

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:

<http://sourceforge.net/projects/pywin32/>

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 con-

ta 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(\
        "Word.Application")
    solo_alfanumerici = re.compile(\
        r"^[^a-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()
```



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 764.

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(\
    "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

```

C:\WINDOWS\system32\cmd.exe

C:\Work\python>python conta_word.py docs\*.
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`.

diversi dalle lettere, dalle cifre e dallo spazio; possiamo utilizzarla per sostituire le occorrente trovate:

```
solo_alfanumerici = re.compile(\
    r"^[^a-z0-9 ]")
```

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(\
    '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`.



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



Figura 12.2 Ecco cosa accade se eseguiamo `ie.py`.

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/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*.



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)
```

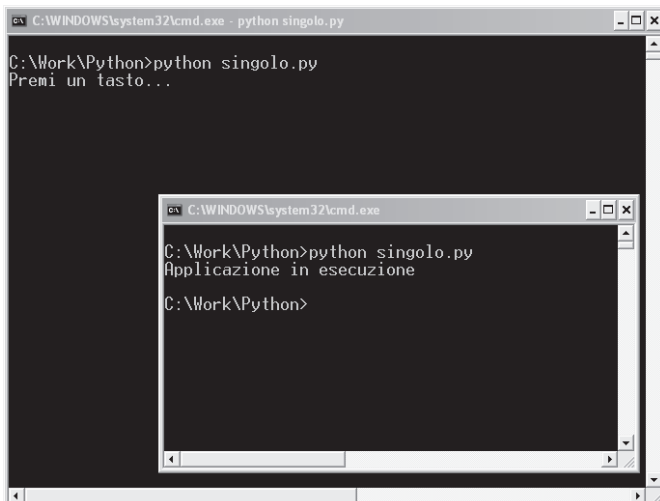


Figura 12.3 Uno script che non può essere in esecuzione due volte.

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.



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.



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()
```



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.GetSystemDirectoryA(\
    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.

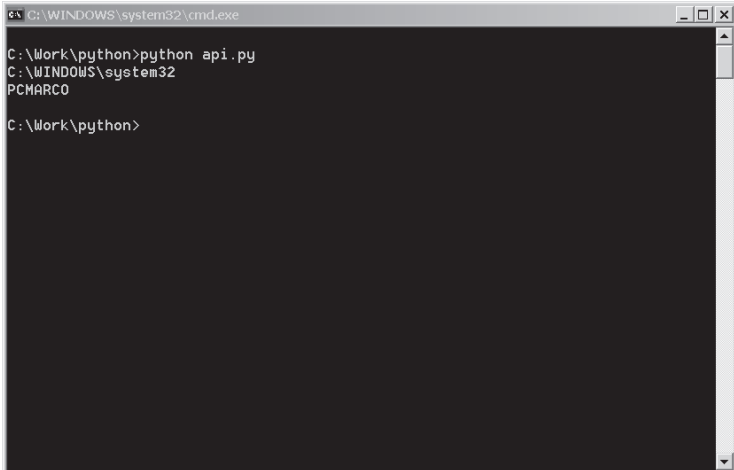
L'API `GetComputerNameA` restituisce il nome del computer sul quale viene eseguita (nel nostro caso `PCMARCO`) e accetta come parametri il puntatore a un buffer di tipo stringa (come il precedente) e un puntatore a una variabile che indica la lunghezza del buffer stesso (creato con `byref` e `c_int64`). 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.

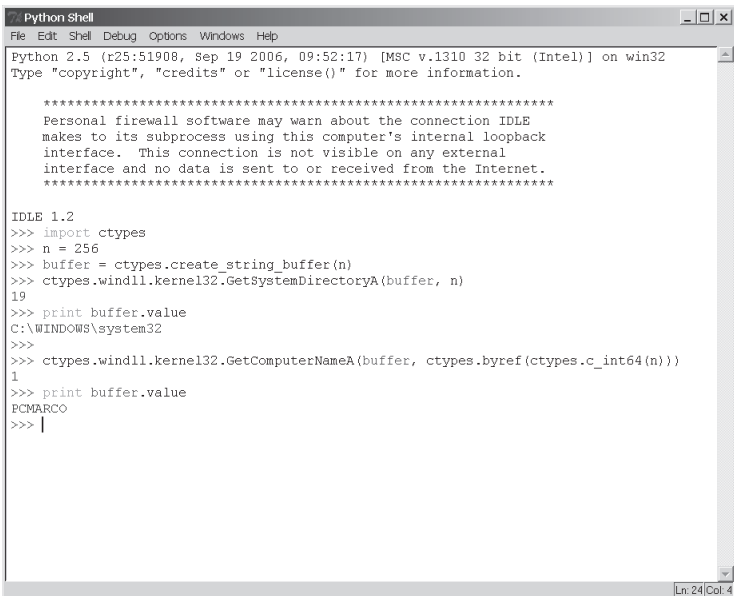


```
C:\WINDOWS\system32\cmd.exe

C:\Work\python>python api.py
C:\WINDOWS\system32
PCMARCO

C:\Work\python>
```

Figura 12.4 L'output di api.py.



```
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 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
>>> |
```

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),
```



```

        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("D8")\
        .Formula
excel.ActiveWorkbook.ActiveSheet.Range("D9")\
        .Formula = "=SUM(D5:D6)*1.2"
excel.ActiveWorkbook.ActiveSheet.Range("D8")\
        .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("D9")\
        .Formula = "=SUM(D5:D6)*1.2"

```

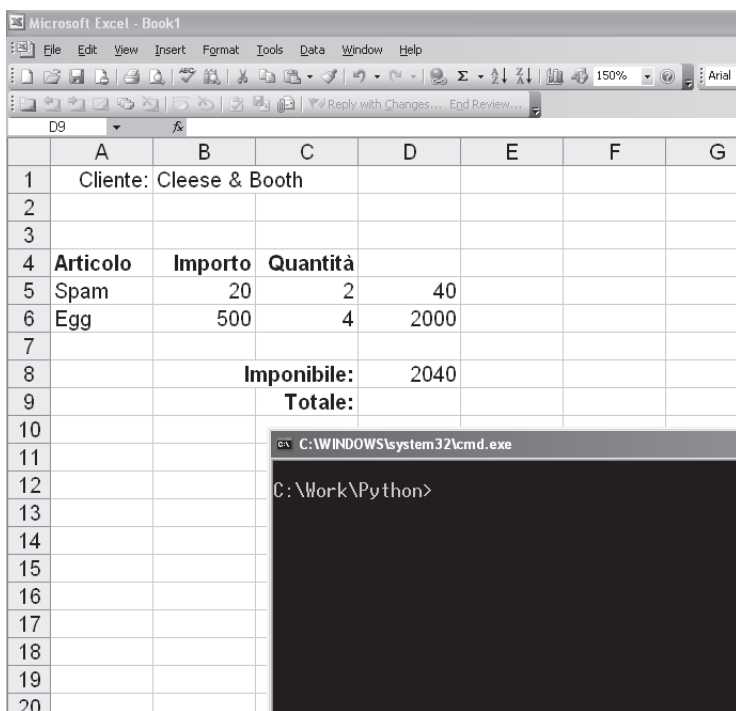


Figura 12.7 La cella D9 del foglio Excel è vuota.

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.

The screenshot shows a Microsoft Excel window titled 'Microsoft Excel - Book1'. The spreadsheet has columns A through G and rows 1 through 17. The data is as follows:

	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							

Overlaid on the bottom right of the Excel window is a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. It shows the following command and output:

```
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.visibile:
        self.nascondi()
    id_notification = (self.hwnd, 0, flags,
                      win32con.WM_USER+23,
                      self.icon, self.suggerimento)
    win32gui.Shell_NotifyIcon(win32gui.NIM_ADD,
                              id_notification)
    self.visibile = True
def nascondi(self):
    if self.visibile:
        id_notification = (self.hwnd, 0)
        win32gui.Shell_NotifyIcon(
            win32gui.NIM_DELETE,
            id_notification)
        self.visibile = 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_LBUTTONDBLCLK:
        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.

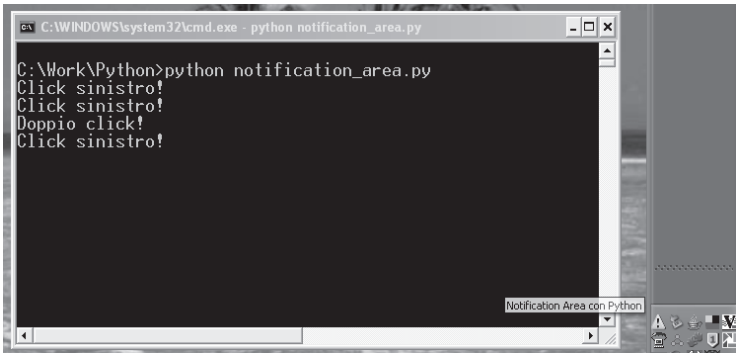


Figura 12.9 L'icona nell'area delle notifiche e il suggerimento visualizzato.

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_RBUTTONDOWN:
        print "Click destro!"
        win32gui.PostQuitMessage(0)
    elif lparam == win32con.WM_LBUTTONDBLCLK:
        print "Doppio click!"

```

Infine, dopo aver creato un'istanza della classe NotificationArea, avviamo il ciclo di gestione dei messaggi di Windows:

```

NotificationArea()
win32gui.PumpMessages()

```



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: se nel vostro sistema non esiste l'applicazione `calc.exe`, il ciclo precedente non avrà mai fine.

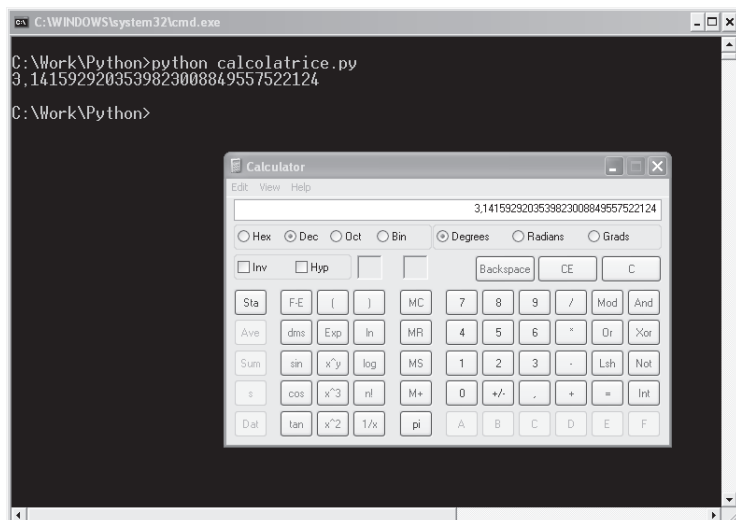


Figura 12.10 La simulazione della pressione di tasti nella Calcolatrice di Windows.

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=")
```



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.

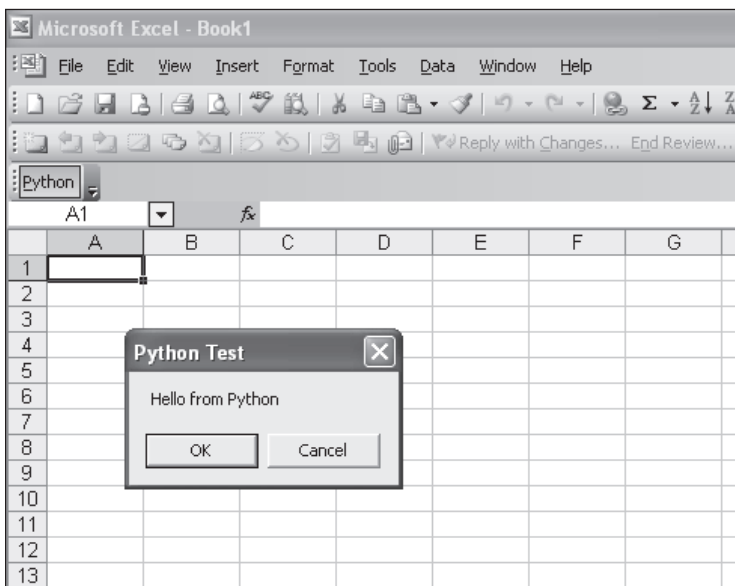


Figura 12.11 Aggiunta di un pulsante nella barra degli strumenti di Excel.