il percorso

Quando lanciate gli script Python da linea di comando, qualche volta torna utile sapere dov'è localizzato lo script corrente sul disco fisso.

Questo è uno degli oscuri e piccoli trucchi che sono virtualmente impossibili da imparare da soli, ma semplici da ricordare una volta visti. La chiave è nel `sys.argv`. Come abbiamo visto in Elaborare XML, `sys.argv` è una lista contenente gli argomenti da linea di comando. Comunque, contiene anche il nome dello script corrente, esattamente come se fosse chiamato da linea di comando, e questa è un'informazione sufficiente per determinarne la locazione.

Esempio 8.3. `fullpath.py`

Se non lo avete ancora fatto, potete scaricare questo ed altri esempi usati in questo libro.

```
import sys, os
```

```
print 'sys.argv[0] =', sys.argv[0]             (1)
pathname = os.path.dirname(sys.argv[0])       (2)
print 'path =', pathname
print 'full path =', os.path.abspath(pathname) (3)
```

(1) Senza curarsi di come lanciate lo script, `sys.argv[0]` conterrà sempre il nome dello script, esattamente come appare da linea di comando. Questo può o meno includere informazioni sul percorso, come vedremo fra breve.

(2) `os.path.dirname` prende un filename come stringa e ritorna la porzione di percorso della directory. Se il filename dato non include informazioni sul percorso, `os.path.dirname` ritorna una stringa vuota.

(3) `os.path.abspath` è il pezzo forte. Prende un percorso, che può essere parziale o anche vuoto, e ritorna un percorso completo.

`os.path.abspath` richiede ulteriori spiegazioni. È molto flessibile; può prendere ogni tipo di percorso.

Esempio 8.4. Ulteriori spiegazioni su `os.path.abspath`

```
>>> import os
>>> os.getcwd()                        (1)
/home/f8dy
>>> os.path.abspath('')                (2)
/home/f8dy
>>> os.path.abspath('.ssh')            (3)
/home/f8dy/.ssh
>>> os.path.abspath('/home/f8dy/.ssh') (4)
/home/f8dy/.ssh
>>> os.path.abspath('.ssh/../foo/')    (5)
/home/f8dy/foo
```

(1) `os.getcwd()` ritorna la directory di lavoro corrente.

(2) Chiamando `os.path.abspath` con una stringa vuota ritorna la directory corrente, come `os.getcwd()`.

(3) Chiamando `os.path.abspath` con un percorso parziale viene costruito un percorso completo, basato sulla directory corrente.

(4) Chiamando `os.path.abspath` con un percorso completo semplicemente restituisce sé stesso.

(5) `os.path.abspath` si occupa anche di *normalizzare* i percorsi che ritorna. Notate che questo esempio funziona anche se non avete realmente una diretory 'foo'. `os.path.abspath` non controlla mai il vostro disco locale; è solo una manipolazione di stringhe.

Nota: os.path.abspath non effettua controlli sui percorsi
I percorsi e i nomi di file che passate a `os.path.abspath` non è necessario che esistano.

Nota: Normalizzare i percorsi
`os.path.abspath` non realizza solo percorsi completi, si occupa anche di normalizzarli. Se siete nella directory `/usr/`, `os.path.abspath('bin/../local/bin')` ritornerà `/usr/local/bin`. Se volete solo normalizzare un percorso senza trasformarlo in un percorso completo, usate invece `os.path.normpath`.

Esempio 8.5. Esempio di output di `fullpath.py`

```
[f8dy@oliver py]$ python /home/f8dy/diveintopython/common/py/fullpath.py (1)
sys.argv[0] = /home/f8dy/diveintopython/common/py/fullpath.py
path = /home/f8dy/diveintopython/common/py
full path = /home/f8dy/diveintopython/common/py
[f8dy@oliver diveintopython]$ python common/py/fullpath.py              (2)
```

```
sys.argv[0] = common/py/fullpath.py
path = common/py
full path = /home/f8dy/diveintopython/common/py
[f8dy@oliver diveintopython]$ cd common/py
[f8dy@oliver py]$ python fullpath.py                                    (3)
sys.argv[0] = fullpath.py
path =
full path = /home/f8dy/diveintopython/common/py
```

(1) Nel primo caso, `sys.argv[0]` include il percorso completo dello script. Possiamo poi usare la funzione `os.path.dirname` per rimuovere il nome dello script e ritornare il nome completo della directory, e `os.path.abspath` ritorna semplicemente ciò che gli diamo.

(2) Se lo script lavora usando un percorso parziale, `sys.argv[0]` conterrà ancora esattamente quello che appare sulla linea di comando. `os.path.dirname` poi ci restituirà un percorso parziale (relativo alla directory corrente), mentre `os.path.abspath` costruirà un percorso completo dal percorso parziale.

(3) Se lo script viene lanciato dalla directory corrente senza dargli alcun percorso, `os.path.dirname` restituirà semplicemente una stringa vuota. Con una stringa vuota, `os.path.abspath` ritorna la directory corrente, che è ciò che vogliamo, dato che lo script è stato lanciato proprio dalla directory corrente.

> **Nota: os.path.abspath è cross−platform**
> Come altre funzioni nei moduli `os` e `os.path`, anche `os.path.abspath` è multi−piattaforma. I vostri risultati sembreranno leggermente differenti dai miei esempi se state lavorando su Windows (che usa i backslash come separatori di percorso) o sul Mac OS (che usa due punti), ma funzionano comunque. Questo è il punto cruciale del modulo `os`.

Appendice. Un lettore era insodddisfatto di questa soluzione, voleva poter lanciare tutti i test delle unità di codice nella directory corrente, non in quella dove si trovava `regression.py`. Suggerì quest'altro approccio:

Esempio 8.6. Lanciare script nella directory corrente

```
import sys, os, re, unittest

def regressionTest():
    path = os.getcwd()       (1)
    sys.path.append(path)    (2)
    files = os.listdir(path) (3)
```

(1) Invece di impostare `path` alla directory dove sta girando lo script, lo impostiamo invece alla corrente directory di lavoro. Questa sarà qualunque directory in cui vi trovavate prima di lanciare lo script, che non è necessariamente la stessa dove si trova lo script. (Leggete questa frase un po' di volte fino a che non l'avete capita.)

(2) Aggiungete questa directory alla libreria di ricerca dei percorsi di Python, così quando importiamo dinamicamente i moduli di test delle unità di codice, più tardi, Python potrà trovarli. Noi non avevamo bisogno di farlo quando `path` era la directory dello script, perché Python cerca sempre in quella directory.

(3) Il resto della funzione rimane invariato.

Questa tecnica vi permetterà di riutilizzare questo script `regression.py` in più progetti. È sufficiente che mettiate lo script in una directory comune e poi anche nella directory del progetto prima di lanciarlo. Tutti quei test delle unità di codice del progetto saranno trovati e provati, invece di quelli ubicati nella directory comune dove si trova `regression.py`.

8.3. Filtrare liste rivisitate

Avete già familiarità con l'utilizzo delle list comprehensions per filtrare le liste. C'è un'altro modo per raggiungere lo stesso risultato, che alcune persone considerano più espressivo.

Python ha una funzione built−in chiamata `filter` che prende due argomenti, una funzione ed una lista, e ritorna una lista. [14] La funzione passata come primo argomento a `filter` deve essa stessa prendere un argomento, e la lista che `filter` ritorna conterrà tutti gli elementi dalla lista passati a `filter` per i quali la funzione elaborata ritorna true.

Capito tutto? Non è difficile come sembra.

Esempio 8.7. Introdurre `filter`

```
>>> def odd(n):           (1)
...     return n%2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, −3]
>>> filter(odd, li)       (2)
[1, 3, 5, 9, −3]
>>> filteredList = []
>>> for n in li:          (3)
...     if odd(n):
...         filteredList.append(n)
...
>>> filteredList
[1, 3, 5, 9, −3]
```

(1) `odd` usa la funzione built−in mod "`%`" per ritornare `1` se `n` è dispari e `0` se `n` è pari.

(2) `filter` prende due argomenti, una funzione (`odd`) e una lista (`li`). Esegue un ciclo nella lista e chiama `odd` con ogni elemento. Se `odd` ritorna un valore vero (ricordate, ogni valore diverso da zero è vero in Python), l'elemento è incluso nella lista ritornata, altrimenti ne è fuori, nello stesso ordine in cui appare nell'originale.

(3) Potreste raggiungere lo stesso risultato con un ciclo `for`. A seconda del vostro stile di programmazione, ciò può sembrare più "diretto", ma funzioni come `filter` sono molto più espressive. Non solo sono più facili da scrivere, ma anche da leggere. Leggere il ciclo `for` è come stare troppo vicino ad un dipinto; vedete tutti i dettagli, ma ci vogliono alcuni secondi per fare un passo indietro e vedere l'intera opera: "Ehi, stavamo semplicemente filtrando una lista!".

Esempio 8.8. `filter` in `regression.py`

```
    files = os.listdir(path)                                (1)
    test = re.compile("test\.py$", re.IGNORECASE)           (2)
    files = filter(test.search, files)                      (3)
```

(1) Come abbiamo visto nella sezione Trovare il percorso, `path` può contenere il percorso completo o parziale della directory dello script corrente, o può contenere una stringa vuota se lo script è stato lanciato dalla directory corrente. D'altra parte, `files` finirà con i nomi dei file nella stessa directory dalla quale lo script è stato lanciato.

(2) Questa è una espressione regolare compilata. Come abbiamo visto nella sezione Rifattorizzazione, se userete la stessa espressione regolare più e più volte, dovreste compilarla per una migliore performance. L'oggetto compile ha un metodo `search` che prende un singolo elemento, la stringa da cercare, se l'espressione regolare rispecchia la stringa. Il metodo `search` ritorna un oggetto `Match` contenente informazioni sul confronto dell'espressione regolare; altrimenti ritorna `None`, il valore nullo di Python.

(3) Per ogni elemento nella lista `files`, chiameremo il metodo `search` dell'oggetto compilato dell'espressione regolare, `test`. Se l'espressione regolare coincide, il metodo ritornerà un oggetto `Match`, che Python considera essere vero, così l'elemento sarà incluso nella lista ritornata da `filter`. Se l'espressione regolare non coincide, il metodo `search` ritornerà `None`, che Python cosidera essere falso, così l'elemento non sarà incluso.

Nota storica. Le versioni di Python anteriori alla 2.0 non hanno le list comprehensions, perciò non potevate filtrare usando le list comprehensions; la funzione `filter` era l'unica possibilità. Anche con l'introduzione delle list comprehensions dalla versione 2.0, alcune persone preferiscono ancora la vecchia `filter` (e la sua funzione gemella, `map`, che vedremo nella prossima sezione). Entrambe queste tecniche funzionano, e nessuna scomparirà, quindi è solo una questione di stile.

Esempio 8.9. Filtrare usando le list comprehensions

```
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
files = [f for f in files if test.search(f)] (1)
```

(1) Questo raggiungerà lo stesso risultato della funzione `filter`. Qual'è la più espressiva? A voi la scelta.

8.4. Rivisitazione della mappatura delle liste

Siete già abituati ad usare la list comprehensions per mappare una lista in un altra. C'è un altro modo per fare la stessa cosa, usando la funzione built−in `map`. Funziona in modo molto simile alla funzione `filter`.

Esempio 8.10. Introducendo `map`

```
>>> def double(n):
...     return n*2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, −3]
>>> map(double, li)                         (1)
[2, 4, 6, 10, 18, 20, 512, −6]
>>> [double(n) for n in li]                 (2)
[2, 4, 6, 10, 18, 20, 512, −6]
>>> newlist = []
>>> for n in li:                            (3)
...     newlist.append(double(n))
...
>>> newlist
[2, 4, 6, 10, 18, 20, 512, −6]
```

(1) `map` richiede una funzione ed una lista [15] e restituisce una nuova lista chiamando la funzione ed ordinandola per ogni elemento della lista. In questo caso la funzione moltiplica semplicemente ogni elemento per 2.

(2) Potreste arrivare allo stesso risultato con una list comprehension. La list comprehension è stata introdotta in Python 2.0; `map` esiste da sempre.

(3) Potreste, se insistete a pensare come un programmatore di Visual Basic, usare un ciclo `for` per fare lo stesso lavoro.

Esempio 8.11. `map` con liste di tipi di dato diversi

```
>>> li = [5, 'a', (2, 'b')]
>>> map(double, li)                         (1)
```

```
[10, 'aa', (2, 'b', 2, 'b')]
```

(1) Come nota a margine, preferisco sottolineare che `map` funziona bene anche con liste di tipi di dato diversi, almeno finché la funzione che state usando gestisce bene i vari tipi di dato. In questo caso, la nostra funzione `double` moltiplica semplicemente l'argomento ricevuto per 2, Python fa La Cosa Giusta a seconda del tipo di dato dell'argomento. Per gli interi, significa proprio moltiplicare per 2; per le stringhe, significa concatenare la stringa con se stessa; per le tuple, significa creare una nuova tupla che ha tutti gli elementi dell'originale e ancora, a seguire, un suo duplicato.

Bene, abbiamo giocato abbastanza. Diamo un'occhiata ad un po' di codice vero.

Esempio 8.12. `map` in `regression.py`

```
filenameToModuleName = lambda f: os.path.splitext(f)[0] (1)
moduleNames = map(filenameToModuleName, files)          (2)
```

(1) Come abbiamo visto nella sezione Usare le funzioni `lambda`, `lambda` definisce una funzione inline. Come abbiamo visto nell'Esempio 4.36, Dividere i pathnames , `os.path.splitext` riceve come parametro un nome di file e restituisce una tupla (*`nome, estensione`*). Quindi `filenameToModuleName` è una funzione che riceverà un nome di file, toglierà l'estensione e ne restituirà solamente il nome.

(2) Chiama `map` per ogni nome di file elencato in files, passa il `nome-file` alla nostra funzione `filenameToModuleName` e restituisce una lista di valori risultante da ogni chiamata della funzione. In altre parole, eliminiamo l'estensione di ogni nome di file e conserviamo la lista di tutte queste parti di nomi in `moduleNames`.

Come vedremo nel resto del capitolo, possiamo estendere questo atteggiamento di pensiero data-centrico adattandolo al nostro scopo, che in definitiva è eseguire un singolo test che contenga tutti quei singoli test che ci servono.

8.5. Programmazione data-centrica

Per adesso probabilmente vi starete grattando la testa chiedendovi perché questo modo di fare sia migliore rispetto all'usare un ciclo `for` e chiamate dirette a funzioni. Ed è una domanda perfettamente legittima. È principalmente una questione di prospettiva. Usare `map` e `filter` vi obbliga a focalizzare l'attenzione sui dati.

In questo caso abbiamo cominciato senza nessun dato; la prima cosa che abbiamo fatto è stata ottenere la directory dello script in esecuzione e una lista di files in questa directory. Questo è stato il nostro punto di partenza che ci ha fornito dei dati veri su cui lavorare: una lista di nomi di files.

Comunque, sapevamo di non doverci occupare di tutti quei files, ma solo di quelli che sono configurazioni di test. Avevamo *troppi dati* e dovevamo `filtrarli`. Come potevamo sapere quali dati scegliere? Avevamo bisogno di un test per decidere, quindi ne abbiamo definito uno e lo abbiamo passato alla funzione `filter`. In questo caso abbiamo usato una espressione regolare per decidere, ma il concetto rimane lo stesso indipendentemente dal modo in cui strutturiamo il test.

A questo punto avevamo i nomi dei files di ogni configurazione di test (e solo le configurazioni di test, visto che ogni altra cosa era stata esclusa dal filtro), ma in realtà volevamo il nome dei moduli. Avevamo le informazioni giuste ma nel *formato sbagliato*. Abbiamo quindi definito una funzione per trasformare ogni nome di file in un nome di modulo ed abbiamo mappato la funzione sull'intera lista. Da un nome di file, possiamo avere un nome di modulo; da una lista di nomi di file, possiamo avere una lista di nomi di moduli.

Al posto di un `filtro`, potevamo usare un ciclo `for` con una condizione `if`. Invece di `map`, potevamo usare un ciclo `for` con una chiamata a funzione. Però usare un ciclo `for` come questi è laborioso. Nel caso migliore, è una semplice perdita di tempo; nel caso peggiore, può introdurre bugs misteriosi. Per esempio, dovremo in ogni caso

inventare un modo per decidere "questo file è una configurazione di test?"; questa è la logica specifica della nostra applicazione, e nessun linguaggio può scriverla per noi. Ma una volta trovata, dobbiamo davvero crearci il problema di definire una lista vuota, scrivere un ciclo `for`, impostare un `if`, `aggiungere` a mano alla nuova lista ogni elemento che ha passato il test e tenere traccia di quale variabile contiene i nuovi dati filtrati e quale contiene i vecchi dati integri? Perché non definire la condizione di test e poi lasciare che Python faccia il resto del lavoro per noi?

Certamente, potevate tentare di essere fantasiosi e cancellare gli elementi sul posto senza creare una nuova lista. Ma sareste stati bruciati da questo metodo ancora prima di cominciare. Cercare di modificare una struttura di dati sulla quale state iterando può essere rischioso. Cancellate un elemento, andate all'elemento successivo e ne avete appena saltato uno. Ma Python è un linguaggio che lavora in questo modo? Quanto tempo ci avete messo per capirlo? I programmatori sprecano troppo tempo e fanno troppi errori connessi a questioni puramente tecniche come questa, e questo non ha nessun senso. Non migliora per niente il vostro programma; è solo spreco di energie.

Io ho resistito alla list comprehension quando ho iniziato ad imparare Python, ed ho resistito a `filter` e `map` ancora più a lungo. Ho insistito per rendermi la vita più difficile, ancorandomi ai familiari cicli `for` ed `if` di una programmazione incentrata sul codice. Ed i miei programmi in Python sembravano scritti in Visual Basic, descrivendo ogni passo di ogni operazione in ogni funzione. Avevano lo stesso tipo di piccoli problemi e bugs oscuri. E questo non aveva senso.

Lasciamo andare tutto. Il codice laborioso non è importante. I dati sono importanti. Ed i dati non sono difficili. Sono solo dati. Se sono troppi, li filtriamo. Se non sono quelli che vogliamo, li mappiamo. Concentratevi sui dati; lasciatevi alle spalle i lavori inutili.

8.6. Importare dinamicamente i moduli

Ok, abbiamo filosofeggiato abbastanza. Vediamo come si importano dinamicamente i moduli.

Per cominciare osservate come vengono importati normalmente i moduli. La sintassi `import` *`module`* rimanda al percorso di ricerca per trovare un modulo e lo importa per nome. Si possono anche importare più moduli contemporaneamente usando una lista separata da virgole, è stato fatto nella prima riga di codice di questo capitolo.

Esempio 8.13. Importare contemporaneamente più moduli

```
import sys, os, re, unittest (1)
```

(1) Questa riga importa quattro moduli in una istruzione: `sys` (per funzioni di sistema ed accesso ai parametri della riga di comando), `os` (per operazioni legate al sistema operativo, come ad esempio elencare directory), `re` (per le espressioni regolari) e `unittest` (per i test delle unità di codice).

Adesso facciamo la stessa cosa, però con l'importazione dinamica.

Esempio 8.14. Importare moduli dinamicamente

```
>>> sys = __import__('sys')           (1)
>>> os = __import__('os')
>>> re = __import__('re')
>>> unittest = __import__('unittest')
>>> sys                               (2)
>>> <module 'sys' (built-in)>
>>> os
>>> <module 'os' from '/usr/local/lib/python2.2/os.pyc'>
```

(1) La funzione built-in `__import__` ha lo stesso obiettivo dell'istruzione `import`, però è una vera funzione ed accetta una stringa come argomento.

(2) La variabile `sys` adesso è il modulo `sys`, come se avessimo fatto `import sys`. La variabile `os` adesso è il modulo `os`, e così via.

Quindi `__import__` importa moduli, però con una stringa come argomento. In questo caso il modulo che abbiamo importato era un parametro statico, ma poteva essere una variabile o anche il risultato di una chiamata a funzione. Inoltre la variabile a cui assegnamo il modulo può avere un nome qualsiasi o anche essere un elemento di una lista.

Esempio 8.15. Importare dinamicamente una lista di moduli

```
>>> moduleNames = ['sys', 'os', 're', 'unittest'] (1)
>>> moduleNames
['sys', 'os', 're', 'unittest']
>>> modules = map(__import__, moduleNames)        (2)
>>> modules                                       (3)
[<module 'sys' (built-in)>,
<module 'os' from 'c:\Python22\lib\os.pyc'>,
<module 're' from 'c:\Python22\lib\re.pyc'>,
<module 'unittest' from 'c:\Python22\lib\unittest.pyc'>]
>>> modules[0].version                            (4)
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
>>> import sys
>>> sys.version
'2.2.2 (#37, Nov 26 2002, 10:24:37) [MSC 32 bit (Intel)]'
```

(1) `moduleNames` è una semplice lista di stinghe. Niente di strano, eccetto che le stringhe potrebbero essere nomi di moduli che, nel caso serva, possiamo importare.

(2) Sorpresa, volevamo importarli, e lo abbiamo fatto mappando la funzione `__import__` sulla lista. Ricordate, `map` prende ogni elemento della lista (`moduleNames`) e chiama la funzione (`__import__`) una volta per ogni elemento della lista, compone una lista con i risultati e la restituisce.

(3) Adesso da una lista di stringhe abbiamo creato una lista di veri e propri moduli. (Gli indirizzi potrebbero essere diversi, dipendono dal sistema operativo dove è installato Python, dalle fasi della luna, etc.)

(4) Per aver conferma che sono moduli veri, andiamo a vedere alcuni attributi. Ricodate, `modules[0]` *è* il modulo `sys` ed infatti `modules[0].version` *è* identico a `sys.version`. Ovviamente sono disponibili anche tutti gli attributi ed i metodi degli altri moduli della lista. Ma non c'è niente di magico né nell'istruzione `import`, né nei moduli. Infatti anche i moduli sono oggetti. Tutto è un oggetto.

Ora dovremmo essere in grado di comprendere il significato della maggior parte degli esempi di questo capitolo.

8.7. Mettere assieme il tutto (parte 1)

Ora abbiamo imparato abbastanza da poter scomporre le prime sette linee dell'esempio di questo capitolo: leggere una directory ed importare alcuni dei moduli che vi sono contenuti.

Esempio 8.16. La funzione `regressionTest`, parte 1

```
def regressionTest():
    path = os.path.abspath(os.path.dirname(sys.argv[0]))
    files = os.listdir(path)
    test = re.compile("test\.py$", re.IGNORECASE)
```

```
    files = filter(test.search, files)
    filenameToModuleName = lambda f: os.path.splitext(f)[0]
    moduleNames = map(filenameToModuleName, files)
    modules = map(__import__, moduleNames)
```

Osserviamolo linea per linea, interattivamente. Assumendo che la directory corrente sia `c:\diveintopython\py`, che contiene gli esempi di questo libro, compreso lo script di questo capitolo. Come abbiamo visto in *Trovare il percorso*, la directory degli script sarà contenuta nella variabile `path`, per cui ne assegnamo il valore e procediamo.

Esempio 8.17. Passo 1: Leggere i nomi dei file

```
>>> import sys, os, re, unittest
>>> path = r'c:\diveintopython\py'
>>> files = os.listdir(path)
>>> files (1)
['BaseHTMLProcessor.py', 'LICENSE.txt', 'apihelper.py', 'apihelpertest.py',
'argecho.py', 'autosize.py', 'builddialectexamples.py', 'dialect.py',
'fileinfo.py', 'fullpath.py', 'kgptest.py', 'makerealworddoc.py',
'odbchelper.py', 'odbchelpertest.py', 'parsephone.py', 'piglatin.py',
'plural.py', 'pyfontify.py', 'regression.py', 'roman.py', 'romantest.py',
'uncurly.py', 'unicode2koi8r.py', 'urllister.py', 'kgp', 'roman',
'colorize.py']
```

(1) `files` è una lista di tutti i file e le directory nella directory degli script. (Se avete già eseguito alcuni degli esempi, potrete anche notare, tra gli altri, anche qualche file `.pyc`.)

Esempio 8.18. Passo 2: Filtrare per trovare i file necessari

```
>>> test = re.compile("test\.py$", re.IGNORECASE)          (1)
>>> files = filter(test.search, files)                     (2)
>>> files                                                  (3)
['apihelpertest.py', 'kgptest.py', 'odbchelpertest.py', 'romantest.py']
```

(1) Questa espressione regolare corrisponderà ad ogni stringa che termina con `test.py`. È importante notare che è necessario far precedere il punto dal carattere escape ("\"), visto che un punto, all'interno di una espressione regolare, normalmente significa "corrisponde ad ogni singolo carattere".

(2) L'espressione regolare compilata agisce come una funzione, così può essere usata per filtrare la grande lista di file e directory, per trovare quelli che corrispondono all'espressione regolare.

(3) In questo modo è rimasta solo la lista degli script di test, poiché questi sono i soli con nomi del tipo `QUALCOSAtest.py`.

Esempio 8.19. Passo 3:Trasformare i nomi dei file in nomi di moduli

```
>>> filenameToModuleName = lambda f: os.path.splitext(f)[0] (1)
>>> filenameToModuleName('romantest.py')                    (2)
'romantest'
>>> filenameToModuleName('odchelpertest.py')
'odbchelpertest'
>>> moduleNames = map(filenameToModuleName, files)          (3)
>>> moduleNames                                             (4)
['apihelpertest', 'kgptest', 'odbchelpertest', 'romantest']
```

(1) Come si è visto in *Usare le funzioni lambda*, `lambda` è un modo veloce, anche se poco elegante, per creare delle semplici funzioni su una sola riga. Questa accetta come argomento un

nome di file con estensione e restituisce solo il nome vero e proprio, grazie all'uso della funzione standard di libreria `os.path.splitext` vista nell'Esempio 4.36, Dividere i pathnames .

(2) `filenameToModuleName` è una funzione. Non c'è niente di magico nelle funzioni `lambda` rispetto alle normali funzioni definite con l'istruzione `def`. `filenameToModuleName` può essere chiamata come ogni altra funzione e fa esattamente ciò che si desiderava: rimuove l'estensione dal suo argomento.

(3) Ora la funzione può essere applicata ad ogni file della lista dei file di test delle unità, utilizzando `map`.

(4) Il risultato è proprio quello cercato: una lista di moduli, sotto forma di stringhe.

Esempio 8.20. Passo 4: Trasformare i nomi di moduli in moduli

```
>>> modules = map(__import__, moduleNames)                (1)
>>> modules                                               (2)
[<module 'apihelpertest' from 'apihelpertest.py'>,
<module 'kgptest' from 'kgptest.py'>,
<module 'odbchelpertest' from 'odbchelpertest.py'>,
<module 'romantest' from 'romantest.py'>]
>>> modules[-1]                                           (3)
<module 'romantest' from 'romantest.py'>
```

(1) Come si è visto in *Importare dinamicamente i moduli*, si possono utilizzare `map` ed `__import__` insieme per trasformare una lista di nomi di moduli (sotto forma di stringhe) in veri moduli (ai quali si può accedere come ad ogni altro modulo).

(2) `modules` è una lista di moduli, alla quale si può accedere come ad ogni altro modulo.

(3) L'ultimo modulo nella lista *è* il modulo `romantest`, proprio come se si fosse utilizzato `import romantest`. Come si vedrà nella sezione successiva, non solo si può accedere a questo modulo, istanziare le classi contenute in esso e chiamarne le funzioni, si può fare introspezione nel modulo per scoprire quali classi e funzioni contiene.

8.8. Dentro PyUnit

Spiacente, questa parte del capitolo ancora non è stata scritta. Provate a controllare http://diveintopython.org/ per gli aggiornamenti.

---

[14] Tecnicamente, il secondo argomento di `filter` può essere una qualsiasi sequenza, incluse liste, tuple e classi che si comportano come liste, definendo il metodo speciale `___getitem__`. Se possibile, `filter` restituirà lo stesso tipo di dato che ha ricevuto, quindi filtrando una lista si ottiene una lista, invece filtrando una tupla si ottiene una tupla.

[15] Ancora, devo sottolineare che `map` può ricevere una lista, una tupla, o un qualsiasi oggetto simile ad una sequenza. Controllate la nota precedente riguardo a `filter`.

Appendice A. Ulteriori letture

Capitolo 1. Installare Python

Capitolo 2. Conoscere Python

- 2.3. Documentare le funzioni
    - PEP 257 definisce le convenzioni sulle `doc string`.
    - *Python Style Guide* discute di come scrivere una buona `doc string`.
    - *Python Tutorial* discute delle convenzioni per spaziare le `doc string`.
- 2.4. Tutto è un oggetto
    - *Python Reference Manual* spiega esattamente cosa significa che tutto in Python è un oggetto, perché alcune persone sono pedanti e amano discutere lungamente su questo genere di cose.
    - eff−bot riassume gli oggetti in Python.
- 2.5. Indentare il codice
    - *Python Reference Manual* discute i problemi dell'indentazione cross−platform e mostra vari errori di indentazione.
    - *Python Style Guide* discute sul buon stile di indentazione.
- 2.6. Testare i moduli
    - *Python Reference Manual* discute dettagli di basso livello sull'importazione dei moduli.
- 2.7. Introduzione ai dizionari
    - *How to Think Like a Computer Scientist* insegna ad usare i dizionari e mostra come usarli per modellare matrici rade.
    - Python Knowledge Base contiene molti esempi di codice sull'uso dei dizionari.
    - Python Cookbook parla di come ordinare il valori di un dizionario per chiave.
    - *Python Library Reference* riassume tutti i metodi dei dizionari.
- 2.8. Introduzione alle liste
    - *How to Think Like a Computer Scientist* spiega le liste e spiega come funziona il passaggio di liste come argomenti di funzione.
    - *Python Tutorial* mostra come usare liste come pile e code.
    - Python Knowledge Base risponde a domande comuni sulle liste e contiene esempi di codice che le utilizza le liste.
    - *Python Library Reference* riassume tutti i metodi delle liste.
- 2.9. Introduzione alle tuple
    - *How to Think Like a Computer Scientist* parle delle tuple e mostra come concatenarle.
    - Python Knowledge Base mostra come ordinare una tupla.
    - *Python Tutorial* mostra come definire una tupla con un elemento.
- 2.10. Definire le variabili
    - *Python Reference Manual* mostra esempi su quando potete saltare il carattere di continuazione linea e quando dovete usarlo.
- 2.11. Assegnare valori multipli in un colpo solo
    - *How to Think Like a Computer Scientist* mostra come usare assegnamenti multi−variabile per invertire il valore di due variabili.

- 2.12. Formattare le stringhe
    - *Python Library Reference* riassume tutti i caratteri di formattazione delle stringhe.
    - *Effective AWK Programming* discute tutti i caratteri di formattazione e le tecniche di formattazione avanzate come specificare lunghezza, precisione e il riempimento di zeri.
- 2.13. Mappare le liste
    - *Python Tutorial* discute di un altro modo per mappare le liste usando la funzione built−in `map`.
    - *Python Tutorial* mostra come annidare le list comprehensions.
- 2.14. Concatenare liste e suddividere stringhe
    - Python Knowledge Base risponde a domande comuni sulle stringhe e molti esempi di codice che usano le stringhe.
    - *Python Library Reference* riassume tutti i metodi delle stringhe.
    - *Python Library Reference* documenta il modulo `string`.
    - *The Whole Python FAQ* spiega perché `join` è un metodo di string invece che di list.

Capitolo 3. La potenza dell'introspezione

- 3.2. Argomenti opzionali ed argomenti con nome
    - *Python Tutorial* discute esattamente quando e come argomenti predefiniti vengano valutati, che ha importanza quando il valore predefinito è una lista o un'espressione con effetti collaterali.
- 3.3. type, str, dir, ed altre funzioni built−in
    - *Python Library Reference* documenta tutte le funzioni built−in e tutte le eccezioni built−in.
- 3.5. Filtrare le liste
    - *Python Tutorial* tratta di un altro modo di filtrare le liste usando la funzione built−in `filter`.
- 3.6. Le particolarità degli operatori and e or
    - Python Cookbook tratta di alternative al trucchetto `and−or` .
- 3.7. Usare le funzioni lambda
    - Python Knowledge Base discute l'utilizzo delle funzioni `lambda` per chiamare le funzioni indirettamente.
    - *Python Tutorial* mostra come accedere alle variabili esterne dall'interno di una funzione `lambda` . La PEP 227 spiega come questo cambierà nelle future versioni di Python.
    - *The Whole Python FAQ* ha esempi di codice offuscato, composto da una funzione `lambda`.

Capitolo 4. Una struttura orientata agli oggetti

- 4.2. Importare i moduli usando from module import
    - eff−bot mostra altri esempi su `import module` *contro* `from module import`.
    - Nel *Python Tutorial* è possibile leggere di tecniche avanzate d'importazione, tra cui `from module import *`.
- 4.3. Definire classi
    - *Learning to Program* una semplice introduzione alle classi.
    - *How to Think Like a Computer Scientist* mostra come utilizzare le classi per modellare specifici tipi di dato.

- ♦ *Python Tutorial* dà uno sguardo in profondità alle classi, spazi dei nomi ed ereditarietà.
- ♦ Python Knowledge Base risponde a domande comuni sulle classi.

- 4.4. Istanziare classi

  - ♦ *Python Library Reference* riassume gli attributi built−in come `__class__`.
  - ♦ *Python Library Reference* documenta il modulo `gc`, che vi dà un controllo a basso livello sulla garbage collection di Python.

- 4.5. UserDict: una classe wrapper

  - ♦ *Python Library Reference* documenta il modulo `UserDict` ed il modulo `copy`.

- 4.7. Metodi speciali di classe avanzati

  - ♦ *Python Reference Manual* documenta tutti i metodi speciali per le classi.

- 4.9. Funzioni private

  - ♦ Nel *Python Tutorial* troverete informazioni difficilmente rintracciabili altrove sulle variabili private.

- 4.10. Gestire le eccezioni

  - ♦ *Python Tutorial* discute della definizione e del sollevamento delle vostre eccezioni, ed infine della gestione di eccezioni multiple, tutto in una volta.
  - ♦ *Python Library Reference* riassume tutte le eccezioni built−in.
  - ♦ *Python Library Reference* documenta il modulo `getpass`.
  - ♦ *Python Library Reference* documenta il modulo `traceback`, che consente un accesso a basso livello agli attributi delle funzioni dopo che è stata sollevata un'eccezione.
  - ♦ *Python Reference Manual* discute del funzionamento interno del blocco `try...except`.

- 4.11. Oggetti file

  - ♦ *Python Tutorial* discute della lettura e della scrittura dei file, come la lettura una riga per volta in una lista.
  - ♦ eff−bot discute l'efficienza e le prestazioni di alcuni sistemi di lettura di file.
  - ♦ Python Knowledge Base risponde alle domande più frequenti sui file.
  - ♦ *Python Library Reference* riassume tutti i metodi degli oggetti file.

- 4.13. Ancora sui moduli

  - ♦ Il *Python Tutorial* tratta esattamente di come e quando gli argomenti predefiniti vengono valutati.
  - ♦ Il *Python Library Reference* documenta il modulo `sys`.

- 4.14. Il modulo os

  - ♦ Python Knowledge Base risponde a domande sul modulo `os` .
  - ♦ *Python Library Reference* documenta il modulo `os` ed anche il modulo `os.path`.

Capitolo 5. Elaborare HTML

- 5.4. Introdurre BaseHTMLProcessor.py

  - ♦ W3C discute dei riferimenti a caratteri ed ad entità.
  - ♦ *Python Library Reference* conferma i vostri sospetti che il modulo `htmlentitydefs` sia esattamente ciò che sembra.

- 5.9. Introduzione alle espressioni regolari

  - ♦ Regular Expression HOWTO discute delle espressioni regolari e di come usarle in Python.
  - ♦ *Python Library Reference* riassume il modulo `re` .

- 5.10. Mettere tutto insieme

    - Pensavate che stessi scherzando sull'idea del server−side scripting. Così ho fatto, fino a che non ho trovato un dialettizatore. Non ho idea se sia implementato in Python, ma la home page della mia compagnia è felice. Sfortunatamente, il codice sorgente non sembra essere disponibile.

Capitolo 6. Elaborare XML

- 6.4. Unicode

    - Unicode.org è la home page dello standard unicode, inclusa una breve introduzione tecnica.
    - Unicode Tutorial ha alcuni esempi su come usare le funzioni unicode di Python, incluso come forzare Python a coercizzare l'unicode in ASCII anche quando non lo vuole.
    - Unicode Proposal è l'originale specifica tecnica per le funzionalità unicode di Python. Solo per per programmatori esperti dell'unicode.

Capitolo 7. Test delle unità di codice

- 7.1. Immergersi

    - Questo sito fornisce ulteriori informazioni sui numeri romani, incluso un affascinante resoconto su come i Romani ed altre civiltà li usavano nella vita reale (breve resoconto: in modo approssimativo ed inconsistente).
- 7.2. Introduzione al modulo romantest.py

    - La pagina web di PyUnit contiene un'approfondita descrizione di come usare l'infrastruttura del modulo `unittest`, incluse speciali caratteristiche non discusse in questo capitolo.
    - La FAQ di PyUnit spiega perché il codice di test è salvato separatamente dal codice da validare.
    - Il *Python Library Reference* riassume le caratteristiche del modulo `unittest`.
    - ExtremeProgramming.org tratta del perché è oppportuno scrivere i test delle unità di codice.
    - Il Portland Pattern Repository contiene una discussione in continuo aggiornamento sui test di unità, inclusa una loro definizione standard, ragioni per cui si dovrebbe cominciare con lo scrivere i test delle unità di codice e molti altri studi approfonditi.
- 7.15. Sommario

    - XProgramming.com include riferimenti utili a scaricare infrastrutture per i test delle unità di codice per molti differenti linguaggi.

Capitolo 8. Programmazione orientata ai dati

Appendice B. Recensione in 5 minuti

Capitolo 1. Installare Python

- 1.1. Qual è il Python giusto per te?

  Benvenuto in Python. Immergiamoci.
- 1.2. Python su Windows

  Su Windows, avete diverse alternative per installare Python.
- 1.3. Python su Mac OS X

  Con Mac OS X, ci sono due possibili alternative per installare Python: o lo si installa, oppure no. Probabilmente, voi volete installarlo.
- 1.4. Python su Mac OS 9

  Mac OS 9 non include alcuna versione di Python, ma l'installazione è molto semplice e non ci sono altre alternative.
- 1.5. Python su RedHat Linux

  Per installare su RedHat Linux, avete bisogno di scaricare l'RPM da http://www.python.org/ftp/python/2.3.2/rpms/ ed installarlo con il comando **rpm**.
- 1.6. Python su Debian GNU/Linux

  Se siete abbastanza fortunati da utilizzare Debian GNU/Linux, potete installare con il comando **apt**.
- 1.7. Installare dai sorgenti

  Se preferite compilare dai sorgenti, potete scaricare i sorgenti di Python da http://www.python.org/ftp/python/2.3.2/ ed effettuare i soliti **`configure`**, **`make`**, **`make install`**.
- 1.8. La shell interattiva

  Adesso che abbiamo Python, che cos'è questa shell interattiva che abbiamo lanciato?
- 1.9. Sommario

  Dovreste adesso avere una versione di Python installata che lavora per voi.

Capitolo 2. Conoscere Python

- 2.1. Immergersi

  Ecco un completo e funzionante programma in Python.
- 2.2. Dichiarare le funzioni

  Python supporta le funzioni come molti altri linguaggi, ma non necessita di header file separati come il C++ o sezioni di `interfaccia`/`implementazione` come il Pascal. Quando avete bisogno di una funzione, basta che la dichiariate e la definiate.
- 2.3. Documentare le funzioni

  Potete documentare una funzione Python dandole una `doc string` (stringa di documentazione ndr.).

- 2.4. Tutto è un oggetto

  Una funzione, come ogni altra cosa in Python, è un oggetto.
- 2.5. Indentare il codice

  Le funzioni in Python non hanno un `inizio` o `fine` esplicito, nessuna parentesi graffa che indichi dove il codice inizia o finisce. L'unico delimitatore sono i due punti (`":"`) e l'indentazione del codice.
- 2.6. Testare i moduli

  I moduli in Python sono oggetti ed hanno diversi attributi utili. Potete usarli per testare i moduli dopo che li avete scritti.
- 2.7. Introduzione ai dizionari

  Uno dei tipi predefiniti in Python è il dizionario che definisce una relazione uno−a−uno tra chiavi e valori.
- 2.8. Introduzione alle liste

  Le liste sono, tra i tipi di dati in Python, la vera forza motrice. Se la vostra sola esperienza con le liste sono gli array in Visual Basic o (Dio ve ne scampi!) il datastore di Powerbuilder, fatevi forza e osservate come si usano in Python.
- 2.9. Introduzione alle tuple

  Una tupla è una lista immutabile. Una tupla non può essere modificata in alcun modo una volta che è stata creata.
- 2.10. Definire le variabili

  Python possiede il concetto di variabili locali e globali come molti altri linguaggi, ma non contempla la dichiarazione esplicita delle variabili. Le variabili cominciano ad esistere quando assegnamo loro un valore e vengono automaticamente distrutte quando non servono più.
- 2.11. Assegnare valori multipli in un colpo solo

  Una delle più comode scorciatoie di programmazione in Python consiste nell'usare sequenze per assegnare valori multipli in un colpo solo.
- 2.12. Formattare le stringhe

  Python supporta la formattazione dei valori nelle stringhe. Sebbene ciò possa comprendere espressioni molto complicate, il modo più semplice per utilizzarle consiste nell'inserire valori in una stringa attraverso l'istruzione `%s`.
- 2.13. Mappare le liste

  Una delle più potenti caratteristiche di Python sono le list comprehension (descrizioni di lista), che utilizzano una tecnica compatta per mappare una lista in un'altra, applicando una funzione ad ognuno degli elementi della lista.
- 2.14. Concatenare liste e suddividere stringhe

  Avete una lista di coppie chiave−valore nella forma *`chiave=valore`* e volete concatenarle in una singola stringa. Per concatenare qualunque lista di stringhe in una singola stringa, usate il metodo `join` di un oggetto stringa.
- 2.15. Sommario

Il programma `odbchelper.py` ed il suo output dovrebbero ora esservi perfettamente chiari.

Capitolo 3. La potenza dell'introspezione

- 3.1. Immergersi

  Qui abbiamo un programma Python completo e funzionante. Dovreste essere in grado di capire un bel po' di cose solamente guardando l'esempio. Le linee numerate illustrano concetti trattati nella sezione Conoscere Python. Non preoccupatevi se il resto del codice sembra intimidatorio; imparerete tutto nel corso di questo capitolo.
- 3.2. Argomenti opzionali ed argomenti con nome

  Python permette agli argomenti delle funzioni di avere un valore predefinito; se la funzione è chiamata senza l'argomento, l'argomento prende il suo valore predefinito. Inoltre gli argomenti possono essere specificati in qualunque ordine usando gli argomenti con nome. Le stored procedure in SQL Server Transact/SQL possono farlo; se siete dei guru nella programmazione via script di SQL Server potete saltare questa parte.
- 3.3. type, str, dir, ed altre funzioni built−in

  Python ha un piccolo insieme di funzioni built−in estremamente utili. Tutte le altre funzioni sono suddivise in moduli. Si tratta di una decisione coscienziosa, per preservare il nucleo del linguaggio dall'essere soffocato come in moltri altri linguaggi di script (coff coff,Visual Basic).
- 3.4. Ottenere riferimenti agli oggetti usando getattr

  Già sapete che le funzioni in Python sono oggetti. Quello che non sapete è che potete ottenere un riferimento da una funzione senza conoscere il suo nome fino al momento dell'esecuzione, utilizzando la funzione `getattr`.
- 3.5. Filtrare le liste

  Come già si è detto, Python offre potenti strumenti per mappare delle liste in liste corrispondenti, per mezzo delle "list comprehension". Queste possono essere combinate con un meccanismo di filtro, facendo in modo che alcuni elementi delle liste vengano mappati ed altri vengano ignorati completamente.
- 3.6. Le particolarità degli operatori and e or

  In Python, gli operatori `and` e `or` eseguono le operazioni di logica booleane che ci si aspetta dal loro nome, ma non restituiscono valori booleani; essi restituiscono invece il valore di uno degli elementi che si stanno confrontando.
- 3.7. Usare le funzioni lambda

  Python supporta un'interessante sintassi che vi permette di definire al volo delle piccole funzioni di una sola riga. Derivate dal Lisp, queste funzioni `lambda` possono essere usate ovunque sia richiesta una funzione.
- 3.8. Unire il tutto

  L'ultima riga di codice, l'unica che ancora non abbiamo analizzato, è quella che svolge l'intero lavoro. Ma adesso il lavoro è semplice, perché ogni cosa di cui abbiamo bisogno è già impostata come serve a noi. Tutte le tessere sono al loro posto; è ora di buttarle giù.
- 3.9. Sommario

Il programma `apihelper.py` ed il suo output ora dovrebbero essere chiari.

Capitolo 4. Una struttura orientata agli oggetti

- 4.1. Immergersi

  Eccovi un completo e funzionante programma in Python. Leggete la `doc string` del modulo, le classi e le funzioni, in modo da avere un'idea di cosa faccia e come lavori questo programma. Come al solito, non curatevi delle parti che non comprendete; il resto del capitolo è fatto per loro.
- 4.2. Importare i moduli usando from module import

  Python utilizza due modi per importare i moduli. Entrambi sono utili e dovreste sapere quando usarli. Il primo, `import module`, l'avete già visto nel capitolo 2. Il secondo, raggiunge lo stesso scopo, ma funziona in modo differente.
- 4.3. Definire classi

  Python è completamente orientato agli oggetti: potete definire le vostre classi, ereditare dalle vostre classi o da quelle built−in ed istanziare le classi che avete definito.
- 4.4. Istanziare classi

  Istanziare una classe in Python è molto semplice. Per istanziare una classe, basta semplicemente chiamare la classe come se fosse una funzione, passandole gli argomenti che il metodo `__init__` definisce. Il valore di ritorno sarà il nuovo oggetto.
- 4.5. UserDict: una classe wrapper

  Come avete visto, `FileInfo` è una classe che agisce come un dizionario. Per esplorare ulteriormente questo aspetto, diamo uno sguardo alla classe `UserDict` nel modulo `UserDict`, che è l'antenato della nostra classe `FileInfo`. Non è nulla di speciale; la classe è scritta in Python e memorizzata in un file `.py`, proprio come il nostro codice. In particolare, è memorizzata nella directory `lib` della vostra installazione di Python.
- 4.6. Metodi speciali per le classi

  In aggiunta ai normali metodi delle classi, c'è un certo numero di metodi speciali che le classi Python possono definire. Invece di essere chiamati direttamente dal vostro codice (come metodi normali), i metodi speciali sono chiamati per voi da Python in particolari circostanze o quando viene adoperata una certa sintassi.
- 4.7. Metodi speciali di classe avanzati

  Ci sono molti più metodi speciali dei soli `__getitem__` e `__setitem__`. Alcuni di questi vi permettono di emulare funzionalità che non avreste mai conosciuto.
- 4.8. Attributi di classe

  Conoscete già gli attributi dato, che sono variabili di una specifica istanza di una classe. Python supporta anche gli attributi di classe, che sono variabili riferite alla classe stessa.
- 4.9. Funzioni private

  Come molti linguaggi, anche Python ha il concetto di funzioni private, che non possono essere chiamate dall'esterno del loro modulo; metodi privati di una classe, che non possono essere chiamati dall'esterno della loro classe ed attributi privati, che non possono essere referenziati dall'esterno della loro classe. Al contrario di molti linguaggi, il fatto che una funzione, metodo o attributo in Python sia pubblico o privato viene determinato interamente

dal suo nome.
- 4.10. Gestire le eccezioni

  Come molti linguaggi orientati agli oggetti, Python consente la manipolazione delle eccezioni tramite i blocchi `try...except`.
- 4.11. Oggetti file

  Python ha una funzione built−in, `open`, per aprire un file su disco. `open` ritorna un oggetto di tipo file, che dispone di metodi e attributi per ottenere informazioni sul file aperto e manipolarlo.
- 4.12. Cicli for

  Come molti altri linguaggi, Python ha i cicli `for`. L'unica ragione per cui non si sono visti finora è perché Python è valido per così tante cose che spesso non se ne sente tanto il bisogno.
- 4.13. Ancora sui moduli

  I moduli, come qualunque altra cosa in Python, sono oggetti. Una volta importato, si può sempre ottenere un riferimento ad un modulo attraverso il dizionario globale `sys.modules`.
- 4.14. Il modulo os

  Il modulo `os` ha molte funzioni utili per manipolare file e processi e `os.path` ha molte funzioni per manipolare percorsi di file e directory.
- 4.15. Mettere tutto insieme

  Ancora una volta tutte le nostre pedine del domino sono al loro posto. Abbiamo visto come funziona ogni linea di codice, adesso facciamo un passo indietro e vediamo come il tutto venga messo insieme.
- 4.16. Sommario

  Adesso il programma `fileinfo.py` dovrebbe avere un senso compiuto.

Capitolo 5. Elaborare HTML

- 5.1. Immergersi

  Spesso in comp.lang.python trovo domande del genere: "Come posso creare una lista di tutti gli [headers | immagini | collegamenti] presenti nel mio documento HTML?" "Come posso [analizzare | tradurre | modificare] il testo del mio documento HTML senza modificare i tag?" "Come posso [aggiungere | rimuovere | mettere tra apici] gli attributi di tutti i tag del mio HTML in una volta sola?" Questo capitolo risponderà a tutte queste domande.
- 5.2. Introduciamo sgmllib.py

  L'elaborazione dell'HTML è suddivisa in tre passi: spezzare l'HTML nei suoi elementi costitutivi, giocherellare con questi pezzi, infine ricostruire i pezzi nuovamente nell'HTML. Il primo passo è fatto da `sgmllib.py`, che è parte della libreria standard di Python
- 5.3. Estrarre informazioni da documenti HTML

  Per estrarre informazioni da documenti HTML, create una sottoclasse `SGMLParser` e definite metodi per ogni tag o altro che volete catturare.
- 5.4. Introdurre BaseHTMLProcessor.py

  `SGMLParser` non produce nulla da solo. Analizza, analizza e analizza, e chiama un metodo

per ogni cosa interessante che trova, ma il metodo non fa nulla. `SGMLParser` è un *consuma* HTML: prende il codice HTML e lo divide in piccoli pezzi strutturati. Come avete visto nella sezione precedente, potete creare una sottoclasse di `SGMLParser` per definire classi che catturano tag specifici e producono cose utili, come la lista dei collegamenti delle pagine web. Adesso facciamo un ulteriore passo avanti, per definire la classe che cattura ogni cosa che `SGMLParser` rilascia, e quindi ricostruire l'intero documento HTML. In termini tecnici, questa classe sarà un *produttore* di codice HTML.

- 5.5. locals e globals

  Python dispone di due funzioni built−in, `locals` e `globals` che forniscono un accesso basato sui dizionari alle variabili locali e globali.
- 5.6. Formattazione di stringhe basata su dizionario

  Esiste una forma alternativa per la formattazione di stringhe che usa i dizionari al posto delle tuple di valori.
- 5.7. Virgolettare i valori degli attributi

  Una domanda ricorrente su comp.lang.python è: "Io ho un gruppo di documenti HTML con i valori degli attributi espressi senza virgolette e voglio virgolettarli tutti in maniera opportuna. Come posso farlo?" [10] (Questa situazione è generalmente causata da un responsabile di progetto, convertito alla religione "HTML è uno standard", che si aggiunge ad un grosso progetto e annuncia che tutte le pagine HTML devono essere validate da un verificatore di HTML. Avere i valori degli attributi senza virgolette è una violazione comune dello standard HTML.) Qualunque sia la ragione, è facile rimediare ai valori degli attributi senza virgolette, se si filtra il documento HTML attraverso `BaseHTMLProcessor`.
- 5.8. Introduzione al modulo dialect.py

  La classe `Dialectizer` è una semplice (e un po' stupida) specializzazione della classe `BaseHTMLProcessor`. Questa classe sottopone un blocco di testo ad una serie di sostituzioni, ma al contempo fa in modo che tutto ciò che è racchiuso in un blocco `<pre>...</pre>` rimanga inalterato.
- 5.9. Introduzione alle espressioni regolari

  Le espressioni regolari sono un strumento potente (e piuttosto standardizzato) per cercare, sostituire o analizzare del testo contenente complessi schemi di caratteri. Se avete già usato espressioni regolari in altri linguaggi (come il Perl), potete saltare questa sezione e leggere direttamente il sommario del modulo `re` per avere una panoramica delle funzioni disponibili e dei loro argomenti.
- 5.10. Mettere tutto insieme

  È tempo di mettere a frutto tutto quello che abbiamo imparato finora. Spero che abbiate prestato attenzione.
- 5.11. Sommario

  Python fornisce uno strumento potente, il modulo `sgmllib.py`, per manipolare codice HTML trasformando la sua struttura in un oggetto modello. È possibile usare questo strumento in molti modi diversi.

Capitolo 6. Elaborare XML

- 6.1. Immergersi

Ci sono due principali metodi per lavorare con l'XML. Il primo è chiamato SAX ("Simple API for XML"), funziona leggendo l'XML un pezzo per volta e chiamando un metodo per ogni elemento trovato. Se avete letto il capitolo Elaborare HTML, dovrebbe risultarvi familiare, perché è così che lavora il modulo `sgmllib`. L'altro è chiamato DOM ("Document Object Model"), funziona leggendo l'intero documento XML in un'unica volta e creando una rappresentazione interna dell'XML basata su classi native Python collegate in una struttura ad albero. Python ha dei moduli standard per entrambi i tipi di parsing, ma questo capitolo affronterà solo l'uso del DOM.

- 6.2. Package

  Analizzare un documento XML è estremamente semplice: una sola riga di codice. Comunque, prima di studiare questa riga di codice, dobbiamo fare una piccola deviazione per parlare dei package.
- 6.3. Analizzare XML

  Come stavo dicendo, analizzare un documento XML è molto semplice: una riga di codice. Dove andare poi, dipende da voi.
- 6.4. Unicode

  Unicode è un sistema per rappresentare i caratteri di tutti i differenti linguaggi del mondo. Quando Python analizza un documento XML, tutti i dati sono immagazzinati in memoria sottoforma di unicode.
- 6.5. Ricercare elementi

  Attraversare documenti XML passando da un nodo all'altro può essere noioso. Se state cercando qualcosa in particolare, bene in profondità nel vostro documento XML, c'è una scorciatoia che potete usare per trovarlo più velocemente: `getElementsByTagName`.
- 6.6. Accedere agli attributi di un elemento

  Gli elementi XML possono avere uno o più attributi ed è incredibilmente semplice accedervi una volta che avete analizzato il documento XML.
- 6.7. Astrarre le sorgenti di ingresso

  Uno dei punti di forza di Python è il suo binding dinamico, ed uno degli usi più potenti del binding dinamico sono gli *oggetti file* (ovvero, che si comportano come dei file).
- 6.8. Standard input, output, ed error

  Gli utenti UNIX hanno già familiarità con i concetti di standard input, standard output e standard error. Questa sezione è per il resto di voi.
- 6.9. Memorizzare i nodi e ricercarli

  `kgp.py` impiega alcuni trucchi che potrebbero esservi utili nella elaborazione XML. Il primo trucco consiste nel trarre vantaggio dalla consistente struttura dei documenti in input per costruire una cache di nodi.
- 6.10. Trovare i figli diretti di un nodo

  Un'altra utile tecnica quando analiziamo documenti XML consiste nel trovare tutti gli elementi figli diretti di un particolare elemento. Per esempio, nel nostro file grammar, un elemento `ref` può avere alcuni elementi `p`, ognuno dei quali può contenere molte cose, inclusi altri elementi `p`. Vogliamo trovare gli elementi `p` che sono figli del `ref`, non gli elementi `p` che sono figli degli altri elementi `p`.
- 6.11. Create gestori separati per tipo di nodo

Il terzo suggerimento utile per processare gli XML comporta la separazione del vostro codice in funzioni logiche, basate sui tipi di nodi e nomi di elementi. I documenti XML analizzati sono fatti di vari tipi di nodi, di cui ognuno rappresenta un oggetto Python. Il livello base del documento stesso è rappresentato da un oggetto `Document`. `Document` contiene uno o più oggetti `Element` (per i tags XML reali), ognuno dei quali può contenere altri oggetti `Element`, oggetti `Text` (per parti di testo), od oggetti `Comment` (per commenti incorporati). Python rende semplice scrivere uno smistatore per separare la logica per ciascun tipo di nodo.

- 6.12. Gestire gli argomenti da riga di comando

  Python supporta pienamente la creazione di programmi che possono essere eseguiti da riga di comando, completi di argomenti, sia con flag nel formato breve che nel formato esteso per specificare le varie opzioni. Nessuno di questi è specifico di XML, ma questo script fa un buon uso dell'interpretazione da riga di comando, dunque pare un buon momento per menzionarla.
- 6.13. Mettere tutto assieme

  Abbiamo coperto una buona distanza. Facciamo un passo indietro e vediamo come si mettono assieme tutti i pezzi.
- 6.14. Sommario

  Python viene rilasciato con delle potenti librerie per l'analisi e la manipolazione dei documenti XML. `minidom` prende un file XML e lo analizza in un albero di oggetti Python, permettendo l'accesso casuale ad elementi arbitrari. Inoltre, questo capitolo mostra come Python può essere usato per creare dei "veri" script da riga di comando autonomi, completi di flag da riga di comando, argomenti da riga di comando, gestione degli errori ed anche la possibilità di acquisire come input il risultato di un programma precedente.

Capitolo 7. Test delle unità di codice

- 7.1. Immergersi

  Nei capitoli precedenti, ci si è "tuffati" immediatamente nell'analisi del codice, cercando poi di capirlo il più velocemente possibile. Ora che avete immagazzinato un po' di Python, faremo un passo indietro e analizzeremo ciò che si fa *prima* di scrivere il codice.
- 7.2. Introduzione al modulo romantest.py

  Ora che abbiamo completamente definito il comportamento che ci aspettiamo dalle nostre funzioni di conversione, faremo qualcosa di inaspettato: scriveremo un modulo di test che faccia eseguire a queste funzioni il proprio codice e verifichi che esse si comportino come vogliamo. Avete letto bene: andremo a scrivere del codice per verificare altro codice che non abbiamo ancora scritto.
- 7.3. Verificare i casi di successo

  La parte fondamentale della verifica delle unità di codice è costruire i singoli test. Un test risponde ad una singola domanda sul codice che si sta verificando.
- 7.4. Verificare i casi di errore

  Non è abbastanza verificare che la nostra funzione abbia successo quando gli input sono validi; occorre anche verificare che la funzione vada in errore quando riceve input non validi. E non basta che vada in errore: deve farlo nel modo che ci si aspetta.
- 7.5. Verificare la consistenza

Spesso vi capiterà di scoprire che un'unità di codice contiene un insieme di funzioni reciproche, di solito in forma di funzioni di conversione, laddove una converte A in B e l'altra converte B in A. In questi casi, è utile creare un "test di consistenza" per essere sicuri che si possa converire A in B e poi riconvertire B in A senza perdere precisione, incorrere in errori di arrotondamento o in qualche altro malfunzionamento.

- 7.6. roman.py, fase 1

  Ora che i nostri test delle unità di codice sono pronti, è tempo di cominciare a scrivere il codice che stiamo cercando di verificare con i nostri test. Faremo questo in più fasi, in modo che si possa osservare dapprima come tutti i test falliscano, e poi come a poco a poco abbiano successo man mano che riempiamo gli spazi vuoti all'interno del modulo `roman.py`.
- 7.7. roman.py, fase 2

  Ora che abbiamo delineato l'infrastruttura del modulo `roman`, è tempo di cominciare a scrivere il codice e passare con successo qualche test.
- 7.8. roman.py, fase 3

  Adesso che la funzione `toRoman` si comporta correttamente con input validi (numeri da `1` a `3999`), è tempo di fare in modo che lo faccia anche con input non validi (qualsiasi altra cosa).
- 7.9. roman.py, fase 4

  Ora che la funzione `toRoman` è completa, è tempo di cominciare a scrivere il codice di `fromRoman`. Grazie alla nostra struttura dati dettagliata che mappa i singoli numeri romani elementari negli interi corrispondenti, la cosa non è più difficile dell'aver scritto la funzione `toRoman`.
- 7.10. roman.py, fase 5

  Ora che `fromRoman` funziona correttamente con input validi, è tempo di far combaciare l'ultimo pezzo del puzzle: farla funzionare con input non validi. Senza troppi giri di parole cerchiamo di osservare una stringa e di determinare se è un numero romano valido. Questo è ancor più difficoltoso se paragonato agli input validi in `toRoman`, ma noi abbiamo a disposizione un potente strumento a disposizione: le espressioni regolari
- 7.11. Come gestire gli errori di programmazione

  A dispetto dei nostri migliori sforzi per scrivere test completi per le unità di codice, capita di fare degli errori di programmazione, in gergo chiamati bachi ("bug"). Un baco corrisponde ad un test che non è stato ancora scritto.
- 7.12. Gestire il cambiamento di requisiti

  A dispetto dei vostri migliori sforzi di bloccare i vostri clienti in un angolo per tirargli fuori gli esatti requisiti del software da sviluppare, sotto la minaccia di sottoporli ad orribili operazioni con forbici e cera bollente, i requisiti cambieranno lo stesso. Molti clienti non sanno cosa vogliono fino a quando non lo vedono, ed anche allora, non sono così bravi a dettagliare esattamente il progetto, a tal punto da fornire indicazioni che possano risultare utili. Ed anche nel caso lo siano, chiederanno sicuramente di più per la prossima versione del software. Siate quindi preparati ad aggiornare i vostri test quando i requisiti cambieranno.
- 7.13. Rifattorizzazione

  La cosa migliore nel fare test esaustivi delle unità di codice non è la sensazione piacevole che si ha quando tutti i test hanno finalmente successo, e neanche la soddisfazione di quando qualcun altro ti rimprovera di aver scombinato il loro codice e tu puoi effettivamente *provare* che non è vero. La cosa migliore nell'effettuare i test delle unità di codice è la sensazione che

ti lascia la libertà di rifattorizzare senza provare rimorsi.

- 7.14. Postscritto

  Un lettore intelligente dopo aver letto la sezione precedente ha subito fatto il passo successivo. Il più grosso grattacapo (e peggioratore delle prestazioni) nel programma com'è scritto al momento è l'espressione regolare, che è necessaria perché non abbiamo altro modo di decomporre un numero romano. Ma ci sono solo 5000 numeri romani: perché non ci costruiamo all'inizio una tavola di riferimento e non usiamo quella? Questa idea è ancora migliore una volta capito che è possibile rimuovere del tutto l'uso delle espressioni regolari. Una volta costruita la tavola di riferimento per convertire interi in numeri romani, è possibile costruire la tavola inversa per convertire i numeri romani in interi.
- 7.15. Sommario

  Il test delle unità di codice è un concetto forte che, se opportunamente implementato, può ridurre sia i costi di manutenzione che aumentare la flessibilità in ogni progetto a lungo termine. È però anche importante capire che i test delle unità di codice non sono una panacea, una bacchetta magica che risolve tutti i problemi, o una pallottola d'argento. Scrivere dei buoni test è difficile, e mantenerli aggiornati richiede disciplina (specialmente quando i clienti vi stanno chiedendo a gran voce la soluzione di qualche problema critico nel software). I test delle unità di codice non sono un sostituto per le altre forme di verifica del codice, tra cui i test funzionali, i test di integrazione e i test di accettazione. Rappresentano tuttavia una pratica realizzabile, e funzionano, ed una volta che li avrete provati vi chiederete come avete fatto ad andare avanti senza di loro.

Capitolo 8. Programmazione orientata ai dati

- 8.1. Immergersi

  Nel capitolo relativo al Test delle unità di codice, ne abbiamo discusso la filosofia e ne abbiamo vista l'implementazione in Python. Questo capitolo si dedicherà a tecniche specifiche di Python, più avanzate, basate sul modulo `unittest`. Se non avete letto il capitolo Test delle unità di codice, vi perderete a metà lettura. Siete stati avvertiti.
- 8.2. Trovare il percorso

  Quando lanciate gli script Python da linea di comando, qualche volta torna utile sapere dov'è localizzato lo script corrente sul disco fisso.
- 8.3. Filtrare liste rivisitate

  Avete già familiarità con l'utilizzo delle list comprehensions per filtrare le liste. C'è un'altro modo per raggiungere lo stesso risultato, che alcune persone considerano più espressivo.
- 8.4. Rivisitazione della mappatura delle liste

  Siete già abituati ad usare la list comprehensions per mappare una lista in un altra. C'è un altro modo per fare la stessa cosa, usando la funzione built−in `map`. Funziona in modo molto simile alla funzione `filter`.
- 8.5. Programmazione data−centrica

  Per adesso probabilmente vi starete grattando la testa chiedendovi perché questo modo di fare sia migliore rispetto all'usare un ciclo `for` e chiamate dirette a funzioni. Ed è una domanda perfettamente legittima. È principalmente una questione di prospettiva. Usare `map` e `filter` vi obbliga a focalizzare l'attenzione sui dati.
- 8.6. Importare dinamicamente i moduli

Ok, abbiamo filosofeggiato abbastanza. Vediamo come si importano dinamicamente i moduli.

- 8.7. Mettere assieme il tutto (parte 1)

  Ora abbiamo imparato abbastanza da poter scomporre le prime sette linee dell'esempio di questo capitolo: leggere una directory ed importare alcuni dei moduli che vi sono contenuti.

- 8.8. Dentro PyUnit

  Spiacente, questa parte del capitolo ancora non è stata scritta. Provate a controllare http://diveintopython.org/ per gli aggiornamenti.

Appendice C. Consigli e trucchi

Capitolo 1. Installare Python

Capitolo 2. Conoscere Python

- 2.1. Immergersi

  Suggerimento: Lanciate il modulo (Windows)
  Nel Python IDE di Windows, potere lanciare un modulo con File−>Run... (Ctrl−R). File−>Run... (**Ctrl−R**). L'output viene mostrato nella finestra interattiva.

  Suggerimento: Lanciate il modulo (Mac OS)
  Nel Python IDE di Mac OS, potete lanciare un modulo con Python−>Run window... (**Cmd−R**), ma c'è prima un'opzione importante che dovete settare. Aprite il modulo nell'IDE, fate comparire il menu delle opzioni dei moduli cliccando sul triangolo nero posto nell'angolo in alto a destra della finestra e siate sicuri che "Run as __main__" sia settato. Questa impostazione verrà salvata con il modulo, così avrete bisogno di impostarla una volta sola per ogni modulo.

  Suggerimento: Lanciate il modulo (UNIX)
  Sui sistemi UNIX−compatibili (incluso Mac OS X), potete lanciare un modulo direttamente dalla linea di comando: `python odbchelper.py`
- 2.2. Dichiarare le funzioni

  Nota: Python contro Visual Basic: ritornare un valore
  Nel Visual Basic, le funzioni (che ritornano un valore) iniziano con `function`, e le procedure (che non ritornano un valore) iniziano con `sub`. Non esistono procedure in Python. Sono tutte funzioni, ritornano un valore, anche se è `None` ed iniziano tutte con `def`.

  Nota: Python contro Java: ritornare un valore
  In Java, C++, ed in altri linguaggi con tipi di dato statici, dovete specificare il tipo del valore di ritorno della funzione e di ogni suo argomento. In Python non si specifica mai espressamente nessun tipo di dato. A seconda del valore che assegnerete, Python terrà conto del tipo di dato internamente.
- 2.3. Documentare le funzioni

  Nota: Python contro Perl: citazione
  Le triple virgolette sono anche un semplice modo per definire una stringa con singole e doppie virgolette, come il `qq/.../` del Perl.

  Nota: Perché le doc string sono la cosa giusta
  Molti IDE di Python usano la `doc string` per rendere disponibile una documentazione context−sensitive, così quando scrivete il nome di una funzione, la sua `doc string` appare come tooltip. Questo può essere incredibilmente utile, ma la sua qualità si basa unicamente sulla `doc string` che avete scritto.
- 2.4. Tutto è un oggetto

  Nota: Python contro Perl: import
  `import` in Python è come `require` in Perl. Una volta che avete importato un modulo Python, accedete alle sue funzioni con *`module.function`*; una volta che avete richiesto un modulo Perl, accedete alle sue funzioni con *`modulo::funzione`*.
- 2.5. Indentare il codice

  Nota: Python contro Java: separazione degli statement
  Python usa il ritorno a capo per separare le istruzioni e i due punti e l'indentazione per separare i blocchi di codice. C++ e Java usano un punto e virgola per separare le istruzioni e

le parentesi graffe per separare i blocchi di codice.

- 2.6. Testare i moduli

  Nota: Python contro C: confronto e assegnamento
  Come il C, Python usa `==` per i confronti e `=` per gli assegnamenti. Al contrario del C, Python non supporta gli assegnamenti in una singola riga, così non c'è modo di assegnare per errore il valore che pensavate di comparare.

  Suggerimento: if __name__ su Mac OS
  In MacPython c'è un passaggio ulteriore per far funzionare il trucco `if __name__`. Fate apparire le opzioni del modulo cliccando sul triangolo nero, nell'angolo in altro a destra della finestra e assicuratevi che "Esegui come __main__" sia selezionato.

- 2.7. Introduzione ai dizionari

  Nota: Python contro Perl: dizionari
  Un dizionario in Python è come una hash in Perl. In Perl, le variabili che memorizzano degli hash cominciano sempre con il carattere `%`; in Python, le variabili possono avere qualsiasi nome, e Python tiene traccia del loro tipo internamente.

  Nota: Python contro Java: dizionari
  Un dizionario in Python è come l'istanza di una classe `Hashtable` in Java.

  Nota: Python contro Visual Basic: dizionari
  Un dizionario Python è come l'istanza di un oggetto `Scripting.Dictionary` in Visual Basic.

  Nota: I dizionari non sono ordinati
  I dizionari non hanno il concetto di ordinamento tra elementi. È errato dire che gli elementi sono "disordinati"; sono semplicemente non ordinati. Si tratta di un'importante distinzione che vi darà noia quando vorrete accedere agli elementi di un dizionario in un ordine specifico e ripetibile (ad esempio in ordine alfabetico di chiave). Ci sono dei modi per farlo, semplicemente non sono predefiniti nel dizionario.

- 2.8. Introduzione alle liste

  Nota: Python contro Perl: liste
  Una lista in Python è come un array in Perl. In Perl, le variabili che contengono array hanno un nome che inizia sempre con il carattere `@`; in Python le variabili possono avere qualunque nome, è il linguaggio che tiene traccia internamente del tipo di dato.

  Nota: Python contro Java: liste
  Una lista in Python è molto più di un array in Java (sebbene possa essere usato allo stesso modo, se davvero non volete altro). Un'analogia migliore sarebbe la classe `Vector`, che può contenere oggetti di tipo arbitrario e si espande automaticamente quando vi si aggiungono nuovi elementi.

  Nota: Cos'è vero in Python?
  Prima della versione 2.2.1, Python non aveva un tipo di dato booleano distinto dai numeri interi. Per compensare, Python accettava quasi tutto in un contesto booleano (come un comando `if`), secondo la seguente regola: `0` è falso; tutti gli altri numeri sono veri. Una stringa vuota (`""`) è falsa; tutte le altre stringhe sono vere. Una lista vuota (`[]`) è falsa; tutte le altre liste sono vere. Una tupla vuota (`()`) è falsa; tutte le altre tuple sono vere. Un dizionario vuoto (`{}`) è falso; tutti gli altri dizionari sono veri. Queste regole valgono ancora in Python 2.2.1 e oltre, ma ora si può usare anche un valore booleano vero e proprio, che ha valore `True` o `False`. Notate le maiuscole; questi valori, come tutto il resto in Python, sono case−sensitive.

- 2.9. Introduzione alle tuple

  Nota: Tuple all'interno di liste e di tuple
  Le tuple possono essere convertite in liste e viceversa. La funzione built−in `tuple` prende una lista e ritorna una tupla con gli stessi elementi, mentre la funzione `list` prende una tupla e ritorna una lista. In effetti, `tuple` congela una lista e `list` scongela una tupla.
- 2.10. Definire le variabili

  Nota: Comandi multilinea
  Quando un comando è diviso su più linee attraverso il continuatore di linea ("\"), la linea seguente può essere indentata in ogni maniera; Le rigide regole di indentazione di Python non vengono applicate. Se il vostro Python IDE auto−indenta la linea continuata, probabilmente dovreste accettare il suo default a meno che non abbiate ottime ragioni per non farlo.

  Nota: Comandi impliciti multilinea
  A rigor di termini, espressioni fra parentesi, parentesi quadre o graffe (come per la definizione di un dizionario), possono essere divise in linee multiple con o senza il carattere di continuazione linea("\"). Io preferisco includere il backslash anche quando non è richiesto perché credo che renda il codice più leggibile, ma è solo una questione di stile.
- 2.12. Formattare le stringhe

  Nota: Python contro C: formattazione delle stringhe
  La formattazione delle stringhe in Python utilizza la stessa sintassi della funzione C `sprintf`.
- 2.14. Concatenare liste e suddividere stringhe

  Importante: non potete concatenare oggetti che non siano stringhe.
  `join` funziona solamente su liste di stringhe, non applica alcuna forzatura sul tipo. Concatenare una lista che contiene uno o più oggetti che non sono di tipo stringa genererà un'eccezione.

  Nota: ricercare con split
  `anystring``.split (`*`separatore`*`, 1)` è un'utile tecnica quando volete cercare in una stringa la presenza di una particolare sottostringa e quindi utilizzare tutto ciò che c'è prima di detta sottostringa (che va a finire nel primo elemento della lista ritornata) e tutto quello che c'è dopo (che va a finire nel secondo elemento).

Capitolo 3. La potenza dell'introspezione

- 3.2. Argomenti opzionali ed argomenti con nome

  Nota: La chiamata a funzione è flessibile
  L'unica cosa che dovete fare per chiamare una funzione è specificare un valore (in qualche modo) per ogni argomento richiesto; il modo e l'ordine in cui lo fate è a vostra discrezione.
- 3.3. type, str, dir, ed altre funzioni built−in

  Nota: Python è auto−documentante
  Python è dotato di un eccellente manuale di riferimento, che dovreste usare spesso per conoscere tutti i moduli che Python ha da offrire. Ma mentre nella maggior parte dei linguaggi vi ritroverete a dover tornare spesso sul manuale (o pagine man o ... Dio vi aiuti, MSDN) per ricordare come si usano questi moduli, Python è largamente auto−documentante.
- 3.6. Le particolarità degli operatori and e or

  Importante: Per usare and−or in modo efficace
  Il trucchetto `and−or`, cioè *`bool`* `and a or b`, non funziona come il costrutto C *`bool`* `? a : b` quando `a` ha un valore falso in un contesto booleano.
- 3.7. Usare le funzioni lambda

  Nota: lambda è opzionale
  l'utilizzo o meno delle funzioni `lambda` è questione di stile. Usarle non è mai necessario; in

ogni caso in cui vengono usate è possibile definire una funzione normale e usarla al posto della funzione `lambda`. Le funzioni `lambda` sono comode, ad esempio, nei casi in cui si voglia incapsulare codice specifico, non riutilizzato, senza riempire il programma di piccolissime funzioni.

- 3.8. Unire il tutto

  Nota: Python contro SQL: confrontare valori nulli
  In SQL, dovete usare `IS NULL` invece di `= NULL` per confrontare un valore nullo. In Python, potete usare indifferentemente `== None` o `is None`, ma `is None` è più efficiente.

Capitolo 4. Una struttura orientata agli oggetti

- 4.2. Importare i moduli usando from module import

  Nota: Python contro Perl: from module import
  `from *module* import *` in Python è simile allo `use *module*` in Perl; `import *module*` nel Python è come il `require *module*` del Perl.

  Nota: Python contro Java: from module import
  `from *module* import *` in Python è come `import *module*.*` in Java; `import *module*` in Python è simile a `import *module*` del Java.

- 4.3. Definire classi

  Nota: Python contro Java: pass
  Lo statement `pass` in Python è come un paio di graffe vuote (`{}`) in Java od in C.

  Nota: Python contro Java: antenati
  In Python, l'antenato di una classe è semplicemente elencato tra parentesi subito dopo il nome della classe. Non c'è alcuna keyword come la `extends` di Java.

  Nota: Ereditarietà multipla
  Anche se non la discuterò in profondità nel libro, Python supporta l'ereditarietà multipla. Nelle parentesi che seguono il nome della classe, potete elencare quanti antenati volete, separati da virgole.

  Nota: Python contro Java: self
  Per convenzione, il primo argomento di ogni metodo di una classe (il riferimento all'istanza corrente) viene chiamato `self`. Questo argomento ricopre il ruolo della parola riservata `this` in C++ o Java, ma `self` non è una parola riservata in Python, è semplicemente una convenzione sui nomi. Non di meno, vi prego di non chiamarlo diversamente da `self`; è una convenzione molto forte.

  Nota: Quando usare self
  Quando definite i vostri metodi nella classe, *dovete* elencare esplicitamente `self` come il primo argomento di ogni metodo, incluso `__init__`. Quando chiamate il metodo di una classe antenata dall'interno della vostra classe, *dovete* includere l'argomento `self`. Invece, quando chiamate i metodi della vostra classe dall'esterno, non dovete specificare affatto l'argomento `self`, lo saltate per intero e Python automaticamente aggiunge il riferimento all'istanza per voi. Temo che inizialmente possa confondere; non è proprio inconsistente ma può sembrarlo perché fa affidamento su una distinzione (tra metodi bound ed unbound) che ancora non conoscete.

  Nota: i metodi __init__
  i metodi `__init__` sono opzionali, ma quando ne definite uno, dovete ricordarvi di

chiamare esplicitamente il metodo `__init__` dell'antenato. Questo è generalmente vero; quando un discendente vuole estendere il comportamento di un antenato, il metodo del discendente deve esplicitamente chiamare il metodo dell'antenato nel momento opportuno e con gli opportuni argomenti.

- 4.4. Istanziare classi

  Nota: Python contro Java: istanziazione di classi
  In Python, semplicemente chiamate una classe come se fosse una funzione per creare una sua nuova istanza. Non c'è alcun operatore esplicito new come in C++ o in Java.

- 4.5. UserDict: una classe wrapper

  Suggerimento: Aprire i moduli rapidamente
  Nella IDE di Python su Windows, potete aprire rapidamente qualunque modulo nel vostro library path usando File−>Locate... (**Ctrl−L**).

  Nota: Python contro Java: overload di funzioni
  Java e Powerbuilder supportano l'overload di funzioni per lista di argomenti, cioè una classe può avere più metodi con lo stesso nome, ma diverso numero di argomenti o argomenti di tipo diverso. Altri linguaggi (principalmente PL/SQL) supportano l'overload di funzioni per nome di argomento; cioè una classe può avere più metodi con lo stesso nome e lo stesso numero di argomenti dello stesso tipo, ma i nomi degli argomenti sono diversi. Python non supporta nessuno di questi; non ha nessuna forma di overload di funzione. I metodi sono definiti solamente dal loro nome e ci può essere solamente un metodo per ogni classe con un dato nome. Così, se una classe discendente ha un metodo `__init__`, questo sovrascrive *sempre* il metodo `__init__` dell'antenato, anche se il discendente lo definisce con una lista di argomenti diversa. La stessa regola si applica ad ogni altro metodo.

  Nota: Guido sulle classi derivate
  Guido, l'autore originale di Python, spiega l'override dei metodi in questo modo: "Classi derivate possono sovrascrivere i metodi delle loro classi base. Siccome i metodi non hanno privilegi speciali quando chiamano altri metodi dello stesso oggetto, un metodo di una classe base che chiama un altro metodo definito nella stessa classe base, può infatti finire per chiamare un metodo di una classe derivata che lo sovrascrive. (Per i programmatori C++ tutti i metodi in Python sono effettivamente virtual.)" Se questo per voi non ha senso (confonde un sacco pure me), sentitevi liberi di ignorarlo. Penso che lo capirò più avanti.

  Nota: Inizializzare sempre gli attributi
  Assegnate sempre un valore iniziale a tutti gli attributi di un'istanza nel metodo `__init__`. Vi risparmierà ore di debugging più tardi, spese a tracciare tutte le eccezioni `AttributeError` che genera il programma perché state referenziando degli attributi non inizializzati (e quindi inesistenti).

- 4.6. Metodi speciali per le classi

  Nota: Chiamare altri metodi di classe
  Quando accedete agli attributi di una classe, avete bisogno di qualificare il nome dell'attributo: `self.`*`attributo`*. Quando chiamate altri metodi di una classe, dovete qualificare il nome del metodo: `self.`*`metodo`*.

- 4.7. Metodi speciali di classe avanzati

  Nota: Python contro Java: uguaglianza ed identità
  In Java, determinate quando due variabili di tipo stringa referenziano la stessa memoria fisica utilizzando `str1 == str2`. Questa è chiamata *identità di oggetto*, ed è espressa in Python come `str1 is str2`. Per confrontare valori di tipo stringa in Java, dovreste usare `str1.equals(str2)`; in Python, usereste `str1 == str2`. Programmatori Java a cui è stato insegnato a credere che il mondo è un posto migliore in quanto `==` in Java confronta l'identità invece del valore potrebbero avere qualche difficoltà nell'acquisire questo nuovo

"concetto" di Python.

Nota: Modelli fisici contro modelli logici
Mentre altri linguaggi orientati agli oggetti vi lasciano definire il modello fisico di un oggetto ("questo oggetto ha il metodo `GetLength`"), i metodi speciali di classe Python come `__len__` vi permettono di definire il modello logico di un oggetto ("questo oggetto ha una lunghezza").

- 4.8. Attributi di classe

  Nota: Attributi di classe in Java
  In Java, sia le variabili statiche (chiamate attributi di classe in Python) che le variabili di istanza (chiamate attributi dato in Python), sono definite immediatamente dopo la definizione della classe (il primo con la parola chiave `static`, il secondo senza). In Python, solo gli attributi di classe posso essere definiti così; gli attributi dato sono definiti nel metodo `__init__`.

- 4.9. Funzioni private

  Nota: Cos'è privato in Python?
  Se il nome di una funzione, metodo di classe o attributo in Python inizia con (ma non finisce con) due underscore, è privato; ogni altra cosa è pubblica.

  Nota: Convenzioni sui nomi dei metodi
  In Python, tutti i metodi speciali (come `__setitem__`) e gli attributi built−in (come `__doc__`) seguono una convenzione standard sui nomi: iniziano e finiscono con due underscore. Non nominate i vostri metodi e attributi in questa maniera, servirà solo a confondervi. Inoltre, in futuro, potrebbe anche confondere altri che leggeranno il vostro codice.

  Nota: Metodi non protetti
  Python non ha il concetto di metodi protetti di classe (accessibili solo nella loro stessa classe ed in quelle discendenti). I metodi di classe sono o privati (accessibili unicamente nella loro stessa classe) o pubblici (accessibili ovunque).

- 4.10. Gestire le eccezioni

  Nota: Python contro Java: gestire le eccezioni
  Python usa `try...except` per gestire le eccezioni e `raise` per generarle. Java e C++ usano `try...catch` per gestirle e `throw` per generarle.

- 4.14. Il modulo os

  Nota: Quando usare il modulo os
  Ogniqualvolta sia possibile, dovreste usare le funzioni in `os` e `os.path` per file, directory e manipolazione dei percorsi. Questi moduli sono dei wrapper ai moduli specifici per piattaforma, per cui funzioni come `os.path.split` funzionano su UNIX, Windows, Mac OS ed ogni altra possibile piattaforma supportata da Python.

Capitolo 5. Elaborare HTML

- 5.2. Introduciamo sgmllib.py

  Importante: Evoluzione del linguaggio: DOCTYPE
  Python 2.0 ha un bug per il quale `SGMLParser` non riconosce del tutto le dichiarazioni (`handle_decl` non viene mai chiamato), questo implica che i `DOCTYPE` vengano ignorati senza che sia segnalato. Questo è stato corretto in Python 2.1.

  Suggerimento: Specificare argomenti da linea di comando in Windows
  Nella IDE di Python su Windows, potete specificare argomenti da linea di comando nella finestra di dialogo "Run script". Argomenti multipli vanno separati da spazi.

- 5.4. Introdurre BaseHTMLProcessor.py

  Importante: Processare il codice HTML con uno script migliorato
  La specifica HTML richiede che tutto ciò che non è HTML (come JavaScript lato client) debba essere racchiuso tra commenti HTML, ma non in tutte le pagine ciò è stato fatto (e tutti i moderni browsers chiudono un occhio in merito). `BaseHTMLProcessor` non perdona; se uno script è inserito in modo improprio, verrà analizzato come se fosse codice HTML. Per esempio, se lo script contiene i simboli di minore e maggiore, `SGMLParser` potrebbe pensare, sbagliando, di aver trovato tag e attributi. `SGMLParser` converte sempre i tag e i nomi degli attributi in lettere minuscole, il che potrebbe bloccare lo script, e `BaseHTMLProcessor` racchiude sempre i valori tra doppi apici (anche se originariamente il documento HTML usava apici singoli o niente del tutto), cosa che di sicuro interromperebbe lo script. Proteggete sempre i vostri script lato client inserendoli dentro commenti HTML.
- 5.5. locals e globals

  Importante: Evoluzione del linguaggio: gli scope nidificati
  Python 2.2 introdusse un subdolo ma importante cambiamento che modificò l'ordine di ricerca dei namespace: gli scope nidificati. Nelle versioni di Python precedenti al 2.2, quando vi riferite ad una variabile dentro ad una funzione nidificata o una funzione `lambda`, Python ricercherà quella variabile nel namespace corrente della funzione, e poi nel namespace del modulo. Python 2.2 ricercherà la variabile nel namespace della funzione corrente, *poi nel namespace della funzione genitore*, e poi nel namespace del modulo. Python 2.1 può lavorare in ogni modo; predefinitamente, lavora come Python 2.0, ma potete aggiugere la seguente linea di codice in cima al vostro modulo per farlo funzionare come Python 2.2:

  ```
  from __future__ import nested_scopes
  ```

  Nota: Accedere dinamicamente alle variabili
  Usando le funzioni `locals` e `globals`, potete ottenere il valore di variabili arbitrarie dinamicamente, rendendo disponibile il nome della variabile come una stringa. Questo rispecchia la funzionalità della funzione `getattr`, che vi permette di accedere a funzioni arbitrarie dinamicamente, rendendo disponibile il nome della funzione come stringa.
- 5.6. Formattazione di stringhe basata su dizionario

  Importante: Problemi di performance con locals
  Usare una formattazione di stringhe basata su dizionari con `locals` è un modo conveniente per ottenere una formattazione di stringhe complessa in maniera più leggibile, ma ha un prezzo. C'è una minima perdita di performance nell'effettuare una chiamata a `locals`, in quanto `locals` costruisce una copia dello spazio dei nomi locale.

Capitolo 6. Elaborare XML

- 6.2. Package

  Nota: Di che cosa è fatto un package
  Un package è una directory con il file speciale `__init__.py` dentro. Il file `__init__.py` definisce gli attributi ed i metodi del package. Non deve obbligatoriamente definire nulla; può essere un file vuoto, ma deve esistere. Ma se `__init__.py` non esiste, la directory è soltanto una directory, non un package e non può essere importata, contenere moduli o package annidati.
- 6.6. Accedere agli attributi di un elemento

  Nota: Attributi XML ed attributi Python
  Questa sezione potrebbe risultare un po' confusa a causa della sovrapposizione nella terminologia. Gli elementi in un documento XML hanno degli attributi ed anche gli oggetti Python hanno degli attributi. Quando analizziamo un documento XML, otteniamo un gruppo di oggetti Python che rappresentano tutti i pezzi del documento XML ed alcuni di questi

oggetti rappresentano gli attributi degli elementi XML. Ma anche gli oggetti (Python) che rappresentano gli attributi (XML) hanno attributi, che vengono utilizzati per accedere alle varie parti degli attributi (XML) che l'oggetto rappresenta. Ve l'ho detto che vi avrebbe confuso. Sono aperto a suggerimenti su come distinguere le due cose in maniera più chiara.

Nota: Gli attributi non hanno ordinamento
Come un dizionario, gli attributi di un elemento XML non hanno ordinamento. Gli attributi *possono essere* elencati in un certo ordine nel documento XML originale e gli oggetti `Attr` *possono essere* elencati in un certo ordine quando il documento XML viene interpretato in oggetti Python, ma questi ordinamenti sono arbitrari e non dovrebbero avere alcun significato particolare. Dovreste sempre accedere agli attributi in base al nome, come nel caso delle chiavi di un dizionario.

Capitolo 7. Test delle unità di codice

- 7.2. Introduzione al modulo romantest.py
  Nota: Avete unittest?
  `unittest` è incluso con Python 2.1 e successivi. Gli utilizzatori di Python 2.0 possono scaricarlo da `pyunit.sourceforge.net`.
- 7.8. roman.py, fase 3
  Nota: Sapere quando si deve smettere di scrivere il programma
  La cosa più importante che test delle unità di codice, condotti in modo esaustivo, possano comunicarci è il momento in cui si deve smettere di scrivere un programma. Quando tutti i test di un modulo hanno successo, è il momento di smettere di scrivere quel modulo.
- 7.10. roman.py, fase 5
  Nota: Cosa fare quando tutti i test passano con successo
  Quando tutti i test passano con successo, smettete di scrivere codice.
- 7.13. Rifattorizzazione
  Nota: Compilare le espressioni regolari
  Ogni qualvolta si ha intenzione di usare più di una volta una espressione regolare, la si dovrebbe prima compilare per ottenerne il corrispondente oggetto pattern, e quindi chiamare direttamente i metodi di tale oggetto.

Capitolo 8. Programmazione orientata ai dati

- 8.2. Trovare il percorso
  Nota: os.path.abspath non effettua controlli sui percorsi
  I percorsi e i nomi di file che passate a `os.path.abspath` non è necessario che esistano.

  Nota: Normalizzare i percorsi
  `os.path.abspath` non realizza solo percorsi completi, si occupa anche di normalizzarli. Se siete nella directory `/usr/`, `os.path.abspath('bin/../local/bin')` ritornerà `/usr/local/bin`. Se volete solo normalizzare un percorso senza trasformarlo in un percorso completo, usate invece `os.path.normpath`.

  Nota: os.path.abspath è cross−platform
  Come altre funzioni nei moduli `os` e `os.path`, anche `os.path.abspath` è multi−piattaforma. I vostri risultati sembreranno leggermente differenti dai miei esempi se state lavorando su Windows (che usa i backslash come separatori di percorso) o sul Mac OS (che usa due punti), ma funzionano comunque. Questo è il punto cruciale del modulo `os`.

Appendice D. Elenco degli esempi

Capitolo 1. Installare Python

- 1.2. Python su Windows
  - ♦ Esempio 1.1. La IDE di ActivePython
  - ♦ Esempio 1.2. IDLE (Python GUI)
- 1.3. Python su Mac OS X
  - ♦ Esempio 1.3. Usare la versione di Python preinstallata su Mac OS X
  - ♦ Esempio 1.4. L'IDE MacPython in Mac OS X
  - ♦ Esempio 1.5. Due versioni di Python
- 1.4. Python su Mac OS 9
  - ♦ Esempio 1.6. La IDE MacPython su Mac OS 9
- 1.5. Python su RedHat Linux
  - ♦ Esempio 1.7. Installazione su RedHat Linux 9
- 1.6. Python su Debian GNU/Linux
  - ♦ Esempio 1.8. Installare su Debian GNU/Linux
- 1.7. Installare dai sorgenti
  - ♦ Esempio 1.9. Installare dai sorgenti
- 1.8. La shell interattiva
  - ♦ Esempio 1.10. Primi passi nella shell interattiva

Capitolo 2. Conoscere Python

- 2.1. Immergersi
  - ♦ Esempio 2.1. odbchelper.py
  - ♦ Esempio 2.2. Output di odbchelper.py
- 2.2. Dichiarare le funzioni
  - ♦ Esempio 2.3. Dichiarare la funzione buildConnectionString
- 2.3. Documentare le funzioni
  - ♦ Esempio 2.4. Definire la doc string della funzione buildConnectionString
- 2.4. Tutto è un oggetto
  - ♦ Esempio 2.5. Accedere alla doc string della funzione buildConnectionString
  - ♦ Esempio 2.6. Percorso di ricerca per l'importazione
- 2.5. Indentare il codice
  - ♦ Esempio 2.7. Indentazione della funzione buildConnectionString
- 2.6. Testare i moduli
  - ♦ Esempio 2.8. Il trucco if __name__
  - ♦ Esempio 2.9. Se importate il modulo __name__

- 2.7. Introduzione ai dizionari
    - Esempio 2.10. Definire un dizionario
    - Esempio 2.11. Modificare un dizionario
    - Esempio 2.12. Mischiare tipi di dato in un dizionario
    - Esempio 2.13. Cancellare elementi da un dizionario
    - Esempio 2.14. Le stringhe sono case−sensitive
- 2.8. Introduzione alle liste
    - Esempio 2.15. Definire una lista
    - Esempio 2.16. Indici di liste negativi
    - Esempio 2.17. Sezionare una lista.
    - Esempio 2.18. Sezionamento abbreviato
    - Esempio 2.19. Aggiungere elementi ad una lista
    - Esempio 2.20. Ricerca in una lista
    - Esempio 2.21. Togliere elementi da una lista
    - Esempio 2.22. Operatori su liste
- 2.9. Introduzione alle tuple
    - Esempio 2.23. Definire una tupla
    - Esempio 2.24. Le tuple non hanno metodi
- 2.10. Definire le variabili
    - Esempio 2.25. Definire la variabile myParams
    - Esempio 2.26. Riferirsi ad una variabile non assegnata (unbound, "slegata" n.d.T.)
- 2.11. Assegnare valori multipli in un colpo solo
    - Esempio 2.27. Assegnare valori multipli in un colpo solo
    - Esempio 2.28. Assegnare valori consecutivi
- 2.12. Formattare le stringhe
    - Esempio 2.29. Introduzione sulla formattazione delle stringhe
    - Esempio 2.30. Formattazione delle stringhe contro concatenazione
- 2.13. Mappare le liste
    - Esempio 2.31. Introduzione alle list comprehension
    - Esempio 2.32. List comprehension in buildConnectionString
    - Esempio 2.33. chiavi, valori ed elementi
    - Esempio 2.34. List comprehension in buildConnectionString, passo per passo
- 2.14. Concatenare liste e suddividere stringhe
    - Esempio 2.35. Concatenare una lista in buildConnectionString
    - Esempio 2.36. Output di odbchelper.py
    - Esempio 2.37. Suddividere una stringa
- 2.15. Sommario
    - Esempio 2.38. odbchelper.py
    - Esempio 2.39. Output di odbchelper.py

Capitolo 3. La potenza dell'introspezione

- 3.1. Immergersi

    - Esempio 3.1. apihelper.py
    - Esempio 3.2. Esempio di utilizzo di apihelper.py
    - Esempio 3.3. Utilizzo avanzato di apihelper.py
- 3.2. Argomenti opzionali ed argomenti con nome
    - Esempio 3.4. help, una funzione con due argomenti opzionali
    - Esempio 3.5. Chiamate valide per help
- 3.3. type, str, dir, ed altre funzioni built−in
    - Esempio 3.6. Introduzione a type
    - Esempio 3.7. Introduzione a str
    - Esempio 3.8. Introduzione a dir
    - Esempio 3.9. Introduzione a callable
    - Esempio 3.10. Attributi e funzioni built−in
- 3.4. Ottenere riferimenti agli oggetti usando getattr
    - Esempio 3.11. Introduzione a getattr
    - Esempio 3.12. getattr in apihelper.py
- 3.5. Filtrare le liste
    - Esempio 3.13. Sintassi per filtrare una lista
    - Esempio 3.14. Introduzione alle liste filtrate
    - Esempio 3.15. Filtrare una lista in apihelper.py
- 3.6. Le particolarità degli operatori and e or
    - Esempio 3.16. Introduzione all'operatore and
    - Esempio 3.17. Introduzione all'operatore or
    - Esempio 3.18. Introduzione al trucchetto and−or
    - Esempio 3.19. Quando il trucchetto and−or fallisce
    - Esempio 3.20. Usare il trucchetto and−or in modo sicuro
- 3.7. Usare le funzioni lambda
    - Esempio 3.21. Introduzione alle funzioni lambda
    - Esempio 3.22. funzioni lambda lambda in apihelper.py
    - Esempio 3.23. split senza argomenti
    - Esempio 3.24. Assegnare una funzione a una variabile
- 3.8. Unire il tutto
    - Esempio 3.25. Il succo di apihelper.py
    - Esempio 3.26. Ottenere dinamicamente una doc string
    - Esempio 3.27. Perchè usare str su una stringa di documentazione?
    - Esempio 3.28. Introduzione al metodo ljust
    - Esempio 3.29. Stampare una lista
    - Esempio 3.30. Il succo di apihelper.py, rivisitato
- 3.9. Sommario
    - Esempio 3.31. apihelper.py
    - Esempio 3.32. Output di apihelper.py

Capitolo 4. Una struttura orientata agli oggetti

- 4.1. Immergersi

- Esempio 4.1. fileinfo.py
- Esempio 4.2. Output di fileinfo.py

- 4.2. Importare i moduli usando from module import

- Esempio 4.3. Sintassi basilare di from module import
- Esempio 4.4. import module contro from module import

- 4.3. Definire classi

- Esempio 4.5. La più semplice classe Python
- Esempio 4.6. Definire la classe FileInfo
- Esempio 4.7. Inizializzazione della classe FileInfo
- Esempio 4.8. Realizzare la classe FileInfo

- 4.4. Istanziare classi

- Esempio 4.9. Creare un'istanza di FileInfo
- Esempio 4.10. Proviamo ad implementare un memory leak

- 4.5. UserDict: una classe wrapper

- Esempio 4.11. Definire la classe UserDict
- Esempio 4.12. Metodi comuni di UserDict

- 4.6. Metodi speciali per le classi

- Esempio 4.13. Il metodo speciale __getitem__
- Esempio 4.14. Il metodo speciale __setitem__
- Esempio 4.15. Sovrascrivere __setitem__ in MP3FileInfo
- Esempio 4.16. Impostare il nome di un MP3FileInfo

- 4.7. Metodi speciali di classe avanzati

- Esempio 4.17. Altri metodi speciali in UserDict

- 4.8. Attributi di classe

- Esempio 4.18. Introdurre gli attributi di classe
- Esempio 4.19. Modificare gli attributi di classe

- 4.9. Funzioni private

- Esempio 4.20. Provare ad invocare un metodo privato

- 4.10. Gestire le eccezioni

- Esempio 4.21. Aprire un file inesistente
- Esempio 4.22. Supportare le funzionalità specifiche per piattaforma

- 4.11. Oggetti file

- Esempio 4.23. Aprire un file
- Esempio 4.24. Leggere un file
- Esempio 4.25. Chiudere un file
- Esempio 4.26. Oggetti file in MP3FileInfo

- 4.12. Cicli for

- Esempio 4.27. Introduzione al ciclo for
- Esempio 4.28. Semplici contatori
- Esempio 4.29. Iterare sugli elementi di un dizionario
- Esempio 4.30. Ciclo for in MP3FileInfo

- 4.13. Ancora sui moduli
    - ♦ Esempio 4.31. Introduzione a sys.modules
    - ♦ Esempio 4.32. Usare sys.modules
    - ♦ Esempio 4.33. L'attributo di classe __module__
    - ♦ Esempio 4.34. Uso di sys.modules in fileinfo.py
- 4.14. Il modulo os
    - ♦ Esempio 4.35. Costruire pathnames
    - ♦ Esempio 4.36. Dividere i pathnames
    - ♦ Esempio 4.37. Elencare directory
    - ♦ Esempio 4.38. Elencare directory in fileinfo.py
- 4.15. Mettere tutto insieme
    - ♦ Esempio 4.39. listDirectory
- 4.16. Sommario
    - ♦ Esempio 4.40. fileinfo.py

Capitolo 5. Elaborare HTML

- 5.1. Immergersi
    - ♦ Esempio 5.1. BaseHTMLProcessor.py
    - ♦ Esempio 5.2. dialect.py
    - ♦ Esempio 5.3. Output di dialect.py
- 5.2. Introduciamo sgmllib.py
    - ♦ Esempio 5.4. Test di esempio di sgmllib.py
- 5.3. Estrarre informazioni da documenti HTML
    - ♦ Esempio 5.5. Introdurre urllib
    - ♦ Esempio 5.6. Introdurre urllister.py
    - ♦ Esempio 5.7. Utilizzare urllister.py
- 5.4. Introdurre BaseHTMLProcessor.py
    - ♦ Esempio 5.8. Introdurre BaseHTMLProcessor
    - ♦ Esempio 5.9. BaseHTMLProcessor output
- 5.5. locals e globals
    - ♦ Esempio 5.10. Introdurre locals
    - ♦ Esempio 5.11. Introdurre globals
    - ♦ Esempio 5.12. locals è in sola lettura, globals no
- 5.6. Formattazione di stringhe basata su dizionario
    - ♦ Esempio 5.13. Introduzione alla formattazione di stringhe basata su dizionario
    - ♦ Esempio 5.14. Formattazione di stringhe basata su dizionario in BaseHTMLProcessor.py
- 5.7. Virgolettare i valori degli attributi
    - ♦ Esempio 5.15. Valori degli attributi tra virgolette
- 5.8. Introduzione al modulo dialect.py

    - Esempio 5.16. Gestire tag specifici
    - Esempio 5.17. SGMLParser
    - Esempio 5.18. Ridefinizione del metodo handle_data
- 5.9. Introduzione alle espressioni regolari
    - Esempio 5.19. Cercare corrispondenze alla fine di una stringa
    - Esempio 5.20. Cercare la corrispondenza con parole complete
- 5.10. Mettere tutto insieme
    - Esempio 5.21. La funzione translate, parte prima
    - Esempio 5.22. La funzione translate, parte seconda: sempre più curioso
    - Esempio 5.23. La funzione translate, parte terza

Capitolo 6. Elaborare XML

- 6.1. Immergersi
    - Esempio 6.1. kgp.py
    - Esempio 6.2. toolbox.py
    - Esempio 6.3. Esempio di output di kgp.py
    - Esempio 6.4. Semplice output di kgp.py
- 6.2. Package
    - Esempio 6.5. Caricare un documento XML (una rapida occhiata)
    - Esempio 6.6. Disposizione dei file di un package
    - Esempio 6.7. I package sono anche dei moduli
- 6.3. Analizzare XML
    - Esempio 6.8. Caricare un documento XML (questa volta per davvero)
    - Esempio 6.9. Ottenere i nodi figli
    - Esempio 6.10. toxml funziona su ogni nodo
    - Esempio 6.11. I nodi figli possono essere di testo
    - Esempio 6.12. Tirare fuori tutti i nodi di testo
- 6.4. Unicode
    - Esempio 6.13. Introduzione sull'unicode
    - Esempio 6.14. immagazzinare caratteri non−ASCII
    - Esempio 6.15. sitecustomize.py
    - Esempio 6.16. Effetti dell'impostazione di una codifica predefinita
    - Esempio 6.17. russiansample.xml
    - Esempio 6.18. Analizzare russiansample.xml
- 6.5. Ricercare elementi
    - Esempio 6.19. binary.xml
    - Esempio 6.20. Introduzione a getElementsByTagName
    - Esempio 6.21. Ogni elemento è ricercabile
    - Esempio 6.22. La ricerca è ricorsiva
- 6.6. Accedere agli attributi di un elemento
    - Esempio 6.23. Accedere agli attributi di un elemento
    - Esempio 6.24. Accedere agli attributi individuali
- 6.7. Astrarre le sorgenti di ingresso

- Esempio 6.25. Analizzare XML da file
- Esempio 6.26. Analizzare XML da una URL
- Esempio 6.27. Analizzare XML da una stringa (il metodo facile ma inflessibile)
- Esempio 6.28. Introduzione a StringIO
- Esempio 6.29. Analizzare XML da una stringa (alla maniera degli oggetti file)
- Esempio 6.30. openAnything
- Esempio 6.31. Usare openAnything in kgp.py

- 6.8. Standard input, output, ed error
  - Esempio 6.32. Introduzione a stdout e stderr
  - Esempio 6.33. Redigere l'output
  - Esempio 6.34. Redirigere le informazioni di errore
  - Esempio 6.35. Concatenare comandi
  - Esempio 6.36. Leggere dallo standard input in kgp.py
- 6.9. Memorizzare i nodi e ricercarli
  - Esempio 6.37. loadGrammar
  - Esempio 6.38. Usare gli elementi ref in memoria
- 6.10. Trovare i figli diretti di un nodo
  - Esempio 6.39. Trovare elementi figli diretti
- 6.11. Create gestori separati per tipo di nodo
  - Esempio 6.40. Nomi di classi di oggetti XML analizzati
  - Esempio 6.41. analizzare, un generico smistatore di nodi XML
  - Esempio 6.42. Funzioni chiamate dall'analizzatore di smistamento
- 6.12. Gestire gli argomenti da riga di comando
  - Esempio 6.43. Introduzione a sys.argv
  - Esempio 6.44. Il contenuto di sys.argv
  - Esempio 6.45. Introduzione a getopt
  - Esempio 6.46. Gestire gli argomenti da riga di comando in kgp.py

Capitolo 7. Test delle unità di codice

- 7.2. Introduzione al modulo romantest.py
  - Esempio 7.1. Il modulo romantest.py
- 7.3. Verificare i casi di successo
  - Esempio 7.2. testToRomanKnownValues
- 7.4. Verificare i casi di errore
  - Esempio 7.3. Verificare la funzione toRoman con input non validi
  - Esempio 7.4. Verificare fromRoman con input non validi
- 7.5. Verificare la consistenza
  - Esempio 7.5. Verificare toRoman in confronto con fromRoman
  - Esempio 7.6. Verificare rispetto a maiuscolo/minuscolo
- 7.6. roman.py, fase 1
  - Esempio 7.7. Il modulo roman1.py

- Esempio 7.8. Output del modulo romantest1.py eseguito su roman1.py
- 7.7. roman.py, fase 2
  - Esempio 7.9. roman2.py
  - Esempio 7.10. Come funziona toRoman
  - Esempio 7.11. Output di romantest2.py a fronte di roman2.py
- 7.8. roman.py, fase 3
  - Esempio 7.12. roman3.py
  - Esempio 7.13. Osservare toRoman gestire input non corretti
  - Esempio 7.14. Output di romantest3.py a fronte di roman3.py
- 7.9. roman.py, fase 4
  - Esempio 7.15. roman4.py
  - Esempio 7.16. Come funziona fromRoman
  - Esempio 7.17. Output di romantest4.py a fronte di roman4.py
- 7.10. roman.py, fase 5
  - Esempio 7.18. Controllare la presenza delle migliaia
  - Esempio 7.19. Controllare la presenza delle centinaia
  - Esempio 7.20. roman5.py
  - Esempio 7.21. Output di romantest5.py a confronto con roman5.py
- 7.11. Come gestire gli errori di programmazione
  - Esempio 7.22. Il baco
  - Esempio 7.23. Verificare la presenza del baco (romantest61.py)
  - Esempio 7.24. Output di romantest61.py a fronte di roman61.py
  - Esempio 7.25. Eliminazione del baco (roman62.py)
  - Esempio 7.26. Output di romantest62.py a fronte di roman62.py
- 7.12. Gestire il cambiamento di requisiti
  - Esempio 7.27. Modificare i test per tener conto di nuovi requisiti (romantest71.py)
  - Esempio 7.28. Output di romantest71.py a fronte di roman71.py
  - Esempio 7.29. Trasformare in codice i nuovi requisiti (roman72.py)
  - Esempio 7.30. Output di romantest72.py a fronte di roman72.py
- 7.13. Rifattorizzazione
  - Esempio 7.31. Compilare le espressioni regolari
  - Esempio 7.32. Uso di un espressione regolare compilata in roman81.py
  - Esempio 7.33. Output di romantest81.py a fronte di roman81.py
  - Esempio 7.34. roman82.py
  - Esempio 7.35. Output di romantest82.py a fronte di roman82.py
  - Esempio 7.36. roman83.py
  - Esempio 7.37. Output di romantest83.py a fronte di roman83.py
- 7.14. Postscritto
  - Esempio 7.38. roman9.py
  - Esempio 7.39. Output di romantest9.py a fronte di roman9.py

Capitolo 8. Programmazione orientata ai dati

- 8.1. Immergersi

- Esempio 8.1. regression.py
- Esempio 8.2. Semplice output di regression.py

- 8.2. Trovare il percorso

  - Esempio 8.3. fullpath.py
  - Esempio 8.4. Ulteriori spiegazioni su os.path.abspath
  - Esempio 8.5. Esempio di output di fullpath.py
  - Esempio 8.6. Lanciare script nella directory corrente
- 8.3. Filtrare liste rivisitate

  - Esempio 8.7. Introdurre filter
  - Esempio 8.8. filter in regression.py
  - Esempio 8.9. Filtrare usando le list comprehensions
- 8.4. Rivisitazione della mappatura delle liste

  - Esempio 8.10. Introducendo map
  - Esempio 8.11. map con liste di tipi di dato diversi
  - Esempio 8.12. map in regression.py
- 8.6. Importare dinamicamente i moduli

  - Esempio 8.13. Importare contemporaneamente più moduli
  - Esempio 8.14. Importare moduli dinamicamente
  - Esempio 8.15. Importare dinamicamente una lista di moduli
- 8.7. Mettere assieme il tutto (parte 1)

  - Esempio 8.16. La funzione regressionTest, parte 1
  - Esempio 8.17. Passo 1: Leggere i nomi dei file
  - Esempio 8.18. Passo 2: Filtrare per trovare i file necessari
  - Esempio 8.19. Passo 3:Trasformare i nomi dei file in nomi di moduli
  - Esempio 8.20. Passo 4: Trasformare i nomi di moduli in moduli

Appendice E. Storia delle revisioni

Diario delle Revisioni	
Revisione 2.0	30 Gennaio 2003
• Questa è la traduzione letterale, in lingua Italiana della versione **4.4** del presente libro. Non è stata modificata alcuna parte del testo originale. • Per coloro che si trovassero un po' spaesati dalla difficoltà del testo e si volessero fare una cultura in merito, è opportuno segnalare che presso il sito http://www.zonapython.it è possibile reperire le traduzioni di alcuni testi, di base e non, tra cui i citati più volte in questo documento, ovvero: la "PEP:8" sullo stile per il codice Python, il "tutorial" di Guido Van Rossum e l'"How to Think Like a Computer Scientist" di Allen B. Downey, Jeffrey Elkner e Chris Meyers. • Hanno contribuito ad aggiornare questo documento: Ferdinando Ferranti, Francesco Bochicchio, Marco Marconi, Matteo Giacomazzi, Matteo Bertini, Marco Barisione e Marco Mariani. Un sentito ringraziamento a tutti coloro che hanno collaborato a questa traduzione, contribuendo così ad aumentare la documentazione disponibile in lingua Italiana sul linguaggio di programmazione Python. • Eventuali aggiornamenti li potrete reperire presso il sito http://it.diveintopython.org/. Per eventuali suggerimenti, segnalazioni di refusi ecc. ecc. contattate Ferdinando Ferranti.	
Revisione 1.0	15 Gennaio 2002
• Traduzione letterale, in lingua Italiana della versione **4.1** del presente libro. Non è stata in alcun modo modificata alcuna parte del testo originale. • Questo progetto è nato perché continuamente veniva consigliato questo libro a chi desiderava approfondire le proprie conoscenze su Python. A questo punto, Ferdinando Ferranti ne ha proposto la traduzione, sulla mailing list Italiana dedicata al linguaggio e sul relativo news group, offrendo la propria disponibilità per coordinarla. A questo appello hanno risposto in tanti, a seguito l'elenco, in ordine di contribuzione: Francesco Bochicchio, Marco Marconi, Matteo Giacomazzi, Matteo Bertini, Marco Barisione, Marco Mariani, Riccardo Galli e Gerolamo Valcamonica. Un sentito ringraziamento a tutti coloro che hanno collaborato a questa traduzione, contribuendo così ad aumentare la documentazione disponibile in lingua Italiana sul linguaggio di programmazione Python. • Eventuali aggiornamenti li potrete reperire presso il sito http://diveintopython.org/, visto che l'originale, nelle prossime versioni, includerà anche la presente traduzione. Per eventuali suggerimenti, segnalazioni di refusi ecc. ecc. contattate Ferdinando Ferranti.	

Appendice F. Circa questo libro

Questo libro è stato scritto in DocBook XML usando Emacs, convertito in HTML usando il processore SAXON XSLT di Michael Kay, con la versione personalizzata dei fogli di stile XSL di Norman Walsh. Successivamente è stato convertito in PDF usando HTMLDoc ed in puro testo usando w3m. I listati dei programmi e gli esempi sono stati colorati usando una versione aggiornata del pyfontify.py di Just van Rossum `pyfontify.py`, che è inclusa negli script usati per gli esempi.

Se siete interessati ad approfondire DocBook per scrivere documentazione tecnica, potete scaricare i sorgenti XML e gli scripts per ricompilarli, includono i fogli di stile personalizzati XSL usati per creare tutti i vari formati di questo libro. Dovreste anche leggere il classico libro, *DocBook: The Definitive Guide*. Se avete intenzione di scrivere documentazione professionale in DocBook, vi consiglio di iscrivervi alla relativa mailing lists.

Appendice G. GNU Free Documentation License

Version 1.1, March 2000

> Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

G.0. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

G.1. Applicability and definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats

suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

G.2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.3. Copying in quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

G.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the

previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

G.5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

G.6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

G.7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self−contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

G.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

G.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

G.10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

G.11. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front−Cover Texts being LIST, and with the Back−Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front−Cover Texts, write "no Front−Cover Texts" instead of "Front−Cover Texts being LIST"; likewise for Back−Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendice H. Python 2.1.1 license

H.A. History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI) in the Netherlands as a successor of a language called ABC. Guido is Python's principal author, although it includes many contributions from others. The last version released from CWI was Python 1.2. In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI) in Reston, Virginia where he released several versions of the software. Python 1.6 was the last of the versions released by CNRI. In 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. Python 2.0 was the first and only release from BeOpen.com.

Following the release of Python 1.6, and after Guido van Rossum left CNRI to work with commercial software developers, it became clear that the ability to use Python with software available under the GNU Public License (GPL) was very desirable. CNRI and the Free Software Foundation (FSF) interacted to develop enabling wording changes to the Python license. Python 1.6.1 is essentially the same as Python 1.6, with a few minor bug fixes, and with a different license that enables later versions to be GPL−compatible. Python 2.1 is a derivative work of Python 1.6.1, as well as of Python 2.0.

After Python 2.0 was released by BeOpen.com, Guido van Rossum and the other PythonLabs developers joined Digital Creations. All intellectual property added from this point on, starting with Python 2.1 and its alpha and beta releases, is owned by the Python Software Foundation (PSF), a non−profit modeled after the Apache Software Foundation. See http://www.python.org/psf/ for more information about the PSF.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

H.B. Terms and conditions for accessing or otherwise using Python

H.B.1. PSF license agreement

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.1.1 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 2.1.1 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001 Python Software Foundation; All Rights Reserved" are retained in Python 2.1.1 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 2.1.1 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 2.1.1.
4. PSF is making Python 2.1.1 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 2.1.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.1.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.1.1, 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. 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 2.1.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

H.B.2. 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.

H.B.3. CNRI open source GPL−compatible license agreement

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 (c) 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.

H.B.4. CWI permissions statement and disclaimer