



THiNKCODE.TV

PROGRAMMARE IN PYTHON

Marco Beri



Cosa accomuna Google, Star Wars e la NASA?



PROGRAMMARE IN PYTHON

Marco Beri



Cosa accomuna Google, Star Wars e la NASA?

PROGRAMMARE IN PYTHON

Marco Beri



Titolo: Programmare in Python

Autore: Marco Beri

Editore: ThinkCode.TV

Email: info@thinkcode.tv

Sito Web: <http://it.thinkcode.tv>

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

Copyright 2010 © ThinkCode Labs, Inc. Tutti i diritti riservati. Questo libro è rilasciato ad uso personale e ne è vietata la distribuzione, anche solo parziale.

L'edizione cartacea è in vendita [online](#).

INTRODUZIONE

Se state leggendo queste righe, ci sono solo due possibilità: avete sentito parlare di Python e volette capire di cosa si tratta oppure vi piacciono i serpenti. Nel secondo caso mi auguro che stiate sfogliando queste pagine in libreria, perché questo libro in realtà non parla di animali ma di un linguaggio di programmazione. Sono stato tentato di chiamarlo il linguaggio di programmazione, ma si sa che nel mondo informatico le guerre di religione, per fortuna solo virtuali, sono all'ordine del giorno, per cui mi sono trattenuto. Battute a parte, Python è davvero un linguaggio di programmazione meraviglioso e spero che questo libro riesca a convincervi a utilizzarlo sul serio, sia che siate dei veri appassionati, sia che siate degli informatici di professione (e non è escluso che siate entrambe le cose).

Se dovessi scegliere una caratteristica peculiare di

Python, direi che è un linguaggio che permette di dedicarsi completamente al problema che si deve risolvere, spesso senza pensare alle istruzioni da usare e concentrandosi solo sulla ricerca della soluzione. Subito dopo non posso non dire anche che mi permette di leggere i miei programmi, a distanza di mesi, e capire comunque cosa mi passava per la testa quando li ho scritti. Se vi è mai capitata questa (dis)avventura capirete benissimo a cosa mi riferisco.

Ho avuto modo di vedere e provare sul campo diversi linguaggi di programmazione, ma mai nessuno mi ha divertito quanto Python. Divertire può sembrare un verbo fuori luogo, ma a mio parere è proprio quello che più si adatta all'uso di questo bellissimo linguaggio: è potente e versatile ma allo stesso tempo semplice e lineare; è multipiattaforma; è veloce da utilizzare; è facilmente estendibile; è intuitivo e, ultimo ma non meno importante, è gratuito.

Per poter seguire questo libro non è necessaria

alcuna conoscenza di Python, mentre sarà utile aver utilizzato in passato un qualsiasi linguaggio di programmazione.

La versione di Python che utilizzeremo sarà la 2.5, ma se avete già installato una versione precedente non ci sono problemi: vanno benissimo anche la versione 2.4 o la 2.3 e, probabilmente, per la quasi totalità degli esempi presenti in questo libro, anche versioni più vecchie non vi daranno problemi. Non è invece importante che utilizziate come sistema operativo Windows (XP, 2000, 2003, NT, 95, 98 e così via) oppure Linux, Mac OS X, OS2, Amiga e mi fermo qui, perché la lista è assai più lunga: vanno tutti bene!

Nel primo capitolo affronteremo la domanda principale: che cos'è Python?

Nel secondo capitolo impareremo a installare Python.

Nei capitoli da terzo al nono affronteremo i

rudimenti del linguaggio, i suoi tipi di dati, la sintassi, le eccezioni, le funzioni, le classi, l'input e l'output.

Nel decimo capitolo entreremo nel dettaglio dei moduli (le potentissime librerie già pronte per l'uso).

Nell'undicesimo capitolo scriveremo un'applicazione completa partendo da zero.

Nel dodicesimo capitolo esamineremo PyWin32, le potenti estensioni dedicate a Windows, indispensabili per chi ha a che fare con questo sistema operativo (non preoccupatevi: sono completamente gratuite anche queste!).

Nel tredicesimo capitolo daremo un veloce sguardo ad alcuni argomenti avanzati, per capire ancora meglio le enormi potenzialità di Python. Tra le altre cose realizzeremo un programma di installazione completamente indipendente per l'applicazione che realizzeremo nell'undicesimo

capitolo.

Nel quattordicesimo e ultimo capitolo elencheremo alcune risorse utili disponibili su Internet. Un link curioso è quello relativo al primo messaggio scritto da Guido Van Rossum su Python nel lontano febbraio del 1991.

Buona lettura!

L'autore

Marco Beri si laurea in Scienze dell'Informazione nel 1990, periodo oramai definibile come la preistoria del settore.

Il computer è prima di tutto un suo hobby e anche per questo si innamora di Python a prima vista nel lontano 1999, dopo aver sperimentato una ventina di altri linguaggi.

Fa di tutto, riuscendoci, per portarlo nella sua azienda, dove dal 1997 occupa il ruolo di

responsabile dello sviluppo software.

Sposato dal 1991 con Lucia, ha due figli, Alessandro e Federico, e gli altri suoi hobby sono il gioco in ogni sua forma (pallacanestro compresa), i Lego (che come tutti sanno rientrano nella categoria arte) e la lettura.

Potete scrivergli all'indirizzo di posta elettronica:
mberi@linkgroup.it.

Convenzioni utilizzate nel libro

Nel libro sono presenti alcune etichette che identificano tipi particolari di informazioni relative agli argomenti trattati.

Terminologia Questa etichetta segnala e spiega termini nuovi o poco familiari.

Suggerimento Questa etichetta contrassegna spunti e consigli per risparmiare tempo ed evitare confusioni,

come per esempio il modo più semplice per eseguire una determinata operazione.

Attenzione Accanto a questa etichetta sono riportate indicazioni da non trascurare per evitare le difficoltà in cui gli utenti, soprattutto alle prime armi, possono imbattersi.

Riferimenti Questa etichetta contiene indicazioni circa altri testi o siti Internet in cui approfondire gli argomenti trattati.

Nota Una nota contiene informazioni interessanti, talvolta tecniche, relative all'argomento trattato. Talvolta riporta approfondimenti e curiosità.

A Lucia, Alessandro e Federico

CAPITOLO 1

Cos'è Python?

La domanda è, non a caso, la prima delle FAQ (Frequently Asked Question: le domande più frequenti) presenti sul sito ufficiale di Python <http://www.python.org>. Leggiamo insieme la risposta e non preoccupiamoci se qualche termine ci sembra un po' criptico, perchè nei paragrafi e quindi nei capitoli successivi avremo modo di affrontare nel dettaglio ogni definizione:

“Python è un linguaggio di programmazione interpretato, interattivo e orientato agli oggetti. Incorpora al suo interno moduli, eccezioni, tipizzazione dinamica, tipi di dati di altissimo livello e classi. Python combina un'eccezionale potenza con una sintassi estremamente chiara. Ha interfacce verso molte chiamate di sistema, oltre che verso diversi ambienti grafici, ed è estendibile

in C e in C++. È inoltre usabile come linguaggio di configurazione e di estensione per le applicazioni che richiedono un'interfaccia programmabile. Da ultimo, Python è portabile: può girare su molte varianti di Unix, sul Mac, sui PC con MS-DOS, Windows, Windows NT e OS/2.”

A chi dobbiamo una meraviglia del genere? A un geniale signore olandese che risponde al nome di Guido Van Rossum. Curiosamente Guido non è la traduzione italiana del suo nome, è proprio il suo nome originale in olandese.

Guido, nel lontano Natale del 1989, invece di passare le vacanze a decorare l'albero, decise di scrivere un linguaggio che correggesse la maggior parte, se non tutti, i difetti che secondo lui erano presenti negli altri linguaggi.

Per nostra fortuna, Guido Van Rossum era, ed è tuttora, un grandissimo esperto di linguaggi di programmazione e questo ha fatto sì che fin da subito la sua creatura avesse un notevole successo,

dapprima tra i colleghi del centro di ricerca dove lavorava in quel periodo e poi, dopo la pubblicazione su Usenet nel febbraio del 1991, in tutto il mondo.

Qualcuno può domandarsi perché una persona decida di donare all'umanità quello che ha creato, dedicandovi così tanto del proprio tempo libero senza ricevere in cambio altro che gratitudine. È la stessa domanda, senza risposta, che potremmo porre a Linus Torvalds, autore della prima versione del kernel di Linux o, ancora meglio, ad Albert Sabin, se fosse ancora vivo, scopritore del vaccino della poliomielite che si rifiutò sempre di brevettare.

Nel mondo libero di Internet questo accade spesso e, talvolta, si instaura un circolo virtuoso in cui le persone restituiscono qualcosa in cambio, migliorando, correggendo e diffondendo ciò che viene reso disponibile gratuitamente. Python è sicuramente un esempio lampante di questo fenomeno: attualmente esistono ben 68

(sessantotto!) sviluppatori ufficiali del linguaggio anche se Guido Van Rossum rimane il solo e unico BDFL (Benevolent Dictator For Life: benevolo dittatore a vita) di Python; in altre parole è colui che ha l'ultima e definitiva parola in presenza di dispute informatiche.

Il termine Python deriva dalla passione che il suo creatore ha per il quasi omonimo gruppo di comici inglesi degli anni sessanta, i Monty Python, i quali a loro volta scelsero il proprio nome perché “suonava divertente”. In un’altra FAQ si dice espressamente che è possibile, anzi consigliato, fare riferimento a scenette o a giochi di parole dei Monty Python. Non è però l’unica volta in cui questi comici hanno dato il nome a qualcosa che avesse a che fare con l’informatica: il termine spam, oramai tristemente noto a chiunque abbia a che fare con la posta elettronica, deriva da un loro famosissimo sketch, in cui l’improbabile menu di un ancora più improbabile ristorante era composto da un’infinita lista di piatti, tutti invariabilmente accompagnati da montagne di spam, un tipo di

carne macinata in scatola, non molto appetibile.

Vediamo ora in dettaglio le diverse definizioni contenute nella risposta alla domanda che dà il titolo a questo capitolo.

Interpretato, interattivo

Per la maggior parte dei linguaggi di programmazione, le operazioni necessarie all'esecuzione di un programma comprendono la scrittura del codice sorgente, la compilazione, talvolta il linkaggio delle librerie (se non conoscente il significato di questo termine meglio per voi: con Python non vi serve saperlo) e infine l'esecuzione del programma eseguibile così ottenuto.

Python vi permette invece di eseguire direttamente il codice sorgente che avete scritto (per questo è detto interpretato) o, addirittura, di scrivere le istruzioni direttamente dal suo prompt dei comandi senza bisogno di creare o editare un file sorgente

(per questo è detto interattivo).

Certo quest'ultima modalità d'uso può sembrarvi bizzarra, ma vedremo che per iniziare da zero, per provare alcune istruzioni nuove o per testare piccole parti dei vostri programmi, questa modalità è estremamente comoda e veloce.

Per quelli che non sanno aspettare, che hanno già scaricato e installato da soli Python, facciamo un piccolo salto avanti per dare un rapido sguardo a un esempio di interattività.

Ecco Python in modalità a riga di comando; per maggior chiarezza (e solo in questo esempio) quello che abbiamo digitato appare in neretto, mentre il resto è il testo visualizzato dall'interprete):

```
C:\> python
Python 2.5c1 (r25c1:51305, Aug 17 2006,
10:41:11)
[MSC v.1310 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or
```

```
"license" for more information.  
>>> print "Ciao mondo!"  
Ciao mondo!  
>>> a = 4  
>>> b = a * 2  
>>> print b  
8  
>>>
```

Nota I caratteri >>> rappresentano il prompt dell'interprete dei comandi di Python.

Nella Figura 1.1 è visibile Python in modalità finestra.

Orientato agli oggetti

Questa espressione è la traduzione letterale dell'abusatissimo termine inglese “object-oriented”. Non è il caso di entrare troppo nel dettaglio di questo paradigma di programmazione, su cui sono già stati versati fiumi di inchiostro senza giungere a una definizione condivisa e universalmente accettata; possiamo dire a grandi

linee che seguire questo paradigma vuol dire pensare alla soluzione di un problema non in termini di una successione di istruzioni, ma di oggetti e dei relativi attributi.

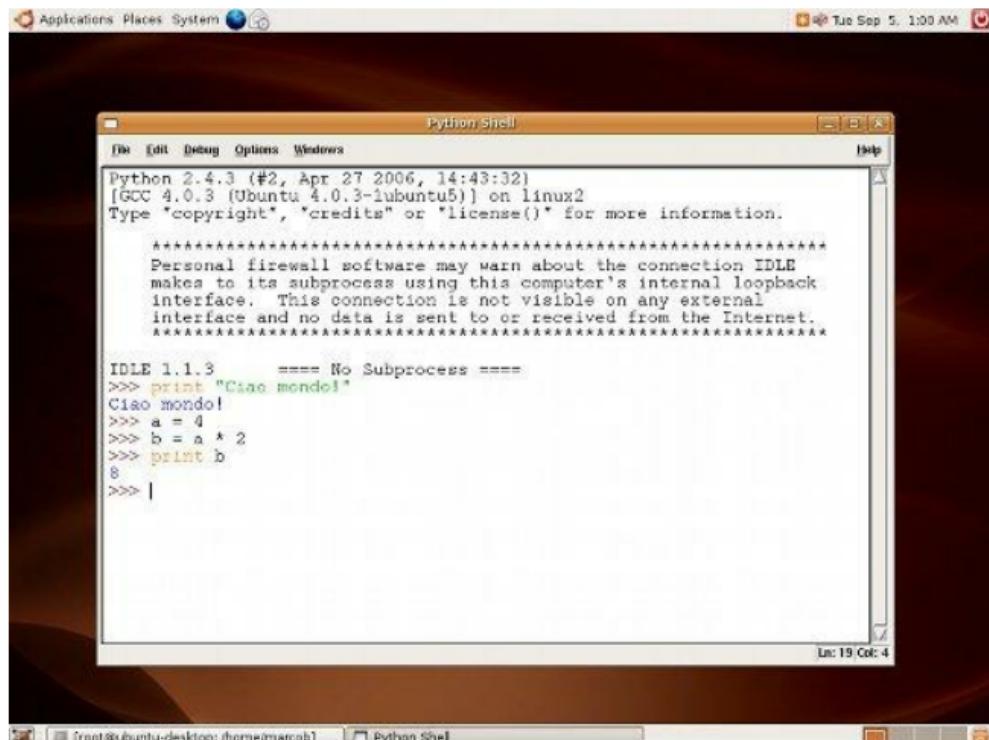


Figura 1.1 IDLE di Python nella distribuzione Ubuntu di Linux.

Un esempio pratico può aiutarci a comprendere meglio questo concetto. Immaginiamo di dover

realizzare una libreria che renda disponibili ai nostri colleghi le funzionalità caratteristiche di lettura e scrittura di variabili in un file di configurazione. Utilizzando un approccio procedurale cominceremmo a scrivere una serie di funzioni per chi dovrà usare questa libreria: CreaFile, LeggiFile, LeggiVariabile, ScriviVariabile e così via. Ogni funzione, molto probabilmente, dovrà ricevere in ingresso il nome del file, motivo per cui dovremo ricordarci il suo nome durante tutto l'uso di queste funzioni. Inoltre ogni funzione dovrà occuparsi di aprire, scrivere o leggere e chiudere il file. Ogni funzione potrà fallire per diversi motivi (file non esistente in lettura, spazio su disco terminato in scrittura e così via), per cui dovremo leggere il valore restituito e poi eventualmente andare a verificare con un'altra funzione quale sia il messaggio di errore completo.

Con un linguaggio orientato agli oggetti possiamo affrontare il problema in maniera totalmente diversa. Per esempio possiamo definire un oggetto FileConfigurazione. Questo oggetto, al momento

della creazione, richiederà il nome del file e da questo momento in poi non dovremo più preoccuparci di ricordarlo. Inoltre possiamo salvare all'interno dell'oggetto lo stato del file fisico, in modo da non doverlo aprire ogni volta. Il nostro oggetto fornirà delle funzionalità di lettura e scrittura delle variabili, di reperimento dello stato del file e così via.

In linea teorica è possibile programmare seguendo questo modello con molti linguaggi, ma vedremo come con Python definire un oggetto, crearlo e utilizzarlo sia estremamente facile.

Per i curiosi ecco in Python la definizione di una classe, Frutto, e la creazione di una sua istanza, mela:

```
>>> class Frutto:  
...     tipo = "vegetale"  
...  
>>> mela = Frutto()  
>>> print mela.tipo  
vegetale  
>>>
```

Nota La sequenza di simboli "... " indica la continuazione sulla riga seguente di un comando che richiede più istruzioni.

Moduli

Installando Python ci troviamo automaticamente a disposizione una grande quantità di librerie pronte per l'uso e in grado di fornire un'enorme quantità di codice già testato e funzionante. Le librerie si chiamano moduli.

Come si importa un modulo in Python?

Nell'esempio seguente importeremo il modulo `smtplib` e lo utilizzeremo per inviare un messaggio di posta elettronica:

```
>>> import smtplib  
>>> host=smtplib.SMTP("mail.server.it")  
>>>  
ret=host.sendmail("otello@venezia.it",  
"desdemona@venezia.it",  
"Cara Desdy, dov'eri ieri?")  
>>>
```

Non preoccupatevi se dalla seconda istruzione in poi non vi è tutto chiaro: l'esempio vuole solo mostrare quanto è facile (anche in modalità interattiva) caricare e utilizzare una libreria (in questo caso il modulo `smtplib`).

Eccezioni

La gestione degli errori in Python è simile a quella di altri linguaggi, come per esempio Java, dato che utilizza il concetto di eccezione (exception).

Un'eccezione è provocata da un evento anomalo o imprevisto che cambia il normale flusso di esecuzione del codice. Un'eccezione, per esempio, può essere dovuta a un input non valido da parte di un utente (un valore alfabetico al posto di un valore numerico) oppure a un'anomalia hardware (tentare di scrivere un file su un hard disk pieno).

Le eccezioni possono essere previste dal programmatore, e in tal caso si dicono eccezioni gestite (handled), oppure possono essere del tutto impreviste, e in tal caso si dicono non gestite

(unhandled).

Se non avete mai avuto modo di utilizzare un linguaggio che preveda le eccezioni, è possibile che queste poche righe non vi abbiano chiarito del tutto le idee. Non allarmatevi: vista l'importanza del tema c'è un intero capitolo dedicato a questo argomento.

Per gli impazienti, ecco un esempio di eccezione non gestita in Python:

```
>>> a = 0
>>> b = 10
>>> print b/a
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or
modulo by zero
>>>
```

Ecco ora la stessa eccezione, questa volta gestita con l'istruzione composta try / except:

```
>>> a = 0
>>> b = 10
```

```
>>> try:  
...     print b/a  
... except ZeroDivisionError:  
...     print "Divisione per zero"  
...  
Divisione per zero  
>>>
```

Tipizzazione dinamica

In un linguaggio di programmazione la tipizzazione delle variabili può essere statica o dinamica. Nel primo caso il programmatore deve dichiarare esplicitamente il tipo della variabile prima di utilizzarla. Nella tipizzazione dinamica è l'interprete (o il compilatore) che, in base al valore assegnato alla variabile, ne decide il tipo.

Python utilizza la tipizzazione dinamica, ma nonostante questo è un linguaggio fortemente tipizzato. Per esempio non è possibile sommare una variabile stringa a una variabile numerica senza convertire esplicitamente quest'ultima in una variabile stringa; in caso contrario scatenereste

un'eccezione!):

```
>>> a = "totale "
>>> b = 10
>>> print a + b
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and
'int' objects
>>>
```

Quindi si deve usare la seguente forma:

```
>>> a = "totale "
>>> b = 10
>>> print a + str(b)
'totale 10'
>>>
```

Tipi di dati di alto livello

Oltre ai tipi di dati nativi più usuali che si possono utilizzare in Python (interi, stringhe, boolean, float) ve ne sono altri estremamente specializzati e utili: liste, tuple, set e dizionari.

La lista è un elenco ordinato di oggetti non necessariamente dello stesso tipo. È possibile aggiungere alla lista nuovi elementi in fondo, all'inizio o in qualunque altra posizione, estrarre un elemento o anche una sequenza di elementi. È possibile creare cicli su tutti gli elementi, cercare un elemento e così via. Probabilmente tutto quello che vi può venire in mente di fare con un elenco ordinato di oggetti può essere fatto facilmente con una lista.

Ecco un esempio di lista:

```
>>> a = [1, 2, 3]
>>> print a
[1, 2, 3]
>>> a.append("stella")
>>> print a
[1, 2, 3, 'stella']
>>>
```

La tupla è abbastanza simile alla lista, tranne per il fatto che è immutabile: una volta assegnata non può più essere modificata. Dunque a una tupla non potete togliere o aggiungere alcun elemento. Le

tuple sono utili quando i dati che contengono non devono mai essere modificati una volta assegnati. Rispetto alle liste sono molto più efficienti per tempo di esecuzione e consumo di memoria, ma in cambio offrono molte funzionalità in meno.

Ecco un esempio di tupla:

```
>>> elenco = (3, "14", 15, "92")
>>> print len(elenco)
4
>>>
```

Un set è un vero e proprio insieme di elementi non ordinati e senza duplicati. Se inizializziamo un set con un elenco contenente elementi ripetuti, questi appariranno una volta sola nel set. Con i set è possibile fare tutte le classiche operazioni che si possono fare con gli insiemi: unione, differenza e intersezione; esiste anche una quarta operazione che è l'intersezione asimmetrica, la quale individua gli elementi che sono presenti in uno e uno solo dei set.

Ecco un esempio di set:

```
>>> primi = set(["pasta", "minestra",
"uova"])
>>> secondi = set(["carne", "uova"])
>>> print primi - secondi
set(['pasta', 'minestra'])
>>> print primi & secondi
set(['uova'])
>>>
```

Un dizionario è una collezione di oggetti di qualunque tipo che possono essere reperiti tramite una chiave. La chiave può essere un intero, una stringa, persino una tupla e in generale qualunque oggetto immutabile (quindi non una lista). Si può anche pensare a un dizionario come a un elenco non ordinato di coppie di chiavi e valori.

Ecco un esempio di dizionario:

```
>>> anni = {'lucia': 42, 'ale': 12,
'fede': 9}
>>> print anni['lucia']
42
>>> print anni.keys()
```

```
[‘fede’, ‘ale’, ‘lucia’]
```

```
>>>
```

Sintassi estremamente chiara

La sintassi di Python è talmente chiara e intuitiva che spesso si intuisce il modo giusto di usare un comando anche senza consultare la documentazione. Inoltre, leggendo un programma, anche in assenza di commenti, è spesso possibile comprenderne lo scopo.

C’è una particolarità importantissima nella sintassi di Python, così importante che da sola è spesso oggetto di violente dispute tra i sostenitori e i detrattori di questo linguaggio: l’indentazione. Python utilizza l’indentazione per delimitare blocchi di istruzioni. Il linguaggio non prevede parentesi, sintassi end-if, next, enddo o altro: solo l’indentazione.

In questo caso un esempio è d’obbligo per tutti, non solo per i curiosi. Proviamo a leggere queste

righe di codice Python:

```
if persona.anni < 18:  
    print "Accesso negato"  
    accesso = False  
elif persona.anni > 80:  
    print "Non è il caso..."  
    accesso = False  
else:  
    accesso = True  
    if persona.isUomo():  
        print "Benvenuto"  
    else:  
        print "Benvenuta"  
return accesso
```

È visibile a colpo d'occhio dove termina il corpo di ogni istruzione if o di ogni istruzione else o elif (che sta per else if).

Questo invece è lo stesso frammento di programma scritto in C:

```
if (persona.anni < 18) {  
printf("Accesso negato\n");  
accesso = 0; }  
else if (persona.anni > 80) {
```

```
printf("Non è il caso...\n");
accesso = 0; }
else { accesso = -1;
if (persona.isUomo == -1)
printf("Benvenuto\n");
else
printf("Benvenuta\n");
return(accesso); }
```

D'accordo, in quest'ultimo caso il programmatore non è stato particolarmente ordinato, ma ha comunque scritto del codice accettabile per un compilatore C. In Python questo non si può proprio fare: non si può essere disordinati perché il programma non funzionerebbe.

Estendibile in C e in C++

Python, pur essendo un linguaggio interpretato, è incredibilmente veloce ed efficiente. In alcuni rari casi può però servirvi tutta la potenza del processore, anche quella parte dedicata all'interprete Python, oppure volette riutilizzare delle librerie compatibili solo con il linguaggio C o C++. Bene, in entrambi i casi è possibile, anzi

facile, creare dei moduli di estensione per superare questi problemi. Questi moduli possono essere poi importati come ogni altro modulo standard.

Con le API (Application Programming Interface) di Python, se lo desiderate e siete particolarmente creativi, potete addirittura implementare nuovi tipi di dati pronti per l'uso nei vostri programmi.

Fate attenzione però: prima di pensare che sia necessario sviluppare un nuovo modulo di estensione, verificate su Internet che tale modulo non sia già disponibile. Tanto per fare un esempio, esiste tutta una serie di moduli per il calcolo scientifico che mettono Python in grado di competere in termini di velocità con i programmi scritti in C o in C++. Potete reperire questi moduli e il relativo codice sorgente sul sito <http://www.scipy.org/>.

Usabile come linguaggio di configurazione

Non esiste quasi nessun programma che non preveda la possibilità da parte dell'utente di configurare in qualche modo il suo comportamento. Finché si tratta di qualche parametro non vi sono problemi: un form dove l'utente può inserire e salvare i valori necessari è più che sufficiente.

Pensiamo invece al caso in cui il nostro applicativo, magari già scritto in un linguaggio non interpretato, debba poter essere configurato in maniera più complessa. Addirittura in alcuni casi gli utenti avanzati devono poter cambiare il comportamento del programma in base ad alcune situazioni. Il nostro programma richiede quindi una cosiddetta interfaccia programmabile. In questi casi la soluzione migliore è quella di fornire una sorta di linguaggio di scripting, da utilizzare in fase di configurazione.

Purtroppo la realizzazione di un programma in grado di comprendere un linguaggio di scripting è

un compito eccezionalmente oneroso. Ma Python ci viene in aiuto anche questa volta: con poche righe di codice possiamo includere il suo interprete nel nostro programma. A questo punto i file di configurazione possono contenere persino dei piccoli programmi scritti in Python.

Proviamo a immaginare un programma gestionale nel cui file di configurazione possiamo scrivere:

```
codici_esenti = (10, 20, 33, 44)
if codice_settore in codici_esenti:
    totale_fattura = imponibile
else:
    totale = imponibile * (1 + iva)
```

Quante release e quante installazioni di nuove versioni potremmo evitare? Ci basterà inviare il nuovo file di configurazione per email...

Portable

È l'ultima definizione e abbiamo vita facile nel dimostrare la portabilità di Python. Se scriviamo

un programma in Python, oltre a poterlo eseguire con il nostro sistema operativo, potremo impiegare tutti quelli più diffusi, come Windows (in tutte le sue versioni), Mac OS X e Linux (tutte le distribuzioni); possiamo perfino inviarlo perché possa essere eseguito da chi utilizza uno di questi sistemi operativi: AIX, AROS, AS/400 (OS/400), BeOS, OS/2, OS/390 e z/OS, Palm OS, PlayStation e PSP (no, non è uno scherzo), Psion, QNX, RISC OS (ex Acorn), Series 60, Sparc Solaris, VMS, VxWorks, Windows CE e Pocket PC, Sharp Zaurus e MorphOS.

E se per caso il sistema che utilizzate non è presente in questo elenco, non disperate: vi basta avere un compilatore C per poter scaricare il codice sorgente e creare una versione personalizzata di Python!

CAPITOLO 2

Come installare Python

Per poter installare Python dovremo scaricare la versione adeguata alla nostra piattaforma e, contestualmente, troveremo le istruzioni più aggiornate per l'installazione. Per questo motivo non dedicheremo troppo tempo a questa fase, ma cercheremo più che altro di dare le indicazioni utili per essere poi autonomi nella ricerca delle soluzioni ai problemi che potremmo incontrare. Come recitava un famoso proverbio inglese “dai dello spam a un uomo e lo sfamerai per un giorno, insegnagli a macinare un maiale e lo sfamerai per sempre”.

Per chi si è già dimenticato lo ripetiamo: lo spam è la carne in scatola resa famosa dai Monty Python. Scrivere un libro su Python senza far riferimento ogni tanto allo spam è considerato politicamente

scorretto dalla comunità pythonista.

Windows

Per installare Python su Windows basta accedere al sito ufficiale <http://www.python.org> e fare clic sulla voce Download, normalmente presente nel menu a sinistra. A questo punto arriveremo a una pagina dove troveremo un elenco di versioni di Python: scegliamo la versione 2.5.

C'è da dire che al posto della versione 2.5 potremmo trovare una nuova versione, come la 2.5.1: scegliete pure l'ultima versione che trovate.

Se non dovessimo riuscire trovare l'elenco, possiamo andare “da soli” alla pagina che ci serve digitandone l'indirizzo URL:

<http://www.python.org/download/releases/>

Attenzione *Così facendo rischiamo di non scoprire se è stata rilasciata una versione*

più recente di quella che digitiamo nell'indirizzo URL!

Giunti alla pagina dedicata alla versione che ci interessa, possiamo fare clic sul link Python 2.5 Windows Installer. Subito dopo il nostro browser ci chiederà di salvare un file con un nome più o meno simile a questo: python-2.5.msi.

A partire dalla versione 2.4, Python è distribuito come file in formato Microsoft Installer, con estensione .msi. Per scoprire se la nostra versione di Windows supporta tale formato ci basterà fare clic sul file. Se partirà una schermata di installazione simile a quella rappresentata nella Figura 2.1, non dovremo fare altro che seguire le solite istruzioni che incontriamo quando installiamo un programma per Windows, lasciando, senza preoccuparci troppo, tutte le opzioni di default.

Se invece il nostro sistema operativo non riconosce il formato .msi ci apparirà una

schermata simile a quella rappresentata nella Figura 2.2.



Figura 2.1 Schermata di installazione iniziale di Python in formato Microsoft Installer (file con estensione .msi).



Impossibile aprire il file:

File: python-2.5.msi

Per aprire il file occorre indicare il programma con cui è stato creato. È possibile eseguire una ricerca automatica sul Web o selezionare manualmente il programma da un elenco sul computer in uso.

Scegliere l'operazione da effettuare.

- Utilizza il servizio di ricerca sul Web per trovare il programma
- Seleziona il programma da un elenco

OK

Annulla

Figura 2.2 Il sistema operativo non riconosce il formato Microsoft Installer.

In questo caso, non perdiamoci assolutamente d'animo, facciamo clic su Annulla e torniamo alla pagina web dedicata alla versione che abbiamo scaricato. Qui troveremo i link che rimandano alle pagine del sito web Microsoft dal quale potremo scaricare gratuitamente (da non crederci...) la versione di Microsoft Installer adatta al nostro sistema operativo. Ne esiste una versione per

Windows 95/98 (il cui file di installazione si chiama InstMsiA.exe) e una per Windows NT 4.0/2000 (InstMsiW.exe). Windows XP e seguenti sono invece in grado di comprendere direttamente il formato .msi. Se non abbiamo Windows XP ma siamo riusciti comunque a installare Python, nessun mistero: molti programmi Microsoft, per esempio Word o Excel, portano in dote Microsoft Installer.

Arrivati a questo punto, anche i più sfortunati avranno avuto la possibilità di iniziare l'installazione. Completati i passi richiesti, nel menu dei programmi installati sul sistema troveremo una voce Python 2.5. Apriamola e scegliamo Python (command line). Evviva! Ecco la finestra dell'interprete di Python e finalmente possiamo provare il nostro primo programma:

```
>>> print "Ciao mondo!"  
Ciao mondo!  
>>>
```

Attenzione Un'ultima indicazione utile per

un uso più comodo di Python in Windows: se non riusciamo a eseguire direttamente l'interprete da una finestra del prompt dei comandi (il comando cmd per intenderci) è probabile che nella variabile di sistema PATH non sia presente la directory di installazione. Nel caso della versione 2.5, se non abbiamo scelto una directory diversa, tale directory sarà C:\python25. Possiamo aggiungerla manualmente dal Pannello di controllo, scegliendo Sistema, quindi la scheda Avanzate e infine il pulsante Variabili d'ambiente.

Linux

È estremamente probabile che sul nostro sistema Linux, Python sia già presente. Per verificarlo basta aprire una finestra terminale e provare a eseguire direttamente il comando python. Se ci apparirà l'ormai usuale prompt dei comandi >>> non dovremo fare altro.

Se invece Python non c'è, allora abbiamo due alternative: cercare una versione binaria di Python in un formato compatibile per la nostra distribuzione di Linux (.RPM per RedHat, .deb per Debian o Ubuntu e così via) oppure utilizzare un programma di installazione dei pacchetti (apt-get su Debian o Synaptic Package Manager su Ubuntu).

In realtà abbiamo una terza alternativa: scaricare e compilare Python partendo direttamente dal codice sorgente. A questa opzione è dedicato un intero paragrafo di questo capitolo.

Mac OS X

Anche nel caso di Mac OS X è molto probabile che Python sia già installato sul nostro sistema. Di nuovo, per appurarlo, ci basta aprire una finestra terminale ed eseguire il comando `python`.

Se Python non dovesse essere presente sul nostro sistema, avremo comunque vita facile, visto che la

versione binaria per l'installazione su Mac OS X è già pronta e disponibile sul sito di Python in formato .dmg (disk image) compatibile sia per Mac su PowerPC sia per Mac su Intel.

La possiamo scaricare direttamente dalla pagina dedicata alla versione che ci interessa. Per installarla basta fare clic sul file appena scaricato e anche con Mac OS X siamo pronti a partire per l'avventura pythoniana!

Compilazione dal codice sorgente

Per compilare Python a partire dal codice sorgente dobbiamo innanzitutto recuperare proprio i file di codice sorgente. Visitiamo quindi la pagina della versione che ci interessa. Abbiamo già visto più volte in dettaglio questa operazione nelle sezioni precedenti, per cui la ripetiamo in breve. In fondo è facile supporre che chi voglia compilare Python abbia una certa dimestichezza con il proprio sistema operativo e con l'informatica in generale.

Andiamo nella seguente pagina per scegliere la versione:

<http://www.python.org/download/>

Oppure digitiamo direttamente l'indirizzo URL:

<http://www.python.org/download/releases/>

Ora cerchiamo il paragrafo che inizia con le parole “All others” (tutti gli altri), il quale conterrà il link che rimanda al file del codice sorgente: Python-2.5.tgz. Una volta scaricato il file dobbiamo espandere i file in esso contenuti. Il formato .tgz dovrebbe essere noto a tutti gli utenti dei sistemi operativi Linux e Mac, un po’ meno agli utenti Windows. Per recuperare un’utility in grado di espandere tale formato, basta cercare su Google le stringhe tgz e windows e, voilà, ecco trovato l’indirizzo URL che ci serviva:

<http://www.gzip.org>

Espandiamo i file di codice sorgente in una

directory a nostro piacimento, apriamo un finestra terminale (o il prompt dei comandi se siamo in Windows) e posizioniamoci nella directory in questione.

Ora dobbiamo fare una distinzione. La compilazione dei file di codice sorgente in Linux o Mac OS X è semplice: basta eseguire la “solita danza” dei comandi gunzip, tar, configure, make e make install. Il comando gunzip espande il file .tgz in un file .tar da cui il comando tar estrae i file di codice sorgente. Il comando configure verifica la presenza dei programmi necessari alla compilazione (e le relative versioni), creando un apposito makefile. Il comando make compila i file di codice sorgente, creando i file per l’installazione. Infine il comando make install copia i file nelle directory di destinazione corrette. Se qualcosa non dovesse funzionare è probabile che il problema riguardi il comando configure, per cui possiamo leggere il file config.log alla ricerca della possibile soluzione del problema. Prima di riprovare la “danza” dei comandi, assicuriamoci

di eseguire il comando make clean.

In Windows le cose sono diverse. Innanzitutto dobbiamo accertarci che sul sistema sia installato Microsoft Visual C++ 7.1 o 8.0. Quindi dobbiamo aprire il workspace pcbuild.sln presente nella directory PCBuild (o PCBuild8 in caso di MS Visual C++ 8.0). Solo a questo punto possiamo lanciare il comando per la creazione degli eseguibili.

Riferimenti Se, durante la compilazione dei file di codice sorgente, ci siamo scontrati con qualche problema insormontabile, una insostituibile fonte di informazioni è <http://groups.google.com>. Se nella ricerca aggiungiamo la stringa *group:comp.lang.python*, saremo sicuri di trovare informazioni relative al nostro linguaggio preferito.

CAPITOLO 3

Pronti? Via!

Siamo sicuri che, se siete arrivati sin a qui, avrete già provato qualche esempio che vi abbiamo presentato. È tuttavia opportuno iniziare dal più classico degli esempi: l'ubiquo hello world! (“ciao mondo!”).

Come abbiamo già detto, uno dei vantaggi di un linguaggio interpretato è la rapidità di prototipazione: dal prompt di Python possiamo subito digitare e provare le istruzioni. Se questo approccio ha senso per programmi piuttosto semplici, è però vero che nella maggior parte dei casi scriveremo un file di codice sorgente, che poi eseguiremo con l’interprete Python.

Così faremo anche con questo primo esempio.

IDLE

Tra le voci del menu di Python è presente “IDLE (Python GUI)”. Questo strumento integrato ci permette di editare, eseguire e testare i nostri programmi scritti in Python.

Nota Anche questo acronimo, che ufficialmente sta per “*Integrated DeveLopment Environment*”, è in realtà un richiamo ai nostri cari Monty Python: uno di loro infatti si chiama Eric Idle.

Eseguiamo quindi IDLE (Python GUI) e ci apparirà una finestra come quella rappresentata nella Figura 3.1.

Digitiamo:

```
print "Ciao mondo!"
```

e premiamo il tasto INVIO.

Abbiamo scritto ed eseguito il nostro primo

programma in Python.

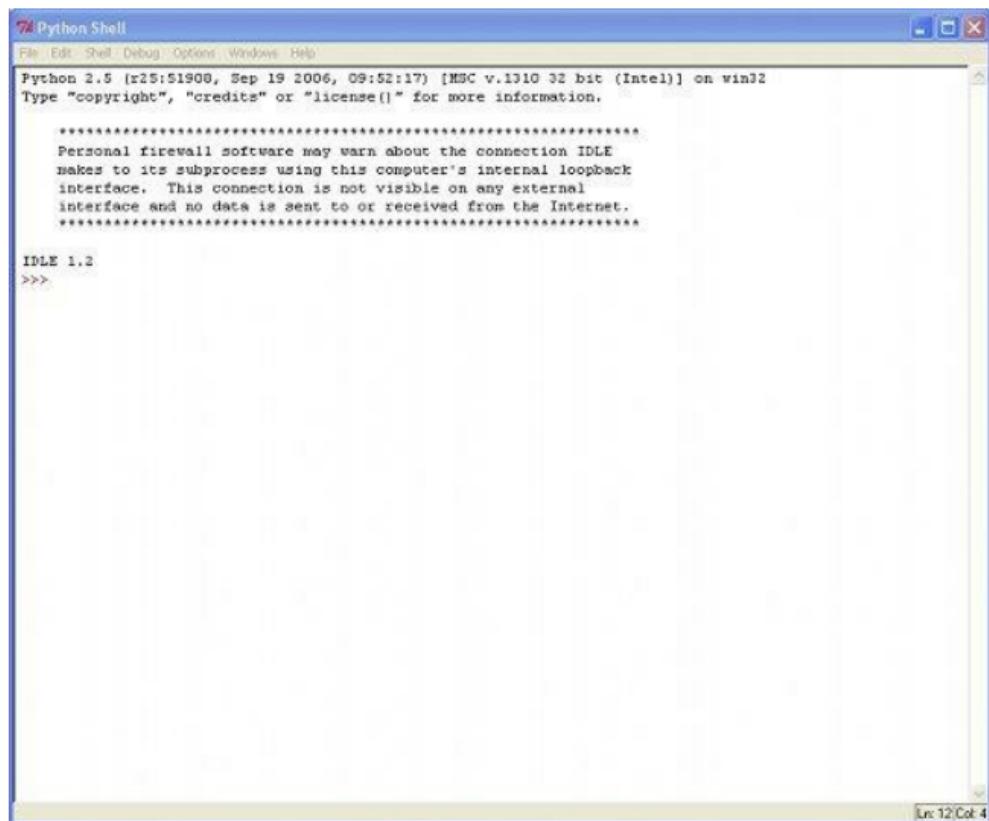


Figura 3.1 La schermata iniziale di IDLE.

Indubbiamente è stato facile, ma non è certo molto comodo dover digitare ogni volta il codice di questo utilissimo programma, per cui ora proveremo a salvare il codice sorgente in un file,

in modo da poterlo facilmente riutilizzare a nostro piacimento.

Editing

Eseguiamo il comando File/New Window scegliendolo dal menu di IDLE o premendo CTRL-N. La finestra che ci apparirà, rappresentata nella Figura 3.2, sarà simile alla precedente, con una differenza sostanziale: non è presente il classico prompt dei comandi di Python, i tre caratteri >>>, che ormai dovremmo aver imparato a riconoscere. Questa finestra ci permette infatti di editare e salvare il codice Python.

Ora digitiamo nuovamente il codice:

```
print "Ciao mondo!"
```

Attenzione Python, come ormai quasi tutti i linguaggi di programmazione (tranne rarissime eccezioni), distingue tra lettere minuscole e maiuscole. Per cui occorre fare attenzione a non utilizzare Print o PRINT al-

posto del comando corretto, print.



Figura 3.2 La nuova finestra di editing.

Questa volta non premiamo il tasto INVIO ma scegliamo il comando File/Save, oppure premiamo CTRL-S, e salviamo il file con il nome hello.py in una directory a scelta.

Attenzione I file di codice sorgente dei programmi Python hanno l'estensione .py. Pur potendo, in linea teorica, utilizzare

un'estensione diversa per il nostro script, è bene non farlo, perché il comando import richiede tale estensione. Come abbiamo visto nel primo capitolo e come vedremo più volte in seguito, il comando import è assolutamente fondamentale in Python, perché permette l'importazione delle librerie all'interno di un nostro programma.

La finestra, come possiamo vedere nella Figura 3.3, adesso sarà leggermente cambiata: il titolo mostrerà il nome del file e la directory in cui l'abbiamo salvato.

Esecuzione

A questo punto possiamo provare a eseguire il programma; possiamo farlo direttamente da IDLE. Pertanto scegliamo il comando Run/Run Module oppure premiamo direttamente il tasto F5.

hello.py - C:/work/python/hello.py



File Edit Format Run Options Windows Help

```
print "Ciao mondo!"
```

Ln: 2 Col: 0

Figura 3.3 Il codice sorgente del nostro primo programma.

A questo punto la finestra iniziale di IDLE verrà riportata in primo piano e potremo vedere l'output del nostro programma, così come appare nella Figura 3.4.

Il messaggio RESTART, preceduto e seguito da una serie di caratteri “=” ci permette di distinguere a colpo d'occhio le diverse esecuzioni dei programmi.

The screenshot shows the Python Shell window with the title bar "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main window displays the following text:

```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2
>>> print "Ciao mondo!"
Ciao mondo!
>>> ===== RESTART =====
>>>
Ciao mondo!
>>> |
```

In the bottom right corner, there is a status bar with "Ln: 17 Col: 4".

Figura 3.4 L'output del nostro primo programma.

Avrete sicuramente notato (a schermo) i vari colori con cui l'editor di IDLE ha evidenziato il programma. Questa è una funzionalità davvero utile, perché permette di accorgersi di alcuni errori già in fase di editing del codice sorgente. Per i più pignoli, i colori sono completamente configurabili, insieme a molte altre funzionalità di IDLE, come possiamo vedere dalla finestra rappresentata nella Figura 3.5, raggiungibile con il comando Options/Configure IDLE.

Riga di comando

Tutto quello che abbiamo fatto finora in questo capitolo utilizzando IDLE, può essere ottenuto con un editor eseguendo il file dalla riga di comando.



Fonts/Tabs | Highlighting | Keys | General

Custom Highlighting

Choose Colour for :

Normal Text

Foreground Background

```
#you can click here
#to choose items
def func(param):
    """string"""
    var0 = 'string'
    var1 = 'selected'
    var2 = 'found'
    var3 = list(None)

    error cursor |
    shell stdout stderr
```

Save as New Custom Theme

Highlighting Theme

Select :

a Built-in Theme

a Custom Theme

IDLE Classic

- no custom themes -

Delete Custom Theme

Ok

Apply

Cancel

Help

Figura 3.5 Configurazione della colorazione della sintassi.

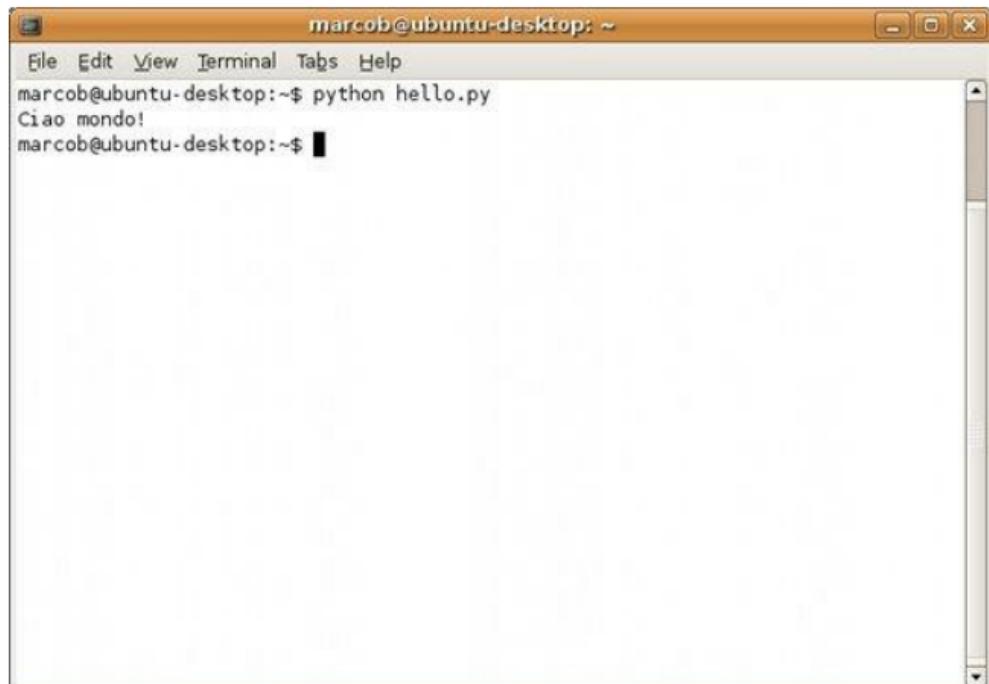
Nota Per scrivere programmi Python esistono moltissimi editor; due su tutti: vi ed Emacs che tra l'altro offrono alcune modalità che facilitano l'editing dei file di codice sorgente scritti in Python. Per esempio visualizzano con colori differenti le istruzioni, le variabili e le costanti. Il vantaggio non indifferente di IDLE è però il fatto di essere stato creato appositamente per lo sviluppo di programmi Python. È quindi perfettamente integrato con l'interprete e permette di analizzare e collaudare facilmente il codice; inoltre, essendo disponibile con l'installazione di Python senza bisogno di fare altro, ha lo stesso aspetto e lo stesso funzionamento su ogni sistema operativo.

Supponiamo ora di aver scritto e salvato il file hello.py utilizzando il nostro editor preferito. Possiamo facilmente eseguirlo lanciando dalla riga di comando la seguente istruzione:

```
python hello.py
```

Facile, vero?

Le finestre rappresentate nelle Figure 3.6 e 3.7 mostrano rispettivamente l'output prodotto su un sistema Linux (Ubuntu 6.06) e su un sistema Windows (prompt dei comandi in Windows XP Professional).



The screenshot shows a terminal window titled "marcob@ubuntu-desktop: ~". The window has a standard window title bar with icons for minimize, maximize, and close. Below the title bar is a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area of the terminal contains the following text:

```
File Edit View Terminal Tabs Help
marcob@ubuntu-desktop:~$ python hello.py
Ciao mondo!
marcob@ubuntu-desktop:~$ █
```

Figura 3.6 L'output in Linux.

C:\WINDOWS\system32\cmd.exe

C:\Work\Python>python hello.py

Ciao mondo!

C:\Work\Python>

Figura 3.7 L'output in Windows.

Conclusioni

Con questo capitolo termina la trattazione degli aspetti secondari di Python; a partire dal prossimo capitolo ci dedicheremo ai costrutti veri e propri del linguaggio (sì, lo sappiamo, non vedevate l'ora....).

A questo punto dovreste essere in grado di

scrivere dei file di codice sorgente in Python, salvarli ed eseguirli con una delle modalità appena descritte. Se così non fosse, prima di proseguire date una seconda lettura a queste pagine.

CAPITOLO 4

I tipi di dati

Dopo tante “lungaggini burocratiche”, entriamo finalmente nel vivo del linguaggio. Entriamo alla grande, esplorando uno degli aspetti più peculiari di Python: i suoi spettacolari tipi di dati.

Cominceremo facendo rapidamente la conoscenza dei comandi `dir` e `type`, utilissimi quando si inizia a utilizzare Python, ma anche dopo! Quindi passeremo alle caratteristiche delle liste, delle stringhe, delle tuple, degli insiemi e dei dizionari (in Python sono rispettivamente i tipi `list`, `str`, `tuple`, `set` e `dict`), dei dati numerici. Infine daremo un’occhiata anche ad alcuni tipi un po’ speciali: `True`, `False` e `None` (che in italiano vogliono dire `Vero`, `Falso` e `Nulla`).

Attenzione *Nei prossimi paragrafi*

utilizzeremo IDLE per collaudare i vari esempi d'uso delle variabili. Come vedremo, per inizializzare una variabile basta scegliere un nome e poi assegnarle un valore. Questo è vero sia nella modalità interattiva sia nella scrittura di codice sorgente. Attenzione però: anche se non siamo obbligati a dichiarare una variabile, non possiamo usarne una non ancora inizializzata. Python ci lascia la massima libertà, ma cerca sempre di ridurre al minimo ogni possibilità di errore.

Il comando dir

Il comando dir si utilizza per lo più in modalità interattiva. Ci permette di visualizzare un elenco degli attributi dell'oggetto passato come argomento, qualunque esso sia.

Vediamolo all'opera con un modulo:

```
>>> import smtplib
```

```
>>> dir(smtplib)
['CRLF', 'OLDSTYLE_AUTH', 'SMTP',
'SMTPAuthenticationError',
'SMTPConnectError',
'SMTPDataError', 'SMTPEException',
'SMTPHeloError',
'SMTPRecipientsRefused',
'SMTPResponseException',
'SMTPSenderRefused',
'SMTPServerDisconnected',
'SMTP_PORT', 'SSLFakeFile',
'SSLFakeSocket',
'__all__', '__builtins__', '__doc__',
'__file__', '__name__', 'base64', 'email',
'encode_base64',
'hmac', 'quoteaddr', 'quotedata', 're',
'socket',
'stderr']
>>>
```

Tutti gli elementi della lista sono funzioni o costanti del modulo smtplib (una possibilità fondamentale quando vorremo inviare un messaggio email da un nostro programma; abbiamo già visto all'opera questa possibilità in un esempio del Capitolo 1).

Nota Spesso, molti degli elementi restituiti da dir sono delle stringhe che presentano una sequenza di due caratteri “_” all'inizio e alla fine. Per ora possiamo tranquillamente ignorarli: sono metodi particolari che vengono richiamati dall'interprete per le operazioni standard. Per esempio il metodo `__str__` di un dato oggetto viene richiamato da Python ogni volta che è richiesta la conversione in stringa dell'oggetto o una sua rappresentazione visualizzabile dal comando `print`.

Vediamo ora l'output di dir con una stringa:

```
>>> s="Ciao"  
>>> dir(s)  
['__add__', '__class__', '__contains__',  
 '__delattr__', '__doc__', '__eq__',  
 '__ge__',  
 '__getattribute__', '__getitem__',  
 '__getnewargs__', '__getslice__',  
 '__gt__',  
 '__hash__', '__init__', '__le__',  
 '__lt__',  
 '__ne__',  
 '__reduce__',  
 '__reduce_ex__',  
 '__repr__',  
 '__str__',  
 '__subclasshook__']
```

```
'__len__', '__lt__', '__mod__', '__mul__',  
['__ne__','  
'__new__', '__reduce__',  
'__reduce_ex__',  
'__repr__', '__rmod__', '__rmul__',  
'__setattr__',  
'__str__', 'capitalize', 'center',  
'count',  
'decode', 'encode', 'endswith',  
'expandtabs',  
'find', 'index', 'isalnum', 'isalpha',  
'isdigit',  
'islower', 'isspace', 'istitle',  
'isupper', 'join',  
'ljust', 'lower', 'lstrip', 'partition',  
'replace',  
'rfind', 'rindex', 'rjust',  
'rpartition', 'rsplit',  
'rstrip', 'split', 'splitlines',  
'startswith',  
'strip', 'swapcase', 'title',  
'translate', 'upper',  
'zfill']  
>>>
```

Non facciamoci spaventare da questo lungo elenco: Python è un linguaggio potente e i suoi tipi

di dati non sono da meno. Nel paragrafo dedicato alle stringhe vedremo in dettaglio il funzionamento della maggior parte di questi metodi.

Il comando type

Il comando type visualizza il tipo dell'oggetto che viene passato come argomento. Vediamo alcuni rapidi esempi:

```
>>> n=1
>>> type(n)
<type 'int'>
>>> s="ciao"
>>> type(s)
<type 'str'>
>>> import smtplib
>>> type(smtplib)
<type 'module'>
>>>
```

Ogni qualvolta avremo un dubbio su un elemento restituito da dir (sarà una costante, una funzione oppure una classe?), type ci verrà in soccorso.

Proviamo a vedere come ci può aiutare con alcuni degli elementi restituiti da dir per il modulo smtplib:

```
>>> import smtplib
>>> dir(smtplib)
['CRLF', 'OLDSTYLE_AUTH', 'SMTP',
 'SMTPAuthenticationError',
 'SMTPConnectError',
 'SMTPDataError', 'SMTPException',
 'SMTPHeloError',
 'SMTPRecipientsRefused',
 'SMTPResponseException',
 'SMTPSenderRefused',
 'SMTPServerDisconnected',
 'SMTP_PORT', 'SSLFakeFile',
 'SSLFakeSocket',
 '__all__', '__builtins__', '__doc__',
 '__file__', '__name__', 'base64', 'email',
 'encode_base64',
 'hmac', 'quoteaddr', 'quotedata', 're',
 'socket',
 'stderr']
>>>
```

Cosa sarà mai l'attributo CRLF?

```
>>> type(smtplib.CRLF)
<type 'str'>
>>>
```

Una stringa! Va bene, era facile, ci si poteva arrivare dal nome. Infatti è la costante che contiene i codici CR e LF, che stanno per Carriage Return (ritorno carrello) e Line Feed (nuova riga).

Nota I caratteri CR e LF sono ben conosciuti a tutti coloro che hanno avuto a che fare con il trasferimento di file tra sistemi Unix e sistemi Windows. Unix utilizza come terminatore di una riga dei file di testo (e quindi anche dei file di codice sorgente) il solo codice LF, mentre Windows utilizza la sequenza CR-LF. Questo fatto può essere di per sé fonte di problemi, ma la situazione è a volte peggiorata da alcuni programmi di trasferimento file, che effettuano automaticamente questa conversione. Questa traduzione può essere utile per i file di testo, ma se viene erroneamente effettuata per i file binari (per

*esempio per i file compressi in formato ZIP)
il risultato sarà quasi sempre un discreto
mal di testa per il malcapitato che ne è
vittima.*

SMTP che cosa rappresenta?

```
>>> type(smtplib.SMTP)
<type 'classobj'>
>>>
```

Una classe! Ma quali metodi conterrà questa classe? Riproviamo con dir:

```
>>> dir(smtplib.SMTP)
['docmd', 'does_esmtp', 'ehlo',
 'ehlo_resp', 'expn',
 'file', 'getreply', 'has_extn', 'helo',
 'helo_resp', 'help', 'login', 'mail',
 'noop',
 'putcmd', 'quit', 'rcpt', 'rset',
 'send',
 'sendmail', 'set_debuglevel',
 'starttls', 'verify',
 'vrfy']
>>>
```

A questo punto, con un po' di immaginazione, possiamo intuire che sendmail è il metodo che serve per inviare un messaggio di posta elettronica.

Spesso ci sarà capitato di dover consultare la documentazione di un linguaggio di programmazione per farci ricordare il nome di un metodo o di una costante: con Python, dir e type ci eviteranno un sacco di volte questa fatica...

Riferimenti La documentazione di Python è molto ben realizzata ma, purtroppo, è disponibile solo in inglese. Per fortuna esiste anche un sito dedicato alla traduzione in italiano della documentazione ufficiale: <http://www.python.it>. È probabile che non vi troverete l'ultima versione tradotta, ma anche la penultima sarà più che sufficiente per risolvere la stragrande maggioranza dei dubbi.

Le liste

Cosa ci fa venire in mente la parola lista? Quasi certamente penseremo alla lista della spesa. Quali sono le sue caratteristiche principali? In estrema sintesi possiamo affermare che una lista è un elenco ordinato che contiene elementi eterogenei, ovvero di diverso tipo.

Bene, in Python una lista è proprio questo: un elenco ordinato di elementi eterogenei.

Proviamo subito a definire una semplice lista contenente qualche valore numerico scelto a caso:

```
>>> elenco = [23, 9, 1964]  
>>> elenco  
[23, 9, 1964]  
>>>
```

Attenzione In questo esempio dobbiamo prestare attenzione ai caratteri “[“ e “]”. Sono proprio queste ultimi, le parentesi quadre, che indicano a Python che vogliamo creare una lista.

Sì, facile; ma non avevamo detto che la lista poteva contenere elementi eterogenei?

```
>>> elenco = [23, 9, 1964]
>>> varie = [1, "pippo", elenco]
>>>
```

Questo esempio è un pochino più interessante: il primo elemento della lista è un numero, il secondo è una stringa e il terzo è un'altra lista. Proviamo ora a visualizzare il contenuto della nuova lista:

```
>>> varie
[1, 'pippo', [23, 9, 1964]]
>>>
```

Abbiamo definito una lista che contiene un'altra lista. Non male.

Non ci basta? Allora esageriamo:

```
>>> import smtplib
>>> varie.append(smtplib)
>>> varie
[1, 'pippo', [23, 9, 1964], <module
'smtplib' from
```

```
'C:\python25\lib\smtplib.py'>]  
>>>
```

Ora la nostra lista, oltre a contenere dei valori semplici e un'altra lista, contiene addirittura una libreria importata.

Quello di inserire in una lista un modulo importato è un colpo a effetto (probabilmente inutile).

Abbiamo però visto il funzionamento del metodo append, che aggiunge un elemento in fondo alla lista.

Attenzione In Python il punto separa un oggetto da un suo attributo o da un suo metodo.

Bene, ora che abbiamo imparato a creare le liste, come possiamo estrarre un elemento a piacere? Semplice:

```
>>> elenco = [23, 9, 1964]  
>>> elenco[0]  
23  
>>>
```

Per estrarre l'elemento nella prima posizione abbiamo utilizzato l'indice 0. Questo perchè in Python la numerazione delle posizioni all'interno di una lista parte da 0: data una lista di n elementi, questi sono numerati da 0 a n - 1.

Che cosa succede se specifichiamo un indice che non esiste? Viene scatenato un errore o più precisamente un'eccezione, argomento al quale è dedicato un intero capitolo:

```
>>> elenco = [23, 9, 1964]
>>> elenco[4]
Traceback (most recent call last):
File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

L'errore “list index out of range” indica proprio il fatto che l'indice utilizzato si trova al di fuori della “portata” della lista.

Come possiamo estrarre l'elemento che si trova nell'ultima posizione? In questo caso abbiamo due possibilità:

```
>>> elenco = [23, 9, 1964]
>>> elenco[2]
1964
>>> elenco[-1]
1964
>>>
```

Se conosciamo la lunghezza della lista, possiamo indicare direttamente l'indice dell'ultimo elemento; in alternativa possiamo, assai più comodamente, indicare l'indice -1. Python sa che quando l'indice è negativo deve partire dal fondo. Gli elementi di una lista di lunghezza n possono essere individuati in ordine inverso con un indice che va da -1 a -n.

A proposito di dimensioni, la funzione len ci permette di scoprire la lunghezza di una lista:

```
>>> elenco = [23, 9, 1964]
>>> len(elenco)
3
>>>
```

Possiamo anche estrarre più elementi in un colpo solo? Ma certo, basta specificare due indici,

separati dal simbolo “::”:

```
>>> elenco = [23, 9, 1964]
>>> elenco[0:2]
[23, 9]
>>> elenco[1:3]
[9, 1964]
```

Possiamo usare gli indici negativi anche con il simbolo “::”:

```
>>> elenco[1:-1]
[9]
>>> elenco[0:-1]
[23, 9]
>>>
```

Possiamo perfino omettere uno dei due valori, a sinistra o a destra di “::”; in tal caso Python raggiunge automaticamente il primo o l’ultimo indice della lista:

```
>>> elenco = [23, 9, 1964]
>>> elenco[1:]
[9, 1964]
>>> elenco[:2]
[23, 9]
```

```
>>> elenco[:]
[23, 9, 1964]
>>>
```

Quest'ultimo esempio è da tenere a mente in quanto rappresenta il modo più comodo e veloce per duplicare una lista:

```
>>> elenco = [23, 9, 1964]
>>> elenco2 = elenco[:]
>>> elenco2
[23, 9, 1964]
>>>
```

Questo particolare metodo di accesso a uno o più elementi di una lista è detto slicing (da to slice = affettare); si tratta di uno di quegli aspetti di Python che lo rendono così piacevole da utilizzare.

Attenzione In Python lo slicing è utilizzabile con tutti i tipi di dati che sono sequenziali. Per esempio è utilizzabile anche con le stringhe che, come vedremo nel prossimo paragrafo, possono essere considerate come una sequenza ordinata di caratteri.

Lo slicing può essere usato anche per assegnare in un colpo solo più elementi di una lista:

```
>>> elenco = [23, 9, 1964]
>>> elenco[0:2] = [13, 8]
>>> elenco
[13, 8, 1964]
>>>
```

oppure per inserire nuovi elementi:

```
>>> elenco = ['a', 'b', 'c']
>>> elenco[1:1] = [1, 2, 3]
>>> elenco
['a', 1, 2, 3, 'b', 'c']
>>>
```

o anche per cancellare degli elementi:

```
>>> elenco = ['a', 1, 2, 'b', 'c']
>>> elenco[1:3] = []
>>> elenco
['a', 'b', 'c']
>>>
```

Abbinando fra loro questi ultimi esempi, proviamo a immaginare come sia possibile cancellare gli

ultimi due elementi di una lista. Elementare, Watson:

```
>>> elenco = ['a', 'b', 'c', 1, 2]
>>> elenco[-2:] = []
>>> elenco
['a', 'b', 'c']
>>>
```

In questo paragrafo abbiamo già visto all'opera il metodo append. Ma quali altri metodi possono essere applicati a una lista? Come facciamo a scoprirlo?

Ebbene sì: con il nostro vecchio amico dir:

```
>>> varie = [1, 'marco', [23, 9, 1964] ]
>>> dir(varie)
['__add__', '__class__', '__contains__',
 '__delattr__', '__delitem__',
 '__delslice__',
 '__doc__', '__eq__', '__ge__',
 '__getattribute__',
 '__getitem__', '__getslice__', '__gt__',
 '__hash__', '__iadd__', '__imul__',
 '__init__',
 '__iter__', '__le__', '__len__']
```

```
'__lt__', '__mul__', '__ne__', '__new__',
'__reduce__',
'__reduce_ex__',
'__reversed__',
'__rmul__', '__setattr__',
'__setitem__',
'__setslice__', '__str__', 'append',
'count',
'extend', 'index', 'insert', 'pop',
'remove',
'reverse', 'sort']
>>>
```

Ignoriamo tranquillamente gli attributi che iniziano e terminano con la sequenza “ ” e vediamo in dettaglio gli altri: append, count, extend, index, insert, pop, remove, reverse e sort.

Sappiamo già usare il primo, append: consente di aggiungere un nuovo elemento in fondo alla lista.

insert permette di inserire un elemento nella posizione desiderata:

```
>>> varie = ['pluto', 'pippo']
>>> varie.insert(0, 'nuovo')
```

```
>>> varie  
['nuovo', 'pluto', 'pippo']  
>>>
```

Per inserire un elemento in fondo alla lista possiamo usare un numero qualsiasi, uguale o maggiore alla lunghezza della lista.

Quindi possiamo impiegare la seguente forma:

```
>>> varie = ['pluto', 'pippo']  
>>> varie.insert(2, 'fondo')  
>>> varie  
['pluto', 'pippo', 'fondo']  
>>>
```

ma anche:

```
>>> varie = ['pluto', 'pippo']  
>>> varie.insert(42, 'fondo')  
>>> varie  
['pluto', 'pippo', 'fondo']  
>>>
```

Il metodo extend accetta come parametro una seconda lista, che viene “appesa” in fondo alla

lista principale:

```
>>> varie = ['pluto', 'pippo']
>>> aggiunta = ['a', 'b', 'c']
>>> varie.extend(aggiunta)
>>> varie
['pluto', 'pippo', 'a', 'b', 'c']
>>>
```

La differenza tra i metodi append ed extend è chiara: il primo aggiunge in fondo alla lista un elemento semplice, mentre il secondo aggiunge in fondo alla lista principale un'intera lista; in pratica è un po' come usare un append per tutti gli elementi della lista da aggiungere.

Siete stanchi di sentir parlare di metodi che allungano la nostra lista? È comprensibile, sicuramente ci capiterà anche di dover rimuovere qualche elemento dalla lista. I metodi remove e pop servono proprio a questo.

Il metodo remove è speculare rispetto a insert, in quanto rimuove l'elemento corrispondente all'indice passato come argomento:

```
>>> elenco = ['a', 'b', 1, 'c']
>>> elenco.remove(2)
>>> elenco
['a', 'b', 'c']
>>>
```

Il metodo `pop` (onomatopeico) è molto particolare, in quanto abbina le due funzionalità di estrazione e rimozione di un elemento. Può essere usato con o senza indice; in quest'ultimo caso estrae l'ultimo elemento della lista:

```
>>> elenco = ['a', 'b', 'c']
>>> elenco.pop()
'c'
>>> elenco
['a', 'b']
>>> elenco.pop(0)
'a'
>>> elenco
['b']
>>>
```

Il metodo `count` conta gli elementi della lista che sono uguali all'argomento passato:

```
>>> varie = [1, 2, 3, 4, 1]
```

```
>>> varie.count(1)
2
>>> varie.count(5)
0
>>>
```

Attenzione, perché count non esplora ricorsivamente eventuali liste presenti all'interno della lista principale:

```
>>> varie = [1, 'marco', [23, 9, 1964] ]
>>> varie.count(23)
0
>>>
```

L'elemento 23 è, sì, presente, ma solo all'interno della lista che rappresenta il terzo elemento della lista principale, per questo motivo non viene trovato da count.

index restituisce la posizione all'interno della lista dell'elemento passato come argomento:

```
>>> elenco = ['a', 'b', 'c']
>>> elenco.index('b')
1
```

```
>>>
```

Attenzione, perchè se l'elemento non esiste viene scatenato un errore (sì, giusto, si chiama eccezione):

```
>>> elenco = ['a', 'b', 'c']
>>> elenco.index('e')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
>>>
```

Il metodo reverse inverte semplicemente una lista:

```
>>> elenco = ['c', 'b', 'a']
>>> elenco.reverse()
>>> elenco
['a', 'b', 'c']
>>>
```

Il metodo sort dispone in ordine alfabetico e/o numerico gli elementi della lista:

```
>>> elenco = [44, 'c', 1, 'b', 'a']
>>> elenco.sort()
>>> elenco
```

```
[1, 44, 'a', 'b', 'c']
```

```
>>>
```

I numeri vengono disposti in ordine all'inizio della lista, quindi vengono elencati i valori alfanumerici, in ordine alfabetico. Normalmente il metodo sort dovrebbe essere applicato a liste contenenti elementi omogenei, ma nulla ci vieta di impiegarlo anche su liste di elementi eterogenei.

Le stringhe

Senz'altro qualcuno si domanderà che cosa ci potrà mai essere di particolare e innovativo nelle stringhe Python. Effettivamente, di solito una stringa non è altro che una sequenza di caratteri. Ma attenzione: in Python tutto ciò che è una sequenza ha accesso alle potentissime funzionalità di slicing che abbiamo appena visto all'opera per le liste. Inoltre una stringa, come ogni altro elemento in Python, è un oggetto, dotato di metodi che ne permettono una facile gestione.

Vediamo subito qualche esempio di slicing con le stringhe in Python:

```
>>> s = "0123456789"
>>> s[:5] # I primi 5 caratteri
'01234'
>>> s[-5:] # Gli ultimi 5 caratteri
'56789'
>>> s[1:-1] # Tutti i caratteri tranne
il primo e l'ultimo
'12345678'
>>>
```

Nota In Python il carattere “#” indica l’inizio di un commento. Inserire un commento in modalità interattiva ha senso solo quando occorre fare degli esempi; sarà invece molto più utile durante la stesura di programmi veri e propri. Lo sappiamo tutti che è utile commentare bene i propri file di codice sorgente, vero?

C’è ancora un tipo di slicing che non abbiamo esaminato. Possiamo utilizzare un terzo parametro che indica, oltre all’inizio e alla fine della nostra

“fetta”, ogni quanti elementi dobbiamo estrarre uno. Forse è meglio chiarire il concetto con un esempio:

```
>>> s = "0123456789"  
>>> s[::2] # Prendi un carattere ogni due  
'02468'  
>>> s[1::2] # Questa volta partì dal secondo  
'13579'  
>>>
```

Un’importante differenza rispetto alle liste consiste nel fatto che le stringhe sono oggetti immutabili. Non possiamo assegnare un nuovo valore a una parte di una stringa:

```
>>> s = "ab-de"  
code >>> s[2] = "c"  
  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
s[2] = "c"  
TypeError: 'str' object does not support  
item  
assignment
```

```
>>>
```

Per poterlo fare dobbiamo riassegnare l'intera stringa:

```
>>> s = "ab-de"  
code >>> s = s[:2] + "c" + s[3:]  
>>> s  
'abcde'  
>>>
```

Possiamo delimitare una stringa con doppi apici; in tal caso la stringa può anche contenere degli apici singoli:

```
>>> s = "Mi piace l'uva"  
>>>
```

Oppure possiamo utilizzare come delimitatore l'apice singolo e allora all'interno della stringa potremo utilizzare gli apici doppi:

```
>>> s = 'Ho visto i "Monty Python": che  
risate!'  
>>>
```

Per inserire in una stringa un carattere di “a capo” dobbiamo utilizzare il simbolo backslash (la barra rovesciata), seguito dal carattere “n”, ovvero la sequenza “\n”:

```
code >>> s = "1\n2\n3\nVia!"  
>>> print s  
1  
2  
3  
Via!  
>>>
```

Nota La sequenza “\n” è un simbolo speciale. Ne esistono diversi altri ma quelli più importanti sono per l'appunto “\n” Line Feed (a capo), “\r” Carriage Return (ritorno carrello), “\b” Backspace (indietro di un carattere), “\t” Tab (carattere di tabulazione) e “\xHH” che permette di inserire il simbolo il cui codice ASCII è il valore esadecimale HH.

Spesso ci capiterà di dover scrivere delle stringhe su più righe, come stringhe di documentazione dei

programmi. È possibile farlo utilizzando una sequenza di tre apici doppi o singoli per racchiudere una stringa multiriga.

```
>>> s = """  
Facile scrivere stringhe  
su più righe...  
E posso usare apici: 'spam'  
O doppi apici: "egg"  
E finisco qui.  
"""
```

```
>>> print s  
  
Facile scrivere stringhe  
su più righe...  
E posso usare apici: 'spam'  
O doppi apici: "egg"  
E finisco qui.
```

```
>>>
```

Per concatenare tra di loro due stringhe possiamo usare il simbolo +:

```
code >>> s = "Precipite"  
>>> p = "volissimevolmente"
```

```
>>> s + p  
'Precipitevolissimevolmente'  
>>>
```

Possiamo perfino moltiplicare una stringa, usando il simbolo *:

```
>>> s = "spam"  
>>> s * 5  
'spamspamspamspamspam'  
>>>
```

Anche le stringhe offrono dei metodi che permettono di manipolarle con facilità. Sono davvero molti: capitalize, center, count, decode, encode, endswith, expandtabs, find, index, isalnum, isalpha, isdigit, islower, isspace, istitle, isupper, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill.

Esamineremo in dettaglio solo i principali; la documentazione di Python ci può venire in aiuto per scoprire il funzionamento degli altri metodi

(ma un pizzico di intuizione e il prompt di Python spesso sarà più che sufficiente).

find ricerca un carattere in una stringa:

```
>>> s = "Troviamo la x in questa  
stringa"  
>>> s.find("x")  
12  
>>>
```

strip rimuove gli spazi all'inizio e alla fine di una stringa:

```
>>> s = " egg "  
>>> s.strip()  
'egg'  
>>>
```

replace sostituisce una sottostringa con un'altra:

```
>>> s = "Non mi piacciono i Monty  
Python"  
>>> s.replace("Non", "Ma come")  
'Ma come mi piacciono i Monty Python'  
>>>
```

split e join vanno di pari passo: il primo spezza una stringa in una lista di più parti, mentre il secondo riunisce una lista per formare un'unica stringa:

```
>>> s = "uno due tre"  
>>> s.split(" ")  
['uno', 'due', 'tre']  
>>>  
>>> "/".join(["12", "10", "1492"])  
'12/10/1492'  
>>>
```

Forse l'uso del metodo join ci sembrerà un po' strano; in realtà, se ci pensiamo bene, è perfettamente corretto. join non può essere un metodo della lista; deve essere un metodo della stringa utilizzata per unire gli elementi della lista.

Non possiamo usare split specificando come argomento una stringa vuota per separare tutti i caratteri di una stringa. Per ottenere questo risultato possiamo però usare la funzione predefinita list:

```
code >>> s = "123"  
>>> list(s)  
['1', '2', '3']  
>>>
```

Le tuple

Le tuple sono sequenze di oggetti eterogenei (in questo senso sono simili alle liste) ma sono immutabili (e in questo senso sono simili alle stringhe). Vediamo la nostra prima tupla:

```
>>> t = ('basta', 'con', 'le', 'liste')  
>>> t  
('basta', 'con', 'le', 'liste')  
>>>
```

Le parentesi tonde “(“ e “)” indicano a Python che vogliamo creare una tupla.

Anche con le tuple possiamo usare lo slicing:

```
>>> t = (1, 2, 3, 'stella')  
>>> t[:3]  
(1, 2, 3)  
>>> t[-1]
```

```
'stella'
```

```
>>>
```

Ma non possiamo modificarne alcuna parte:

```
>>> t = (1, 'x', 3, 'stella')
>>> t[1] = 2
```

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
t[1] = 'x'
TypeError: 'tuple' object does not
support item
assignment
>>>
```

Le tuple non hanno alcun metodo e l'unico modo che abbiamo per sapere se un elemento è presente in una tupla consiste nell'impiegare l'operatore in:

```
>>> fibonacci = (1, 1, 2, 3, 5, 8, 13,
21)
>>> 8 in fibonacci
True
>>> 9 in fibonacci
False
>>>
```

Nota *True* e *False* sono i valori booleani *vero* e *falso* in Python: li vedremo più in dettaglio tra qualche paragrafo.

Come possiamo definire una tupla contenente un solo elemento? Con questo accorgimento:

```
>>> singolo = (1) # No, non così...
>>> singolo
1
>>> singolo = (1,) # Così invece sì!
>>> singolo
(1,)
>>>
```

Nota Qualcuno si domanderà a cosa servono le tuple, visto che non offrono niente in più delle liste ma presentano molte funzionalità in meno. È una domanda legittima: il motivo principale sta proprio nella loro immutabilità che gli permette di fungere da indici per i dizionari (argomento che verrà trattato in uno dei prossimi paragrafi).

Gli insiemi

I set (insiemi) sono un potente tipo di dati che possiamo usare quando dobbiamo gestire gruppi di elementi non ordinati e senza duplicati.

A differenza dei tipi di dati che abbiamo visto fino a questo momento, per creare un set dobbiamo usare una parola chiave specifica invece che una carattere speciale:

```
>>> insieme = set(['pippo', 'pluto',  
'paperino'])  
>>> insieme  
set(['pippo', 'paperino', 'pluto'])  
>>>
```

La parola chiave da impiegare per creare un insieme è proprio set. L'argomento può essere una sequenza qualsiasi.

Quindi possiamo impiegare una lista come nell'esempio precedente, ma anche una tupla o una stringa:

```
>>> lettere = set("hello")
>>> lettere
set(['h', 'e', 'l', 'o'])
>>>
```

Attenzione Potete notare come la stringa “hello”, pur contenendo 5 caratteri ha generato un insieme di soli quattro caratteri (il carattere “l” appare una volta sola nell’insieme): gli insiemi non possono contenere elementi duplicati.

Le operazioni che possiamo effettuare sugli insiemi sono tutte quelle classiche dell’insiemistica tradizionale.

L’insieme unione, con il carattere “|” (la barra verticale):

```
>>> piccoli = set(['topo', 'mosca'])
>>> grandi = set(['orso', 'balena'])
>>> piccoli | grandi
set(['topo', 'orso', 'mosca', 'balena'])
>>>
```

L'insieme differenza, con il carattere “-” (il meno):

```
>>> piccoli = set(['topo', 'mosca'])
>>> bianchi = set(['colomba', 'topo'])
>>> piccoli - bianchi
set(['mosca'])
>>>
```

L'insieme intersezione, con il carattere “&” (la “e” commerciale):

```
>>> piccoli = set(['topo', 'mosca'])
>>> bianchi = set(['colomba', 'topo'])
>>> piccoli & bianchi
set(['topo'])
>>>
```

L'insieme differenza simmetrica, meglio noto come xor, costituito da tutti gli elementi che sono presenti solo in uno dei due insiemi ma non in entrambi, con il carattere “^” (l'accento circonflesso):

```
>>> piccoli = set(['topo', 'mosca'])
>>> bianchi = set(['colomba', 'topo'])
>>> piccoli ^ grandi
set(['colomba', 'mosca'])
```

>>>

I dizionari

Se possiamo eleggere il tipo di dati di Python di cui sentiamo di più la mancanza quando (purtroppo) abbiamo a che fare con un altro linguaggio di programmazione che ne è privo, questo è proprio il dictionary (dizionario).

La definizione ufficiale di un dizionario è array associativo. Ma come quasi tutte le definizioni belle e pompose, non ci dice molto.

Per renderci conto di che cosa stiamo parlando, possiamo esaminare un esempio: dobbiamo creare un dizionario e assegnargli qualche valore:

```
>>> rubrica = dict()  
>>> rubrica['marco'] = '333-123123'  
>>> rubrica['lucia'] = '345-888555'  
>>>
```

Proviamo a visualizzare il dizionario che abbiamo

appena creato:

```
>>> rubrica  
{'marco': '333-123123', 'lucia': '345-  
888555'}  
>>>
```

Ora estraiamo un valore dal dizionario:

```
>>> rubrica['marco']  
'333-123123'  
>>>
```

Quindi proviamo ad aggiungere un nuovo valore:

```
>>> rubrica['difra'] = '332-610610'  
>>>
```

Adesso cominciamo ad afferrare il concetto? Un dizionario è un insieme di oggetti che possiamo estrarre attraverso una chiave. La chiave in questione è quella che abbiamo utilizzato in fase di assegnamento.

Per elencare tutte le chiavi di un dizionario possiamo utilizzare il metodo keys:

```
>>> rubrica.keys()  
['difra', 'marco', 'lucia']  
>>>
```

Come possiamo sapere se una determinata chiave è presente nel dizionario? Il metodo `has_key` fa al caso nostro:

```
>>> rubrica.has_key('silvio')  
False  
>>> rubrica.has_key('marco')  
True  
>>>
```

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51906, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2

>>> 2**10000

19950631168807583848837421626835850838234968318861924540520089498529438830221946631919
96168403619459789933112942320912427155649134941378111759378593209632395785573004679379
4526765246551266059895502550086918193311542508608460618046855090748660896248880904898
948380092353941633257850621568309473902556913288065225096643874441046759871626985453222
86853816169431577562964076283688076073222853509164147618395638145896946389941084096053
62678210646214273333940365255656495306031426802349694003359343166514592977732796657756
06172582031407994198179607378245683762800037302885487251900834464581454650557929601414
8339216157345861392570953797691192778008269577356744412306201875783632550272832378927
07103738028663930314281332414016241956716905740614196543423246388012488561473052074319
92259611796250130992860241708340807605932320161268492280496255841312844061536738951487
11425631511108974551420331820293164095759646475601040584584156507204496286701651506
1920631004186422275908670900574606417586951911456050682515204060075198422618980592371
180544478807290639524258433922198072704473162376760846613033778076039803413197133493
65462270056316993745550824178097281098329131440357187752476850985727693792643322159939
987688666080836883783802764328277517237657572747841122943897338108616074232532919748
1312019760417828196569747589816453125843135959662784130128185406283476649088690521047
58088261582396198577012240704433058307586903931960460340497315658320867210591330090375
28234155397453943977152574552905102123109473216107534748257407752739863482894983407569
379556466386218745694992790165721037013644331358172143117913982229830845847334440270964
18285100507292774836455057863450110085296781238947392869954083434615880704395911898581
51457791771436196987281314594837832020814749821718580113890712282509058268174362205774
75921417653715687725614904582904992461028630081535583308130101987675856234343538955409
17562340084488752616264356846883351946372037729324009445624692325435040067802727383775
53764067268986362410374914109667185570507590981002467898801782719259533812824219540283
0275940848955014676668389697996886241636313376393903373455801407636741877711055384225
7394991101864682196965816514851304942236994771476306915546821768287620036277725772378
13653316111968112807926694818872012986436607685516398605346022978715575179473852463694
46923087894265948217008051120322365496288169035739121368338393591756418733850510970271
613915439590991598154654417336311656936031224993796999922678173235802311186264457529
91357581750081998392362846152498810889602322443621737716180863570154684840586223297928
53875623486556440536962622018963571028812361567912543338303270029097668650568557157505
51672751889919412971133769014991618131517154400772865057318955745092033018530484711381
83154073240533190384620840364217637039115506397890007428536721962809034779745333204683
68795868580237925218629120080742019551317948157624448298518461509704888027274721574688
13159475040973211508049819045580341682694978714131606321068639151168177430479259670937
6L

>>>

Ln: 14 Col: 4

Figura 4.1 Un numero intero decisamente inusuale.

Ovviamente possiamo anche cancellare un elemento dal dizionario; basta specificare la sua chiave con il comando del:

```
>>> del rubrica['difra']
>>> rubrica
{'marco': '333-123123', 'lucia': '345-
888555'}
>>>
```

Che cosa possiamo assegnare a un elemento di un dizionario? Qualsiasi oggetto Python, perfino una funzione o una classe (anche se non abbiamo ancora visto come si usano).

E che cosa possiamo utilizzare come chiave di un elemento? Qualsiasi oggetto immutabile, quindi una stringa, un intero ma anche una tupla.

Attenzione Una tupla può essere utilizzata come chiave in un dizionario solo se non contiene, direttamente o indirettamente, altri oggetti modificabili. Una tupla che contiene una lista non può quindi essere

usata come chiave di un dizionario. Non si può usare neppure un insieme, perché è un oggetto modificabile. Esiste però una versione immutabile che si chiama frozenset (letteralmente “insieme congelato”) che è per i set quello che le tuple sono per le liste; è immutabile e quindi può essere usato come chiave di un dizionario.

Anche i dizionari hanno vari metodi (nessuno ne dubitava): clear, copy, fromkeys, get, has_key, items, iteritems, iterkeys, itervalues, keys, pop, popitem, setdefault, update, values.

Alcuni di essi (come keys e has_key) sono già stati brevemente introdotti negli esempi precedenti. Tra tutti gli altri esamineremo solo i più utili.

clear cancella tutti gli elementi di un dizionario.

```
>>> biblio = {'bibbia': 0,  
             'peopleware': 42 }  
>>> biblio  
{'peopleware': 42, 'bibbia': 0}
```

```
>>> biblio.clear()  
>>> biblio  
{ }  
>>>
```

Attenzione Un dizionario può essere inizializzato con le parentesi graffe. Un dizionario vuoto è indicato da due parentesi graffe vuote.

get permette di estrarre un valore di default qualora la chiave specificata non sia presente:

```
>>> sport = { 'corsa': 10,  
              'basket': 13 }  
>>> sport.get('corsa', 0)  
10  
>>> sport.get('tennis', 0)  
0  
>>>
```

setdefault permette di estrarre un valore di default, ma aggiungendolo al dizionario qualora la chiave specificata non esista:

```
>>> ricetta = { 'uova': 2,
```

```
'farina': 100,  
'burro': 30 }  
>>> ricetta.setdefault('zucchero', 200)  
200  
>>> ricetta  
{'zucchero': 200, 'burro': 30, 'farina':  
100,  
'uova': 2}  
>>>
```

values è il gemello speculare di keys: invece delle chiavi estraе tutti gli elementi:

```
>>> azioni = {'fiat': 2,  
             'link': 99,  
             'oracle': 3}  
>>> azioni.values()  
[2, 3, 99]  
>>>
```

Infine items è la “somma” di keys e values, infatti restituisce la lista delle coppie chiave/valore del dizionario (sotto forma di tuple):

```
>>> frutta = {'mele': 123,  
              'banane': 5}  
>>> frutta.items()
```

```
[('banane', 5), ('mele', 123)]  
>>>
```

I numeri

Cosa mai potrà aggiungere Python a un tipo di dati così comune come i dati numerici? Cominciamo dagli interi e proviamo a calcolare la decimillesima potenza di 2:

```
xmlns:xml="http://www.w3.org/XML/1998/nar  
xml:lang="it">>>> 2 ** 10000
```

Nella Figura 4.1 possiamo vedere il risultato dell'operazione.

Proprio così: non c'è limite alle dimensioni di un numero intero. O meglio, il limite è solo quello fisico della memoria del computer su cui stiamo utilizzando Python.

Ovviamente possiamo utilizzare anche i numeri in virgola mobile (attenzione però, in questo caso il limite c'è e dipende dalla piattaforma che stiamo

utilizzando):

```
>>> 2.3456 / 7.89  
0.29728770595690751  
>>>
```

Per i palati più esigenti esistono anche i numeri complessi, identificati dal suffisso j o J:

```
>>> a = -1j # unità immaginaria  
>>> a.real  
0.0  
>>> a.imag  
-1.0  
>>> a ** 0.5 # radice quadrata di -1  
(0.70710678118654757-  
0.70710678118654746j)  
>>>
```

True, False e None

Anche Python, come ogni linguaggio che si rispetti, ha i suoi valori booleani vero e falso, che corrispondono alle stringhe True e False.

```
>>> numeri_pari = (2, 4, 6, 8)
```

```
>>> x = 1 in numeri_pari
>>> x
False
>>> y = 4 in numeri_pari
>>> y
True
>>>
```

Se vogliamo inizializzare un valore booleano
dobbiamo ricordarci di non usare gli apici come
per le stringhe, quindi:

```
>>> flag1 = True # Così va bene
>>> flag2 = "False" # Così non ci siamo
proprio
>>>
```

Ed eccoci finalmente arrivati alla degna
conclusione di questa lunga passeggiata tra i tipi di
dati di Python: il nulla. Il tipo di dati None (nulla)
può assumere un solo valore, uguale proprio a
None. Viene utilizzato in tutte le situazioni in cui
intendiamo indicare la mancanza di un valore
definito. Per esempio possiamo utilizzarlo per
inizializzare il valore di alcune variabili che
devono esistere ma non hanno alcun valore

iniziale.

Nota Quando parleremo dei parametri opzionali delle funzioni vedremo come il tipo di dati `None` può essere utilizzato per inizializzarli in maniera pulita.

Per definire `None`, come per `True` e `False`, non dobbiamo mai usare gli apici:

```
>>> nulla = None # Così va bene  
>>> qualcosa = "None" # Questo non è un  
verò None  
>>>
```

Conversioni

Sicuramente ci capiterà di dover trasformare un valore da un tipo di dati a un altro; per esempio potremmo dover trasformare una lista in una tupla o un intero in una stringa e così via. Per fare questo dobbiamo utilizzare le funzioni definite `str`, `list`, `tuple`, `int`.

Vediamo qualche esempio:

```
>>> t = ('a', 'b', 'c') # Una tupla
>>> l = [1, 2, 3] # Una lista
>>> s = "ciao" # Una stringa
>>> i = 42 # Un intero
>>> str(i) # Da intero a stringa
"42"
>>> tuple(l) # Da lista a tupla
(1, 2, 3)
>>> list(t) # Da tupla a lista
['a', 'b', 'c']
>>> int("1997") # Da stringa a intero
1997
>>> int("FF", 16) # Da esadecimale a
intero
255
>>>
```

CAPITOLO 5

La sintassi

Adesso che abbiamo conosciuto i tipi di dati di Python, dobbiamo scoprire quali comandi possiamo usare per operare su di essi.

Prima di tutto, prenderemo in esame l'indentazione; poi vedremo le istruzioni per il controllo di flusso if, for, while, break, continue e pass.

L'indentazione

L'indentazione è l'aspetto sintattico che più differenzia Python dagli altri linguaggi di programmazione. Non possiamo non affrontare di petto la questione: l'ordine e la leggibilità del codice sorgente che sono imposti da questa regola sono impagabili e assolutamente superiori alla sua

apparente rigidità.

Diamo uno sguardo al seguente esempio di codice Python che abbiamo già visto nel primo capitolo:

```
if persona.anni < 18:  
    print "Accesso negato"  
    accesso = False  
elif persona.anni > 80:  
    print "Non è il caso..."  
    accesso = False  
else:  
    accesso = True  
    if persona.isUomo():  
        print "Benvenuto"  
    else:  
        print "Benvenuta"  
return accesso
```

È facile vedere a colpo d'occhio dove termina un blocco di istruzioni e dove inizia il successivo.

Questo invece è lo stesso programma scritto in C da un programmatore particolarmente “perverso”:

```
if (persona.anni < 18) {  
printf("Accesso negato\n");
```

```
accesso = 0; } else if (persona.anni >
80) {
printf("Non è il caso...\n");
accesso = 0; } else { accesso = -1;
if (persona.isUomo == -1)
printf("Benvenuto\n"); else
printf("Benvenuta\n"); } return(accesso);
```

Chiunque sia stato così sfortunato da dover eseguire la manutenzione di un frammento di codice scritto più o meno in questa maniera, probabilmente ricorda ancora il tenore dei suoi pensieri in quei tragici momenti...

L'esempio che abbiamo appena presentato è sicuramente un eccesso; qualunque programmatore con un po' di sale in zucca non si azzarderebbe mai a scrivere del codice sorgente in tal modo. Tuttavia nulla impedisce di farlo: magari per la fretta di qualche scadenza imminente può sfuggire qualche riga di codice disordinata e di difficile interpretazione.

Nota Esiste addirittura un concorso internazionale per programmi scritti in C in

maniera intenzionalmente incomprensibile: “The International Obfuscated C Code Contest”. Il sito dove possiamo trovare delle vere opere d’arte come quella rappresentata nella Figura 5.1 è <http://www.ioccc.org>. Si tratta di un programma realmente funzionante, scritto in linguaggio C.

In Python dobbiamo proprio scordarci di essere disordinati: il disordine non ci è permesso a livello di sintassi del linguaggio.

Ma non illudiamoci: anche in Python esiste la possibilità di scrivere “brutti” programmi, non esiste un silver bullet (un proiettile d’argento) che risolva ogni problema nel campo della programmazione. Ma già essere quanto meno costretti all’ordine, elimina in partenza una delle principali fonti di incomprensioni nella lettura e nella manutenzione del codice sorgente.

Suggerimento In IDLE possiamo indentare il codice premendo il tasto TAB, e de-

indentarlo premendo il tasto BACKSPACE.
Normalmente IDLE propone già l'indentazione corretta: per esempio, nella riga che segue un'istruzione if o while il cursore si posiziona già a un livello di indentazione superiore rispetto alla riga precedente.



Figura 5.1 Un candidato al concorso 2004 per il codice sorgente più illeggibile.

Un altro grande vantaggio dell'indentazione è il fatto che la regola di chiusura di un blocco di codice è universale: per chiudere un blocco non dovremo più ricordarci di specificare end, endif, fi, parentesi graffe o tonde, wend, endwhile, next, loop, end procedure o end function e chi più ne ha più ne metta. Ci basta quest'unica, semplice e meravigliosa regola: rispettare l'indentazione del codice.

Nota Volendo essere precisi, esiste anche un possibile rovescio della medaglia: i caratteri di tabulazione. Un carattere di tabulazione non può, a priori, essere interpretato alla stessa maniera da un editor piuttosto che da un altro, esistono programmatore che configurano tabulazioni di 4 spazi, altri di 8, altri ancora di 3 e così via. È chiaro che un codice sorgente in cui sono stati utilizzati insieme caratteri di tabulazione e normali spazi può essere fonte di malintesi sia per i programmatore sia per l'interprete Python. La soluzione è, a nostro

avviso, un po' drastica: non usare mai i caratteri di tabulazione nella scrittura di codice Python. Esiste perfino un'opzione dell'interprete Python in modalità a riga di comando (-t) che visualizza dei warning nel caso in cui venga impiegata un'indentazione mista. Inoltre, per evitare all'origine questo problema, tutti gli editor più diffusi permettono di forzare l'espansione dei caratteri di tabulazione nel corrispondente numero di spazi.

L'istruzione if

L'if è forse l'istruzione più comune fra tutti i linguaggi di programmazione. Vediamo un esempio di if, completo in tutte le sue possibilità:

```
import random
n = random.randrange(1, 100) # a caso
tra 1 e 99
if n < 20:
    print "bassino"
elif n > 80:
```

```
print "altino"  
else:  
    print "mezzano"
```

Nota Questa volta, per comodità, abbiamo omesso il prompt di Python, ma nulla ci vieta di provare questo esempio anche in modalità interattiva.

La sintassi dell'istruzione if è semplice, non c'è molto altro da dire dopo aver visto questo esempio. Possiamo limitarci a osservare l'uso del carattere ":" che indica la fine del test che controlla il flusso del programma. Potete inserire tutti gli elif di cui avete bisogno e l'ultimo else non è obbligatorio.

Attenzione In Python non esiste il corrispondente dell'istruzione "switch" in C o del "Select Case" in Visual Basic. Dobbiamo pertanto usare una sequenza di if ... elif ... elif ... else. La filosofia di Python è chiara, quasi sempre esiste un solo modo di scrivere una data sequenza di istruzioni:

quello giusto.

In Python i test booleani utilizzano una convenzione ormai consolidata nella stragrande maggioranza dei linguaggi: `==` per il test di uguaglianza, `>`, `>=`, `<=`, `<` per i test comparativi e indifferentemente `!=` o `≠` per il test di disuguaglianza.

L'istruzione for

for è l'istruzione che in Python ci permette di definire delle iterazioni. Per decidere quante iterazioni svolgere e su quale elemento eseguire l'iterazione dobbiamo usare qualunque oggetto che possa essere considerato una sequenza.

Ormai abbiamo già visto parecchi esempi di sequenze: liste, tuple, stringhe sono tutti dati che possiamo utilizzare per le iterazioni con for:

```
>>> for c in "abc":  
print c  
a
```

```
b  
c  
>>>
```

Ma come fare se vogliamo semplicemente eseguire un'iterazione per un certo numero di volte e non abbiamo una sequenza adatta alla bisogna? In questo caso useremo il costrutto range:

```
>>> for n in range(4) :  
print n  
1  
2  
3  
4  
>>>
```

Con range possiamo specificare l'inizio, la fine e il passo dell'iterazione:

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

Attenzione Non è consigliabile modificare la lista sulla quale si basa l'iterazione del `for`. Se proprio dobbiamo farlo, dovremmo passare a `for` una copia della lista anziché la lista stessa. Ci ricordiamo come si duplica facilmente una lista? Basta utilizzare lo *slicing* senza indici: “`[:]`”.

Fino a qui il costrutto `for` non è molto diverso dall'omologo costrutto presente in altri linguaggi di programmazione. Ma ancora una volta Python fa qualcosa di più per noi, permettendoci di eseguire un ciclo su più variabili contemporaneamente.

Definiamo un dizionario ed eseguiamo un loop sui suoi elementi:

```
>>> rubrica = dict()
>>> rubrica['marco'] = '333-123123'
>>> rubrica['lucia'] = '345-888555'
>>> for key, value in rubrica.items():
    print key, value
```

```
marco 333-123123
lucia 345-888555
```

```
>>>
```

Questa forma è possibile perchè, come abbiamo visto nel capitolo dedicato ai tipi di dati, il metodo items di un dizionario restituisce una lista di tuple, composte dalla chiave e dal rispettivo valore. for determina un ciclo su ogni elemento di questa lista e assegna alle variabili key e value i due valori presenti in ciascuna tupla.

Spesso ci capiterà di voler contare gli elementi di una sequenza mentre la esaminiamo. Una funzione pronta all'uso per svolgere questa operazione si chiama enumerate:

```
>>> cose = ["spam", "egg", "python"]
```

```
>>> for cont, val in enumerate(cose):  
    print cont, val
```

```
0 spam
```

```
1 egg
```

```
2 python
```

```
>>>
```

Se invece abbiamo liste distinte e vogliamo

definire dei cicli su di esse possiamo usare la funzione zip:

```
>>> nomi = ["Marco", "Lucia", "Alex",
"Fede"]
>>> capelli = ["pochi", "castani",
"castani", "biondi"]
>>> anni = [42, 42, 12, 9]
>>> for n, c, a in zip(nomi, capelli,
anni):
    print "%s ha %s capelli e %d anni" %
(n, c, a)
```

```
Marco ha pochi capelli e 42 anni
Lucia ha castani capelli e 42 anni
Alex ha castani capelli e 12 anni
Fede ha biondi capelli e 9 anni
>>>
```

L'istruzione while

L'istruzione while è simile a for, ma differisce per un solo particolare: l'iterazione non si basa su una sequenza, ma continua fintantoché rimane verificata la condizione passata per argomento:

```
>>> x = 1
>>> while x < 5:
    print x
    x += 1
1
2
3
4
>>>
```

Nota All'apparenza, in questo esempio, dopo il `while` non il codice non è indentato: in realtà l'indentazione c'è ma non si vede, perchè stiamo usando IDLE in modalità interattiva; per questo motivo il `while` viene subito dopo il prompt, mentre le due righe seguenti sono indentate di un blocco.

Le istruzioni `break` e `continue`

Potrà succedere che, sia con `for` che con `while`, capiti la necessità di terminare bruscamente il ciclo o di passare all'iterazione successiva: per queste due operazioni si utilizzano rispettivamente `break` e `continue`.

```
for record in carica_dati():
    if errore(record):
        print "Errore non previsto"
        break # Esci dal ciclo for
    if vuoto(record):
        continue # Passa al ciclo
seguente
gestisci_record(record)
```

Ora che abbiamo conosciuto `break` e `continue`, facciamo un piccolo passo indietro. Le istruzioni `for` e `while` di Python hanno anche un'altro particolare: l'`else`. Ma come? L'`else` non era una clausola dell'`if`? È vero, però ci può tornare comodo anche con i cicli. Le istruzioni che scriveremo nella parte `else` verranno eseguite al termine del ciclo, sempre che non decidiamo di terminare il ciclo con un `break`.

```
for num in range(100):
    for fat in range(2, num):
        if num % fat == 0:
            print num, "è", fat, "*",
num/fat
            break
    else:
        print num, "è un numero primo"
```

Questo breve programma calcola e visualizza tutti numeri primi compresi fra 1 e 99; per i numeri non primi visualizza invece la prima fattorizzazione trovata. Possiamo provarlo anche in modalità interattiva in IDLE (come potete vedere nella Figura 5.2).

L'istruzione pass

In fase di stesura iniziale di un programma, potrà capitarcirci di voler scrivere un costrutto if o while senza alcuna istruzione, tranne un commento. Come possiamo dire a Python che nel blocco che stiamo scrivendo non c'è nessuna istruzione? Se dopo un if non inseriamo alcuna indentazione, l'interprete ci segnalerà un errore.

La soluzione è rappresentata dall'istruzione pass, che non fa assolutamente nulla, se non "tenere occupato il posto". Vediamo un esempio per chiarire meglio il suo funzionamento:

```
record_letti = carica_dati()
```

```
if record_letti < 1:  
    # DA COMPLETARE IN SEGUITO:  
    # verificare perchè non ci sono dati  
    pass  
gestione_dati()
```

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2

```
>>> for num in range(100):
    for fat in range(2, num):
        if num % fat == 0:
            print num, "è", fat, "", num/fat
            break
        else:
            print num, "è un numero primo"
```

```
0 è un numero primo
1 è un numero primo
2 è un numero primo
3 è un numero primo
4 è 2 * 2
5 è un numero primo
6 è 2 * 3
7 è un numero primo
8 è 2 * 4
9 è 3 * 3
10 è 2 * 5
11 è un numero primo
12 è 2 * 6
13 è un numero primo
14 è 2 * 7
15 è 3 * 5
16 è 2 * 8
17 è un numero primo
18 è 2 * 9
19 è un numero primo
20 è 2 * 10
21 è 3 * 7
22 è 2 * 11
23 è un numero primo
```

Ln 120 Col 4

Figura 5.2 Un programma di sette righe che calcola e visualizza i numeri primi.

Attenzione In realtà “pass” può servire non

solo in una fase iniziale della stesura di un programma. Quando, nella gestione di un'eccezione, non vogliamo fare nulla, nella relativa parte di codice (obbligatoria) scriveremo solo l'istruzione pass. Capiremo meglio il significato di questa nota nel prossimo capitolo, dedicato alle eccezioni.

CAPITOLO 6

Le eccezioni

Negli esempi presentati sinora abbiamo già fatto la conoscenza, anche se in modo superficiale, delle eccezioni di Python, le exception.

Il meccanismo delle eccezioni è presente anche in altri linguaggi (Java e C++ tanto per fare un paio di esempi sicuramente noti) ed è ampiamente collaudato. Le eccezioni sono eventi scatenati da errori di varia natura.

Quando un'eccezione si verifica in un nostro programma, all'interno di un contesto d'errore che abbiamo previsto, possiamo gestirla (in inglese to handle); in questo caso parliamo dunque di handled exception.

Se invece un'eccezione si verifica in un contesto

imprevisto, non possiamo gestirla (unhandled exception, eccezione non gestita) e quindi causerà l'uscita dal blocco di codice in cui si presenta. Se però il blocco di codice più esterno ha previsto l'eccezione scatenata, potrà gestirla, altrimenti l'esecuzione uscirà anche da questo blocco e così via; alla fine l'eccezione potrebbe provocare l'uscita dal blocco più esterno, ovvero dal programma stesso.

Vedremo ora cosa vuol dire gestire e anche sollevare (da `to raise` in inglese) un'eccezione.

Attenzione Potrebbe sembrare assurdo il fatto che sia possibile “sollevare” un'eccezione, visto che il nostro compito dovrebbe essere quello di prevederle. In realtà uno degli aspetti più interessanti delle eccezioni è proprio questo: la possibilità di sollevare un'eccezione in modo che altri parti del programma si preoccupino di gestirla. Possiamo pensare a un modulo, una libreria di Python, dove

accade un problema imprevisto (per esempio in fase di scrittura di un file si è esaurito lo spazio su disco): come deve comportarsi? Come può conoscere il modo migliore per gestire l'eccezione, visto che non può sapere in quale contesto viene utilizzato? Il modo migliore è proprio quello di sollevare un'eccezione, lasciando che venga gestita dal codice chiamante.

Le istruzioni try ed except

Per imparare a gestire un'eccezione, vediamo come generarne una. Una tra le eccezioni più facili da ottenere è, senz'ombra di dubbio l'eccezione generata dalla divisione per zero:

```
>>> 1/0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or
modulo by zero
>>>
```

In questo caso l'eccezione è stata generata in modalità interattiva; per questo motivo ci siamo trovati di nuovo con il prompt di Python pronto per ricevere nuovi comandi, come se niente fosse.

Vediamo cosa accade invece se generiamo la stessa eccezione in uno script come il seguente, che chiameremo `div_by_zero.py`:

```
a = 1
b = 0
print "Risultato:", a/b
print "Fine del programma"
```

L'output prodotto da questo script è rappresentato nella Figura 6.1.

```
C:\Work\Python>python div_by_zero.py
Risultato:
Traceback (most recent call last):
  File "div_by_zero.py", line 3, in <module>
    print "Risultato:", a/b
ZeroDivisionError: integer division or modulo by zero
C:\Work\Python>
```

Figura 6.1 Un’eccezione non gestita: Unhandled Exception.

Il nostro script è rovinosamente entrato in crash (in inglese crash significa crollare e dobbiamo ammettere che rende molto meglio l’idea che non l’equivalente in italiano). Non c’è traccia del nostro indispensabile messaggio finale.

Proviamo ora a gestire l’eccezione con questa nuova versione dello script `div_by_zero2.py`:

```
a = 1  
b = 0  
try:  
    print "Risultato:", a/b  
except ZeroDivisionError:  
    print "Divisione per zero!"  
print "Fine del programma"
```

Ora l'output dello script è diverso, vediamolo nella Figura 6.2.

Adesso le cose vanno molto meglio: al posto delle righe incomprensibili della versione precedente, ora viene presentato un messaggio più comprensibile e abbiamo anche il messaggio finale del nostro programma.

Osservando la nostra versione dello script può sorgere spontanea qualche domanda: cosa sarebbe successo se ci fossero state altre istruzioni tra l'errore e l'except? E se non ci fosse stata nessuna eccezione, cosa sarebbe accaduto alle istruzioni contenute nell'except? E se poi fosse stata sollevata un'eccezione diversa da ZeroDivisionError?

```
C:\Work\Python>python div_by_zero2.py
Risultato: Divisione per zero!
Fine del programma
```

```
C:\Work\Python>
```

Figura 6.2 Un'eccezione gestita: Handled Exception.

È pertanto importante comprendere il funzionamento dell'istruzione composta try...except. Il blocco di istruzioni che segue il try, se non viene sollevata nessuna eccezione, viene eseguito fino in fondo; quindi il controllo passa alle istruzioni che seguono il blocco except, il quale non viene quindi eseguito. Se invece all'interno del blocco try viene sollevata

un'eccezione, allora tutte le eventuali istruzioni che seguono quella che ha scatenato l'errore vengono saltate e il controllo passa immediatamente al blocco except, sempre che questo contenga l'eccezione corretta. Se invece l'except riguarda un altro tipo di eccezione, viene saltata l'esecuzione dell'intero blocco di istruzioni che contiene il try e il controllo passa al blocco più esterno che offre un except di tipo corretto e in ultima istanza al sistema operativo.

È ovviamente possibile prevedere più blocchi except per ogni try, ognuno con le istruzioni dedicate ad uno o più tipi di eccezioni. L'ultima clausola except può non specificare alcuna eccezione; in questo caso, quest'ultima clausola (denominata default except) individuerà ogni eccezione non gestita precedentemente.

Vediamo una porzione di codice contenente due except:

```
try:  
    d = dict(arg)
```

```
except TypeError:  
    print "arg di tipo errato"  
except NameError:  
    print "arg non dichiarato"
```

Vediamo che sono previsti due blocchi except per due diversi tipi di eccezioni: `TypeError` e `NameError`. Nella Figura 6.3 vediamo come nel primo caso, dove non viene dichiarato `arg`, venga eseguito il blocco del secondo `except`.

Nel secondo caso invece, `arg` viene dichiarato, ma il suo tipo non è accettabile per il comando `dict`; pertanto viene eseguito il blocco del primo `except`.

L'istruzione `raise`

Il comando `raise` permette di sollevare un'eccezione oppure, se viene chiamato senza parametri, di passare il controllo di un'eccezione al blocco di istruzioni più esterno.

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2

```
>>> try:
    d = dict(arg)
except TypeError:
    print "arg di tipo errato"
except NameError:
    print "arg non dichiarato"

arg non dichiarato
>>> arg = 33
>>> try:
    d = dict(arg)
except TypeError:
    print "arg di tipo errato"
except NameError:
    print "arg non dichiarato"

arg di tipo errato
>>>
```

Ln: 29 Col: 4

Figura 6.3 Un blocco try contenente due blocchi except.

Ecco un esempio che presenta entrambe le situazioni:

```
try:
    print "Adesso solleveremo
un'eccezione"
    raise ValueError, "Bingo!"
```

```
except:  
    print "Un'eccezione si e'  
sollevata!"  
    raise
```

Nella Figura 6.4 possiamo osservare l'output di questa porzione di codice eseguita in IDLE.

La prima riga di output è quella che abbiamo gestito direttamente nel corpo del try e la seconda è prodotta dal blocco except; in entrambi i casi viene impiegata l'istruzione print. Quindi, dopo aver eseguito un raise senza parametri, il controllo è stato restituito all'interprete interattivo, il quale ha visualizzato le ultime righe che contengono il tipo di eccezione oltre al messaggio "Bingo!" che abbiamo chiesto di visualizzare.

```
Python Shell
File Edit Shell Debug Options Windows Help
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2
>>> try:
    print "Adesso solleveremo un'eccezione"
    raise ValueError, "Bingo!"
except:
    print "Un'eccezione si e' sollevata!"
    raise

Adesso solleveremo un'eccezione
Un'eccezione si e' sollevata!

Traceback (most recent call last):
  File "<pyshell#0>", line 3, in <module>
    raise ValueError, "Bingo!"
ValueError: Bingo!
>>>

Ln 26| Col 4
```

Figura 6.4 raise con e senza parametri..

Le istruzioni finally ed else

Il blocco try...except prevede la clausola opzionale finally e, come per il for, anche else. Il blocco di istruzioni contenuto in finally viene eseguito sempre e comunque (finally in inglese significa “alla fine” e non “finalmente”). Il blocco di

istruzioni contenuto in else viene invece eseguito solo se tutto è andato bene e non sono state sollevate eccezioni.

Il seguente script di esempio finally-else.py ci chiarirà le idee meglio di qualsiasi definizione:

```
import sys
quofo = int(sys.argv[1])
dividendo = int(sys.argv[2])
try:
    quofoiente = quofo / dividendo
except ZeroDivisionError:
    print "Denominatore uguale a zero"
else:
    print "Quoziente uguale a",
    quofoiente
finally:
    print "Ho finito"
```

Le prime tre righe del programma possono non essere del tutto comprensibili. La prima importa la libreria di sistema “sys”, mentre la seconda e la terza caricano nelle due variabili quofo e dividendo i primi due parametri passati a finally-else.py dalla

riga di comando.

Nella Figura 6.5 possiamo osservare due esecuzioni distinte dello script, dove nella prima definiremo volutamente un dividendo uguale a 0, mentre nella seconda forniremo due valori corretti.

Nel primo caso, dove viene sollevata un’eccezione, viene visualizzato il messaggio di errore (except) e infine il messaggio di conclusione del programma (finally).

Nel secondo caso, senza eccezioni, viene visualizzato il messaggio contenente il risultato (else) e infine, nuovamente, il messaggio di conclusione del programma (finally).

La funzione exc_info

Nell’ultimo esempio abbiamo visto come importare la libreria sys e come usare la lista argv che corrisponde all’elenco dei parametri passati dalla riga di comando.

C:\Work\python>python finally-else.py 10 0
Denominatore uguale a zero
Ho finito
C:\Work\python>python finally-else.py 9 3
Quoziente uguale a 3
Ho finito
C:\Work\python>

Figura 6.5 Due esecuzioni del programma finally-else.py.

Nella libreria sys esiste anche una funzione utilissima che ha a che fare con le eccezioni: `exc_info`.

Quando ci troviamo a gestire un'eccezione sconosciuta, per esempio con una `default except`, come possiamo accedere alla descrizione dell'ultima eccezione? Proprio con `exc_info`.

Vediamo come utilizzare questa funzione con lo script finally_else2.py:

```
import sys
quoto = int(sys.argv[1])
dividendo = int(sys.argv[2])
try:
    quoziante = quoto / dividendo
except:
    print "Eccezione"
    print sys.exc_info()[0]
    print sys.exc_info()[1]
    print sys.exc_info()[2]
else:
    print "Quoziente uguale a",
    quoziante
finally:
    print "Ho finito"
```

Nella Figura 6.6 possiamo vedere l'output della nuova versione dello script. I tre valori della lista sys.exc_info sono rispettivamente il tipo di eccezione scatenata, l'eventuale valore ad essa associato (per intenderci il secondo parametro del raise) e un oggetto traceback.

```
C:\Work\python>python finally-else2.py 10 0
Eccezione
<type 'exceptions.ZeroDivisionError'>
integer division or modulo by zero
<traceback object at 0x008FF468>
Ho finito
C:\Work\python>
```

Figura 6.6 Esecuzione di finally-else2.py.

Nota Il traceback è un oggetto avanzato che permette di visualizzare lo stack del programma al momento in cui si è verificata l'eccezione. In parole povere, l'elenco di script attualmente caricati da Python a partire dallo script più “esterno”. Nell'esempio precedente, se avessimo visualizzato il traceback con la funzione print_tb del modulo omonimo, avremmo

*ottenuto solo la seguente riga:
File “finally-else2.py”, line 5, in <module>*

CAPITOLO 7

Le funzioni

Dopo aver parlato dei dati e delle istruzioni, possiamo cominciare a mettere tutto assieme, per imparare a definire nuove funzioni in Python. Queste nozioni ci serviranno anche nel prossimo capitolo, dedicato alla creazione di classi; pertanto, anche se pensiamo di utilizzare esclusivamente un approccio orientato agli oggetti, non possiamo prescindere dalla conoscenza di questo aspetto di Python.

Nota Una caratteristica interessante di Python è il fatto che non è il linguaggio a scegliere per noi il paradigma di programmazione da usare: procedurale, orientato agli oggetti o funzionale. Con Python possiamo utilizzare uno qualsiasi di questi approcci e addirittura possiamo

adottare approcci misti.

Definire una funzione

L'istruzione che ci permette di creare una funzione è def. Come di consueto facciamo subito un esempio:

```
def fact(n):  
    if n < 2:  
        return 1  
    return n * fact(n - 1)
```

La funzione fact accetta un unico parametro, di nome n. L'istruzione return viene utilizzata per terminare la funzione restituendo il valore desiderato. Abbiamo così implementato una funzione che restituisce il fattoriale del numero intero passato come argomento (non per nulla, e con tanta fantasia, l'abbiamo chiamata fact).

Vediamo ora la nostra funzione all'opera:

```
>>> print fact(3)
```

```
>>> print fact(300)
3060575122164406360353704612972686293885
>>>
```

Proviamo ora a scrivere la stessa funzione in IDLE, con una piccola aggiunta:

```
def fact(n):
    """
        Questa funzione ritorna il
        fattoriale.
        Il primo parametro indica il numero
        di cui va calcolato il fattoriale
    """
    if n < 2:
        return 1
    return n * fact(n - 1)
```

Facciamo attenzione alla stringa che abbiamo inserito subito dopo la definizione della funzione: è la stringa di documentazione. Proviamo ora a digitare in IDLE `fact` seguito dalla parentesi tonda. Nella Figura 7.1 possiamo vedere la finestra di aiuto (il cosiddetto “helper”) che ci viene mostrata automaticamente e che include l’elenco dei parametri e la prima riga della stringa di

documentazione che abbiamo inserito nel codice.

La stringa di documentazione viene salvata da Python nella variabile `__doc__` che possiamo anche visualizzare integralmente, come vediamo nella Figura 7.2.

Valori restituiti

In Python non esiste il concetto di procedura come in altri linguaggi, intesa come una porzione di codice che viene richiamata e che non restituisce alcun valore. Quando una funzione termina senza che venga incontrata l'istruzione `return`, viene comunque restituito il valore `None`. Lo stesso accade nel caso in cui venga eseguito un `return` non seguito da un valore.

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win 32

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 1.2

```
>>> def fact(n):
    """
    Questa funzione ritorna il fattoriale.
    Il primo parametro indica il numero
    di cui va calcolato il fattoriale
    """
    if n < 2:
        return 1
    return n * fact(n - 1)

>>> fact(
```

```
(n)
    Questa funzione ritorna il fattoriale.
```

[Ln: 22 Col: 9]

Figura 7.1 Se documentiamo il codice, IDLE ci aiuta.

In pratica le funzioni f1, f2 e f3 dell'esempio

seguente, restituiscono tutte None:

```
def f1():
    pass
```

```
def f2():
    return
```

```
def f3():
    return None
```

Possiamo verificarlo in IDLE:

```
>>> print f1(), f2(), f3()
None None None
>>>
```

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win 32

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 1.2

```
>>> def fact(n):
    """
    Questa funzione ritorna il fattoriale.
    Il primo parametro indica il numero
    di cui va calcolato il fattoriale
    """
    if n < 2:
        return 1
    return n * fact(n - 1)

>>> print fact.__doc__
```

Questo programma stampa la stringa di documentazione della funzione `fact`.

Questo programma stampa la stringa di documentazione della funzione `fact`.

>>> |

Figura 7.2 La stringa di documentazione `__doc__`.

Attenzione Nel caso in cui il valore restituito da una funzione sia `None`, IDLE

non lo mostra automaticamente; dobbiamo richiederglielo esplicitamente con il comando print.

Probabilmente ci sarà capitato spesso di dover fare in modo che una funzione restituisca più di un valore. In questi casi, con linguaggi diversi da Python, si doveva ricorrere a vari escamotage, come il passaggio di parametri per indirizzo, la restituzione di valori in un array e così via, inventando.

Con Python ci basta, molto semplicemente, restituire e contemporaneamente assegnare due o più valori:

```
>>> def spam_egg():
    return "Spam", "Egg"
```

```
>>> a, b = spam_egg()
>>> a
'Spam'
>>> b
'Egg'
>>>
```

L'assegnamento in linea di più variabili è davvero una caratteristica utilissima di Python e può essere usata anche con sequenze come liste e tuple:

```
>>> lista = [1, 2]
>>> elem1, elem2 = lista
>>> elem1
1
>>> tupla = ('a', 'b', 'c')
>>> elem1, elem2, elem3 = tupla
>>> elem3
'c'
>>>
```

Il passaggio di argomenti

Nella funzione fact di qualche paragrafo fa, abbiamo visto un esempio molto semplice, che prevedeva il passaggio di un solo parametro.

Per una funzione possiamo ovviamente definire tutti i parametri che vogliamo:

```
def minimo(n1, n2, n3):
    """Questa funzione trova il valore
minimo
```

tra i valori passati per argomento””

```
if n1 < n2 and n1 < n3:  
    return n1  
if n2 < n3:  
    return n2  
return n3
```

I parametri possono anche essere indicati per nome invece che per posizione. La funzione precedente può quindi essere richiamata nei seguenti due modi:

```
minimo(23, 9, 64)
```

```
minimo(n2 = 9, n3 = 64, n1 = 23)
```

Ma esistono altre caratteristiche e modalità per il passaggio dei parametri che ci permettono di definire le funzioni in modo estremamente versatile e potente.

Argomenti opzionali

Per ogni parametro possiamo definire un valore di default opzionale: in assenza del rispettivo

parametro, la funzione impiegherà il valore di default.

Proviamo a digitare la sequenza “int(“ in IDLE e attendere, come si può vedere nella Figura 7.3, la comparsa dell’helper.

Vediamo che la funzione int accetta due parametri: x e base. Dalle parentesi quadre che circondano il parametro base possiamo comprendere che si tratta di un parametro opzionale, ma non sappiamo quale valore assume.

Proviamo a visualizzare integralmente la stringa di documentazione __doc__ (Figura 7.4).

Dalla documentazione scopriamo che la conversione, in assenza del parametro opzionale base, viene eseguita nel sistema decimale; pertanto il valore di default di base è 10.

Verifichiamo in IDLE che sia effettivamente così:

```
>>> int("13")
```

Ora proviamo a specificare esplicitamente una base:

The screenshot shows a Windows application window titled "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main area displays Python version information and a warning about IDLE's connection to its subprocess. Below this, the command "IDLE 1.2" is shown, followed by a user input line starting with ">>> int(" and a help box containing the function signature "int(x[, base]) -> integer". The status bar at the bottom right indicates "Ln: 12 Col: 8".

```
Python 2.5 (r25_51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2
>>> int()
()
```

```
int(x[, base]) -> integer
```

Ln: 12 Col: 8

Figura 7.3 L'helper della funzione int.

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2

```
>>> print int.__doc__
int(x[, base]) -> integer

Convert a string or number to an integer, if possible. A floating point
argument will be truncated towards zero (this does not include a string
representation of a floating point number!) When converting a string, use
the optional base. It is an error to supply a base when converting a
non-string. If the argument is outside the integer range a long object
will be returned instead.
>>> |
```

Figura 7.4 La stringa `__doc__` della funzione `int`.

```
>>> int("13", 10)
13
>>> int("13", 8)
11
>>> int("13", 16)
19
>>>
```

Giusto per ricordarci di questa possibilità,

proviamo a passare i parametri anche per nome, senza rispettare il loro ordine:

```
>>> int(base=8, x="13")
11
>>>
```

Modalità avanzate di passaggio degli argomenti

In alcune situazioni particolari possiamo trovarci nella condizione di dover definire una funzione con un elenco di parametri non noto a priori. Python ci permette di farlo in due diversi modi, che possono addirittura convivere nella stessa funzione:

*argomenti trasforma una lista di parametri (senza nome e di lunghezza arbitraria) in una tupla;

**keyword trasforma una lista di parametri (con nome e di lunghezza arbitraria) in un dizionario.

Come spesso accade in Python, un paio di esempi possono chiarirci le idee. Cominciamo con la

prima tipologia avanzata di passaggio parametri,
la tupla di argomenti posizionali:

```
>>> def spesa(*cosa):
    print type(cosa)
    for x in cosa:
        print x

>>> spesa("spam", "egg", "pane",
"latte")
<type 'tuple'>
spam
egg
pane
latte
>>>
```

Come possiamo vedere, l'elenco di parametri viene trasformato in una tupla, che possiamo scorrere facilmente.

Se però con la stessa funzione utilizziamo un argomento nominale viene sollevata una eccezione:

```
>>> spesa(x = "spam")
```

```
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in
<module>
    spesa(x = "spam")
TypeError: spesa() got an unexpected
keyword argument 'x'
>>>
```

Vediamo ora la seconda modalità avanzata, la tupla di argomenti nominali:

```
>>> def spesa(**cosa):
    print type(cosa)
    for x, y in cosa.items():
        print x, y
>>> spesa(primo="spam", secondo="egg",
         terzo="pane", quarto="latte")

<type 'dict'>
terzo pane
primo spam
secondo egg
quarto latte
>>>
```

Questa volta, al contrario, viene sollevata una

eccezione se usiamo un parametro senza nome,:

```
>>> spesa("spam")  
  
Traceback (most recent call last):  
  File "<pyshell#34>", line 1, in  
<module>  
    spesa("spam")  
TypeError: spesa() takes exactly 0  
arguments (1  
given)  
>>>
```

Ovviamente possiamo utilizzare tutte queste modalit in un'unica funzione:

```
>>> def spesa(dove, *giorni, **cose):  
    print "Devo andare al", dove  
    print "-" * 20  
    print "Nei seguenti giorni:"  
    for giorno in giorni:  
        print giorno  
    print "-" * 20  
    print "A comprare le seguenti cose:"  
    for cosa, quanto in cose.items():  
        print cosa, quanto  
  
>>> spesa("supermercato", "luned",
```

```
"venerdi`",
    latte=1, pane=3)
```

Devo andare al supermercato

Nei seguenti giorni:

lunedì`

venerdì`

A comprare le seguenti cose:

pane 3

latte 1

>>>

Attenzione Quando queste due modalità vengono impiegate insieme, siamo obbligati a specificare prima ***argomenti** e solo di seguito ****keyword**. In caso contrario Python ci segnalerà un errore di sintassi.

Varie ed eventuali

Come abbiamo visto, è molto facile definire una funzione in Python. Ma una cosa importantissima che dobbiamo ancora dire è che in Python una

funzione è un oggetto a tutti gli effetti: possiamo assegnarla a una variabile o, addirittura, passarla come argomento a un'altra funzione.

Se proviamo ad assegnare una funzione a una variabile, possiamo usare la variabile semplicemente al posto della funzione:

```
>>> def stampo(cosa):
    print cosa
>>> f = stampo
>>> f("ciao")
ciao
>>>
```

Proviamo adesso a passare una funzione come argomento a un'altra funzione:

```
>>> def stampo(cosa):
    print cosa
>>> def esegui(funz, arg):
    funz(arg)
>>> esegui(stampo, "ciao")
ciao
>>>
```

Nel prossimo esempio proveremo a creare un dizionario di funzioni:

```
>>> def buttare(cosa):
    print "Dobbiamo buttare", cosa

>>> def tenere(cosa):
    print "Dobbiamo tenere", cosa

>>> funzioni["brutto"] = buttare
>>> funzioni["bello"] = tenere
>>>
```

Ora proviamo ad usare le funzioni che abbiamo assegnato al dizionario:

```
>>> funzioni["brutto"]("spam")
Dobbiamo buttare spam
>>> funzioni["buono"]("egg")
Dobbiamo mangiare egg
>>>
```

Se abbiamo qualche esperienza di programmazione, possiamo sicuramente immaginarci molte situazioni in cui questa versatilità ci sarebbe tornata utile. Ricordiamocelo

quando dovremo decidere il linguaggio con il quale intendiamo scrivere il nostro prossimo programma...

CAPITOLO 8

Le classi

In questo capitolo, dedicato alle classi, metteremo assieme tutto ciò che abbiamo visto finora (i tipi di dati, la sintassi e le funzioni), per imparare a creare nuove classi in Python. Ma non spaventiamoci: la semplicità e la potenza che abbiamo conosciuto finora continueranno ad accompagnarci anche in questo aspetto del linguaggio.

Le classi e le istanze

Questa volta facciamo qualcosa di diverso e partiamo subito con un esempio:

```
class Cibo:  
    pass
```

Fatto: abbiamo già definito la nostra prima classe, Troppo facile, vero? L'unica novità è l'istruzione class, mentre l'istruzione pass è lì solo perché dobbiamo per forza scrivere qualcosa nel corpo della definizione della nostra classe.

Ora proviamo a creare un'istanza della nostra classe:

```
pasta = Cibo()
```

Fatto: basta scrivere il nome della classe seguito dalla coppia di parentesi per far sì che Python ci fornisca un oggetto di quella classe, chiamato anche istanza della classe.

Ora, se proviamo a digitare Cibo e poi pasta, otteniamo questi due risultati:

```
>>> Cibo
<class '__main__.Cibo' at 0x020271E0>
>>> pasta
<__main__.Cibo instance at 0x02029328>
```

Nota I valori possono naturalmente essere

differenti: indicano la posizione in memoria in cui sono collocati gli oggetti visualizzati.

Nel caso di Cibo la parola class, presente nella stringa visualizzata in output, indica che siamo in presenza di una classe.

Nel secondo caso, pasta, la stringa indica invece sia il fatto che siamo in presenza di un'istanza, instance, sia la classe che ne ha permesso la creazione: main.Cibo.

Nota Il nome speciale main indica che la classe Cibo è stata definita interattivamente. Se invece fosse stata definita e importata da un modulo esterno, al posto di main vedremmo il nome del modulo in questione.

I metodi

Bene, adesso che sappiamo come definire una classe e come creare un oggetto appartenente a tale

classe, cosa possiamo fare? Cominciamo ad applicare quanto abbiamo imparato nei capitoli precedenti.

Vediamo una versione un po' più evoluta della nostra classe Cibo:

```
class Cibo:  
    """Un esempio di classe per gestire i cibi"""  
    def __init__(self, proteine=0,  
                 carboidrati=0,  
                 grassi=0):  
        self.proteine = proteine  
        self.carboidrati = carboidrati  
        self.grassi = grassi
```

La prima stringa che abbiamo aggiunto, dovremmo ricordarcelo, è la stringa di documentazione, che viene automaticamente assegnata all'attributo `__doc__` della classe.

Il metodo `__init__` può sembrare un po' criptico: si tratta del metodo costruttore. Quando creiamo un'istanza di una classe è normale voler

specificare alcune caratteristiche speciali e proprie dell'oggetto che stiamo costruendo. Per fare questo possiamo definire un metodo apposito dal nome prestabilito `__init__`, il costruttore appunto, e passargli dei valori all'atto della creazione dell'oggetto. Il primo parametro del costruttore (e di tutti i metodi di una classe) è `self`, il quale permette di accedere all'oggetto creato o, nel caso di `__init__`, che sta per essere creato.

Quando richiamiamo un metodo, non dobbiamo fare espressamente riferimento a `self`; ci pensa Python ad assegnarlo. Il fatto che si chiami `self` è una pura convenzione che, anche se siamo anticonformisti, ci conviene rispettare se teniamo alla leggibilità del nostro codice.

Vediamo ora all'opera il nostro nuovo costruttore:

```
pasta = Cibo(proteine=12,  
carboidrati=72, grassi=1)
```

L'oggetto `pasta` ha ora tre nuovi attributi e possiamo provare a visualizzarne uno:

```
>>> print pasta.carboidrati  
72  
>>>
```

Sarebbe interessante aggiungere un po' di intelligenza alla nostra classe Cibo per calcolare, per esempio, le calorie di un alimento.

Con Python possiamo farlo interattivamente, creando una funzione e assegnandola come metodo della classe:

```
>>> def calcolaCalorie(self):  
        return (self.proteine * 4 +  
                self.carboidrati * 4 +  
                self.grassi * 9)  
>>> Cibo.Calorie = calcolaCalorie  
>>>
```

Attenzione Le parentesi che “inglobano” l'espressione restituita sono necessarie solo perché abbiamo spezzato su più righe una singola istruzione. In questo modo Python non segnala un errore di sintassi dovuto al segno + a fine riga, ma prosegue sulle righe

seguenti fino alla parentesi chiusa. In alternativa possiamo non usare le parentesi e inserire un backslash “\” come ultimo carattere della riga che prosegue a capo.

Abbiamo “appiccicato” a posteriori alla nostra classe Cibo il metodo Calorie e l’oggetto pasta che avevamo già creato lo “riceve” automaticamente:

```
>>> print pasta.Calorie()  
345  
>>>
```

Non è sorprendente Python?

Nota Un metodo è una funzione abbinata all’oggetto, mentre un attributo è un dato che lo caratterizza. Un’altro dei pilastri della programmazione object oriented sta proprio nel fatto che un oggetto “fonde insieme” sia i dati sia la logica che opera su di essi. Questo aspetto è denominato “incapsulamento” in quanto “ingloba” in

un'unica entità (l'oggetto) due aspetti normalmente distinti: i dati (sotto forma di attributi) e il codice che li gestisce (sotto forma di metodi). In questo modo dall'esterno l'oggetto è visto come un'unità, senza bisogno di distinguere le sue caratteristiche interne (nel nostro esempio le tre quantità proteine, carboidrati e grassi) dalla sua logica operativa (nel nostro esempio la funzione che calcola le calorie totali).

L'ereditarietà

L'ereditarietà è uno dei pilastri della programmazione object oriented; ecco la definizione: una classe eredita il comportamento di un'altra classe (la cosiddetta base class) estendendolo però con funzionalità proprie.

Come al solito ricorriamo a un esempio per capire meglio questo concetto, estendendo la nostra classe Cibo:

```
class Verdura(Cibo):
    def __init__(self, proteine=0,
carboidrati=0):
        self.grassi = 0
        self.proteine = proteine
        self.carboidrati = carboidrati
```

La nuova classe Verdura è di tipo Cibo, ma con la particolarità che l'attributo grassi è sempre uguale a 0.

Nota Per i più pignoli: sì, lo sappiamo che alcune verdure in realtà contengono una percentuale, seppur minima, di grassi.

Creiamo un'istanza di questa nuova classe e proviamo a usare un metodo della classe base:

```
>>> melanzane = Verdura(proteine=1.5,
                           carboidrati=2.5)
>>> print melanzane.Calorie()
16
>>>
```

Nella classe Verdura non c'è alcuna traccia del metodo Calorie. Ma siccome tale metodo era

definito nella classe base Cibo, lo possiamo usare tranquillamente anche nella classe derivata.

Nel costruttore abbiamo espressamente ripetuto le istruzioni di assegnamento degli attributi della classe base. Potrebbe essere più comodo (e più semanticamente corretto) richiamare il costruttore della classe base. Inoltre non possiamo sapere cosa verrà aggiunto in futuro nella classe “madre”, per cui è sempre meglio richiamare il suo costruttore.

Vediamo ora una versione modificata della classe Verdura:

```
class Verdura(Cibo):
    def __init__(self, proteine=0,
carboidrati=0):
        Cibo.__init__(self,
proteine = proteine,
carboidrati = carboidrati,
grassi = 0)
```

Osserviamo la sintassi utilizzata per richiamare il metodo costruttore originale: abbiamo usato il

nome della classe base seguito da `__init__` passandogli l'argomento `self`. Possiamo così, se ci serve, riutilizzare il costruttore della classe originaria. In questo caso abbiamo semplicemente forzato a 0 uno dei parametri, grassi, che nella classe “madre” poteva invece essere passato esplicitamente.

I metodi privati

In Python esiste il concetto di metodo o attributo privato presente in altri linguaggi di programmazione, per esempio in C++ e in Java: un metodo o un attributo privato non può essere richiamato dall'esterno della classe cui appartiene.

Ma a differenza di altri linguaggi di programmazione, per ottenere questa “privacy” non abbiamo a disposizione una parola chiave ma solo una convenzione di denominazione (in inglese *naming convention*). Un metodo o un attributo privato deve iniziare (ma non terminare!) con una

sequenza di due caratteri underscore “__”.

Proviamo ora a creare una classe con un metodo privato:

```
>>> class Archivio:  
    def __apri_file(self):  
        print "io sono nascosto"  
>>>
```

Creiamo un’istanza e proviamo ad usare il metodo privato della classe:

```
>>> arc = Archivio()  
>>> arc.__apri_file()  
Traceback (most recent call last):  
  File "<pyshell#55>", line 1, in  
<module>  
    arc.__apri_file()  
AttributeError: Archivio instance has no  
attribute  
'__apri_file'  
>>>
```

Come vediamo, Python si è rifiutato di eseguire il metodo che avevamo definito, rispettando la convenzione che lo rende privato.

Proviamo ora a eseguire dir sulla classe Archivio:

```
>>> dir(Archivio)
['_Archivio__apri_file', '__doc__',
 '__module__']
>>>
```

Il metodo `__apri_file` ha subito un'operazione che in inglese viene definita name mangling, espressione traducibile come “storpiatura” dei nomi.

Qui vediamo all’opera un altro degli aspetti caratteristici di Python: il linguaggio è stato creato per facilitare la vita di chi scrive programmi e, anche in questo caso, lo fa nel modo migliore.

Infatti, se, per qualsiasi motivo, volessimo a ogni costo e a nostro rischio e pericolo accedere a un metodo che avevamo definito come privato, lo possiamo fare:

```
>>> arc = Archivio()
>>> arc._Archivio__apri_file()
```

io sono nascosto
">>>>

I metodi speciali

Nei capitoli precedenti abbiamo visto come la sintassi di Python sia estremamente concisa e potente, pensiamo per esempio allo slicing delle stringhe. Sarebbe bello poter scrivere delle classi che possono essere utilizzate con questa sintassi o magari con gli operatori aritmetici e booleani. Poteva Python deluderci? Certo che no! Infatti è possibile ottenere questo risultato. Come? Ancora una volta con la naming convention.

Nota In altri linguaggi, questo approccio è chiamato “operator overloading” la cui traduzione letterale, “sovraffabbricazione degli operatori”, non è certamente intuitiva (forse sarebbe meglio parlare di “sovraposizione degli operatori”).

Esistono infatti molti nomi di metodo speciali,

special method names, che ci permettono di adattare la nostra classe alla sintassi di Python. L'elenco è assai lungo ed è disponibile nella documentazione; pertanto in queste pagine vedremo solo qualche esempio che ci possa aiutare a capire questo meccanismo.

La funzione nativa `len` ci permette di valutare la lunghezza di una stringa o di una sequenza. Se vogliamo utilizzarla anche con una delle nostre classi, dobbiamo implementare il metodo speciale `__len__` come nell'esempio seguente:

```
>>> class Gag:
    def __init__(self, durata):
        self.durata = durata
    def __len__(self):
        return self.durata
>>>
>>> blue_parrot = Gag(durata=13)
>>> print len(blue_parrot)
13
>>>
```

La funzione nativa `str` ci permette invece di

convertire in stringa qualsiasi oggetto. Il corrispondente metodo speciale è `__str__`:

```
>>> class Gag:  
def __init__(self, titolo):  
    self.titolo = titolo  
def __str__(self):  
    return "Sono la gag '%s'" % self.titolo  
>>>  
>>> spam = Gag("Spam, spam ,spam")  
>>> print str(spam)  
Sono la gag 'Spam, spam ,spam'  
>>>
```

Se invece volessimo utilizzare gli operatori booleani di comparazione `>`, `<`, `==`, `<=` e `>=` i metodi speciali che dovremmo utilizzare sono rispettivamente `__gt__`, `__lt__`, `__eq__`, `__le__`, `__ge__` dove `gt` sta per greater than (maggiore di), `lt` per less than (minore di), `eq` per equal (uguale), `le` per less equal (minore o uguale), `ge` per greater equal (maggiore o uguale). Vediamo un esempio completo dell'uso di questi metodi nella Figura 8.1.

Nota Il lungo elenco dei metodi speciali definiti in Python è presente nella documentazione, che possiamo visualizzare premendo F1 da IDLE; in alternativa tale elenco è accessibile online nel sito di Python. L'indirizzo dal quale possiamo reperire la documentazione riguardante i metodi speciali è:
<http://www.python.org/doc/ref/specialnames..>

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2

```
>>> class Comico:
    def __init__(self, numero_risate):
        self.numero_risate = numero_risate
    def __gt__(self, other):
        return (self.numero_risate >
                other.numero_risate)
    def __lt__(self, other):
        return (self.numero_risate <
                other.numero_risate)
    def __eq__(self, other):
        return (self.numero_risate ==
                other.numero_risate)
    def __le__(self, other):
        return (self.numero_risate <=
                other.numero_risate)
    def __ge__(self, other):
        return (self.numero_risate >=
                other.numero_risate)
```

```
>>>
>>> Cleese = Comico(9)
>>> Boldi = Comico(6)
>>> Palin = Comico(9)
>>>
>>> print Cleese > Boldi
True
>>> print Boldi == Palin
False
>>> print Cleese == Palin
True
>>>
```

Ln: 42 Col: 4

Figura 8.1 I metodi speciali per gli operatori booleani.

CAPITOLO 9

Input e output

Sostanzialmente, tutti i programmi che abbiamo realizzato e che realizzeremo hanno una caratteristica in comune: accettano dei valori in input, svolgono delle operazioni su tali valori e producono qualcos'altro in output.

È una definizione un po' sintetica ma, se ci pensiamo bene, non è molto distante dalla realtà. Le funzioni di input e di output sono quindi essenziali per tutti i linguaggi di programmazione; in questo capitolo vedremo come se la cava Python.

Il comando print

Negli esempi che abbiamo visto nei capitoli precedenti, più volte era presente il comando

print. È un comando davvero semplice da utilizzare e dunque non ci soffermeremo più di tanto su di esso.

Il comando print mostra sullo standard output la lista di espressioni che gli sono state passate, dopo averle opportunamente convertite in stringa:

```
>>> print 1000, "spam", 1, "egg"  
1000 spam 1 egg  
>>>
```

Alla fine dell'elenco, print inserisce automaticamente un carattere di “a capo”, a meno che l'ultimo carattere non sia una virgola:

```
>>> for n in range(10):  
    print "spam",  
spam spam spam spam spam spam spam  
spam spam  
>>>
```

Tipicamente utilizzeremo il comando print in situazioni particolari di scrittura su log o su console. Nelle applicazioni dotate di

un'interfaccia avanzata non ricorreremo mai alla visualizzazione sullo standard output.

Attenzione Esiste in realtà una sintassi particolare del comando `print` che permette l'output su un oggetto di tipo `file`. Questa modalità, detta “`print chevron`”, verrà descritta in uno dei prossimi paragrafi.

L'operatore %

Per chi conosce il linguaggio C, l'operatore `%` è l'equivalente in Python della funzione `sprintf`. Per gli altri, invece, possiamo dire che è un operatore che permette la formattazione avanzata delle stringhe.

Vediamo assieme un semplice esempio:

```
>>> print "Ho mangiato %d uova e %d
spam" % (3, 5)
Ho mangiato 3 uova e 5 spam
>>>
```

L'operatore % cerca nella stringa che lo precede la presenza dei simboli % e li sostituisce con il valore (o i valori) contenuti nella lista che lo segue. Il simbolo % all'interno della stringa da formattare deve essere seguito da un carattere che indica il tipo di dato che si sta inserendo nella stringa. Nell'esempio precedente abbiamo utilizzato %d, che indica la presenza di un numero intero decimale nella lista di valori.

Se il tipo del valore non corrisponde a quello dichiarato, viene sollevata un'eccezione. Nell'esempio seguente vedremo cosa succede se proviamo a visualizzare una stringa usando il carattere d:

```
>>> print "Errore %d" % ("1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    "Errore %d" % ("1")
TypeError: int argument required
>>>
```

Altri caratteri che possiamo utilizzare dopo il carattere % sono i seguenti: f(floating , numero in

virgola mobile), c (carattere singolo), s (stringa). Ve ne sono altri, meno comuni, che potremo trovare nella documentazione di Python.

Qualcuno si chiederà come possiamo stampare il simbolo %. Semplice: specificando un doppio simbolo %:

```
>>> print "In %s %% formatta le %s" %  
('Python',  
     'stringhe')
```

```
In Python % formatta le stringhe  
>>>
```

Con l'operatore % possiamo anche utilizzare alcuni flag per migliorare il controllo sull'output. Per esempio possiamo specificare un numero che stabilisce le dimensioni minime dei campi da visualizzare:

```
>>> for x in range(4):  
    print '%6d' % (9 ** x)  
    1  
    9  
    81
```

>>>

In questo caso abbiamo visualizzato quattro numeri, allineandoli tutti tramite una sequenza di spazi a sinistra (grazie al valore 6 nell'espressione %6d).

Possiamo fare lo stesso utilizzando un valore non indicato esplicitamente:

```
>>> for x in range(4):
    print '%*d' % (6, 9 ** x)
    1
    9
   81
  729
>>>
```

In questo modo possiamo formattare i valori servendoci di un valore variabile e non prefissato.

Come abbiamo visto, i valori vengono formattati e allineati a destra. Con il flag - possiamo allinearli a sinistra:

```
>>> for x in range(4):
print '> %-6d <' % (9 ** x)
> 1           <
> 9           <
> 81          <
> 729         <
>>>
```

E per qualcosa di più speciale possiamo utilizzare assieme all'operatore % i metodi del tipo dati stringa:

```
>>> for man in ("graham", "john",
                  "terry", "michael"):
    print ">%s<" %
(man.capitalize().center(25),)

>           Graham           <
>           John            <
>           Terry           <
>           Michael         <
>>>
```

In questo esempio abbiamo in un colpo solo centrato una lista di stringhe e messo in maiuscolo la lettera iniziale.

Esiste infine una modalità ancora più avanzata, che impiega l'operatore % associato a un dizionario:

```
>>> piramide = {"nome": "Cheope",
                "base": 233,
                "altezza": 137}
>>> print ("Volume piramide di %(nome)s
in metri" +
          " cubi = %(base)d * %(base)d *"
+
          " %(altezza)d / 3") % piramide
```

Volume piramide di Cheope in metri cubi
= 233 * 233 * 137 / 3

```
>>>
```

Nota Per i più curiosi: secondo questo calcolo il volume della piramide di Cheope è di 2.479.197 metri cubi. Comprendendo la copertura (ormai completamente asportata) il volume della piramide superava i 2.600.000 metri cubi.

I file

Per accedere a un file, Python mette a disposizione la classe predefinita `file`. Un oggetto creato tramite questa classe offre diversi metodi che ci permettono di leggere e scrivere il contenuto del file.

Creiamo un oggetto di tipo `file`:

```
>>> f = file("prova.tmp", "w")  
>>>
```

Attenzione *Attenzione: con questa istruzione un eventuale file prova.tmp contenuto nella directory corrente, verrà cancellato!*

Il secondo argomento “`w`” (dall’inglese `write`, scrivere) passato alla classe `file` indica che vogliamo aprire il file in scrittura, eventualmente cancellandolo se il file esiste già. In alternativa possiamo utilizzare “`r`” (`read`, leggere) per aprire in lettura un file già esistente o “`a`” (`append`, aggiungere) per aprire in scrittura un file già esistente, aggiungendo in fondo i nuovi dati.

Proviamo a elencare con dir i metodi di f:

```
>>> dir(f)
['__class__', '__delattr__', '__doc__',
 '__enter__', '__exit__',
 '__getattribute__',
 '__hash__', '__init__', '__iter__',
 '__new__',
 '__reduce__', '__reduce_ex__',
 '__repr__',
 '__setattr__', '__str__', 'close',
 'closed',
 'encoding', 'fileno', 'flush', 'isatty',
 'mode',
 'name', 'newlines', 'next', 'read',
 'readinto',
 'readline', 'readlines', 'seek',
 'softspace',
 'tell', 'truncate', 'write',
 'writelines',
 'xreadlines']
>>>
```

Questo lungo elenco ci lascia intuire che anche con i file Python non lesina metodi che soddisfino anche i palati più esigenti. Come già per altri elenchi simili presentati nei capitoli precedenti,

vediamo rapidamente solo i metodi più importanti, lasciando alla documentazione la descrizione dettagliata di tutti gli altri.

Il metodo `close`, senza argomenti, chiude l'oggetto file rimettendolo a disposizione di altre applicazioni o del sistema operativo. Per poter eseguire altre operazioni di scrittura o lettura dovremo quindi riaprirlo.

L'attributo booleano `closed` dice se il file è già chiuso oppure no.

L'attributo stringa `mode` dice la modalità con la quale è stato aperto il file.

L'attributo stringa `name` dice il nome fisico del file.

`read` seguito da un numero intero `n` legge al massimo `n` byte da un file aperto in lettura. Se nel file sono rimasti meno di `n` byte, vengono restituiti solo quelli presenti nel file.

write seguito da una stringa s scrive il contenuto della stringa in un file aperto in scrittura. I file aperti in scrittura possono anche essere utilizzati con il comando print nella sintassi chiamata chevron:

```
>>> f = file("prova.tmp", "w")
>>> print >> f, "Spam, spam e ancora
spam"
>>> f.close()
>>>
```

readline e readlines leggono rispettivamente una o tutte le righe presenti in un file di testo.

Nel caso di readlines, il valore restituito è la lista delle righe lette, che possiamo scorrere per esempio con for:

```
>>> f = file("prova.tmp", "w")
>>> f.write("Tizio\nCaio\nSempronio\n")
>>> f.close()
>>> f = file("prova.tmp", "r")
>>> for l in f.readlines():
...     print l,
...
Tizio
```

Caio
Sempronio
>>>

I toolkit grafici

Parlare di input e output pensando solo a file, a print o a standard output è sicuramente riduttivo. Ormai chiunque di noi abbia avuto a che fare con un sistema operativo qualsiasi tra le diverse varianti di Linux, Windows, Mac OS X e via dicendo, non può non aver utilizzato una GUI (da Graphic User Interface, interfaccia utente grafica) avanzata.

Come i più ottimisti avranno già intuito, per Python esistono varie librerie grafiche che consentono di realizzare applicativi dotati di un'interfaccia grafica accattivante e che mantengono comunque le caratteristiche di portabilità sulle diverse piattaforme. La programmazione di applicazioni dotate di un'interfaccia grafica avanzata richiederebbe dello spazio che qui non abbiamo, ma cercheremo lo stesso di gettare uno sguardo

oltre l'ostacolo: con un po' di buona volontà e con la documentazione sottomano potremo senza dubbio cavarsela da soli.

Agendo anche noi per una volta da dittatori benevolenti, sceglieremo d'autorità il toolkit wxPython, che possiamo reperire gratuitamente insieme al codice sorgente dal sito <http://www.wxpython.org>. Siamo sicuri che ne esistono molti altri ugualmente potenti e versatili tant'è vero che solo in questo link sul sito ufficiale di Python ne sono elencati oltre venticinque (!):

<http://wiki.python.org/moin/GUIProgrammingWithwxPython>

wxPython

Uno dei grandi vantaggi di wxPython è l'avanzatissima applicazione demo che possiamo scaricare ed eseguire assieme al toolkit. Una volta installato il toolkit, possiamo lanciare la demo da Programmi/ wxPython2.7 Docs Demos and Tools/Run the wxPython DEMO.

Dopo il simpatico splash screen che vediamo in Figura 9.1, ci apparirà la schermata rappresentata nella Figura 9.2.

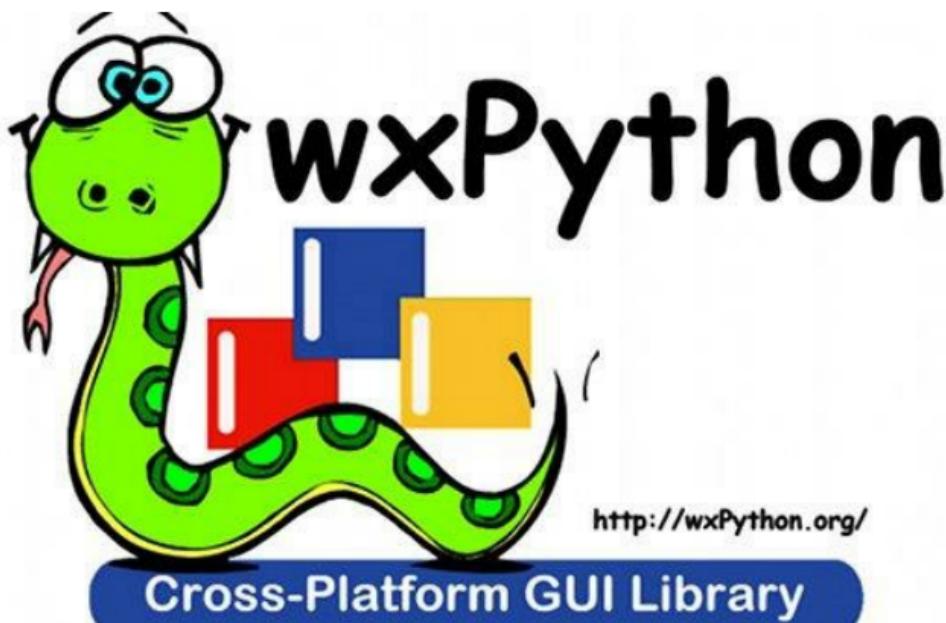


Figura 9.1 Lo splash screen della demo di wxPython.

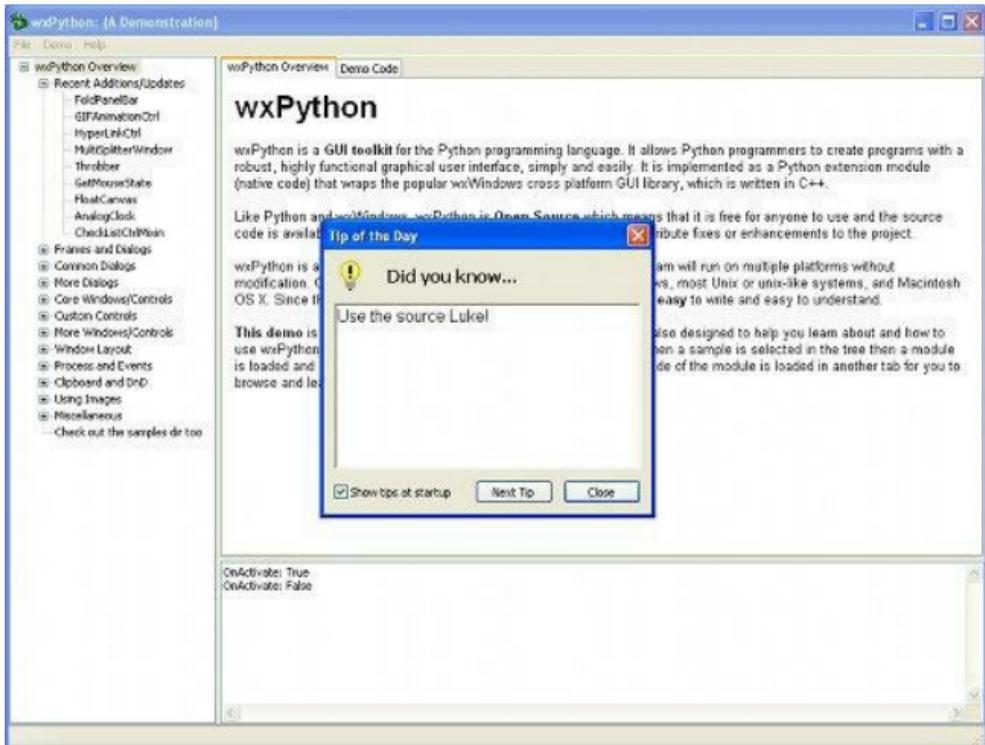


Figura 9.2 Il menu iniziale della demo di wxPython.

In primo piano vediamo una finestra di popup che mostra, a ogni avvio della demo, un suggerimento diverso. Il suggerimento nella Figura 9.2 dice “Use the source Luke!” (Usa il sorgente Luke!). Il riferimento al film Guerre Stellari è ironico, ma la sostanza del suggerimento è cruciale per l’uso

della demo di wxPython: ogni elemento dell'elenco rappresentato nella Figura 9.3 raggruppa diversi esempi e per ogni esempio possiamo vedere il codice, modificarlo e provarlo in tempo reale senza uscire dalla demo.

Proviamo a modificare il codice di un esempio per scoprire quanto sia facile fare esperimenti con questa demo.

Espandiamo la voce More Windows/Controls dell'elenco di gruppi di esempi e facciamo clic sulla voce MaskEditControls.

Dovrebbe apparire una finestra simile a quella rappresentata nella Figura 9.4. Nella parte superiore della finestra compaiono le tre schede MaskEditControls Overview (panoramica dei controlli con maschera di inserimento), Demo Code (codice della demo) e Demo (demo vera e propria). Per ogni singolo esempio dei circa 200 disponibili saranno presenti le stesse informazioni: panoramica, codice sorgente, esempio eseguibile.

Torniamo al nostro esempio e sbizzarriamoci a provare i diversi esempi di controlli avanzati presenti nelle diverse sottoschede della demo.

Proviamo ora a fare una piccola modifica all'esempio originale. Facciamo clic su Demo Code per esaminare il codice sorgente, come indicato nella Figura 9.5.

The screenshot shows a tree view interface titled "wxPython Overview". The root node is expanded, revealing a list of subtopics. Each topic is preceded by a plus sign (+) indicating it can be collapsed. The subtopics are:

- + Recent Additions/Updates
- + Frames and Dialogs
- + Common Dialogs
- + More Dialogs
- + Core Windows/Controls
- + "Book" Controls
- + Custom Controls
- + More Windows/Controls
- + Window Layout
- + Process and Events
- + Clipboard and DnD
- + Using Images
- + Miscellaneous
- + Check out the samples dir too

Figura 9.3 L'elenco dei gruppi di esempi presenti nella demo di wxPython.

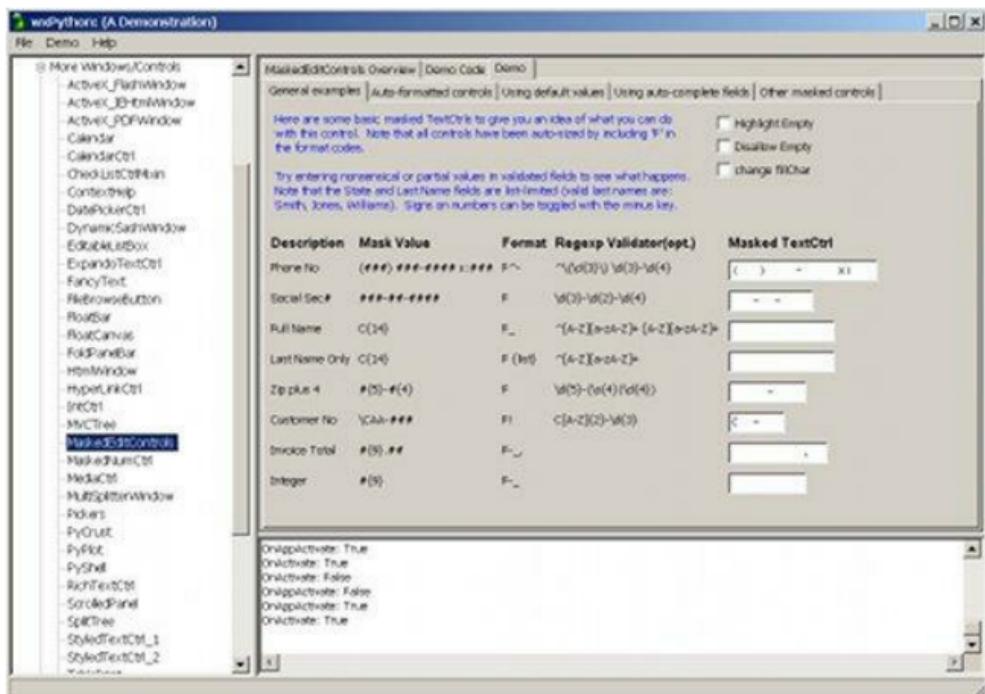


Figura 9.4 L'esempio MaskedEditControls.

The screenshot shows a Windows application window titled "wxPython (A Demonstration)". The menu bar includes "File", "Demo", and "Help". A sidebar on the left lists various wxPython control classes. The main area displays Python code for a "MaskedEditControls" demo. The code defines a class "demoMixin" with a static method "labelGeneralTable" that creates several static text controls with specific descriptions and fonts. Below the code, a status bar shows some wxPython properties.

```
1 import string
2 import sys
3 import traceback
4
5 import wx
6 import wx.lib.masked      as masked
7 import wx.lib.scrolledpanel as scroll
8
9
10
11 class demoMixin:
12     ...
13     # Centralized routines common to demo pages, to remove repetition.
14
15     def labelGeneralTable(self, sizer):
16         description = wx.StaticText(self, -1, "Description")
17         mask = wx.StaticText(self, -1, "Mask Value")
18         formatted = wx.StaticText(self, -1, "Format")
19         regex = wx.StaticText(self, -1, "Regexp Validator(opt.)")
20         ctrl = wx.StaticText(self, -1, "Masked TextCtrl")
21
22         description.SetFont(wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD))
23         mask.SetFont(wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD))
24         formatted.SetFont(wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD))
25         regex.SetFont(wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD))
26
27
28     OnDragactivate: True
29     OnDclick: True
30     OnDright: False
31     OnDdouble: False
32     OnDleft: True
33     OnDrightdown: True
```

Figura 9.5 Il codice sorgente dell'esempio MaskedEditControls.

Modifichiamo la stringa “Description” presente alla riga 16 del codice scrivendo invece “Monty Python”, facciamo clic sul pulsante Save Changes (salva i cambiamenti) e infine facciamo clic sulla scheda Demo. La demo sarà pressoché invariata, tranne che per la scritta che abbiamo modificato e che abbiamo indicato con una freccia nella Figura

9.6. Non possiamo elencare tutti gli esempi a disposizione nella demo: sono moltissimi e spaziano dall'uso del joystick, alla visualizzazione di pagine HTML, di file PDF, di applicazioni Flash (osservate la Figura 9.7; ricordate il gioco Asteroids?).

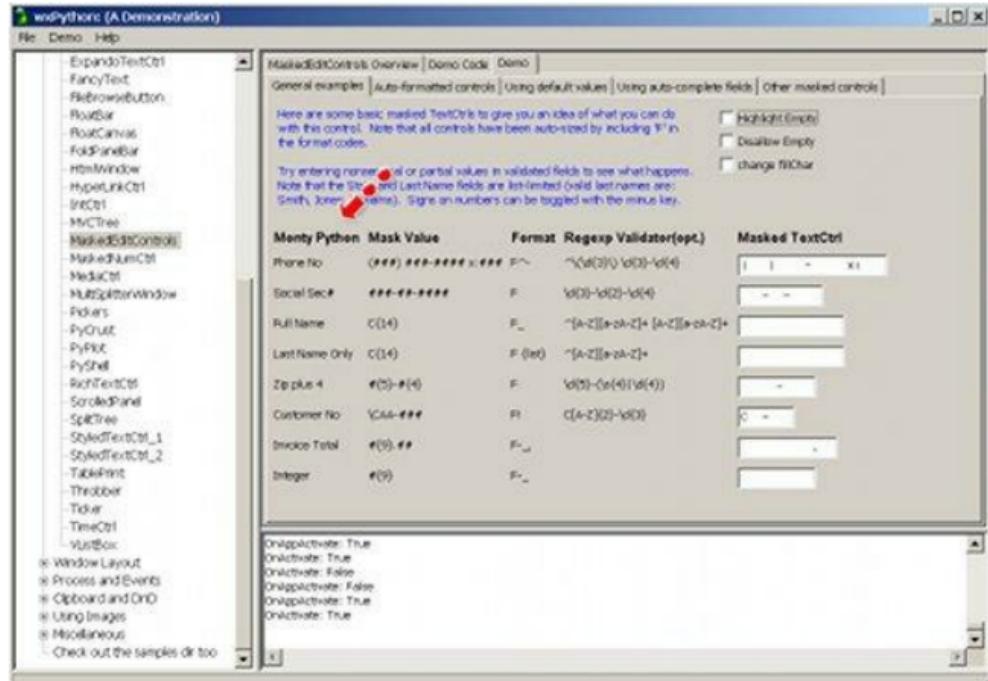


Figura 9.6 L'esempio MaskedEditControls modificato.

In pratica possiamo considerare la demo di wxPython una sorta di raccolta di esempi di codice sorgente che possiamo tranquillamente collaudare, modificare, copiare e utilizzare nei nostri programmi.

Come conclusione di questo brevissima panoramica su uno dei toolkit grafici disponibili per Python, proviamo a creare un'applicazione estremamente semplice dal prompt interattivo dei comandi.

Inseriamo il seguente codice in IDLE:

```
>>> import wx  
>>> app = wx.PySimpleApp()  
>>> frame = wx.Frame(None, -1, "Hello  
World")  
>>> frame.Show(1)  
>>> app.MainLoop()
```

Le Figure 9.8 e 9.9 mostrano il risultato di questo piccolo esempio da IDLE e dal prompt dei comandi di Windows: poter creare delle finestre interattivamente non può lasciarci indifferenti!

Python ci ha sorpreso un'altra volta...

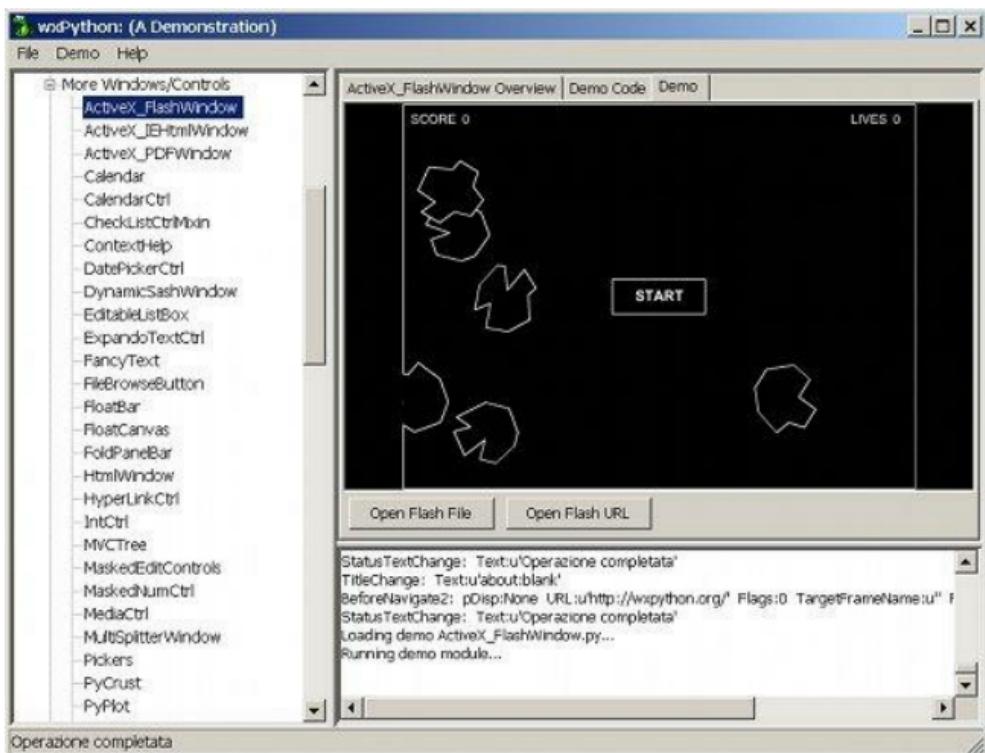


Figura 9.7 L'esempio di visualizzazione di un'applicazione Flash.

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2

```
>>> import wx  
>>> app = wx.PySimpleApp()  
>>> frame = wx.Frame(None, -1, "Hello World")  
>>> frame.Show(1)  
True  
>>> app.MainLoop()
```



Ln: 16 Col: 0

Figura 9.8 Una finestra creata interattivamente con wxPython in IDLE.

C:\>python

```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win  
32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import wx  
>>> app = wx.PySimpleApp()  
>>> frame = wx.Frame(None, -1, "Hello World")  
>>> frame.Show(1)  
True  
>>> app.MainLoop()
```



Figura 9.9 La stessa finestra creata da prompt dei comandi.

CAPITOLO 10

I moduli

L'elenco dei moduli già pronti all'uso con l'installazione base è eccezionalmente ricco. Oltre a questa base di partenza abbiamo a disposizione un ampio e ricchissimo Cheese shop (negozi di formaggi) all'indirizzo:

<http://cheeseshop.python.org/pypi>

Nota È forse inutile sottolineare che “Cheese Shop” è un famosissimo sketch dei Monty Python.

In questo momento vi sono elencati ben 1.782 (!) pacchetti scaricabili e utilizzabili (wxPython, esaminato nel capitolo precedente, è uno di questi).

Dedicare anche una sola riga a ogni pacchetto esaurirebbe tutto lo spazio disponibile per questo libro. Ciononostante possiamo dare uno sguardo ai pacchetti più utilizzati, per comprendere la quantità e la qualità di codice (e quindi di funzionalità), già pronto all'uso e collaudato; i pacchetti non compresi nell'installazione, sono scaricabili gratuitamente da Internet.

sys

Quando eseguiamo il comando import, Python cerca di leggere e caricare in memoria il file che abbiamo specificato nel codice. Ma da dove vengono importati i file? In quali directory Python cerca i file dei moduli?

Il modulo sys è particolarmente importante proprio perché definisce la variabile sys.path che non è altro che l'elenco di directory in cui Python cerca i moduli da caricare. Se non riusciamo a importare un modulo che crediamo di avere installato, può essere utile verificare il valore di sys.path.

Un'altra variabile importante presente in questo modulo è argv. Chi conosce il linguaggio C avrà già intuito il suo significato: è la lista degli argomenti passati al nostro script sulla riga di comando.

Proviamo a creare uno script argomenti.py con queste poche righe di codice:

```
import sys  
for arg in sys.argv:  
    print arg
```

Nella Figura 10.1 possiamo vedere il risultato della sua esecuzione con il passaggio di 4 argomenti: “1”, “2”, “3” e “prova”.

Come possiamo vedere, l'elenco è composto anche dal nome dello stesso script che viene quindi considerato alla stregua dell'argomento numero 0.

Attenzione *Attenzione: tutti gli elementi della lista sys.argv sono considerati*

stringhe.

The screenshot shows a Windows command-line interface (cmd.exe) window. The title bar reads "C:\WINDOWS\system32\cmd.exe". The command prompt is at "C:\Work\Python>". The user has run the command "python argomenti.py 1 2 3 prova". The output shows the program "argomenti.py" printing the integers 1, 2, and 3, followed by the string "prova".

```
C:\Work\Python>python argomenti.py 1 2 3 prova
argomenti.py
1
2
3
prova
C:\Work\Python>
```

Figura 10.1 L'uso della variabile sys.argv.

OS

Il modulo os è estremamente utile quando
dobbiamo interagire con il sistema operativo che
ospita il nostro programma. Ci permette infatti di

eseguire tutta una serie di operazioni senza preoccuparci se siamo su Windows, Linux, Mac OS X e così via. L'elenco di operazioni che possiamo effettuare è incredibilmente ampio per cui, obbligatoriamente, dobbiamo fare una selezione rappresentativa.

`chdir` permette di cambiare la directory di lavoro corrente.

`rename` permette di rinominare un file.

`remove` permette di cancellare un file.

`path.exists` permette di verificare l'esistenza di un file.

`path.getsize` permette di conoscere le dimensioni di un file.

`path.isdir` permette di verificare se una directory esiste.

`path.split`, `path.splitext` e `path.basename`

permettono di separare un percorso nelle tre parti directory, nome del file ed estensione del file.

Vediamo un esempio d'uso per queste ultime tre funzioni:

```
>>> import os  
>>> os.path.split("C:/folder/nome.ext")  
('C:/folder', 'nome.ext')  
>>>  
os.path.splitext("C:/folder/nome.ext")  
('C:/folder/nome', '.ext')  
>>>  
os.path.basename("C:/folder/nome.ext")  
'nome.ext'  
>>>
```

La prima funzione, path.split, ha separato la directory dal nome del file. La seconda, path.splitext, ha separato l'estensione da tutto il resto del percorso. La terza, path.basename, ha estratto il nome del file dal percorso.

zipfile, tarfile, zlib, gzip, bz2

Il modulo zipfile ci permette di aprire, creare, modificare file in formato zip. I moduli tarfile, zlib, gzip e bz2 ci permettono invece di gestire file rispettivamente in formato tar, zlib, gz e bz2.

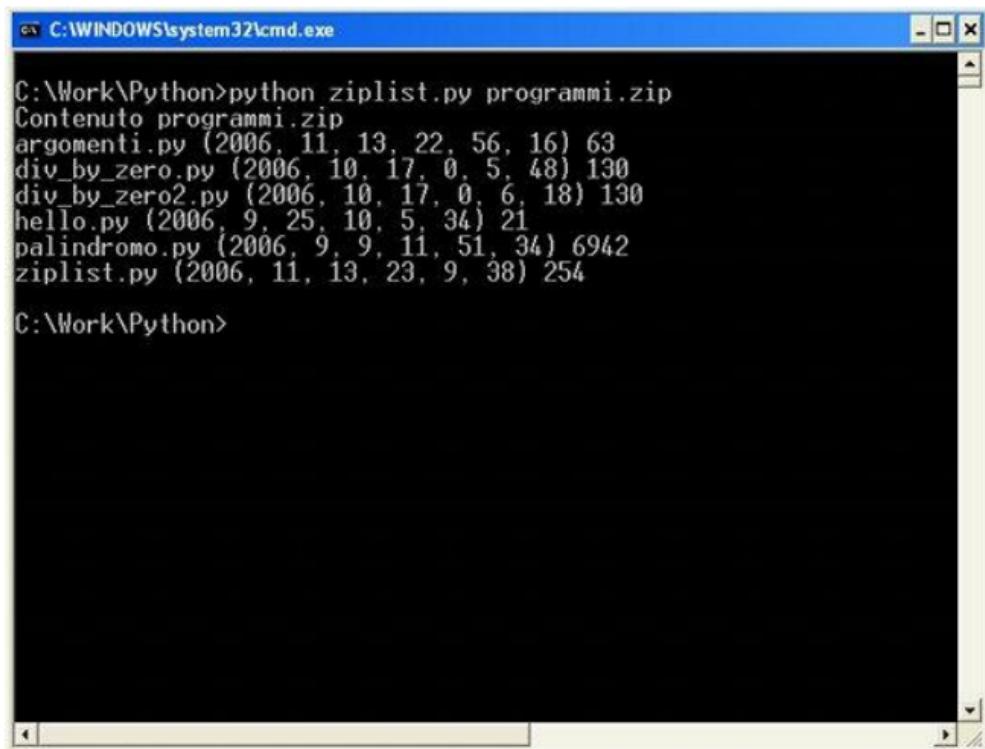
Proviamo a scrivere un programma che elenca i file contenuti nell'elenco di file zip passati sulla riga di comando.

Lo script, che chiameremo ziplist.py, è molto semplice e composto da poche righe di codice:

```
import zipfile, sys
for zip in sys.argv[1:]:
    print "Contenuto %s" % zip
    file_zip = zipfile.ZipFile(zip, "r")
    for info in file_zip.infolist():
        print info.filename,
    info.date_time, \
        info.file_size
```

Nella Figura 10.2 vediamo il risultato dell'esecuzione su un file zip contenente alcuni script.

Anche se lo script è assai semplice e non dovremmo quindi avere molti dubbi nell'interpretarlo, vediamo in dettaglio il suo funzionamento.



The screenshot shows a Windows command-line interface (cmd.exe) window. The title bar reads "C:\WINDOWS\system32\cmd.exe". The command entered is "C:\Work\Python>python ziplist.py programmi.zip". The output lists the contents of the "programmi.zip" file, which includes several Python files and their details:

```
C:\Work\Python>python ziplist.py programmi.zip
Contenuto programmi.zip
argomenti.py (2006, 11, 13, 22, 56, 16) 63
div_by_zero.py (2006, 10, 17, 0, 5, 48) 130
div_by_zero2.py (2006, 10, 17, 0, 6, 18) 130
hello.py (2006, 9, 25, 10, 5, 34) 21
palindromo.py (2006, 9, 9, 11, 51, 34) 6942
ziplist.py (2006, 11, 13, 23, 9, 38) 254
```

C:\Work\Python>

Figura 10.2 Utilizzo della classe ZipFile.

La riga di codice `for zip in sys.argv[1:]`: fa sì che venga saltato il primo argomento che, come

abbiamo visto nel precedente paragrafo, corrisponde al nome dello script. Quindi, dopo un messaggio che indica il file che stiamo per aprire, viene istanziato un oggetto di tipo ZipFile. Infine utilizziamo il metodo infolist che restituisce per ogni file contenuto nell'archivio zip un oggetto di tipo ZipInfo che lo descrive. L'oggetto ZipInfo offre un lungo elenco di attributi, tra cui il nome del file, filename, la data del file, date_time, e le sue dimensioni file_size.

Nella classe ZipFile esistono ovviamente altri metodi che ci permettono di leggere il contenuto dei file contenuti in un archivio .zip o di aggiungervi altri file. Non è invece possibile rimuovere un file da un archivio, per ottenere questo risultato dobbiamo creare un nuovo archivio temporaneo che non contenga i file da eliminare e alla fine sostituirlo all'archivio originale.

Il funzionamento degli altri moduli tarfile, zlib, gzip e bz2 è del tutto analogo.

shelve, pickle

Abbiamo ormai imparato che Python ci permette di creare facilmente oggetti di alto livello. Abbiamo anche visto quanto sia facile leggere il contenuto di un file o creare un nuovo file. Ma come possiamo salvare in un file un oggetto di tipo avanzato per poterlo poi leggere in un secondo momento?

Il modulo shelve è quello che fa per noi. È davvero incredibile la facilità con la quale è possibile creare con questo modulo un piccolo database object oriented:

```
>>> import shelve
>>> db = shelve.open("db")
>>> class Egg:
...     def __init__(self, nome):
...         self.nome = nome
>>> a = Egg("Cleese")
>>> a.nome
'Cleese'
>>> db["jc"] = a
>>> db.close()
```

Con queste poche righe abbiamo appena creato un database di nome db contenente un oggetto di tipo Egg. L'oggetto è stato salvato usando db proprio come un dizionario, specificando una chiave che ci permetterà in un secondo momento di recuperare e rileggere l'oggetto.

Ora chiudiamo l'interprete e riapriamolo per essere sicuri di non avere più nulla in memoria che riguardi il nostro oggetto Egg.

Adesso ridefiniamo la classe Egg e proviamo a rileggere l'oggetto che abbiamo appena salvato:

```
>>> import shelve
>>> class Egg:
...     def __init__(self, nome):
...         self.nome = nome
>>> db = shelve.open("db")
>>> z = db["jc"]
>>> z.nome
'Cleese'
>>>
```

Facile, vero? E con shelve possiamo salvare

praticamente ogni oggetto creato in Python.

Perchè nel titolo di questo paragrafo abbiamo elencato anche il modulo pickle? Perchè questo modulo permette la “serializzazione” e “deserializzazione” di un qualsiasi oggetto Python.

La serializzazione è il processo che trasforma un oggetto in una sequenza di byte e la deserializzazione è il processo inverso. Il modulo shelve usa il modulo pickle sia per trasformare gli oggetti in una sequenza di byte e salvarli su disco sia per svolgere l’operazione inversa. Tutto ciò che può essere serializzato da pickle può quindi essere salvato da shelve.

smtplib, urllib, ftplib

Al giorno d’oggi è difficile che non ci capiti di dover realizzare programmi che hanno in qualche modo a che fare con Internet. Esistono moltissimi moduli Python che ci aiutano in questo compito, ne esamineremo rapidamente solo alcuni.

Dobbiamo inviare un messaggio email? Il modulo che fa per noi è smtplib:

```
import smtplib  
host=smtplib.SMTP("mail.server.it")  
ret=host.sendmail("otello@venezia.it",  
                  "desdemona@venezia.it",  
                  "Cara Desdy, dov'eri ieri?")
```

Difficile trovare un modo più semplice per inviare un messaggio email.

Dobbiamo leggere il contenuto di un sito? Il modulo giusto è urllib:

```
import urllib  
urllib.urlretrieve("http://www.google.co  
\\  
                    "locale.html")
```

Con due sole righe di codice possiamo salvare su un file il contenuto di una pagina Internet.

Dobbiamo collegarci a un sito ftp? Utilizziamo ftplib come indicato nella Figura 10.3.

Qui le righe di codice sono ben cinque... forse troppe?

threading

Il concetto di thread (letteralmente “filo”, “filone”) è estremamente avanzato e complesso. Proviamo a immaginare un programma che deve eseguire più operazioni non in maniera sequenziale ma parallela. L'unica possibilità che abbiamo è quella di utilizzare nel nostro codice dei thread. In alternativa saremmo costretti a eseguire nel codice le operazioni in sequenza, con il rischio che un'operazione lenta ritardi quelle successive, che sono tutte in coda e quindi in attesa della conclusione delle operazioni precedenti.

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2

```
>>> import ftplib
>>> ftp = ftplib.FTP('ftp.cwi.nl')
>>> ftp.login()
'230 Login successful.'
>>> ftp.retrlines('LIST')
drwxrwxr-x 2 ftp      ftp          512 Jan 15 2001 DIENST
drwxr-xr-x 2 ftp      ftp          512 Nov 16 2004 incoming
-rw-r--r--  1 ftp      ftp         1810 Jul 05 2004 incoming.readme
lrwxrwxrwx  1 ftp      ftp          1 Nov 14 2004 people -> .
dr-xr-xr-x  76 ftp     ftp         1536 Oct 24 08:50 pub
drwxrwxr-x 10 ftp     ftp          512 Nov 09 2001 sigchi
-r--r--r--  1 ftp      ftp         2195 May 30 1995 welcome.msg
'226 Directory send OK.'
>>> ftp.quit()
'221 Goodbye.'
>>>
```

Figura 10.3 Utilizzo del modulo ftplib..

Solo la facilità con cui possiamo creare dei thread in Python ci permette di affrontare questo argomento in poco spazio. Proviamo quindi a realizzare un semplice programma che faccia uso dei thread; successivamente esamineremo in dettaglio le istruzioni utilizzate.

Una situazione tipica in cui potremmo voler

eseguire più operazioni contemporaneamente riguarda la realizzazione di un web spider (ragno del web), chiamato anche web crawler (“strisciatore” del web).

Nota *Un web spider è un programma che esplora il web in maniera metodica. Tutti i motori di ricerca utilizzano programmi di questo tipo per esplorare e indicizzare le pagine web.*

Il nostro spider, che chiameremo spider.py, è molto semplice e non fa altro che leggere tre indirizzi web e salvare le pagine web in altrettanti file locali:

```
import threading, urllib, time
urls = ['www.google.com',
        'www.linux.org',
        'www.python.org']
class esplora(threading.Thread):
    def __init__(self, url):
        threading.Thread.__init__(self)
        self.url = url
        self.status = -1
```

```
def run(self):
    print "Sto leggendo %s" %
self.url
    urllib.urlretrieve("http://%s" %
self.url,
                      self.url)
    print "Ho letto e salvato %s" %
self.url,
    print "alle ore %02d:%02d:%02d"
\
                % time.localtime() [3:6]
for url in urls:
    esplora_url = esplora(url)
    esplora_url.start()
```

Se proviamo ad eseguire spider.py otterremo un output simile a quello rappresentato nella Figura 10.4.

Prestiamo attenzione all'ordine in cui lo spider ha esaminato i siti: prima www.google.com, poi www.linux.org e infine www.python.org. Il salvataggio è invece avvenuto in ordine inverso: i tre thread sono partiti in sequenza, hanno lavorato in parallelo e hanno terminato in tempi diversi i rispettivi compiti.

```
C:\Work\python>python spider.py
Sto leggendo www.google.com
Sto leggendo www.linux.org
Sto leggendo www.python.org
Ho letto e salvato www.python.org alle ore 16:28:03
Ho letto e salvato www.linux.org alle ore 16:28:04
Ho letto e salvato www.google.com alle ore 16:28:08
C:\Work\python>
```

Figura 10.4 Output dello script spider.py.

Ma siamo sicuri che spider.py abbia svolto il proprio compito? Proviamo a elencare tutti i file il cui nome inizia per www contenuti nella directory in cui abbiamo eseguito lo script.

La schermata rappresentata nella Figura 10.5 ci conferma che tutto è andato per il verso giusto.

Come avevamo anticipato, diamo un'occhiata alle istruzioni dello script che ci hanno permesso di ottenere questo risultato.

Oltre allo scontato import threading, la prima riga da analizzare è:

```
class esplora(threading.Thread) :
```

Qui abbiamo definito una sottoclasse della classe Threading.Thread. In questo modo la nostra classe ne erediterà tutte le funzionalità. Gli unici metodi di cui possiamo (e normalmente dobbiamo) sostituire il codice sono `__init__` (il costruttore) e `run` (l'esecutore del thread).

Nel nostro metodo costruttore `__init__` abbiamo quindi come prima cosa richiamato il costruttore della classe madre (operazione richiesta per inizializzare il thread):

```
threading.Thread.__init__(self)
```

```
C:\Work\python>python spider.py
Sto leggendo www.google.com
Sto leggendo www.linux.org
Sto leggendo www.python.org
Ho letto e salvato www.python.org alle ore 16:28:03
Ho letto e salvato www.linux.org alle ore 16:28:04
Ho letto e salvato www.google.com alle ore 16:28:08
```

```
C:\Work\python>dir www.*
Volume in drive C is Local Disk
Volume Serial Number is 402A-8893
```

```
Directory of C:\Work\python
```

14/11/2006 16.28	2.741	www.google.com
14/11/2006 16.28	18.493	www.linux.org
14/11/2006 16.28	12.242	www.python.org
	3 File(s)	33.476 bytes
	0 Dir(s)	102.001.238.016 bytes free

```
C:\Work\python>
```

Figura 10.5 Verifica dei file recuperati e salvati da spider.py.

Abbiamo poi definito il metodo “esecutore” run e vi abbiamo inserito il codice da eseguire in parallelo:

```
def run(self):
```

Infine, dopo aver istanziato i nostri 3 thread, li abbiamo lanciati:

```
esplora_url = esplora(url)
esplora_url.start()
```

Tutto qui. Queste semplici istruzioni ci permettono di parallelizzare l'esecuzione del codice. Tutto il resto è frutto della nostra fantasia...

NumPY e SQLAlchemy

I moduli che abbiamo esaminato finora sono solo una piccola parte di quelli presenti nell'installazione di Python. È molto difficile non trovare tra i moduli standard quello che ci serve. Ma, come abbiamo detto, esistono centinaia di package disponibili nel Cheese Shop. Ne sceglieremo due tra i più famosi, rappresentativi della numerosa compagnia: NumPY e SQLAlchemy.

Data la loro complessità e specializzazione, ciascuno di questi due moduli richiederebbe molto

spazio per una trattazione anche solo superficiale per cui, per una volta, non mostreremo alcun esempio completo, che sarebbe di difficile comprensione, ma ci limiteremo a descrivere lo scopo di ognuno di questi due moduli. Chi fosse interessato ad approfondire l'argomento potrà visitare il Cheese Shop.

NumPY

Una delle (poche) critiche che vengono mosse a Python, e a tutti i linguaggi interpretati o semi-interpretati, è quella di essere lento rispetto a linguaggi compilati, come il C o il C++.

Occorre anche dire che i campi d'azione in cui tale differenza di velocità risulta effettivamente percepibile sono in realtà limitati. Uno di questi è senza dubbio il calcolo numerico puro, che viene spesso utilizzato in campo scientifico.

Il package NumPY vuole ovviare a questo problema, mettendoci a disposizione con la

sintassi di Python delle librerie realizzate in modo nativo in codice C, fondendo così la chiarezza e semplicità d'uso di Python con la velocità pura del linguaggio C.

Con NumPY possiamo realizzare programmi che si occupano di calcolo matriciale, di algebra lineare, di trasformate di Fourier e così via, contando sulle prestazioni del codice C e sulla chiarezza e semplicità del codice Python.

SQLAlchemy

Uno dei moduli che abbiamo visto in questo capitolo, shelve, ci permette il salvataggio di oggetti avanzati in un file binario. Al contrario, SQLAlchemy, svolge un'operazione quasi inversa: ci permette di operare su un database relazionale impiegando oggetti avanzati. Il termine tecnico che definisce SQLAlchemy è Object Relational Mapper (letteralmente “mappatore di database relazionali con logica orientata agli oggetti”).

Con SQLAlchemy possiamo quasi dimenticarci del codice SQL, delle tabelle, delle query e via dicendo. Il database viene interpretato da SQLAlchemy come un insieme di collezioni di oggetti, cui è possibile accedere con un livello di astrazione molto elevato.

Per esempio, per estrarre con SQLAlchemy da una ipotetica tabella utenti, tutti coloro il cui nome è “pippo” basta scrivere questo codice:

```
s =  
utenti.select(utenti.c.nome=='Pippo').exe  
lista = s.execute().fetchall()
```

Con una connessione diretta di basso livello avremmo dovuto più o meno scrivere questo codice:

```
c.execute("SELECT * FROM utenti WHERE  
nome='%s'",  
('pippo',))  
lista = c.fetchall()
```

Nel primo esempio non abbiamo scritto una sola

riga di codice SQL, diversamente da quello che abbiamo dovuto fare nel secondo esempio.

CAPITOLO 11

Un'applicazione completa

In questo capitolo costruiremo un'applicazione partendo da zero. Cercheremo di risolvere un problema non banale, applicando tutto ciò che abbiamo imparato fino a questo momento.

Vedremo poi come il nostro programma potrà essere eseguito, senza alcuna modifica al codice sorgente, su più piattaforme.

Il gioco “Full House”

Questo solitario ha delle regole molto semplici, ma ciononostante è piuttosto complesso da risolvere. L'ideatore di questo (e altri) puzzle è Erich Friedman, un professore di matematica della Stetson University in Florida. Chi lo desidera può visitare il suo sito, dove può trovare questo e altri rompicapi:

La Figura 11.1 presenta un problema e la sua soluzione: il percorso inizia dalla S (Start) e termina alla F (Finish).

Come dicevamo, le regole sono semplici. Il campo da gioco è una scacchiera; la maggior parte delle caselle è di colore bianco e le rimanenti sono nere. Lo scopo del gioco è quello di riempire le caselle bianche percorrendole tutte una sola volta, partendo da una casella bianca a piacere. Il percorso deve essere esclusivamente a tratti rettilinei, che possono cambiare direzione solo in corrispondenza del margine della scacchiera, di una casella nera o di una casella già visitata. Per ogni problema esiste una e una sola soluzione.

Osserviamo la Figura 11.1: possiamo notare che dalla lettera S il percorso procede rettilineo verso destra, incontra il margine, gira verso l'alto, incontra di nuovo il margine e così via fino alla lettera F, riempiendo tutte le caselle..

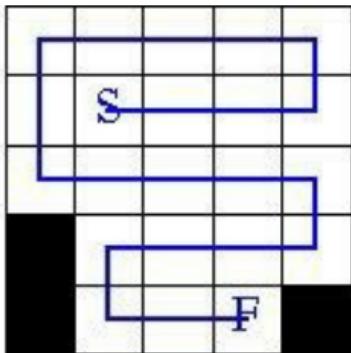
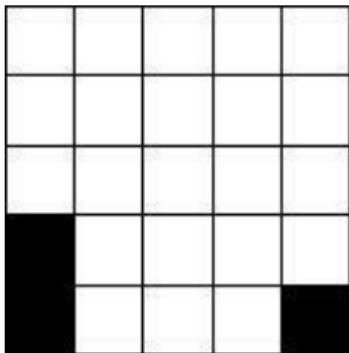


Figura 11.1 Un problema Full House e la sua soluzione.

Se il rompicapo ci sembra troppo facile, è solo perché abbiamo già visto la soluzione. Proviamo a risolvere lo schema rappresentato nella Figura 11.2. Così è un po' più difficile, vero? Ma più avanti vedremo degli schemi ancora più grandi e complessi.

L'aspetto grafico finale

La strada per arrivare alla realizzazione del programma non sarà brevissima e dovremo digitare diverse righe di codice. Per capire subito dove vogliamo arrivare, mostreremo nelle figure

seguenti qualche schermata del programma finale. Siamo convinti che il fatto di sapere cosa troveremo in cima renderà meno faticosa la salita. Alla peggio possiamo anche decidere di non partire...

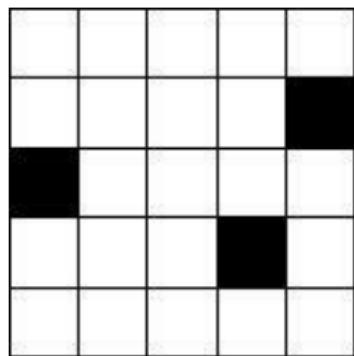


Figura 11.2 Un altro problema Full House.

Nota Il codice sorgente del programma presentato in questo capitolo può essere scaricato all'indirizzo
<http://thinkcode.tv/code/python/fullhouse.zip>

Nella Figura 11.3 possiamo osservare l'interfaccia del gioco, con alcune mosse già effettuate e una parte del menu.

La Figura 11.4 ci mostra un problema difficilissimo da risolvere. Per fortuna, come qualcuno avrà già notato osservando le opzioni del menu visibili nella figura precedente, nel nostro applicativo svilupperemo anche un algoritmo per la “risoluzione” di questi problemi.

Full House



File Edit Problemi

Annulla mossa Ctrl+A

Azzera partita Ctrl+Z

Risolve partita Ctrl+R

Modalità edit Ctrl+E

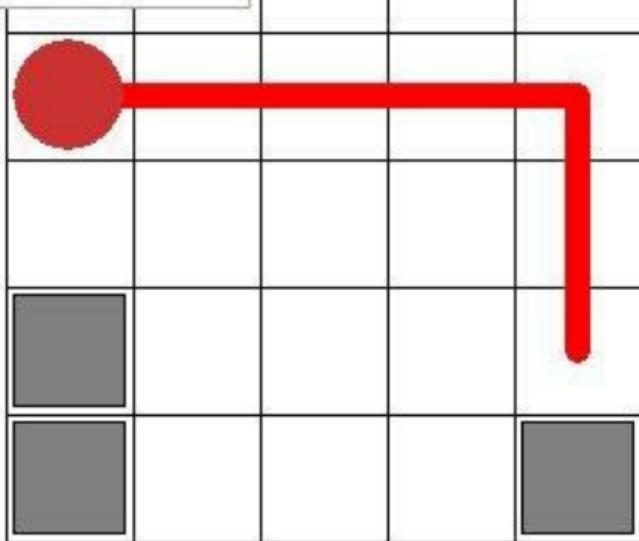


Figura 11.3 L'interfaccia di Full House.

Full House



File Edit Problemi

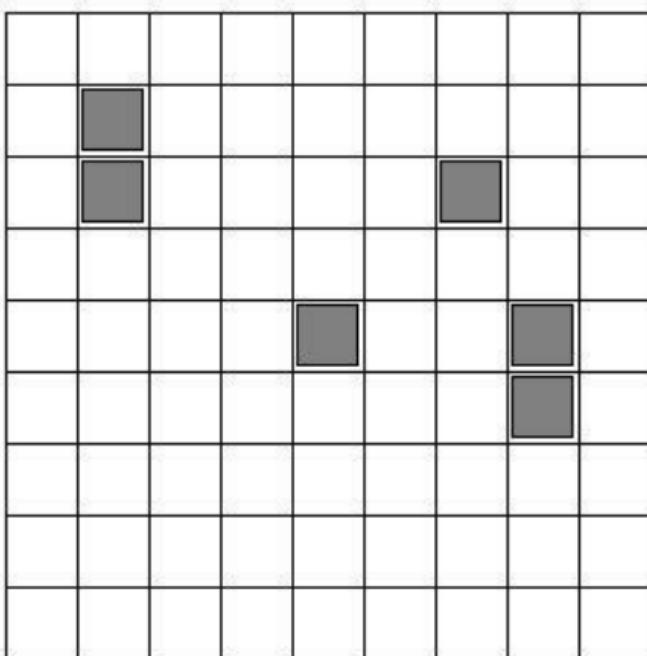


Figura 11.4 Un problema molto difficile.

Nella Figura 11.5 vediamo che il risolutore, dopo aver risolto il problema della Figura 11.2, ci dice che farsi aiutare non vale, è troppo facile!

La creazione di nuovi problemi è l'ultima funzionalità che aggiungeremo al nostro programma. La Figura 11.6 mostra un problema, che abbiamo creato in modalità Edit, per il quale il risolutore ha scoperto che non esiste alcuna soluzione.

Il “motore” del gioco

La prima parte del nostro programma sarà il motore che gestirà il gioco vero e proprio. Definiremo la classe Scacchiera, la classe Posizione e la classe Direzione. Implementeremo anche una funzione Test per verificare il corretto funzionamento del motore.

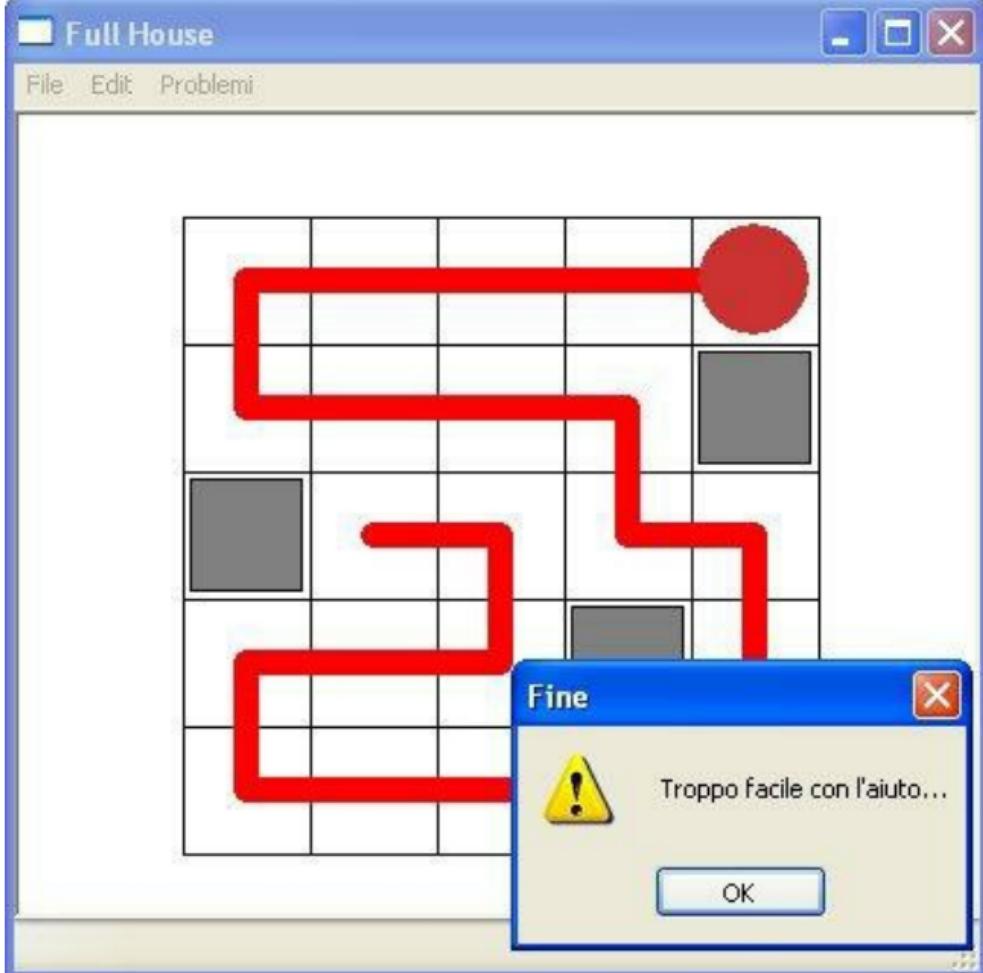


Figura 11.5 Il risolutore in azione.

Il nome dello script è `fullhouse_engine.py`.
Esaminiamolo insieme, riga per riga.

Dapprima dobbiamo definire tre costanti che indicano il possibile contenuto delle varie posizioni della scacchiera:

Caselle

CASELLA_BIANCA = 0

CASELLA_NERA = 1

CASELLA_OCCUPATA = 2

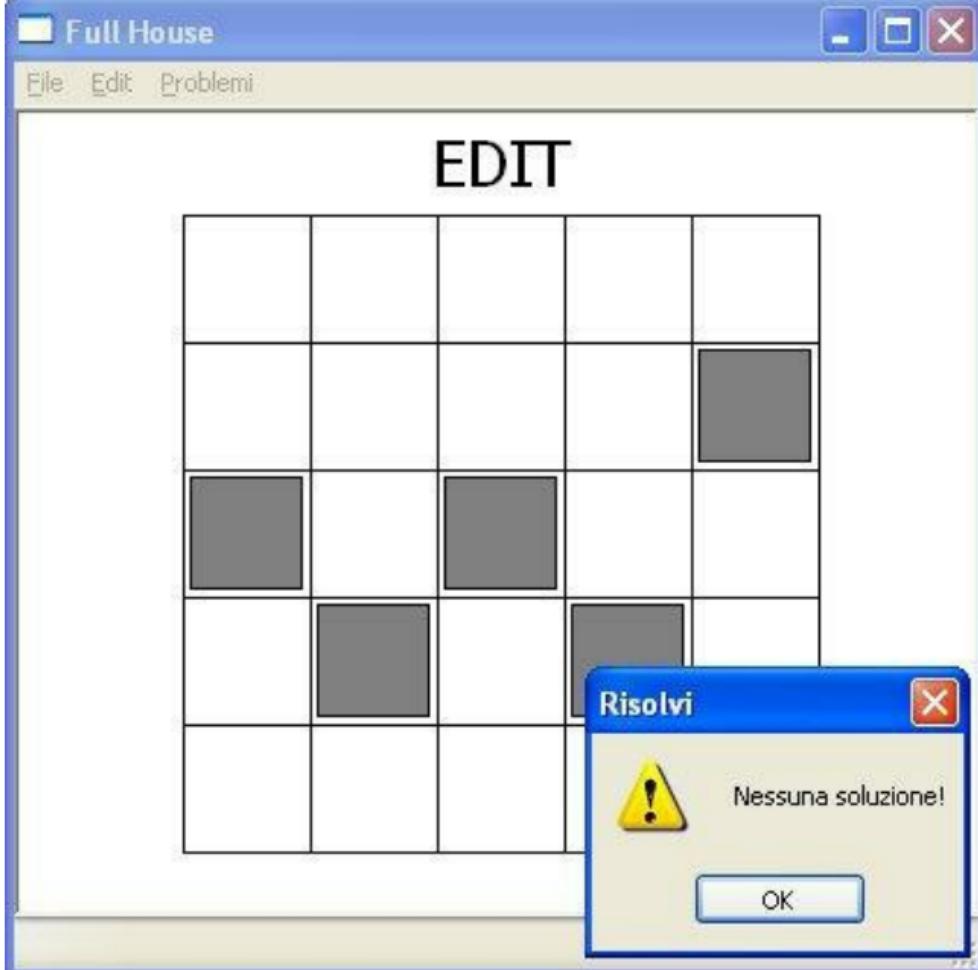


Figura 11.6 La modalità Edit.

L'utilizzo delle costanti è da tenere sempre in considerazione, nel proprio bagaglio di programmazione, qualunque sia il

linguaggio che stiamo utilizzando. Le due righe di codice seguenti svolgono la stessa operazione, ma la loro leggibilità è assai diversa:

matrice[0, 1] = 2

piuttosto che:

matrice[0, 1] = CASELLA_OCCUPATA

Le posizioni della scacchiera possono essere individuate tramite un sistema di coordinate. Nello schema di 5x5 caselle rappresentato nella Figura 11.7, la casella in alto a sinistra è la (0, 0), mentre la casella in basso a destra è la (4, 4). Inoltre la partenza S è nella casella (1, 1) mentre l'arrivo F è nella casella (3, 4).

Durante il gioco, dopo aver stabilito la casella iniziale, le mosse da svolgere vengono individuate dalle direzioni scelte dall'utente. Possiamo per semplicità pensare alle direzioni dei quattro punti cardinali: nord, sud, est e ovest. Per esempio,

sempre sulla base dello schema rappresentato nella Figura 11.7, la casella iniziale è alla coordinata $(1, 1)$, quindi le mosse sono, in sequenza, est, nord, ovest, est, sud, ovest, sud, est.

Possiamo individuare la direzione di spostamento sulla base dei valori che dobbiamo aggiungere alle coordinate di una casella per giungere alla casella di destinazione. Per esempio, se consideriamo la casella di partenza S della Figura 11.8, definita dalle coordinate $(1, 1)$, vediamo che andando verso nord giungiamo nella casella $(1, 0)$, andando verso est giungiamo alla casella $(2, 1)$, andando verso sud giungiamo alla casella $(1, 2)$ e andando verso ovest giungiamo alla casella $(0, 1)$,
Ricapitolando, per andare verso nord dobbiamo sommare alle coordinate di una casella i valori $(0, -1)$, per andare verso est i valori $(1, 0)$, verso sud i valori $(0, 1)$ e verso ovest i valori $(-1, 0)$.

In base a quello che abbiamo appena stabilito, definiamo ora la classe Direzione:

```
# Classe per gestire una direzione di
movimento
class Direzione:
    def __init__(self, d_x, d_y):
        self.d_x = d_x
        self.d_y = d_y
    # Due Direzioni con uguali delta
    # sono uguali
    def __eq__(self, direzione):
        return self.d_x == direzione.d_x
and \
        self.d_y == direzione.d_y
    # Restituisce la direzione opposta
    def Opposta(self):
        if self == NORD:
            return SUD
        elif self == SUD:
            return NORD
        elif self == EST:
            return OVEST
        elif self == OVEST:
            return EST
```

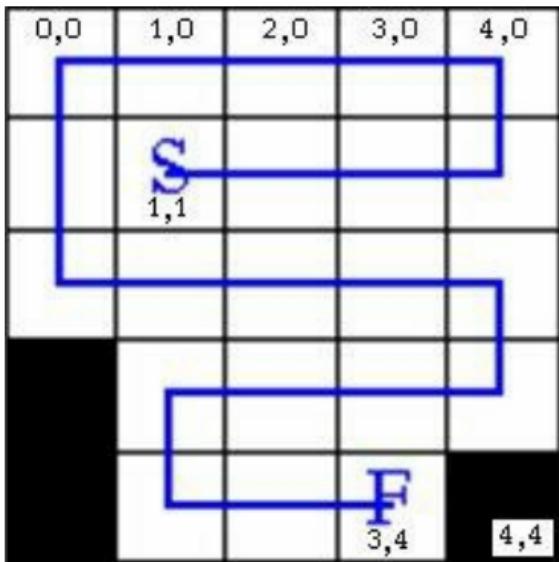


Figura 11.7 La soluzione di un problema e il significato delle coordinate.

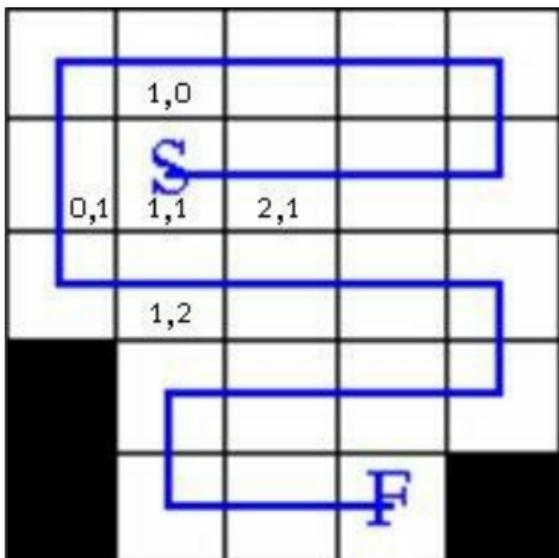


Figura 11.8 Spostamenti da una posizione a un'altra.

Il metodo costruttore `__init__` accetta i valori `d_x` e `d_y`, che individuano una direzione.

Definendo un metodo speciale `__eq__` potremo utilizzare l'operatore `==` tra due istanze diverse. Nel nostro caso `__eq__` ci permette di stabilire quando due direzioni sono uguali. Python non può ovviamente sapere quando due diverse istanze sono per noi identiche dal punto di vista logico. Nello specifico due direzioni coincidono se e solo se hanno gli stessi valori `d_x` e `d_y`.

Il metodo `Opposta`, che utilizzeremo per tornare indietro di una mossa, restituisce la direzione opposta rispetto alla direzione corrente.

Completata la classe `Direzione`, possiamo definire le quattro direzioni cardinali che useremo nel gioco:

```
# Direzioni cardinali
```

```
NORD = Direzione(0, -1)
EST = Direzione(1, 0)
SUD = Direzione(0, 1)
OVEST = Direzione(-1, 0)
DIREZIONI = (NORD, EST, SUD, OVEST)
```

La costante DIREZIONI è solo una variabile di appoggio, che raggruppa tutte le possibili direzioni.

La seconda classe che definiamo è Posizione, che ci permette di gestire la singola posizione sulla scacchiera:

```
# Classe per gestire una posizione sulla
scacchiera
class Posizione:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    # Due posizioni con uguali coordinate
    # sono uguali
    def __eq__(self, posizione):
        return self.x == posizione.x and \
               self.y == posizione.y
    # In base alla direzione restituisce
    # la posizione contigua
```

```
def Contigua(self, direzione):
    return Posizione(self.x + direzione.d_x,
                      self.y + direzione.d_y)

# Controlla se la posizione passata e` 
# in una delle direzioni cardinali
rispetto
# all'istanza self.
# Se lo e`, restituisce la direzione.
# Se non lo e`, restituisce None.
def Direzione(self, posizione):
    # Per essere nella stessa direzione una
    # delle due coordinate deve identica
    if posizione.x == self.x:
        if posizione.y > self.y:
            return SUD
        elif posizione.y < self.y:
            return NORD
        elif posizione.y == self.y:
            if posizione.x > self.x:
                return EST
            elif posizione.x < self.x:
                return OVEST
            return None
```

Il metodo costruttore `__init__` accetta i valori `x` e `y` che individuano la posizione sulla scacchiera.

Come per la classe precedente, il metodo `__eq__` ci permette di stabilire quando due posizioni sono identiche.

Il metodo `Contigua` restituisce la posizione contigua in base alla direzione passata come parametro. Per esempio, data la casella (1, 2), il metodo `Contigua` richiamato con la direzione SUD restituirà la casella (1, 3). Questo metodo ci serve durante il completamento di una mossa per trovare le caselle percorse in una data direzione.

Il metodo `Direzione` verifica se la posizione passata come parametro è in una delle quattro direzioni cardinali rispetto alla posizione corrente. Ci serve per sapere se abbiamo fatto clic su una posizione che individua una mossa accettabile. Per individuare l'eventuale asse dobbiamo verificare quale delle due coordinate è identica (due posizioni con identica coordinata x individuano l'asse NORD / SUD ; due posizioni con identica coordinata y individuano l'asse EST / OVEST). Stabilito l'eventuale asse, controlliamo l'altra

coordinata per capire la direzione precisa.

L'ultima classe che implementiamo nello script principale è la classe Scacchiera:

```
# Classe per gestire la scacchiera di
gioco
class Scacchiera:
def __init__(self, dimensione,
posizioni_nere = ()):
    self.dimensione = dimensione
    self.posizioni_nere =
list(posizioni_nere)
    self.Reset()
# Crea la scacchiera e azzerà le mosse
def Reset(self):
    self.posizioni = []
    self.direzioni = []
    self.matrice = []
    for x in range(self.dimensione):
        self.matrice.append([ CASELLA_BIANCA ] *
self.dimensione)
    for posizione in self.posizioni_nere:
        self.matrice[posizione.x][posizione.y] \
= CASELLA_NERA
    # Controlla se una posizione è
    # percorribile,
    # quindi se è interna alla matrice e se
```

```
non
# e` gia` occupata o nera
def Percorribile(self, posizione):
if posizione.x < 0 or \
posizione.x >= self.dimensione or \
posizione.y < 0 or \
posizione.y >= self.dimensione or not \
self.matrice[posizione.x][posizione.y] \
== CASELLA_BIANCA:
return False
return True
# Occupa la casella della posizione
passata
def Occupa(self, posizione):
self.matrice[posizione.x][posizione.y] =
\
CASELLA_OCCUPATA
# Libera la casella della posizione
passata
def Libera(self, posizione):
self.matrice[posizione.x][posizione.y] =
\
CASELLA_BIANCA
# Controlla se la matrice e` completa e
# quindi risolta
def Risolto(self):
for x in range(self.dimensione):
for y in range(self.dimensione):
if self.Percorribile( \
```

```
Posizione(x, y)):
    return False
    return True
# Controlla se si e` un punto morto
def PuntoMorto(self):
    for direzione in DIREZIONI:
        if self.Percorribile( \
            self.posizioni[-1].Contigua( \
                direzione)):
            return False
    return True
# Muovi dalla posizione corrente nella
# direzione scelta.
# Se percorribile restituisce il numero
# di
# caselle percorse, altrimenti
restituisce 0
def Percorri(self, direzione):
    caselle_percorse = 0
    posizione_corrente = self.posizioni[-1]
    while True:
        posizione_seguente = \
            posizione_corrente.Contigua( \
                direzione)
        if not self.Percorribile( \
            posizione_seguente):
            # Se ho percorso almeno una casella
            # salvo la direzione e la posizione
            # attuale
```

```
if caselle_percorse > 0:  
    self.direzioni.append( \  
        direzione)  
    self.posizioni.append( \  
        posizione_corrente)  
    return caselle_percorse  
caselle_percorse += 1  
posizione_corrente = posizione_seguente  
self.Occupa(posizione_corrente)  
# "Clicca" una casella. Se possibile,  
# questo causa il movimento nella  
direzione  
# prescelta dalla casella.  
# Se e` la prima mossa, setta solo la  
# posizione iniziale.  
# Se e` successiva setta anche la  
# direzione scelta.  
# Se la mossa e` accettabile restituisce  
True,  
# altrimenti False  
def Click(self, posizione):  
    # Innanzitutto la casella deve  
    # essere libera  
    if self.Percorribile(posizione):  
        # Se e` la prima mossa, la salva  
        if len(self.posizioni) == 0:  
            self.Occupa(posizione)  
            self.posizioni.append(posizione)  
        return True
```

```
# Controlla che la casella
# cliccata indichi una direzione
# accettabile
direzione = self.posizioni[ \
-1].Direzione(posizione)
if direzione and \
self.Percorri(direzione) > 0:
return True
return False
# Annulla l'ultima mossa fatta
def Annulla(self):
if len(self.direzioni) > 0:
direzione = \
self.direzioni.pop().Opposta()
posizione_corrente = \
self.posizioni.pop()
while True:
self.Libera(posizione_corrente)
posizione_seguente = \
posizione_corrente.Contigua( \
direzione)
if posizione_seguente == \
self.posizioni[-1]:
break
posizione_corrente = \
posizione_seguente
return True
elif len(self.posizioni) > 0:
self.Libera(self.posizioni.pop())
```

```
return True
return False
# Esplora ricorsivamente la matrice a
# partire dalla posizione corrente
def Esplora(self):
if self.Risolta():
    self.soluzioni.append( \
        (self.posizioni[0],) + \
        tuple(self.direzioni[:]))
return
for direzione in DIREZIONI:
    if self.Click(self.posizioni[ \
        -1].Contigua(direzione)):
        self.Esplora()
        self.Annulla()
# Risolve il problema
def Risolvi(self):
    self.Reset()
    self.soluzioni = []
    for x in range(self.dimensione):
        for y in range(self.dimensione):
            if self.Click(Posizione(x, y)):
                self.Esplora()
                self.Annulla()
    return self.soluzioni
```

Esaminiamo in dettaglio i singoli metodi, il loro funzionamento e il loro utilizzo.

Il costruttore `__init__` accetta le dimensioni della scacchiera e la lista delle coordinate corrispondenti alle caselle nere.

Il metodo `Reset`, richiamato anche dal costruttore, crea le strutture interne, tra cui la matrice quadrata con tutte le posizioni libere, e assegna le caselle nere.

Il metodo `Percorribile` verifica se una coordinata è libera o meno e quindi se è “percorribile”. Per esserlo, la casella corrispondente non deve essere nera e non deve essere già stata occupata durante la soluzione del gioco.

Il metodo `Occupava` rende non percorribile la coordinata corrispondente, mentre il metodo `Libera` la rende nuovamente libera.

Il metodo `Risolta` verifica che tutte le caselle siano nere o già occupate e in tal caso considera risolto il problema corrente.

Il metodo PuntoMorto verifica se la situazione attuale non porta a una soluzione. Nella Figura 11.9 possiamo vedere una situazione di “punto morto”, in cui nessuna mossa è più possibile. Per farlo, il programma verifica semplicemente che le quattro caselle contigue alla casella corrente non siano più percorribili.

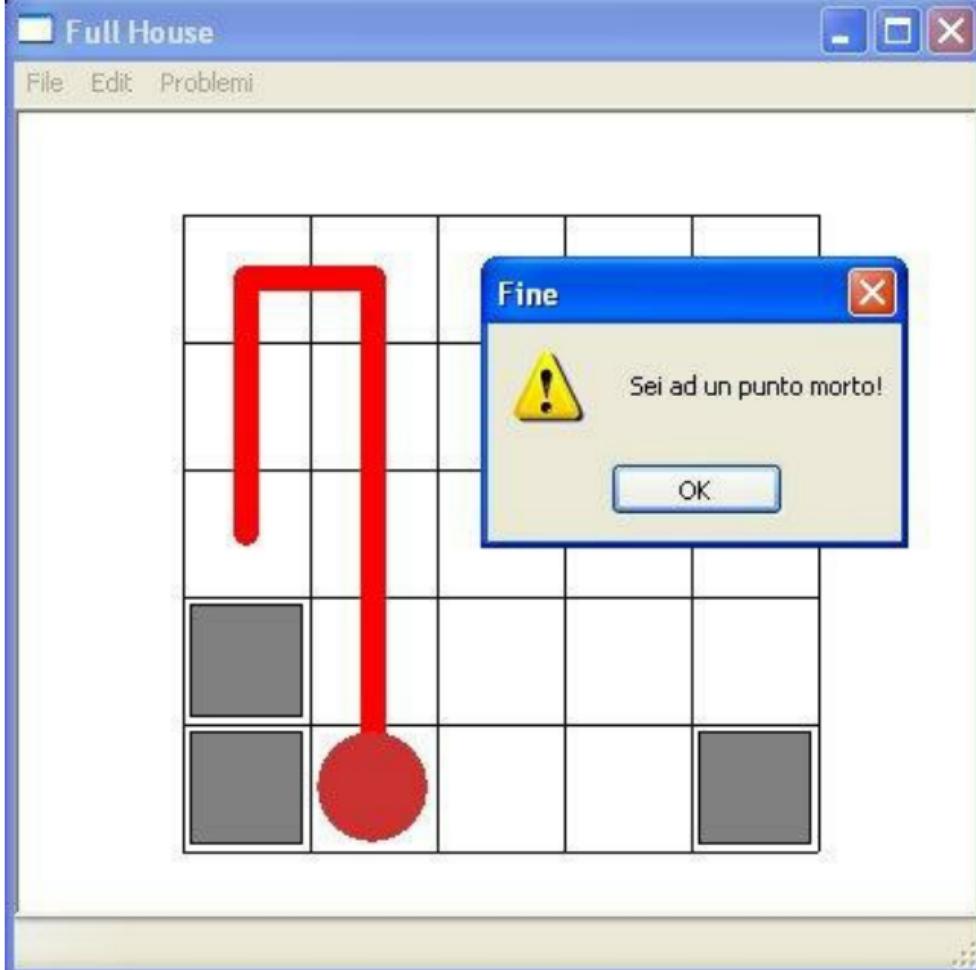


Figura 11.9 Una posizione senza via d'uscita.

Il metodo Percorri, data una direzione, effettua la mossa nella direzione specificata e restituisce il numero di caselle percorse; in caso contrario

restituisce semplicemente 0. Inoltre, solo se la mossa è possibile, memorizza la nuova posizione corrente.

Il metodo Click controlla se la cella è libera e se individua una direzione corretta rispetto alla posizione corrente. In tal caso effettua la mossa.

Il metodo Annulla annulla l'ultima mossa effettuata. Lo utilizzeremo quando il giocatore deciderà di tornare indietro di una mossa perché ha cambiato idea o perché è giunto a un punto morto.

Il metodo Esplora viene inizialmente richiamato dal metodo risolutore Risolvi e poi, ricorsivamente, da se stesso. Tramite un algoritmo di “forza bruta” prova a esplorare la scacchiera in ogni direzione, fino a giungere a un punto morto, in tal caso torna indietro, fino a trovare una soluzione. Ogni soluzione trovata viene memorizzata.

Il metodo Risovi richiama il metodo Esplora per tutte le caselle libere della scacchiera.

Come ultima componente del nostro script realizziamo una funzione Test che verifichi il corretto funzionamento delle classi. Come possiamo verificarlo? Semplice: prendiamo come esempio il problema rappresentato nella Figura 11.10.

La scacchiera ha dimensioni 5x5, le caselle nere si trovano alle coordinate (0, 2), (3, 3) e (1, 4). La soluzione inizia dalla casella (0, 4) e prosegue nelle direzioni ovest, sud, est, sud, est, sud, ovest, nord, est, nord e ovest.

Ora che abbiamo a disposizione un esempio di soluzione possiamo scrivere la funzione Test:

```
# Funzione di test
def Test():
    # Gioco di esempio con relativa
    # soluzione
    dimensione = 5
    posizioni_nere = (Posizione(0, 2),
```

```
Posizione(3, 3),  
Posizione(4, 1))  
soluzione = (Posizione(4, 0),  
OVEST, SUD, EST, SUD, EST, SUD,  
OVEST, NORD, EST, NORD, OVEST)  
# Crea la scacchiera del problema di  
test  
s = Scacchiera(dimensione = dimensione,  
posizioni_nere = posizioni_nere)  
# Verifica che la risoluzione contempi  
una  
# sola soluzione identica a quella nota  
soluzioni = s.Risolvi()  
print "Soluzioni trovate: %d" %  
len(soluzioni)  
if soluzioni[0] == soluzione:  
print "Soluzione corretta trovata"
```

Full House



File Edit Problemi

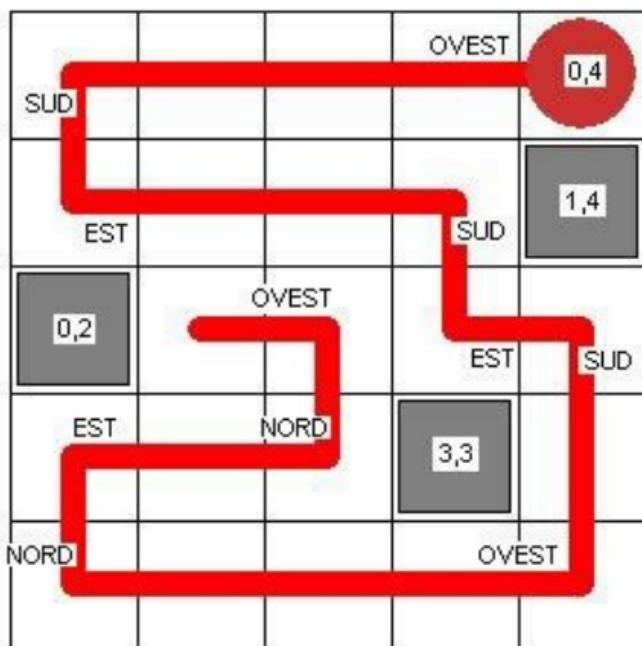


Figura 11.10 Il problema usato come test del nostro script.

La funzione Test crea l'istanza s di tipo Scacchiera contenente le specifiche del problema

rappresentato nella Figura 11.10, definisce la variabile soluzione contenente la casella iniziale e le direzioni per risolvere lo schema, infine prova a richiamare il metodo Risovi verificando che ci sia una sola soluzione e che sia identica a quella nota.

Da ultimo, nel caso in cui il nostro script sia eseguito dalla riga di comando, richiamiamo la funzione Test:

```
# Se eseguito come script, effettua solo il test
if __name__ == "__main__":
    Test()
```

Attenzione La variabile speciale `__name__` viene inizializzata con il nome del modulo, tranne quando questo viene eseguito direttamente dalla riga di comando: in tal caso assume il valore “`__main__`”.

Proviamo ora a eseguire il nostro script dalla riga di comando. Se non abbiamo commesso errori, lo

script dovrà presentare l'output visualizzato nella Figura 11.11.

Senza considerare i commenti, lo script è composto da meno di 150 righe di codice. Siamo assolutamente certi che il gioco Full House, compreso di risolutore, non può essere considerato un'applicazione banale.

The screenshot shows a Windows command-line interface (cmd.exe) window. The title bar reads "C:\WINDOWS\system32\cmd.exe". The command entered is "C:\Work\Python\FullHouse\codice>python fullhouse_engine.py". The output displayed is:

```
C:\Work\Python\FullHouse\codice>python fullhouse_engine.py
Soluzioni trovate: 1
Soluzione corretta trovata
```

The window has standard Windows controls at the top right (minimize, maximize, close) and scroll bars on the right and bottom.

Figura 11.11 Lo script trova correttamente la soluzione.

I problemi

Il secondo script del nostro applicativo definisce vari problemi che il giocatore deve cercare di risolvere.

Lo script si chiama fullhouse_problemi.py e contiene la lista dei problemi disponibili sul sito del professor Friedman:

```
from fullhouse_engine import Scacchiera,  
Posizione  
Problemi = (  
Scacchiera(5, (Posizione(0, 3),  
Posizione(0, 4),  
Posizione(4, 4))),  
Scacchiera(5, (Posizione(0, 2),  
Posizione(3, 3),  
Posizione(4, 1))),  
Scacchiera(5, (Posizione(0, 4),  
Posizione(3, 2),  
Posizione(4, 0))),  
Scacchiera(5, (Posizione(0, 0),
```

```
Posizione(1, 0),  
Posizione(2, 3),  
Posizione(4, 2))),  
Scacchiera(5, (Posizione(0, 0),  
Posizione(2, 1),  
Posizione(3, 4),  
Posizione(4, 2))),  
Scacchiera(5, (Posizione(0, 0),  
Posizione(0, 1),  
Posizione(0, 2),  
Posizione(4, 3),  
Posizione(4, 4))),  
Scacchiera(6, (Posizione(0, 3),  
Posizione(1, 3),  
Posizione(4, 1))),  
Scacchiera(6, (Posizione(0, 0),  
Posizione(1, 2),  
Posizione(0, 4),  
Posizione(0, 5),  
Posizione(3, 5))),  
Scacchiera(6, (Posizione(0, 0),  
Posizione(2, 1),  
Posizione(1, 4),  
Posizione(5, 4),  
Posizione(5, 5))),  
Scacchiera(6, (Posizione(0, 2),  
Posizione(1, 4),  
Posizione(3, 0),  
Posizione(3, 1),
```

```
Posizione(5, 2)),  
Scacchiera(6, (Posizione(0, 2),  
Posizione(1, 4),  
Posizione(3, 1),  
Posizione(4, 1))),  
Scacchiera(6, (Posizione(1, 1),  
Posizione(4, 3),  
Posizione(5, 0),  
Posizione(5, 3))),  
Scacchiera(6, (Posizione(0, 0),  
Posizione(0, 5),  
Posizione(2, 1),  
Posizione(3, 3))),  
Scacchiera(6, (Posizione(0, 0),  
Posizione(0, 5),  
Posizione(1, 2),  
Posizione(4, 4),  
Posizione(5, 2))),  
Scacchiera(6, (Posizione(1, 1),  
Posizione(2, 1),  
Posizione(2, 5),  
Posizione(4, 2),  
Posizione(4, 3),  
Posizione(5, 5))),  
Scacchiera(7, (Posizione(0, 0),  
Posizione(1, 2),  
Posizione(2, 4),  
Posizione(3, 2),  
Posizione(4, 2),
```



```
Posizione(4, 5),  
Posizione(5, 1))),  
Scacchiera(7, (Posizione(0, 0),  
Posizione(0, 1),  
Posizione(1, 3),  
Posizione(2, 5),  
Posizione(3, 3))),  
Scacchiera(7, (Posizione(3, 1),  
Posizione(4, 4),  
Posizione(4, 5),  
Posizione(5, 4),  
Posizione(6, 6))),  
Scacchiera(8, (Posizione(0, 7),  
Posizione(2, 6),  
Posizione(3, 1),  
Posizione(5, 2),  
Posizione(5, 3))),  
Scacchiera(8, (Posizione(0, 0),  
Posizione(1, 6),  
Posizione(2, 1),  
Posizione(3, 7),  
Posizione(4, 5))),  
Scacchiera(8, (Posizione(0, 2),  
Posizione(1, 4),  
Posizione(3, 0),  
Posizione(4, 0),  
Posizione(5, 2))),  
Scacchiera(9, (Posizione(1, 3),  
Posizione(2, 5),
```

```
Posizione(5, 0),  
Posizione(6, 6),  
Posizione(7, 1))),  
Scacchiera(9, (Posizione(4, 1),  
Posizione(4, 2),  
Posizione(4, 3),  
Posizione(6, 2),  
Posizione(7, 4),  
Posizione(8, 6))),  
Scacchiera(9, (Posizione(1, 1),  
Posizione(1, 2),  
Posizione(4, 4),  
Posizione(6, 2),  
Posizione(7, 4),  
Posizione(7, 5))),  
)
```

Abbiamo utilizzato la classe Scacchiera per descrivere i problemi secondo il linguaggio che abbiamo definito nel nostro motore. Ricordiamo che il metodo `__init__` della classe Scacchiera richiede come parametri solo le dimensioni della scacchiera e le coordinate delle caselle nere.

Nella Figura 11.12 vediamo un'anticipazione: la voce che permette all'utente di scegliere uno qualsiasi di questi problemi.

L'interfaccia grafica

Adesso che il motore del gioco è pronto e la lista di problemi da risolvere è compilata, possiamo rivestire il tutto con un'interfaccia grafica accattivante. Nel Capitolo 9 abbiamo visto all'opera il toolkit grafico wxPython: ora proveremo a impiegarlo per il nostro gioco.

Il terzo e ultimo script del programma si chiama fullhouse.py. È il più lungo dei tre e sicuramente il più complesso, ma il risultato è quello che abbiamo visto nelle prime immagini di questo capitolo.

Nella prima riga definiamo l'encoding (la codifica) dei caratteri che vogliamo utilizzare. Questo passo è necessario per via dell'uso delle lettere accentate nei messaggi e nelle voci dei menu:

```
# -*- coding: latin-1 -*-
```

Questa riga non è necessaria se nel programma utilizziamo solo caratteri ASCII non speciali; ma nei messaggi e nel menu del programma ci sono diverse lettere accentate, motivo per cui dobbiamo segnalare all'interprete Python quale encoding deve utilizzare.

Nota Una trattazione estesa dell'encoding esula dallo scopo di questo libro. Occorre però dire che il modello di encoding attuale è stato introdotto in ambito informatico per ovviare alla limitazione che qualunque tipo di codifica con un numero di caratteri limitato avrebbe imposto. Pensiamo solo alla necessità di visualizzare l'alfabeto cirillico o gli ideogrammi cinesi o giapponesi. L'uso del corretto encoding ci permette di farlo.

Quindi importiamo la libreria wx e gli script che abbiamo creato in precedenza:

```
import wx
from fullhouse_engine import Scacchiera,
```

```
\  
Posizione, Direzione  
from fullhouse_problemi import Problemi
```

La prima classe che definiamo è FullHouseWindow, che si occupa di disegnare la finestra principale della scacchiera. Inseriremo poi questa finestra all'interno di un frame con il menu e la barra di stato. Il codice di questa classe è molto lungo, per cui commenteremo i metodi a mano a mano che li incontreremo.

Il costruttore `__init__`, oltre a creare la finestra vera e propria, stabilisce un Bind (un collegamento) per gli eventi `EVT_PAINT` (scatenato dal tracciamento della finestra sullo schermo) e `EVT_LEFT_UP` e `EVT_RIGHT_UP` (scatenati rispettivamente dalla pressione dei tasti sinistro e destro del mouse). In questo modo il verificarsi di questi eventi fa in modo che vengano richiamati di metodi specifici, in particolare i metodi `OnPaint`, `OnLeftClick` e `OnRightClick` descritti più avanti nel codice.

```
# Classe che si occupa di disegnare la
# finestra della schacchiera
class FullHouseWindow(wx.Window):
def __init__(self, parent):
wx.Window.__init__(self, parent,
id = -1,
pos = wx.Point(0, 0),
size = wx.DefaultSize,
style = wx.SUNKEN_BORDER
| wx.WANTS_CHARS
| wx.FULL_REPAINT_ON_RESIZE)
self.SetBackgroundColour( \
wx.NamedColour("white"))
self.Bind(wx.EVT_PAINT, self.OnPaint)
self.Bind(wx.EVT_LEFT_UP,
self.OnLeftClick)
self.Bind(wx.EVT_RIGHT_UP,
self.OnRightClick)
self.scacchiera = Problemi[0]
self.edit_mode = False
```

Il metodo `OnPaint` è il più complesso di tutta l'applicazione. In base alle dimensioni della finestra deve infatti disegnare la scacchiera, le caselle nere e le mosse effettuate.

```
# Disegna il contenuto della finestra
def OnPaint(self, event):
```

```
# Calcola la dimensione della finestra
x, y = self.GetSize()
# Calcola la posizione e la
# dimensione della scacchiera quadrata
# e la dimensione dei riquadri
self.lato_casella = int(min(x, y) * 0.8
/
self.scacchiera.dimensione)
self.lato_scacchiera = self.lato_casella
\
* self.scacchiera.dimensione
self.alto_sinistra = wx.Point((x -
self.lato_scacchiera)/2, (y -
self.lato_scacchiera)/2 * 1.2)
self.bordo = self.lato_casella / 15
self.mezzo_riquadro = wx.Point(
self.lato_casella / 2,
self.lato_casella / 2)
self.raggio = self.lato_casella / 2 - \
self.bordo
# Istanzia l'oggetto per il disegno
w, h = self.GetClientSizeTuple()
buffer = wx.EmptyBitmap(w, h)
dc = wx.PaintDC(self)
# Scrive la modalita` corrente
if self.edit_mode:
font = dc.GetFont()
font.SetPointSize(20)
dc.SetFont(font)
```

```
msg_edit = "EDIT"
w, h = dc.GetTextExtent(msg_edit)
dc.DrawText(msg_edit, (x - w)/2,
(self.alto_sinistra.y - h)/2)
# Disegna la scacchiera
dc.SetPen(wx.Pen(wx.NamedColour("black"),
1, wx.SOLID))
dc.SetBrush(wx.Brush(
wx.NamedColour("grey")))
# Disegna le righe della scacchiera
punto = wx.Point(self.alto_sinistra.x,
self.alto_sinistra.y)
for colonne in range(
self.scacchiera.dimensione + 1):
dc.DrawLine([punto, punto +
wx.Point(0, self.lato_scacchiera)])
punto += wx.Point(self.lato_casella, 0)
# Disegna le colonne della scacchiera
punto = wx.Point(self.alto_sinistra.x,
self.alto_sinistra.y)
for righe in range(
self.scacchiera.dimensione + 1):
dc.DrawLine([punto, punto +
wx.Point(self.lato_scacchiera, 0)])
punto += wx.Point(0, self.lato_casella)
# Disegna le caselle nere della
scacchiera
for posizione_nera in \
self.scacchiera.posizioni_nere:
```

```
riquadro_nero = \
self.CalcolaRiquadro(posizione_nera)
riquadro_nero.Deflate(self.bordo,
self.bordo)
dc.DrawRectangle(riquadro_nero.x,
riquadro_nero.y,
riquadro_nero.width,
riquadro_nero.height)
# Disegna le mosse
if len(self.scacchiera.posizioni) > 0:
dc.SetPen(wx.Pen(wx.NamedColour("red"),
self.lato_casella / 5, wx.SOLID))
for cont, direzione in enumerate(
self.scacchiera.direzioni):
posizione_da = \
self.scacchiera.posizioni[cont]
posizione_finale = \
self.scacchiera.posizioni[
cont + 1]
while True:
posizione_a = \
posizione_da.Contigua(
direzione)
punto_da = \
self.CalcolaRiquadro(
posizione_da).GetTopLeft() \
+ self.mezzo_riquadro
punto_a = \
self.CalcolaRiquadro(
```

```
posizione_a).GetTopLeft() \
+ self.mezzo_riquadro
dc.DrawLine([punto_da,
punto_a])
posizione_da = posizione_a
if posizione_a == \
posizione_finale:
break
# Disegna la mossa iniziale
dc.SetPen(wx.Pen(
wx.NamedColour("orange"),
1, wx.SOLID))
dc.SetBrush(wx.BBrush(
wx.NamedColour("orange")))
centro = self.CalcolaRiquadro(
self.scacchiera.posizioni[0]
).GetTopLeft() + \
self.mezzo_riquadro
dc.DrawCircle(centro.x, centro.y,
self.raggio)
```

Il metodo `OnRightClick` abbina semplicemente la pressione del tasto destro del mouse all'annullamento dell'ultima mossa:

```
# Gestisce il click destro del mouse
# che annulla l'ultima mossa
def OnRightClick(self, event):
```

```
self.Annulla()
```

Il metodo OnLeftClick abbina invece la pressione del tasto sinistro del mouse al tentativo di effettuare una nuova mossa. Per fare questo utilizza il metodo ControllaPunto e Click, descritti più avanti:

```
# Gestisce il click destro del mouse
# che effettua una nuova mossa
def OnLeftClick(self, event):
    point_click = wx.Point(event.m_x,
                           event.m_y)
    x, y = self.ControllaPunto(point_click)
    if x != None:
        self.Click(Posizione(x, y))
```

Il metodo CalcolaRiquadro restituisce il rettangolo del riquadro della scacchiera corrispondente alla posizione passata come parametro. Poiché la finestra è ridimensionabile, dobbiamo poter calcolare facilmente le coordinate di ogni riquadro:

```
# Calcola l'area di un riquadro di una
# posizione tornando il corrispondente
```

```
wx.Rect  
def CalcolaRiquadro(self, posizione):  
    return wx.Rect(self.alto_sinistra.x +  
    self.lato_casella * posizione.x,  
    self.alto_sinistra.y +  
    self.lato_casella * posizione.y,  
    self.lato_casella, self.lato_casella)
```

Il metodo **ControllaPunto** determina se un dato punto si trova all'interno di una posizione; in caso affermativo restituisce la coordinata all'interno della matrice (ricordiamo che le caselle della scacchiera sono individuate da due coordinate x, y):

```
# Controlla se un punto e` interno a un  
# riquadro e ne restituisce le  
coordinate  
# altrimenti restituisce (None, None)  
def ControllaPunto(self, punto):  
    for x in  
        range(self.scacchiera.dimensione):  
            for y in range(  
                self.scacchiera.dimensione):  
                riquadro = \  
                self.CalcolaRiquadro(  
                    Posizione(x, y))
```

```
riquadro.Deflate(self.bordo,
self.bordo)
if riquadro.Contains(punto):
return (x, y)
return (None, None)
```

Il metodo **Annulla** annulla l'ultima mossa e ridisegna tutta la finestra:

```
# Annulla l'ultima mossa
def Annulla(self):
if self.scacchiera.Annulla():
self.Refresh()
```

Il metodo **Azzera** annulla tutte le eventuali mosse effettuate:

```
# Azzera il problema
def Azzera(self):
# Flag vari
self.aiuto = False
self.scacchiera.Reset()
self.Refresh()
```

Il metodo **EditOnOff** seleziona la modalità “edit”, che permette all’utente di creare nuovi problemi cambiando il colore delle caselle:

```
# Modalita` di creazione di un nuovo problema
def EditOnOff(self, set_edit = None):
if set_edit == None:
self.edit_mode = not self.edit_mode
else:
self.edit_mode = set_edit
self.menu_edit_item.Check(self.edit_mode)
self.scacchiera.Reset()
self.Refresh()
```

Il metodo Problema attiva il nuovo problema scelto dall'utente:

```
# Sceglie il problema corrente
def Problema(self, scacchiera):
self.scacchiera = scacchiera
self.Azzera()
```

Il metodo Risolve tenta di risolvere il problema corrente. Può però accadere che il problema corrente, modificato in modalità “edit”, non sia più risolvibile:

```
# Risolve da solo il problema
def Risolve(self):
self.Azzera()
```

```
self.aiuto = True
soluzioni = self.scacchiera.Risolvi()
if len(soluzioni) > 0:
if len(soluzioni) > 1:
self.Messaggio(
"Ci sono %d soluzioni!" %
len(soluzioni), "Risolvi",
wx.OK | wx.ICON_EXCLAMATION)
self.edit_mode = False
self.Click(soluzioni[0][0])
self.Update()
wx.MilliSleep(250)
for direzione in soluzioni[0][1:]:
self.Click(
self.scacchiera.posizioni[
-1].Contigua(direzione))
self.Update()
wx.MilliSleep(250)
self.edit_mode = \
self.menu_edit_item.IsChecked()
self.Refresh()
else:
self.Messaggio("Nessuna soluzione!",
"Risolvi", wx.OK |
wx.ICON_EXCLAMATION)
```

Il metodo `Messaggio` visualizza il messaggio passato col parametro testo:

```
# Visualizza un messaggio
def Messaggio(self, testo, titolo,
stile):
dlg = wx.MessageDialog(self, testo,
titolo, stile)
dlg.ShowModal()
dlg.Destroy()
```

In modalità normale, il metodo Click effettua una nuova mossa. In modalità “edit” inverte, da bianco a nero, e viceversa, il colore della casella selezionata con un clic. In modalità normale, al termine dell’eventuale mossa, verifica che il problema sia stato risolto o che si trovi in un punto morto:

```
# Effettua una nuova mossa oppure, se
# in edit_mode, cambia lo stato di una
casella
def Click(self, posizione):
if self.edit_mode:
if posizione in \
self.scacchiera.posizioni_nere:
self.scacchiera.posizioni_nere.remove(
posizione)
else:
self.scacchiera.posizioni_nere.append(
```

```
posizione)
self.scacchiera.Reset()
self.Refresh()
elif self.scacchiera.Click(posizione):
    self.Refresh()
# Verifica la risoluzione del problema
if self.scacchiera.Risolta():
# Se e` stato scelto l'aiuto
if self.aiuto:
msg = self.Messaggio(
    "Troppo facile con l'aiuto...", 
    "Fine", wx.OK | 
wx.ICON_EXCLAMATION)
else:
msg = self.Messaggio(
    "Hai risolto questo problema!", 
    "Fine", wx.OK | 
wx.ICON_EXCLAMATION)
# Verifica l'arrivo a un punto morto
elif self.scacchiera.PuntoMorto():
msg = self.Messaggio(
    "Sei ad un punto morto!", 
    "Fine", wx.OK | 
wx.ICON_EXCLAMATION)
```

La seconda classe dello script è FullHouseFrame che racchiude una finestra di tipo FullHouseWindow e definisce il menu

dell'applicazione:

```

# Classe frame che gestisce il menu` e
# contiene la finestra della scacchiera
class FullHouseFrame(wx.Frame):
def __init__(self, title):
x, y = wx.ScreenDC().GetSize()
size = (x/3, x/3)
pos = (x/3, y/3)
wx.Frame.__init__(self, parent=None,
id=-1,
title=title, size=size, pos=pos)
self.fullHouseWindow = \
FullHouseWindow(self)
self.CenterOnScreen()
self.CreateStatusBar()
# Menu` principale
menuBar = wx.MenuBar()
menu_principale = wx.Menu()
menu_principale.Append(101, "&Esci",
"Chiude il programma")
menuBar.Append(menu_principale, "&File")
# Menu` mosse
menu_mosse = wx.Menu()
menu_mosse.Append(201,
"&Annulla mossa\tCtrl+A",
"Annulla l'ultima mossa")
menu_mosse.Append(202,
"A&zzera partita\tCtrl+Z",
"
```

```
"Riparte dall'inizio del problema")
menu_mosse.AppendSeparator()
menu_mosse.Append(203,
"&Risolve partita\tCtrl+R",
"Risolve il problema")
menu_mosse.AppendSeparator()
self.fullHouseWindow.menu_edit_item = \
menu_mosse.Append(204,
"Modalità &edit\tCtrl+E",
"Inserisci un nuovo problema",
wx.ITEM_CHECK)
menuBar.Append(menu_mosse, "&Edit")
# Menu` problemi
self.menu_problemi = wx.Menu()
self.menu_problemi.Append(301,
"&Seguente\tCtrl+S",
"Passa al problema seguente")
self.menu_problemi.Append(302,
"&Precedente\tCtrl+P",
"Torna al problema precedente")
self.menu_problemi.AppendSeparator()
self.problema_corrente = 0
for cont, problema in
enumerate(Problemi):
self.menu_problemi.Append(303 + cont,
"Problema %d (%d x %d)" %
(cont + 1, problema.dimensione,
problema.dimensione),
"", wx.ITEM_RADIO )
```

```
menuBar.Append(self.menu_problemi,
"Problemi")
self.SetMenuBar(menuBar)
# Eventi Menu
self.Bind(wx.EVT_MENU, self.Esci,
id=101)
self.Bind(wx.EVT_MENU, self.Annulla,
id=201)
self.Bind(wx.EVT_MENU, self.Azzera,
id=202)
self.Bind(wx.EVT_MENU, self.Risolve,
id=203)
self.Bind(wx.EVT_MENU, self.Edit,
id=204)
self.Bind(wx.EVT_MENU,
self.ProblemaSeguente, id=301)
self.Bind(wx.EVT_MENU,
self.ProblemaPrecedente, id=302)
self.Bind(wx.EVT_MENU, self.Problema,
id=303, id2=302 + len(Problemi))
def Esci(self, event):
    self.Close()
def Azzera(self, event):
    self.fullHouseWindow.Azzera()
def Edit(self, event):
    self.fullHouseWindow.EditOnOff()
def Annulla(self, event):
    self.fullHouseWindow.Annulla()
def Risolve(self, event):
```

```
if self.fullHouseWindow.edit_mode:
    result = wx.ID_OK
else:
    dialog = wx.MessageDialog(self,
    "Sei sicuro di rinunciare a farcela" +
    " da solo?", "Risolve", wx.OK | 
wx.CANCEL | wx.ICON_QUESTION)
    result = dialog.ShowModal()
    dialog.Destroy()
if result == wx.ID_OK:
    self.fullHouseWindow.Risolve()
def Problema(self, event):
    self.problema_corrente = \
event.GetId() - 303
    self.fullHouseWindow.Problema(
    Problemi[self.problema_corrente])
def ProblemaPrecedente(self, event):
    if self.problema_corrente > 0:
        self.problema_corrente -= 1
        self.menu_problemi.GetMenuItems() [
        self.problema_corrente +
        3].Check(True)
    self.fullHouseWindow.Problema(
    Problemi[self.problema_corrente])
def ProblemaSeguente(self, event):
    if self.problema_corrente + 1 < \
len(Problemi):
        self.problema_corrente += 1
        self.menu_problemi.GetMenuItems() [
```

```
self.problema_corrente +
3].Check(True)
self.fullHouseWindow.Problema(
Problemi[self.problema_corrente])
```

Possiamo notare come il contenuto del menu Problemi, che abbiamo visto nella Figura 11.12, non sia prefissato, ma venga definito in base ai problemi che abbiamo inserito nello script fullhouse_problemi. Se dovessimo aggiungere altri problemi, il menu si adatterebbe visualizzandoli tutti automaticamente.

La terza e ultima classe dello script è l'applicazione FullHouse che non fa altro che creare una frame FullHouseFrame e visualizzarla:

```
class FullHouse(wx.App):
def OnInit(self):
fullhouse = FullHouseFrame("Full House")
fullhouse.Show(True)
self.SetTopWindow(fullhouse)
return True
```

La funzione main istanzia un oggetto FullHouse e lo esegue:

```
def main():
    fh = FullHouse(0)
    fh.MainLoop()
if __name__ == "__main__":
    main()
```

Anche in questo caso, come in fullhouse_engine, se il nostro script viene richiamato dalla riga di comando, richiameremo la funzione main.

Una piattaforma diversa

Le immagini di questo capitolo sono state “catturate” su un sistema operativo Windows. Ma cosa succede se prendiamo i nostri tre script e proviamo a eseguirli su una piattaforma diversa? Vediamo un po’ che cosa accade con Linux, in particolare sulla distribuzione Ubuntu.

Senza modificare alcuna riga di codice proviamo a eseguire fullhouse.py: nelle Figure 11.13 e 11.14 vediamo FullHouse in azione. Il Look and Feel è quello del bellissimo desktop di Ubuntu, ma la

funzionalità è immutata. Nella Figura 11.15 abbiamo ingrandito la finestra e FullHouse continua a funzionare egregiamente.

La promessa di sviluppare un applicativo su una piattaforma per poi usarlo su un'altra è stata mantenuta in pieno!

Seguiente

Ctrl+S

Precedente

Ctrl+P

Problema 1 (5 x 5)

Problema 2 (5 x 5)

Problema 3 (5 x 5)

Problema 4 (5 x 5)

Problema 5 (5 x 5)

Problema 6 (5 x 5)

Problema 7 (6 x 6)

● Problema 8 (6 x 6)

Problema 9 (6 x 6)

Problema 10 (6 x 6)

Problema 11 (6 x 6)

Problema 12 (6 x 6)

Problema 13 (6 x 6)

Problema 14 (6 x 6)

Problema 15 (6 x 6)

Problema 16 (7 x 7)

Problema 17 (7 x 7)

Problema 18 (7 x 7)

Problema 19 (7 x 7)

Problema 20 (7 x 7)

Problema 21 (7 x 7)

Problema 22 (7 x 7)

Problema 23 (7 x 7)

Problema 24 (7 x 7)

Problema 25 (8 x 8)

Problema 26 (8 x 8)

Problema 27 (8 x 8)

Problema 28 (9 x 9)

Problema 29 (9 x 9)

Problema 30 (9 x 9)

Figura 11.12 La lista di problemi risolvibili.

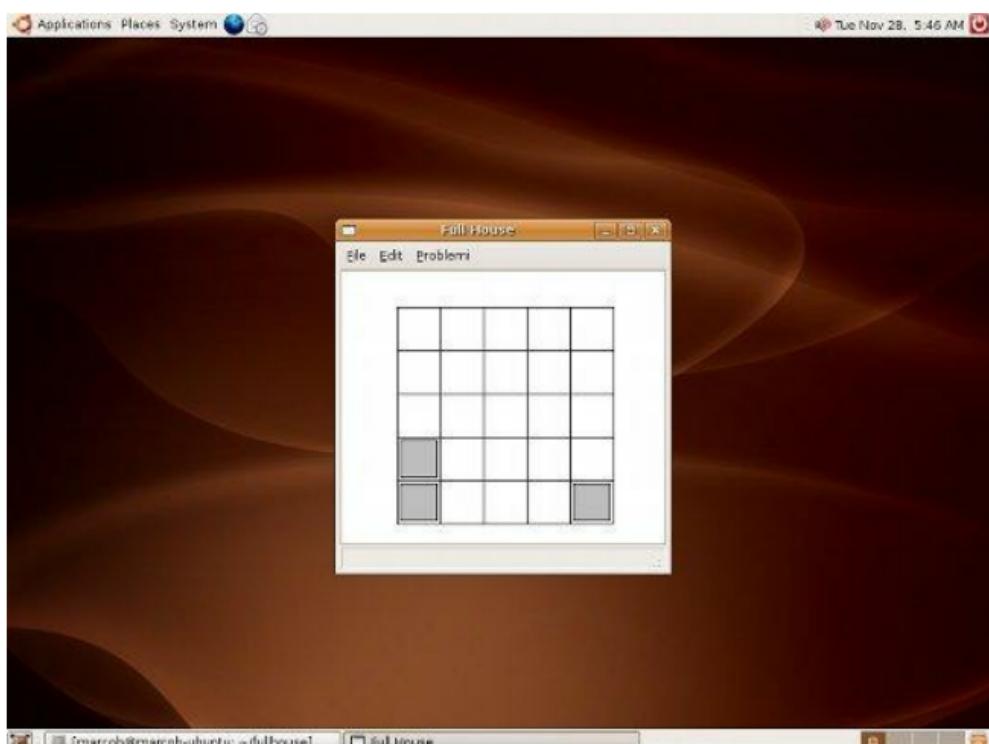


Figura 11.13 FullHouse in azione in Ubuntu Linux.



[marco@marco-ubuntu: ~]\$ fulhouse

Figura 11.14 Il risolutore all'opera.

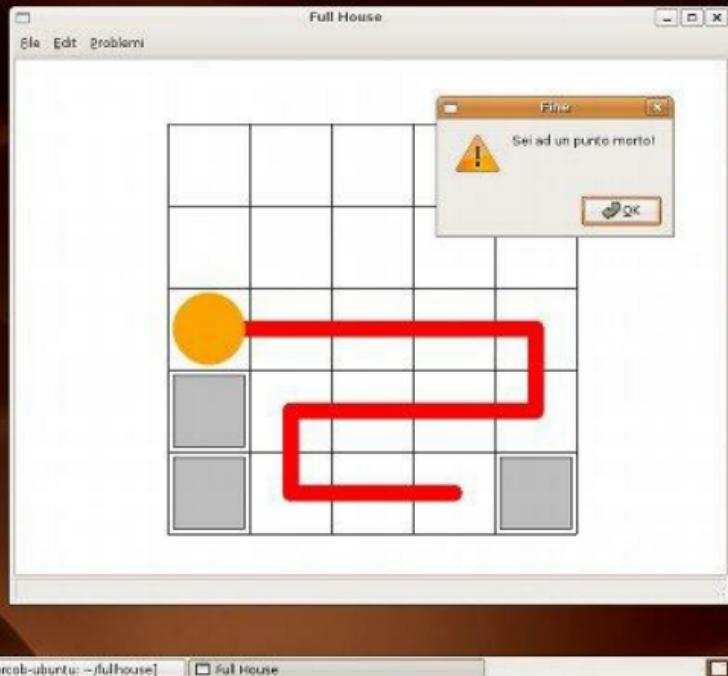


Figura 11.15 La finestra ingrandita.

CAPITOLO 12

PyWin32: le estensioni per Windows

Il rapporto di amore e odio che lega molti di noi a Windows non può prescindere dal fatto che con questo diffusissimo sistema operativo dobbiamo convivere, soprattutto dal punto di vista lavorativo.

Per facilitare il colloquio fra Python e Windows o fra Python e gli applicativi più diffusi (Word, Excel e così via) sono stati sviluppati svariati moduli che sono raccolti sotto il nome di PyWin32.

L'autore di queste indispensabili (almeno nel mondo Windows) librerie è Marc Hammond e il sito da cui possiamo scaricarle è:

Le funzionalità offerte da PyWin32 sono moltissime e un loro elenco non ci aiuterebbe molto a capirne le potenzialità. Preferiamo mostrare una serie di problemi che possiamo incontrare quando abbiamo a che fare con Windows e i suoi applicativi; vedremo come è possibile risolverli in maniera elegante con Python e PyWin32.

Microsoft Word

Word è il word processor (elaboratore di testi) della suite microsoft Office. La sua diffusione all'interno delle aziende è praticamente capillare, per cui non è raro trovarsi a dover gestire dei documenti scritti con questo software.

Proviamo a immaginare la necessità di dover indicizzare in qualche modo una grande quantità di documenti Word, estraendo il testo presente al loro interno. Sarebbe interessante poter automatizzare

un'operazione del genere; con PyWin32 è possibile.

Vediamo un piccolo script che legge il contenuto di tutti i documenti Word passati come parametro sulla riga di comando (accetta anche caratteri speciali come l'asterisco), ne estrae il testo, separa le parole, le conta e mostra le dieci parole più comuni tra tutti i documenti elaborati.

Vediamo ora l'aspetto dello script `conta_word.py`:

```
import win32com.client
import glob, sys, os, re
from collections import defaultdict
from operator import itemgetter
try:
    wordapp =
win32com.client.DispatchEx(\n        "Word.Application")
    solo_alfanumerici = re.compile(\n        r"[^a-zA-Z0-9 ]").sub
    dizionario = defaultdict(int)
    for argomento in sys.argv[1:]:
        for nome_doc in
glob.glob(argomento):
```

```
        print "Sto leggendo %s" %
nome_doc
        doc =
wordapp.Documents.Open(\os.path.abspath(nome_doc)
testo = solo_alfanumerici("",
",
doc.Content.Text.lower())
for parola in testo.split():
    dizionario[parola] += 1
print "\nParole comuni:\n"
for parola, cont in
sorted(dizionario.items(),
key=itemgetter(1),
reverse=True) [:10]:
    print parola, cont
finally:
    wordapp.Quit()
```

Nota Per chi le ha contate veramente, sono più di venti righe, ma solo per motivi di impaginazione.

Nella Figura 12.1 vediamo l'output di questo script su alcuni documenti Word. In questo caso la parola più comune è “di” con ben 1079 occorrenze, seguita da “self” con 904 e da “il” con

Soffermiamoci ora su alcune righe dello script.

Con la seguente istruzione creiamo un oggetto Word.Application che possiamo poi utilizzare all'interno del nostro script:

```
wordapp = win32com.client.DispatchEx(\n    "Word.Application")
```

Le regular expression sono un potente concetto, che permette la manipolazione avanzata delle stringhe. La seguente regular expression individua un oggetto che ricerca in una stringa tutti i caratteri diversi dalle lettere, dalle cifre e dallo spazio; possiamo utilizzarla per sostituire le occorrente trovate:

C:\WINDOWS\system32\cmd.exe

```
C:\Work\python>python conta_word.py docs\x.*  
Sto leggendo docs\Capitolo 1.doc  
Sto leggendo docs\Capitolo 10.doc  
Sto leggendo docs\Capitolo 11.doc  
Sto leggendo docs\Capitolo 12.doc  
Sto leggendo docs\Capitolo 13.doc  
Sto leggendo docs\Capitolo 14.doc  
Sto leggendo docs\Capitolo 2.doc  
Sto leggendo docs\Capitolo 3.doc  
Sto leggendo docs\Capitolo 4.doc  
Sto leggendo docs\Capitolo 5.doc  
Sto leggendo docs\Capitolo 6.doc  
Sto leggendo docs\Capitolo 7.doc  
Sto leggendo docs\Capitolo 8.doc  
Sto leggendo docs\Capitolo 9.doc  
Sto leggendo docs\Introduzione.doc
```

Parole comuni:

```
di 1079  
self 904  
il 764  
posizione 759  
in 668  
la 659  
e 652  
che 570  
un 539  
una 448
```

C:\Work\python>

Figura 12.1 L'output di conta_word.py.

```
solo_alfanumerici = re.compile(\r"^[a-zA-Z0-9 ]").sub
```

Il dizionario che viene creato contiene come chiavi le parole e come valori le occorrenze di ogni parola:

```
for parola in testo.split():
    dizionario[parola] += 1
```

La funzione sorted effettua l'ordinamento della sequenza passata come primo parametro. Il parametro key specifica una funzione che viene utilizzata per estrarre la chiave da ciascun argomento da confrontare. Infine il parametro reverse inverte l'ordinamento. Il risultato, nel nostro caso, è una sequenza ordinata di tuple in cui il primo valore è la parola e il secondo è il numero di occorrenze; quest'ultimo valore viene poi utilizzato per l'ordinamento:

```
for parola, cont in
sorted(dizionario.items(),
       key=itemgetter(1),
       reverse=True)[:10]:
    print parola, cont
```

Solo la fantasia può limitare le possibilità che abbiamo di tradurre in un utilizzo “da programma” gli applicativi che impieghiamo normalmente in modalità interattiva.

Internet Explorer

Un altro software molto diffuso che può essere “telecomandato” con PyWin32 è il browser Internet Explorer. In questo caso ci proponiamo il compito di navigare su un sito, inserire dei dati in un form e poi provare a inviare il form e i suoi dati.

Lo script ie.py è davvero semplicissimo:

```
import win32com.client, time
ie=win32com.client.DispatchEx(\n    'InternetExplorer.Application',None)
ie.Visible=1
ie.Navigate("http://www.google.com")
while ie.Busy:
    time.sleep(0.5)
ie.Document.Forms[0].Elements['q'].value
= \
    'Python -Monty'
ie.Document.Forms[0].submit()
```

Se eseguiamo lo script, accadrà quello che possiamo osservare nella Figura 12.2.

Lo script è semplice e intuitivo, le uniche due righe che meritano qualche chiarimento sono le seguenti:

```
while ie.Busy:  
    time.sleep(0.5)
```

Questo ciclo fa sì che il nostro script attenda che Internet Explorer attenda il download del sito che abbiamo indicato. Al termine del download la variabile ie.Busy diventerà false.



Figura 12.2 Ecco cosa accade se eseguiamo ie.py.

Nota In questo esempio abbiamo scelto di usare Google per comodità. Dobbiamo però dire che il modo migliore di interagire con Google da uno script Python non è certamente quello che abbiamo utilizzato nello script `ie.py`. Le API di Google permettono infatti di interrogare il motore di ricerca senza passare da un browser. Il sito dal quale possiamo scaricare la documentazione delle API di Google è:
http://code.google.com/apis/base/samples/python_sample.html

Singola istanza

In alcune particolari situazioni possiamo volerci accertare che una determinata applicazione non sia in esecuzione due volte: per esempio quando realizziamo un daemon.

Terminologia Un daemon è un programma che normalmente è in esecuzione in background, quindi in attesa di un qualche

evento particolare, senza essere sotto il controllo diretto dell'utente. Un daemon può, per esempio, attendere la ricezione di un messaggio di posta elettronica o la creazione di un file o la pressione di una particolare sequenza di tasti; quando si verifica la condizione prevista, svolge il compito cui è preposto.

In ambiente Windows, come possiamo essere certi che un'applicazione non sia in esecuzione due volte? Per esempio possiamo usare lo script singolo.py:

```
import win32api, winerror, win32event
import msvcrt, sys
nome = "singola-1308196423091964"
mutex = win32event.CreateMutex(None,
False, nome)
if win32api.GetLastError() == \
    winerror.ERROR_ALREADY_EXISTS:
    print "Applicazione in esecuzione"
    sys.exit(-1)
else:
    print "Premi un tasto..."
    msvcrt.getch()
```

```
win32api.CloseHandle(mutex)
```

La Figura 12.3 mostra due finestre distinte in cui è stato lanciato lo stesso script. Mentre nella prima lo script rimane in attesa della pressione di un tasto, nella seconda lo stesso script termina segnalando che l'applicazione è già in esecuzione.

Le istruzioni fondamentali dello script sono le seguenti:

```
nome = "singola-1308196423091964"  
mutex = win32event.CreateMutex(None,  
False, nome)
```

C:\WINDOWS\system32\cmd.exe - python singolo.py

C:\Work\Python>python singolo.py
Premi un tasto...



Figura 12.3 Uno script che non può essere in esecuzione due volte.

In questo modo creiamo un mutex, con un nome complesso scelto in modo da identificare univocamente la nostra applicazione. Il mutex viene mantenuto in vita fino alla fine dell'applicazione, quando viene rilasciato con l'istruzione CloseHandle. È importante notare che,

anche se l'applicazione termina in maniera brutale per qualche errore imprevisto, il mutex viene rilasciato comunque.

Terminologia Un mutex (da “*MUTual EXclusion*”, *mutua esclusione*) è un oggetto che permette di evitare l'accesso simultaneo a risorse condivise tra diversi programmi che se le contendono.

ctypes

ctypes è un modulo che, pur non essendo incluso in PyWin32, fa parte dell'installazione standard di Python e può talvolta essere utile in Windows.

ctypes permette infatti a Python di colloquiare con delle librerie esterne scritte in linguaggio C o comunque richiamabili da questo linguaggio. Di questo insieme fanno parte, oltre che le shared library (librerie condivise) del mondo Unix, anche le ben note DLL di Windows.

Terminologia DLL è un acronimo che deriva

da Dynamic Link Library: librerie “collegate” dinamicamente.

Un semplice esempio pratico è il modo migliore per afferrare il concetto. Nella DLL standard di Windows User32.dll è presente l’API LockWorkStation che permette di ottenere il blocco (lock) del computer sul quale viene eseguita. Possiamo richiamare questa funzione da Python usando l’interfaccia windll di ctypes:

```
import ctypes  
ctypes.windll.user32.LockWorkStation()
```

Attenzione *Se provate a usare questo codice, accertatevi di conoscere la password del sistema su cui lo state sperimentando o rischiate di non poter più uscire dallo stato di “lock” e di dover spegnere brutalmente il computer!*

ctypes permette di richiamare le funzioni di qualsiasi DLL e quindi ci mette a disposizione

anche delle primitive in grado di passare dei puntatori come parametri; spesso questa è una necessità quando si utilizzano le librerie di sistema.

Vediamo lo script api.py di poche righe che ci mostra la chiamata di un paio di API di Windows di questo tipo:

```
import ctypes
n = 256
buffer = ctypes.create_string_buffer(n)
ctypes.windll.kernel32.GetSystemDirectory(
    buffer, n)
print buffer.value
ctypes.windll.kernel32.GetComputerNameA(
    buffer, ctypes.byref(ctypes.c_int64(n)))
print buffer.value
```

L'API GetSystemDirectoryA restituisce la directory di sistema di Windows (nel nostro caso C:\WINDOWS\system32) e accetta come parametri il puntatore a un buffer di tipo stringa (che abbiamo creato con la funzione di ctypes create_string_buffer) e la lunghezza del buffer

stesso.

L'API GetComputerName restituisce il nome del computer sul quale viene eseguita (nel nostro caso PCMARCO) e accetta come parametri il puntatore a un buffer di tipo stringa (come il precedente) e un puntatore a una variabile che indica la lunghezza del buffer stesso (creato con byref e c_int64). Non siamo in grado di spiegare le motivazioni di questo differente standard di chiamata ad API apparentemente così simili nella tipologia di parametri, ma ci basta che Python, grazie a ctypes, possa utilizzarle entrambe.

L'esecuzione dello script api.py dalla riga di comando produrrà un output simile a quello rappresentato nella Figura 12.4 (ovviamente i valori cambieranno a seconda della versione del sistema operativo Windows e del nome del computer).

Ora possiamo collaudare le stesse API dal prompt di Python. Nella Figura 12.5 vediamo come sia

facile utilizzare interattivamente delle DLL di sistema.

Catturare una combinazione di tasti

Negli ultimi paragrafi abbiamo parlato di daemon e di ctypes. Normalmente un daemon può essere in attesa della pressione di uno o più tasti da parte dell'utente; per esempio possiamo decidere che alla pressione contemporanea dei tre tasti Ctrl, Maiusc e P, il nostro daemon si svegli ed effettui l'operazione desiderata.

C:\WINDOWS\system32\cmd.exe

C:\Work\python>python api.py

C:\WINDOWS\system32

PCMARCO

C:\Work\python>

Figura 12.4 L'output di api.py.

The screenshot shows a Python Shell window with the title "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main window displays the following Python code and its output:

```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2
>>> import ctypes
>>> n = 256
>>> buffer = ctypes.create_string_buffer(n)
>>> ctypes.windll.kernel32.GetSystemDirectoryA(buffer, n)
19
>>> print buffer.value
C:\WINDOWS\system32
>>>
>>> ctypes.windll.kernel32.GetComputerNameA(buffer, ctypes.byref(ctypes.c_int64(n)))
1
>>> print buffer.value
PCMARCO
>>> |
```

Ln: 24 Col: 4

Figura 12.5 Uso interattivo delle API di Windows usando puntatori come parametri.

Vediamo come possiamo ottenere questo comportamento con PyWin32 congiuntamente al modulo `ctypes`.

Lo script `hotkey.py` fa al caso nostro:

```
import sys
import ctypes, ctypes.wintypes
import win32con
class MSG(ctypes.wintypes.Structure):
    _fields_ = [ ('hwnd', ctypes.c_int),
    ('message', ctypes.c_uint),
    ('wParam', ctypes.c_int),
    ('lParam', ctypes.c_int),
    ('time', ctypes.c_int),
    ('pt', ctypes.wintypes.POINT) ]
user32 = ctypes.windll.user32
id_hotkey = 1
if not user32.RegisterHotKey(None, \
id_hotkey,
win32con.MOD_CONTROL
| win32con.MOD_SHIFT,
ord('P')):
    sys.exit("Impossibile registrare la
hotkey")
msg = MSG()
while
user32.GetMessageA(ctypes.byref(msg), \
None, 0, 0) != 0:
    if msg.message == win32con.WM_HOTKEY \
and msg.wParam == id_hotkey:
        print "Hotkey premuta..."
        user32.PostQuitMessage(0)
    user32.TranslateMessage(ctypes.byref(msg))
```

```
user32.DispatchMessageA(ctypes.byref(msg))
```

Nella Figura 12.6 vediamo lo script caricato ed eseguito in una finestra di IDLE. Alla pressione simultanea dei tasti Ctrl, Maiusc e P, nella finestra della shell sullo sfondo è comparso il messaggio di avviso che abbiamo inserito nello script.

Alcune istruzioni meritano una descrizione più approfondita.

La classe MSG ci serve per emulare il primo parametro dell'API GetMessageA che, essendo una struttura C, deve essere definita tramite il subclassing della classe ctypes.wintypes.Structure. I singoli elementi della struttura sono a loro volta classi di ctypes:

```
class MSG(ctypes.wintypes.Structure):  
    _fields_ = [ ('hwnd', ctypes.c_int),  
                ('message', ctypes.c_uint),  
                ('wParam', ctypes.c_int),  
                ('lParam', ctypes.c_int),  
                ('time', ctypes.c_int),  
                ('pt', ctypes.wintypes.POINT) ]
```

```
Python 2.5 (r25_51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

Personal firewall software makes to its subprocesses no interface and no data
IDLE 1.2
>>> =====
>>> Hotkey premuta...
>>>
    76 hotkey.py  C:\Work\Python\hotkey.py
File Edit Format Run Options Windows Help
import sys
import ctypes, ctypes.wintypes
# import from ctypes import *
# from ctypes.wintypes import *
import win32con

class MSG(ctypes.wintypes.Structure):
    _fields_ = [('hwnd', ctypes.c_int),
                ('message', ctypes.c_uint),
                ('wParam', ctypes.c_int),
                ('lParam', ctypes.c_int),
                ('time', ctypes.c_int),
                ('pt', ctypes.wintypes.POINT)]

user32 = ctypes.windll.user32

id_hotkey = 1
if not user32.RegisterHotKey(None,
                               id_hotkey,
                               win32con.MOD_CONTROL
                               | win32con.MOD_SHIFT,
                               ord('P')):
    sys.exit("Impossibile registrare la hotkey")

msg = MSG()
while user32.GetMessageA(ctypes.byref(msg), None, 0, 0) != 0:
    if msg.message == win32con.WM_HOTKEY and msg.wParam == id_hotkey:
        print "Hotkey premuta..."
        user32.PostQuitMessage(0)
    user32.TranslateMessage(ctypes.byref(msg))
    user32.DispatchMessageA(ctypes.byref(msg))

Ln 32 Col 0
Ln 15 Col 0
```

Figura 12.6 hotkey.py eseguito in IDLE.

Ora registriamo la combinazione di tasti con l'identificativo `id_hotkey`, che ci permetterà di distinguerla dalle altre:

```
if not user32.RegisterHotKey(None, \
    id_hotkey, \
    win32con.MOD_CONTROL
```

```
| win32con.MOD_SHIFT,  
ord('P')):  
sys.exit("Impossibile registrare la  
hotkey")
```

Nel seguente ciclo rimaniamo in attesa di un messaggio che sia del tipo `win32con.WM_HOTKEY` e che faccia riferimento alla chiave registrata con `id_hotkey`:

```
while  
user32.GetMessageA(ctypes.byref(msg), \  
None, 0, 0) != 0:  
if msg.message == win32con.WM_HOTKEY \  
and msg.wParam == id_hotkey:  
print "Hotkey premuta..."  
user32.PostQuitMessage(0)
```

Ogni altro messaggio viene restituito al sistema che ne svolgerà la gestione di default:

```
user32.TranslateMessage(ctypes.byref(msg))  
user32.DispatchMessageA(ctypes.byref(msg))
```

Excel

In questo capitolo abbiamo già visto come utilizzare Word in un nostro script. Se Word è l'editor di testi per eccellenza nel mondo Windows, Excel ha senz'ombra di dubbio lo stesso rango come foglio di calcolo, per cui merita un paragrafo apposito.

Nello script excel.py proveremo ad accedere a un intervallo di celle, a una singola cella, a una formula e a scrivere una nuova formula:

```
import win32com.client
excel = win32com.client.Dispatch(\"Excel.Application\")
print
excel.ActiveWorkbook.ActiveSheet.Range(\"A1:B1\")
print
excel.ActiveWorkbook.ActiveSheet.Range(\"D8\")
print
excel.ActiveWorkbook.ActiveSheet.Range(\"I
.Formula
excel.ActiveWorkbook.ActiveSheet.Range(\"I
.Formula = \"=SUM(D5:D6)*1.2"
excel.ActiveWorkbook.ActiveSheet.Range(\"I
.Borders(4).Weight = -4138
```

Nelle Figure 12.7 e 12.8 possiamo osservare che la cella D9 dapprima non contiene alcuna formula e, dopo l'esecuzione dello script, contiene la formula “=SUM(D5:D6)*1,2”; inoltre ora vi è una linea orizzontale sotto la cella D8.

Lo script excel.py è estremamente semplice e compatto. Proviamo a osservare la seguente istruzione:

```
excel.ActiveWorkbook.ActiveSheet.Range("I  
.Formula = "=SUM(D5:D6)*1.2"
```

Microsoft Excel - Book1

The screenshot shows a Microsoft Excel spreadsheet titled "Book1". The table has columns labeled A through G and rows numbered 1 through 20. Row 1 contains the text "Cliente: Cleese & Booth". Row 4 is a header row with columns "Articolo", "Importo", and "Quantità". Rows 5 and 6 contain data: "Spam" with Importo 20 and Quantità 40, and "Egg" with Importo 500 and Quantità 2000. Row 8 contains the text "Imponibile: 2040", and row 9 contains the text "Totale:". A Python command prompt window is overlaid on the bottom right of the Excel window, showing the path "C:\Work\Python>".

	A	B	C	D	E	F	G
1	Cliente:	Cleese & Booth					
2							
3							
4	Articolo	Importo	Quantità				
5	Spam	20	2	40			
6	Egg	500	4	2000			
7							
8		Imponibile:		2040			
9			Totale:				
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							

Figura 12.7 La cella D9 del foglio Excel è vuota.

La complessità è solo apparente e comunque la responsabilità non è di Python. Infatti, a partire da `.ActiveWorkbook` fino a `.Formula`, tutta l'istruzione viene passata così com'è a Excel, che si

preoccuperà di interpretarla ed eseguirla.

L'area delle notifiche

L'area delle notifiche (notification area, comunemente ed erroneamente chiamata system tray, vassoio di sistema) è quella porzione di schermo, solitamente all'angolo inferiore destro della barra dei comandi di Windows, che contiene l'orologio e le icone dei programmi che operano in background o che sono stati ridotti a icona ma necessitano di dare comunque un feedback all'utente.

Microsoft Excel - Book1

File Edit View Insert Format Tools Data Window Help

Reply with Changes... End Review... 150% Arial

D9 =SUM(D5:D6)*1,2

	A	B	C	D	E	F	G
1	Cliente: Cleese & Booth						
2							
3							
4	Articolo	Importo	Quantità				
5	Spam	20	2	40			
6	Egg	500	4	2000			
7							
8		Imponibile:	2040				
9		Totale:	2448				
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							

C:\WINDOWS\system32\cmd.exe

```
C:\Work\Python>python excel.py
((u'Cliente:', u'Cleese & Booth'),)
2040.0
=SUM(D5:D6)

C:\Work\Python>
```

Figura 12.8 Ora la cella D9 del foglio Excel contiene la formula desiderata.

Con lo script notification_area.py inseriamo nell'area di notifica un'icona a forma di triangolo giallo con un punto esclamativo, che reagisce ai

clic del mouse e visualizza un suggerimento:

```
import win32gui, win32con
class NotificationArea:
def __init__(self):
self.visibile = False
self.icon = win32gui.LoadIcon(0,
win32con.IDI_EXCLAMATION)
self.suggerimento = \
"Notification Area con Python"
wc = win32gui.WNDCLASS()
wc.lpszClassName =
"PythonNotificationArea"
wc.lpfnWndProc = {
win32con.WM_DESTROY: self.onDestroy,
win32con.WM_USER+23:
self.onNotificationAreaNotify}
self.hwnd = win32gui.CreateWindow(
win32gui.RegisterClass(wc),
"Esempio di Notification Area",
win32con.WS_OVERLAPPED |
win32con.WS_SYSMENU,
0, 0, win32con.CW_USEDEFAULT,
win32con.CW_USEDEFAULT,
0, 0, wc.hInstance, None)
win32gui.UpdateWindow(self.hwnd)
self.visualizza()
def visualizza(self):
flags = win32gui.NIF_ICON | \
```

```
win32gui.NIF_MESSAGE | \
win32gui.NIF_TIP
if self.visible:
    self.nascondi()
id_notification = (self.hwnd, 0, flags,
win32con.WM_USER+23,
self.icon, self.suggerimento)
win32gui.Shell_NotifyIcon(win32gui.NIM_AI
id_notification)
self.visible = True
def nascondi(self):
    if self.visible:
        id_notification = (self.hwnd, 0)
        win32gui.Shell_NotifyIcon(
            win32gui.NIM_DELETE,
            id_notification)
    self.visible = False
def onDestroy(self, hwnd, msg, wparam,
lparam):
    self.nascondi()
    win32gui.PostQuitMessage(0)
def onNotificationAreaNotify(self, hwnd,
msg,
wparam, lparam):
    if lparam == win32con.WM_LBUTTONDOWN:
        print "Click sinistro!"
    elif lparam == win32con.WM_RBUTTONDOWN:
        print "Click destro!"
    win32gui.PostQuitMessage(0)
```

```
elif lparam ==  
win32con.WM_LBUTTONDOWN:dblclk:  
print "Doppio click!"  
return True  
if __name__ == '__main__':  
NotificationArea()  
win32gui.PumpMessages()
```

Nella Figura 12.9 vediamo l'aspetto dell'icona a forma di triangolo giallo con un suggerimento. Nella finestra del prompt dei comandi sono visibili i diversi messaggi prodotti a fronte di alcuni clic del mouse sulla nostra icona.

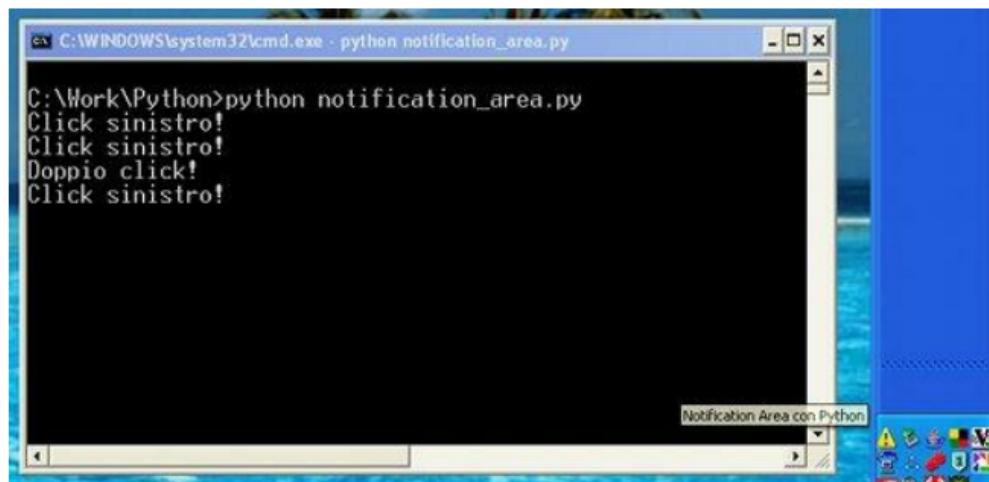


Figura 12.9 L'icona nell'area delle notifiche e il suggerimento visualizzato.

Lo script è abbastanza complesso proprio per via delle strutture di Windows che dobbiamo utilizzare per ottenere il nostro scopo. Per fare maggior chiarezza commentiamo alcune parti dello script.

Con la seguente istruzione carichiamo una delle icone standard di Windows, già a nostra disposizione:

```
self.icon = win32gui.LoadIcon(0,  
win32con.IDI_EXCLAMATION)
```

Ora definiamo una classe di finestre con la quale creeremo poi la finestra che gestirà l'icona:

```
wc = win32gui.WNDCLASS()  
wc.lpszClassName =  
"PythonNotificationArea"
```

Con la seguente istruzione indichiamo gli eventi (o i messaggi) che vogliamo intercettare con la nostra icona:

```
wc.lpfnWndProc = {  
win32con.WM_DESTROY: self.onDestroy,
```

```
win32con.WM_USER+23:  
self.onNotificationAreaNotify}
```

Quindi dobbiamo creare e visualizzare la finestra per la nostra icona:

```
self.hwnd = win32gui.CreateWindow(  
win32gui.RegisterClass(wc),  
"Esempio di Notification Area",  
win32con.WS_OVERLAPPED |  
win32con.WS_SYSMENU,  
0, 0, win32con.CW_USEDEFAULT,  
win32con.CW_USEDEFAULT,  
0, 0, wc.hInstance, None)  
win32gui.UpdateWindow(self.hwnd)  
self.visualizza()
```

Questo metodo risponde agli eventi corrispondenti al clic sinistro, al clic destro (che termina l'esecuzione) e al doppio clic del mouse richiamando altri metodi :

```
def onNotificationAreaNotify(self, hwnd,  
msg,  
wparam, lparam):  
if lparam == win32con.WM_LBUTTONDOWN:  
print "Click sinistro!"
```

```
elif lparam == win32con.WM_RBUTTONUP:  
print "Click destro!"  
win32gui.PostQuitMessage(0)  
elif lparam ==  
win32con.WM_LBUTTONDOWN:  
print "Doppio click!"
```

Infine, dopo aver creato un'istanza della classe `NotificationArea`, avviamo il ciclo di gestione dei messaggi di Windows:

```
NotificationArea()  
win32gui.PumpMessages()
```

Suggerimento Per eseguire il nostro programma senza che venga visualizzata la finestra del prompt dei comandi possiamo usare direttamente dal menu di avvio il comando `pythonw`, che esegue uno script senza creare alcuna finestra.

Applicativi “vecchio stampo”

Comandare con PyWin32 un applicativo che funga da server è abbastanza facile. Quando però ci

troviamo di fronte ad applicativi che non hanno funzioni di server OLE o COM, il discorso non è più così semplice. Con il prossimo script, calcolatrice.py, proveremo ad affrontare una situazione di questo tipo.

Lo script richiama la Calcolatrice di Windows, porta l'applicazione in primo piano, digita i tasti per una divisione, simula la pressione dei tasti Ctrl e Ins per copiare il risultato negli Appunti (in inglese la Clipboard) e infine legge il risultato traendolo dagli appunti:

```
import win32gui, win32api
import win32com.client
import win32clipboard
wscript =
win32com.client.Dispatch("WScript.Shell")
wscript.Run("calc")
while True:
win = win32gui.FindWindow(None,
"Calculator")
if win == 0:
win = win32gui.FindWindow(None, \
"Calcolatrice")
if win != 0:
```

```
break
win32api.Sleep(500)
win32gui.SetForegroundWindow(win)
wscript.SendKeys("{ESC}")
wscript.SendKeys("355/113=")
wscript.SendKeys("^{INSERT}")
win32api.Sleep(500)
win32clipboard.OpenClipboard(0)
print win32clipboard.GetClipboardData(\_
win32clipboard.CF_TEXT)
```

La Figura 12.10 mostra il notevole risultato ottenuto con questo script.

L'oggetto creato con la seguente istruzione si chiama Windows Script Host ed è uno strumento di amministrazione che permette il colloquio non interattivo con gli applicativi Windows:

```
wscript =
win32com.client.Dispatch("WScript.Shell")
```

Con questo oggetto eseguiamo la Calcolatrice:

```
wscript.Run("calc")
```

Con l'API FindWindow cerchiamo la finestra dell'applicazione appena eseguita. Nel ciclo seguente cerchiamo una finestra con titolo "Calculator" o "Calcolatrice" perchè questo cambia a seconda della versione (inglese o meno) di Windows:

```
win = win32gui.FindWindow(None,  
"Calculator")  
if win == 0:  
    win = win32gui.FindWindow(None, \  
"Calcolatrice")
```

Attenzione *Attenzione: se nel vostro sistema non esiste l'applicazione calc.exe, il ciclo precedente non avrà mai fine.*

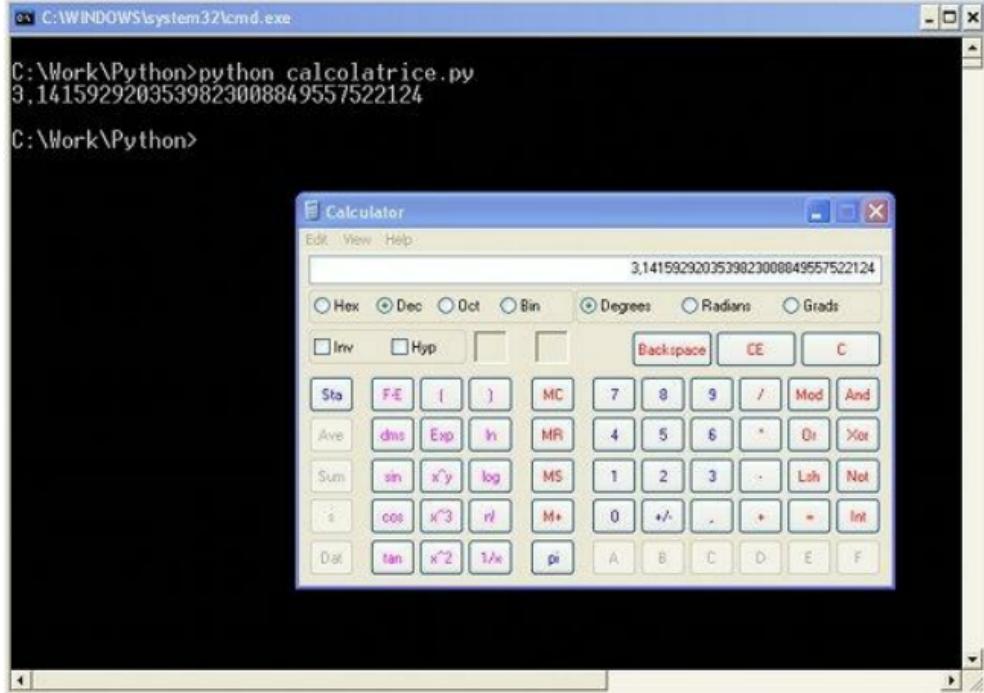


Figura 12.10 La simulazione della pressione di tasti nella Calcolatrice di Windows.

A questo punto, con l'API SetForegroundWindow, portiamo in primo piano la finestra della calcolatrice:

```
win32gui.SetForegroundWindow(win)
```

E finalmente possiamo inviare i tasti desiderati per effettuare la divisione:

```
wscript.SendKeys ("{ESC}")  
wscript.SendKeys ("355/113=")
```

Suggerimento La divisione tra 355 e 113 dà come risultato un numero che è uguale a pi greco, fino alla sesta cifra decimale (compresa).

Simuliamo la pressione dei tasti Ctrl e Ins:

```
wscript.SendKeys ("^ {INSERT}")
```

Dopo aver atteso 500 millisecondi per dare il tempo alla Calcolatrice di elaborare il comando di Copia negli Appunti, possiamo leggere e visualizzare il contenuto degli Appunti stessi:

```
win32api.Sleep(500)  
win32clipboard.OpenClipboard(0)  
print win32clipboard.GetClipboardData(\  
win32clipboard.CF_TEXT)
```

Esempi di PyWin32

Gli script esaminati fino a questo punto, pur non

essendo banali, non raggiungono la complessità degli esempi presenti nell’installazione di PyWin32. La directory nella quale possiamo trovarli è normalmente la seguente:

```
C:\Python25\Lib\site-  
packages\win32com\demos\
```

Lo script excelAddin.py aggiunge un pulsante alla barra degli strumenti di Excel; se premuto, mostra una semplice finestra pop-up. Nella Figura 12.11 vediamo il risultato della sua esecuzione.

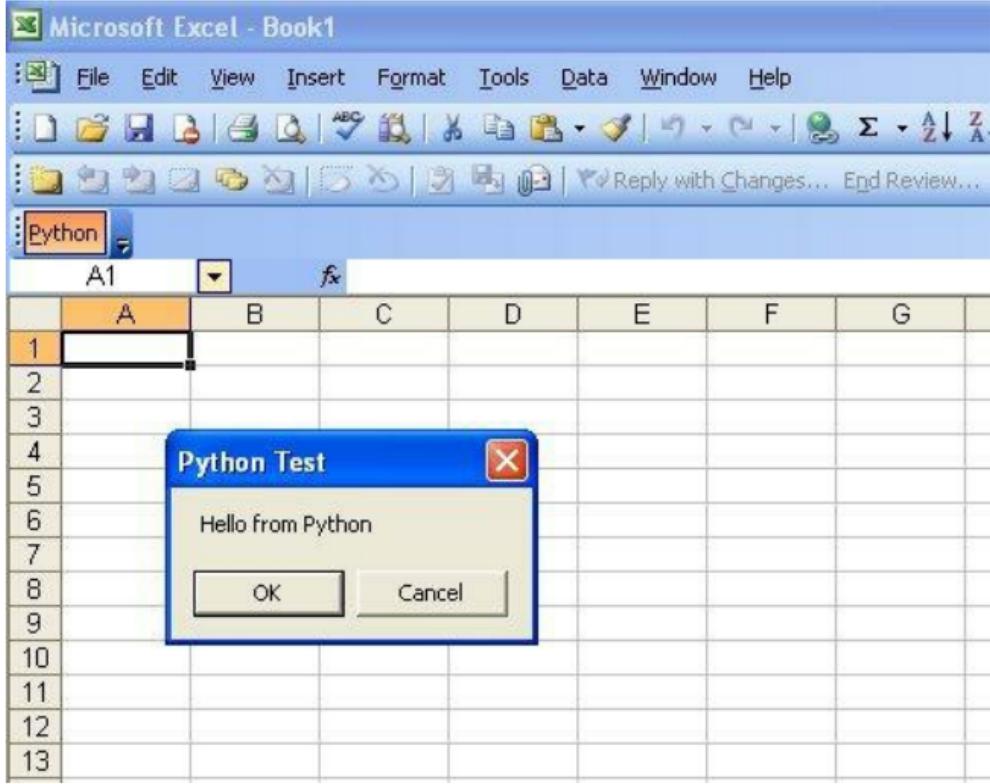


Figura 12.11 Aggiunta di un pulsante nella barra degli strumenti di Excel.

CAPITOLO 13

Argomenti avanzati

Quanto visto finora ci ha permesso di fare la conoscenza delle basi di Python. Esistono però molti altri argomenti, ognuno dei quali meriterebbe una discussione più approfondita. In questo capitolo getteremo uno sguardo su alcuni aspetti avanzati di Python. Per chi fosse interessato all'approfondimento di questi argomenti, la documentazione e il sito di Python rappresentano un'enorme fonte di informazioni. Tra le altre cose impareremo a creare un programma di installazione professionale per il gioco Full House.

Introspezione

La parola introspezione deriva dal latino introspicere, “guardarsi dentro”. L'introspezione di

Python è in fondo la stessa cosa: la capacità di guardarsi dentro e di fornire informazioni su se stesso. Abbiamo già visto nel dettaglio due esempi di questa capacità nel Capitolo 4, con i comandi `dir` e `type`. Esistono però altre funzioni che contribuiscono alla capacità introspettiva di Python: `id` genera in output un numero che corrisponde all'identità dell'oggetto passato come parametro; `callable` dice se il parametro è “richiamabile” (come funzione o come metodo o come classe); `isinstance` dice se il primo argomento è un’istanza della classe passata come secondo argomento; `issubclass` dice se il primo argomento è una sottoclasse, diretta o indiretta, della classe passata come secondo argomento; `help` cerca e visualizza ogni aiuto a disposizione per comprendere il funzionamento dell’oggetto passato come argomento.

Possiamo leggere un interessantissimo e approfondito articolo di Patrick O’Brien sulla capacità introspettiva di Python al seguente indirizzo:

<http://www.ibm.com/developerworks/library/l-pyint.html>

Nell'articolo (che purtroppo è solo disponibile in inglese) troviamo una semplice funzione che ci permette di visualizzare utili informazioni su qualsiasi oggetto:

```
def interrogate(item):
    """Stampa informazioni su un
oggetto."""
    if hasattr(item, '__name__'):
        print "NAME: ", item.__name__
    if hasattr(item, '__class__'):
        print "CLASS: ",
item.__class__.__name__
        print "ID: ", id(item)
        print "TYPE: ", type(item)
        print "VALUE: ", repr(item)
        print "CALLABLE: ", "Yes" if
callable(item) \
    else "No"
    if hasattr(item, '__doc__'):
        doc = getattr(item, '__doc__')
        doc = doc.strip()
        firstline = doc.split('\n')[0]
        print "DOC: ", firstline
```

Vediamo il funzionamento di questa funzione su una stringa, su un intero e... su se stessa nella Figura 13.1.

Un altro spettacolare esempio di introspezione è rappresentato nelle Figure 13.2 e 13.3: dall'interno della demo di wxPython apriamo una shell (Help / Open PyShell Window) con la quale possiamo controllare e modificare interattivamente gli attributi dell'applicazione stessa.

Debugging

Spesso il debug, in Python come in altri linguaggi di programmazione, si effettua con il metodo tradizionale, inserendo dei messaggi all'interno del codice. Talvolta però può essere molto più comodo utilizzare uno strumento più evoluto. In Python questo strumento esiste e si chiama pdb (Python debugger).

pdb può essere utilizzato interattivamente ma può anche essere richiamato dall'interno dei nostri

script. Vediamo un esempio di entrambe le modalità.

The screenshot shows a Python Shell window with the following content:

```
>>> def interrogate(item):
    """Stampa informazioni su item."""
    if hasattr(item, '__name__'):
        print "NAME: ", item.__name__
    if hasattr(item, '__class__'):
        print "CLASS: ", item.__class__.__name__
    print "ID: ", id(item)
    print "TYPE: ", type(item)
    print "VALUE: ", repr(item)
    print "CALLABLE: ", "Yes" if callable(item) \
        else "No"
    if hasattr(item, '__doc__'):
        doc = getattr(item, '__doc__')
        doc = doc.strip()
        firstline = doc.split('\n')[0]
        print "DOC: ", firstline

>>> interrogate("stringa")
CLASS: str
ID: 34152544
TYPE: <type 'str'>
VALUE: 'stringa'
CALLABLE: No
DOC: str(object) -> string
>>> interrogate(23)
CLASS: int
ID: 31151696
TYPE: <type 'int'>
VALUE: 23
CALLABLE: No
DOC: int(x[, base]) -> integer
>>> interrogate(interrogate)
NAME: interrogate
CLASS: function
ID: 33590448
TYPE: <type 'function'>
VALUE: <function interrogate at 0x02008C80>
CALLABLE: Yes
DOC: Stampa informazioni su item.
>>>
```

Ln: 51 Col: 4

Figura 13.1 Alcuni esempi di output della funzione interrogate.

Creiamo uno script lavoro.py come il seguente:

```
# Classe persona
class Persona:
    # Costruttore
    def __init__(self, nome):
        self.nome = nome
        self.capitale = 0

    # Paga la persona aggiungendo il
    # compenso ai suo capitale
    def paga(self, compenso):
        self.capitale == self.capitale +
compenso

# Funzione di test
def Test():
    pippo = Persona('Pippo')
    pippo.paga(100)
    pippo.paga(50)
    print pippo.capitale

if __name__ == "__main__":
    Test()
```

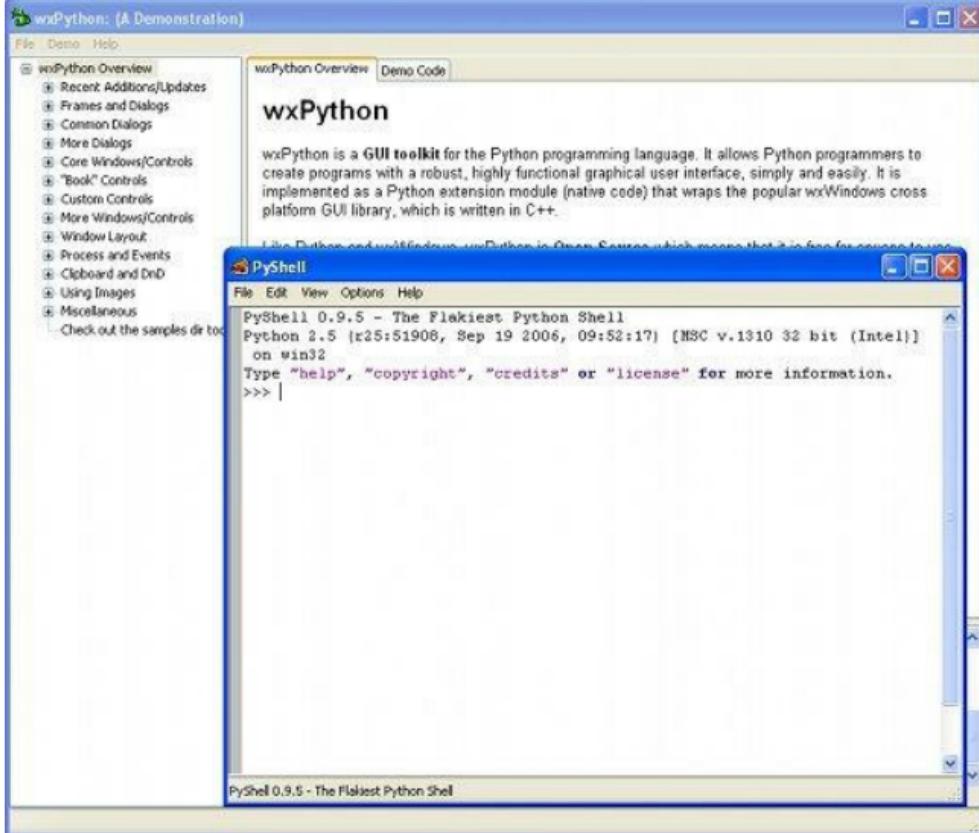


Figura 13.2 La demo di wxPython con la shell interattiva aperta.

Se lo eseguiamo dalla riga di comando potremmo aspettarci di vedere a video la stringa 150 per via dei due pagamenti di 100 e 50. Proviamo a eseguirlo; otteniamo invece quello che si vede nella Figura 13.4.

Nota L'errore nel codice è abbastanza evidente; qualcuno l'avrà senz'altro individuato senza bisogno di usare pdb. Questo è solo un esempio che ci permette di provare il debugger di Python.

Per capire dov'è il bug proviamo a eseguire il debugger dalla riga di comando digitando:

```
python -m pdb lavoro.py
```

Il prompt che ci appare è il seguente:

```
> c:\work\python\lavoro.py(2)<module>()
-> class Persona:
(Pdb)
```

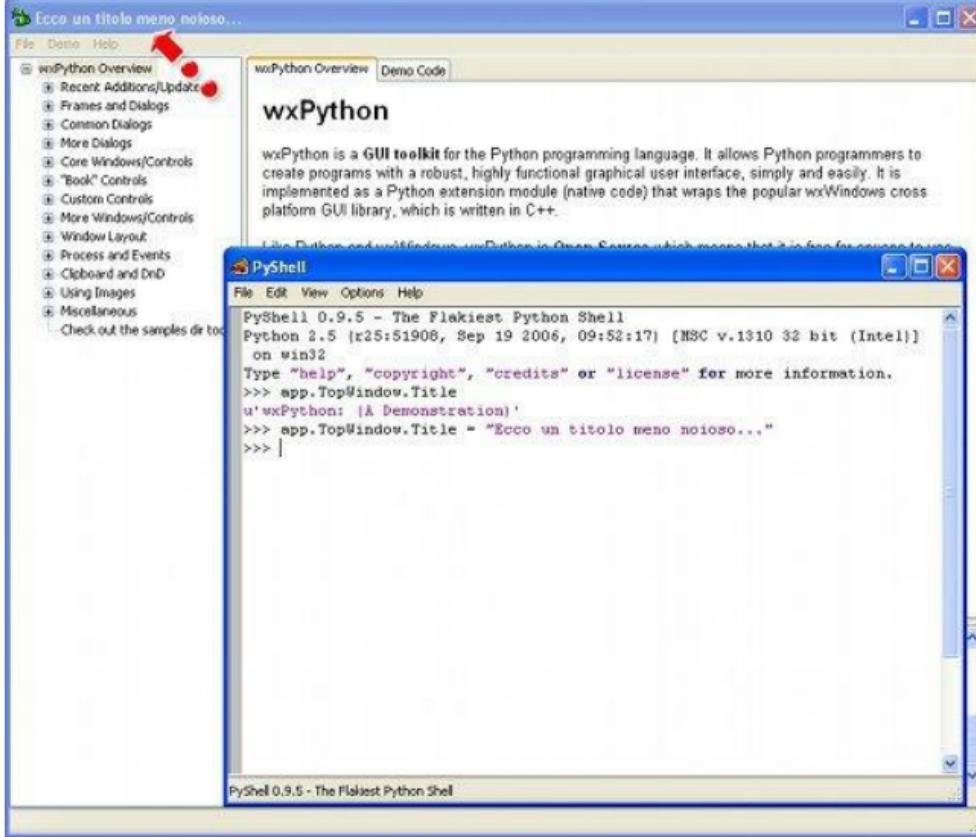


Figura 13.3 Dalla shell abbiamo addirittura modificato il titolo della finestra principale.

C:\WINDOWS\system32\cmd.exe



C:\Work\Python>python lavoro.py
Pippo ha 0 euro

C:\Work\Python>

Figura 13.4 Uno script che non si comporta come ci aspettiamo.

C:\WINDOWS\system32\cmd.exe - python -m pdb lavoro.py

C:\Work\Python>python -m pdb lavoro.py
> c:\work\python\lavoro.py(2)<module>()
-> class Persona:
(Pdb) h

Documented commands (type help <topic>):

```
=====EOF  break  commands  debug  h    l    pp    s    up
a    bt    condition  disable  help  list  q    step  w
alias  c    cont    down    ignore  n    quit  tbreak  whatis
args   cl    continue  enable  j    next  r    u    where
b    clear  d    exit    jump   p    return  unalias
```

Miscellaneous help topics:

```
=====exec  pdb
```

Undocumented commands:

```
=====retval  rv
```

(Pdb)

Figura 13.5 I comandi del debugger pdb.

Se proviamo a digitare help possiamo vedere la lista dei comandi a disposizione, così come sono presentati nella Figura 13.5.

Vorremmo provare a mettere un breakpoint (un punto di interruzione) dove viene sommato il compenso al capitale, nel metodo paga della classe Persona. Proviamo quindi a digitare:

break Persona.paga

Purtroppo ci appare un errore:

```
*** The specified object 'Persona.paga'  
is not a  
function or was not found along  
sys.path.
```

Il motivo è semplice: siamo all'inizio del codice sorgente e l'interprete non ha ancora interpretato il codice seguente. Proviamo allora a mettere un breakpoint nella funzione Test:

```
break Test
```

Questa volta non ci appare alcun messaggio d'errore:

```
(Pdb) break Test  
Breakpoint 1 at  
c:\work\python\lavoro.py:14  
(Pdb)
```

Ora possiamo far partire l'esecuzione con continue ed ecco cosa accade:

```
(Pdb) c
```

```
> c:\work\python\lavoro.py(16) Test()
-> pippo = Persona('Pippo')
(Pdb)
```

Per vedere in che punto del codice ci siamo fermati digitiamo list:

```
(Pdb) list
11 self.capitale == self.capitale +
compenso
12
13 # Funzione di test
14
15 B def Test():
16 -> pippo = Persona('Pippo')
17 pippo.paga(100)
18 pippo.paga(50)
19 print "%s ha %d euro" % (pippo.nome,
pippo.capitale)
20
21 if __name__ == "__main__":
(Pdb)
```

La freccia indica proprio la prima riga della funzione Test. Ora proviamo a inserire un nuovo breakpoint nel metodo Persona.paga e a riprendere l'esecuzione:

```
(Pdb) break Persona.paga
Breakpoint 2 at
c:\work\python\lavoro.py:10
(Pdb) continue
> c:\work\python\lavoro.py(11)paga()
-> self.capitale == self.capitale +
compenso
(Pdb)
```

Eccoci al punto critico. Eseguiamo una singola istruzione digitando step:

```
(Pdb) step
--Return--
> c:\work\python\lavoro.py(11)paga()-
>None
-> self.capitale == self.capitale +
compenso
(Pdb)
```

Il capitale dovrebbe essere stato incrementato; proviamo a visualizzarlo insieme al compenso:

```
(Pdb) print self.capitale, compenso
0 100
(Pdb)
```

Niente da fare: capitale vale ancora 0.

Proviamo a dare un'occhiata alla riga di codice corrente:

```
self.capitale == self.capitale +  
compenso
```

Ma certo! Abbiamo usato “==” invece di “=”. Usciamo dal debugger con quit, correggiamo l'errore e rieseguiamo lo script:

```
C:\Work\Python>python lavoro.py  
Pippo ha 150 euro
```

Finalmente tutto funziona!

set_trace

pdb può però anche essere richiamato esplicitamente nel codice. Proviamo a modificare il metodo paga in questo modo:

```
def paga(self, compenso):  
    import pdb
```

```
pdb.set_trace()
self.capitale = self.capitale +
compenso
```

Abbiamo importato il modulo pdb e richiamato la sua funzione set_trace. Se ora eseguiamo lo script normalmente dalla riga di comando, l'esecuzione prosegue senza interruzioni fino al momento dell'incontro di set_trace: a quel punto ci troviamo nella situazione seguente, in maniera similare (ma più semplice da raggiungere) rispetto a quando abbiamo inserito un breakpoint a mano dal prompt di pdb:

```
> c:\work\python\lavoro.py(13)paga()
-> self.capitale = self.capitale +
compenso
(Pdb)
```

Da qui possiamo comportarci come nell'esempio precedente.

Creare un'applicazione “standalone”

Il problema di consegnare all'utente un'applicazione standalone (letteralmente "che funziona da sola") è abbastanza sentito quando si tratta di script che necessitano di un interprete e Python non fa eccezioni. La piattaforma dove normalmente viene richiesto di consegnare un programma standalone è Windows, per cui proveremo a creare un eseguibile Windows per il gioco Full House che abbiamo scritto nel Capitolo 11.

Tra le diverse opzioni che abbiamo a disposizione per creare eseguibili a partire da script Python scegliamo py2exe che possiamo scaricare dal sito:

<http://www.py2exe.org/>

Una volta ottenuta una distribuzione eseguibile di Full House realizzeremo un programma di installazione grazie a Inno Setup, disponibile all'indirizzo:

<http://www.jrsoftware.org/>

È forse inutile sottolinearlo, ma lo facciamo lo stesso: sono entrambi programmi free e Open Source.

py2exe

L'installazione di py2exe è semplicissima e con uno sguardo alle Figure 13.6, 13.7 e 13.8 possiamo rendercene conto.

Completata rapidamente l'installazione, possiamo già produrre lo script di configurazione per la creazione della nostra versione eseguibile di Full House. Lo script si chiama setup.py, deve essere creato nella directory dei sorgenti di Full House e deve contenere solo le seguenti righe di codice:

Setup

This Wizard will install py2exe on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

This package is a distutils extension to build standalone Windows executable programs from Python scripts.

Author: Thomas Heller

Author_email: theller@python.net

Description: Build standalone executables for Windows

Maintainer: Jimmy Retzlaff

Maintainer_email: jimmy@retzlaff.com

Name: py2exe

Url: <http://www.py2exe.org/>

Version: 0.6.5

Built Tue Jun 20 00:38:10 2006 with distutils-2.4.0

< Back

Next >

Cancel

Figura 13.6 La prima schermata dell'installazione di py2exe.

Setup



Python 2.5 is required for this package. Select installation to use:

Python Version 2.5 (found in registry)

PYTHON
Powered

Python Directory: C:\Python25\

Installation Directory: C:\Python25\Lib\site-packages\

< Back

Next >

Cancel

Figura 13.7 Un'ulteriore schermata di installazione di py2exe.

Setup



Postinstall script finished.
Click the Finish button to exit the Setup wizard.

py2exe is now installed on your machine.

There are some samples in the 'samples' subdirectory.

< Back

Finish

Cancel

Figura 13.8 L'ultima schermata dell'installazione di py2exe.

```
from distutils.core import setup  
import py2exe  
setup(windows=['fullhouse.py'])
```

Per conoscere le opzioni a disposizione di py2exe possiamo utilizzare il comando help da IDLE. Nella Figura 13.9 vediamo la prima parte dell'help in linea. Dall'estensione della barra di scorrimento indicata dalla freccia possiamo

immaginare quante informazioni e opzioni abbiamo a disposizione per risolvere ogni eventuale problema di creazione dei nostri eseguibili.

Proviamo ora a creare la distribuzione eseguibile digitando dalla riga di comando:

```
python setup.py py2exe
```

Non ci spaventiamo per il lunghissimo output che seguirà alla pressione del tasto INVIO. Nelle Figure 13.10 e 13.11 vediamo l'inizio e la fine di questo lungo elenco di messaggi; in mezzo ci sono circa altre 200 righe di output.

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 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 1.2

```
>>> import py2exe
>>> help(py2exe)
Help on package py2exe:
```

NAME

py2exe - builds windows executables from Python scripts

FILE

c:\python25\lib\site-packages\py2exe__init__.py

DESCRIPTION

New keywords for distutils' setup function specify what to build:

console
 list of scripts to convert into console exes

windows
 list of scripts to convert into gui exes

service
 list of module names containing win32 service classes



[Ln: 8 Col: 0]

Figura 13.9 Le opzioni disponibili in py2exe.

```
C:\WINNT\system32\cmd.exe
C:\Work\python\fullhouse>python setup.py py2exe
running py2exe
*** searching for required modules ***
*** parsing results ***
creating python loader for extension 'wx._misc_'
creating python loader for extension 'unicodedata'
creating python loader for extension 'wx._windows_'
creating python loader for extension 'wx._core_'
creating python loader for extension 'wx._gdi_'
creating python loader for extension 'bz2'
creating python loader for extension 'wx._controls_'
*** finding dlls needed ***
*** create binaries ***
*** byte compile python files ***
skipping byte-compilation of C:\Python25\lib\StringIO.py to StringIO.pyc
skipping byte-compilation of C:\Python25\lib\UserDict.py to UserDict.pyc
skipping byte-compilation of C:\Python25\lib\atexit.py to atexit.pyc
skipping byte-compilation of C:\Python25\lib\base64.py to base64.pyc
skipping byte-compilation of C:\Python25\lib\codecs.py to codecs.pyc
skipping byte-compilation of C:\Python25\lib\copy.py to copy.pyc
skipping byte-compilation of C:\Python25\lib\copy_reg.py to copy_reg.pyc
skipping byte-compilation of C:\Python25\lib\encodings\__init__.py to encodings\__init__.pyc
skipping byte-compilation of C:\Python25\lib\encodings\aliases.py to encodings\aliases.pyc
```

Figura 13.10 L'inizio dell'esecuzione di py2exe.

```
C:\WINNT\system32\cmd.exe
*** binary dependencies ***
Your executable(s) also depend on these dlls which are not included,
you may or may not need to distribute them.

Make sure you have the license if you distribute any of them, and
make sure you don't distribute files belonging to the operating system.

ole32.dll - C:\WINNT\system32\ole32.dll
MSVCP71.dll - C:\Python25\MSVCP71.dll
COMCTL32.dll - C:\WINNT\system32\COMCTL32.dll
ADVAPI32.dll - C:\WINNT\system32\ADVAPI32.dll
USER32.dll - C:\WINNT\system32\USER32.dll
WSOCK32.dll - C:\WINNT\system32\WSOCK32.dll
OLEAUT32.dll - C:\WINNT\system32\OLEAUT32.dll
gdiplus.dll - C:\Python25\lib\site-packages\wx-2.7.2-msw-unicode\wx\gdiplus.d
11
WINMM.dll - C:\WINNT\system32\WINMM.dll
RPCRT4.dll - C:\WINNT\system32\RPCRT4.dll
comdlg32.dll - C:\WINNT\system32\comdlg32.dll
SHELL32.dll - C:\WINNT\system32\SHELL32.dll
GDI32.dll - C:\WINNT\system32\GDI32.dll
KERNEL32.dll - C:\WINNT\system32\KERNEL32.dll

C:\Work\python\fullhouse>
```

Figura 13.11 La fine dell'esecuzione di py2exe.

Se tutto è andato a buon fine, troveremo una directory dist dove sarà contenuta una quindicina di file, fra i quali fullhouse.exe. Proviamo a eseguirlo e vedremo la finestra di avvio del nostro gioco. Per poter eseguire il gioco è sufficiente un file Zip contenente tutti i file della directory dist .

InnoSetup

Consegnare un file Zip con qualche file oltre all'eseguibile è già un passo avanti rispetto a chiedere di installare l'interprete Python, wxPython, copiare i tre script di Full House in una directory e lanciare a mano lo script principale. Ma i programmi di installazione professionali sono un'altra cosa.

Il programma più utilizzato per creare le installazioni di programmi scritti in Python per Windows è Inno Setup. Anche in questo caso l'installazione è estremamente lineare. Nelle

Figure da 13.12 a 13.15 vediamo una parte delle schermate che ci accompagneranno nel processo di installazione del programma.

Completata l'installazione possiamo lanciare il programma vero e proprio.

L'interfaccia è molto semplice e possiamo utilizzare il Wizard (procedura guidata) per creare il nostro personale programma di installazione di Full House. Per eseguire il Wizard dal menù scegliamo file / new. Manteniamo tutte le scelte di default tranne le scelte mostrate nelle Figure da 13.16 a 13.20.

Al termine del Wizard ci verrà chiesto di salvare lo script: salviamolo pure in fullhouse.iss, così, in un secondo momento, potremo modificare la nostra installazione senza ripetere tutti i passi svolti finora. Terminato il salvataggio dello script parte la fase finale di compilazione, che crea un file setup.exe nella directory Output.

Benvenuti nel programma di installazione di Inno Setup

Inno Setup versione 5.1.8 sarà installato sul computer.

Si consiglia di chiudere tutte le applicazioni attive prima di procedere.

Premere Avanti per continuare, o Annulla per uscire.



Avanti >

Annulla

Figura 13.12 La prima schermata dell'installazione di Inno Setup.

**Selezione della cartella di installazione**

Dove si vuole installare Inno Setup?



Inno Setup sarà installato nella seguente cartella.

Per continuare, premere Avanti. Per scegliere un'altra cartella, premere Sfoglia.

Sono richiesti almeno 3.4 MB di spazio sul disco.

Figura 13.13 Un'altra schermata dell'installazione di Inno Setup.

Installazione di Inno Setup

Pronto per l'installazione

Il programma di installazione è pronto per iniziare l'installazione di Inno Setup sul computer.



Premere **Installa** per continuare con l'installazione, o **Indietro** per rivedere o modificare le impostazioni.

Cartella di installazione:

C:\Program Files\Inno Setup 5

Cartella del menu Avvio/Start:

Inno Setup 5

Processi aggiornati:

Associa l'estensione .iss a Inno Setup

< Indietro

Installa

Annulla

Figura 13.14 Una schermata successiva.



Completamento dell'installazione di Inno Setup

L'installazione di Inno Setup è stata completata con successo.
L'applicazione può essere eseguita selezionando le relative icone.

Premere Fine per uscire dall'installazione.

Avvia Inno Setup



Fine

Figura 13.15 La fine dell'installazione di Inno Setup.

**Application Information**

Please specify some basic information about your application.

**Application name:****Application name including version:****Application publisher:****Application website:**

bold = required

< Back

Next >

Cancel

Figura 13.16 Il nome e alcuni dati descrittivi di Full House.

**Application Folder**

Please specify folder information about your application.

**Application destination base folder:**

[Program Files folder]

Application folder name:

Full House

Allow user to change the application folder

Other:

The application doesn't need a folder

bold = required

< Back

Next >

Cancel

Figura 13.17 Il nome della directory in cui installare Full House.

**Application Files**

Please specify the files that are part of your application.

**Application main executable file:** C:\Work\python\fullhouse\dist\fullhouse.exe Browse... Allow user to start the application after Setup has finished The application doesn't have a main executable file**Other application files:**

- C:\Work\python\fullhouse\dist_controls_.pyd
- C:\Work\python\fullhouse\dist_core_.pyd
- C:\Work\python\fullhouse\dist_gdi_.pyd
- C:\Work\python\fullhouse\dist_misc_.pyd
- C:\Work\python\fullhouse\dist_windows_.pyd
- C:\Work\python\fullhouse\dist\bz2.pyd
- C:\Work\python\fullhouse\dist\library.zip
- C:\Work\python\fullhouse\dist\MSVCR71.dll
- C:\Work\python\fullhouse\dist\nthon25.dll

 Add file(s)... Add folder... Edit... Remove**bold** = required < Back Next > Cancel

Figura 13.18 I file da includere nell'installazione.

**Application Icons**

Please specify which icons should be created for your application.

**Application Start Menu folder name:**

Full House

- Allow user to change the Start Menu folder name
- Allow user to disable Start Menu folder creation
- Create an Internet shortcut in the Start Menu folder
- Create an Uninstall icon in the Start Menu folder

Other main executable icons:

- Allow user to create a desktop icon
- Allow user to create a Quick Launch icon

bold = required

< Back

Next >

Cancel

Figura 13.19 La voce di menu in cui Inno Setup dovrà inserire Full House.

**Setup Languages**

Please specify which Setup languages should be included.

**Languages:**

- English
- Basque
- Brazilian Portuguese
- Catalan
- Czech
- Danish
- Dutch
- Finnish
- French
- German
- Hungarian
- Italian
- Norwegian

[Select all](#)[Deselect all](#)

bold = required

< Back

Next >

Cancel

Figura 13.20 Le lingue a disposizione per l'installazione.

Se inviamo questo unico file a qualcuno che non ha Python sul proprio sistema, potrà installarlo e divertirsi con Full House senza bisogno di altri file o altri programmi.

Nella Figura 13.21 vediamo in tutto il suo

splendore il programma di installazione di Full House.



Figura 13.21 Il programma di installazione di Full House.

CAPITOLO 14

Qualche link utile

Un capitolo che presenti un elenco di siti diventa obsoleto praticamente nel momento stesso in cui viene compilato. Ciononostante pensiamo che un breve elenco possa essere utile per capire cosa e come possiamo cercare quanto ci serve sul nostro amato linguaggio Python. Tralasciamo Google perchè lo diamo per scontato...

Python.org

Il padre di tutti i siti che hanno a che fare con Python:

<http://www.python.org>

È la sua casa virtuale e vi troveremo sempre le notizie aggiornate sul passato, presente e futuro di

Python. Contiene anche la pagina personale di Guido Van Rossum:

<http://www.python.org/~guido/>

Il primo segno

Forse non è un indirizzo utile, ma sicuramente è interessante:

<http://groups.google.com/group/comp.sys.cs>

È quasi una reliquia, visto che si tratta del primo messaggio scritto da Van Rossum nel febbraio del 1991, quando decise di distribuire liberamente nel mondo la sua creatura.

Comp.lang.python

Il gruppo di discussione dedicato a Python:

<http://groups.google.com/group/comp.lang.python>

Qualsiasi problema dovessimo incontrare nell'uso

di Python, è estremamente improbabile che non sia stato sperimentato e risolto da qualcun altro. E se esiste una risposta, quasi certamente si trova in questo gruppo. Questo gruppo di discussione è anche una fonte importantissima per cercare le risposte ai quesiti più difficili

Google code search

Il motore di ricerca di Google per il codice sorgente free:

<http://www.google.com/codesearch>

Digitando lang:python e una stringa qualsiasi, possiamo cercare tutti i programmi Python disponibili che contengano tale stringa. Può essere un mezzo molto rapido ed efficiente per vedere l'uso nel mondo reale di alcune istruzioni.

ActivePython

La distribuzione Python di ActiveState:

<http://aspn.activestate.com/ASPN/Python>

Questo indirizzo è molto utile, non tanto per la sua distribuzione di Python (a nostro parere è sicuramente preferibile usare quella ufficiale), quanto per i suggerimenti, le soluzioni e tutto quanto possiamo trovare nel Cookbook (libro di ricette).

Planet Python

Le news di Planet Python:

<http://www.planetpython.org/>

Se vogliamo avere notizie aggiornate su Python questo link fa per noi.

PythonWare

I link giornalieri di PythonWare:

<http://www.pythongware.com/daily/index.htm>

Se, oltre alle notizie, vogliamo sapere tutto dei pacchetti appena usciti o appena nati o anche solo ideati, questo è il sito dove cercare.

Django

Il web framework scritto in Python:

<http://www.djangoproject.com>

Questo link è un bonus: se sviluppare siti web fategli una visita...

Table of Contents

Introduzione

Dedica

Capitolo 1

Cos'è Python?

Interpretato,
interattivo
Orientato agli
oggetti
Moduli
Eccezioni
Tipizzazione
dinamica
Tipi di dati di alto
livello
Sintassi
estremamente chiara
Estendibile in C e in
C++
Usabile come

linguaggio di
configurazione
Portabile

Capitolo 2

Come installare Python

Windows

Linux

Mac OS X

Compilazione dal
codice sorgente

Capitolo 3

Pronti? Via!

IDLE

Riga di comando

Conclusioni

Capitolo 4

I tipi di dati

Il comando dir

Il comando type

Le liste

Le stringhe

Le tuple

Gli insiemi

[I dizionari](#)
[I numeri](#)
[True, False e None](#)
[Conversioni](#)

[Capitolo 5](#)

[La sintassi](#)

[L'indentazione](#)
[L'istruzione if](#)
[L'istruzione for](#)
[L'istruzione while](#)
[Le istruzioni break e continue](#)
[L'istruzione pass](#)

[Capitolo 6](#)

[Le eccezioni](#)

[Le istruzioni try ed except](#)
[L'istruzione raise](#)
[Le istruzioni finally ed else](#)
[La funzione exc_info](#)

[Capitolo 7](#)

[Le funzioni](#)

Definire una funzione
Valori restituiti
Il passaggio di argomenti
Varie ed eventuali

Capitolo 8

Le classi

Le classi e le istanze
I metodi
L'ereditarietà
I metodi privati
I metodi speciali

Capitolo 9

Input e output

Il comando print
L'operatore %
I file
I toolkit grafici

Capitolo 10

I moduli

sys
os

zipfile, tarfile, zlib,
gzip, bz2
shelve, pickle
smtplib, urllib,
ftplib
threading
NumPY e
SQLAlchemy

Capitolo 11

Un'applicazione completa

Il gioco “Full House”
L'aspetto grafico finale
Il “motore” del gioco
I problemi
L'interfaccia grafica
Una piattaforma diversa

Capitolo 12

PyWin32: le estensioni per Windows Microsoft Word

Internet Explorer
Singola istanza
ctypes
Catturare una combinazione di tasti
Excel
L'area delle notifiche
Applicativi
“vecchio stampo”
Esempi di PyWin32

Capitolo 13

Argomenti avanzati

Introspezione
Debugging
Creare
un'applicazione
“standalone”

Capitolo 14

Qualche link utile

Python.org
Il primo segno

[Comp.lang.python](#)
[Google code search](#)
[ActivePython](#)
[Planet Python](#)
[PythonWare](#)
[Django](#)