

Python Tips & Tricks v.1

by Italian Python User Group

<http://www.italianpug.org>

11 luglio 2005

Indice

1	Prefazione	5
2	Tipi di dati e oggetti	7
2.1	Un aiuto rapido: help() & dir()	7
2.2	Il tipo di dato char (carattere)	8
2.3	Numeri e valori booleani	9
2.3.1	Cosa é 0, cosa 1	9
2.3.2	Potenza di un numero	10
2.3.3	Come creare un numero random	11
2.3.4	Approssimare un numero	12
2.3.5	True o not(True)	14
2.3.6	Operazioni con i numeri complessi	17
2.4	Stringhe	17
2.4.1	Template di stringhe	17
2.4.2	Sostituire una stringa con un'altra	18
2.4.3	Pulizia stringhe	19
2.4.4	Invertire una stringa	20
2.5	Liste	21
2.5.1	Iterare all'incontrario	21
2.6	Funzioni	22
2.6.1	Creare generatori con yield	22
2.6.2	Funzioni semplici al volo: Lambdas	23
2.7	Classi, variabili e moduli	25
2.7.1	Variabili globali	25
2.7.2	Accesso dinamico al nome delle variabili	26
2.7.3	Inheritance dinamica di classi	27
2.7.4	Importare moduli dinamicamente	29
2.7.5	Estendere l'assegnazione, lettura e cancellazione di un oggetto	30

3	Console e I/O	32
3.1	Come togliere la newline dalla funzione print()	32
3.2	Come catturare l'output di un comando avviato con python	33
3.3	Misurare un file "a crudo"	34
4	Sockets	35
4.1	Stream sockets	35
4.1.1	Listen su piú porte e un unico obiettivo	35
4.1.2	Indirizzo in uso e socket chiusa	37
4.1.3	IPC con le socket in dominio UNIX	37
4.2	World Wide Web	39
4.2.1	Webserver senza files	39
4.2.2	Ricavare il nome di un webserver al volo	42
4.2.3	Download con urllib	43
5	DataBases	45
5.1	Accesso immediato e niente SQL	45
6	Interfaccia grafica	47
6.1	GTK	47
6.1.1	Utilizzo dei threads	47
6.1.2	Bottone con immagine	49
6.1.3	Come compilare le pygtk con py2exe	50
7	Audio	52
7.1	Beep dall'altoparlante	52
8	Compressione	54
8.1	Comprimere con bz2	54
8.2	Comprimere con zlib	55
9	Crittografia	57
9.1	Uso della libreria Crypto	57
10	Programmazione su Windows	60
10.1	Arrestare Windows XP	60
10.2	No more shell	62
11	Programmazione su UNIX	63
11.1	Demonizzazione	63

12 Ottimizzazione, statistica e debug	65
12.1 Libreria profile	65
12.2 Controllare ogni operazione	66
13 Decorazioni e vari trucchetti utili	68
13.1 Aggiungere e togliere elementi da un list con << e >>	68
13.2 Da stringa a comando py: eval()	70
13.3 Coding delle funzioni	71
13.4 For, non fermiamoci alle apparenze	73
13.5 ConfigParser per i propri config files	74
13.6 Come identificare il Sistema Operativo	76
13.7 Commenti su piú righe	77
13.8 Questione di switch	77
14 Note finali	79
15 Autori e contribuenti	80
16 Indice analitico	81
A The GNU General Public License	84

Capitolo 1

Prefazione

License

This work is free software; you can redistribute it and/or modify it under the terms of the **GNU General Public License** as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This work is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You can find a copy of the GNU General Public License version 2 at Appendix [A](#).

Introduzione al CookBook

Questo cookbook fornisce tips & tricks di varia natura, forniti direttamente dagli utenti **IPUG** (*Italian Python User Group*).

Molti sono i motivi che ci hanno spinto a creare questo genere di documento, a partire dalla forza di collaborazione del nostro gruppo alla passione e alla voglia di far contribuire in un unico progetto ogni persona della community italiana sul Python.

Molte volte i programmatori hanno bisogno di effettuare operazioni semplici e veloci che facciano al proprio caso, in modo da aggirare e rendere piú chiaro una parte di codice complessa.

Ci si trova quindi in condizioni di dover chiedere e ricercare nella rete anche piccolezze che però possono salvare il programmatore da varie complicazioni.

Alché si é deciso di istituire questo progetto, con il quale ogni membro puó suggerire dei tips e renderne partecipe l'intera community. L'intento quindi é quello di provvedere ad uno scambio di idee in modo da fornire un aiuto reciproco tra gli sviluppatori.

Ognuno puó quindi contribuire alla realizzazione di questo progetto e contribuirne alla sua crescita.

Ciò di cui si ha bisogno é semplicemente di qualche minuto di tempo per scrivere qualche riga di codice e una spiegazione semplice e chiara di un qualsiasi problema incontrato e risolto nei propri programmi. **L'importante** é non dare per scontato che i propri tips siano inutili, poiché i nuovi programmatori Python hanno bisogno di tutti i punti di riferimento possibili per risolvere anche piccoli errori e di conseguenza sciogliere dubbi e aumentare di un pizzico l'esperienza.

Gli articoli possono essere piccoli e scontati ma a volte sono miniere per chi ha bisogno di uno spunto veloce e semplice su come effettuare alcune operazioni.

Ora vi lascio alla lettura del cookbook, e spero con questa piccola premessa di avervi indotto a partecipare al progetto.

Capitolo 2

Tipi di dati e oggetti

2.1 Un aiuto rapido: `help()` & `dir()`

by The_One

Cosa diavolo fa quella data funzione?

Come si usa?

Qual'è la funzione di quel certo metodo?

In tutti questi casi Python ci viene in aiuto con dei semplici strumenti che fanno al caso nostro. Provate a digitare *help* dalla riga di comando:

```
>>> help
Type help() for interactive help, or help(object) for help about
object.
```

L'output dice tutto. Con *help* potete richiedere aiuto su qualsiasi funzione/classe digitandone semplicemente il nome!

`help()` si basa sulla variabile `__doc__`, che restituisce una documentazione breve su di una funzione/classe specifica:

```
>>> import math
>>> print math.pow.__doc__
pow(x,y)
```

Return `x**y` (x to the power of y).

Restituisce in pratica la stringa di documentazione della funzione stessa, visualizzabile anche con `help(math.pow)`.

Prendiamo in considerazione la seguente funzione `f()`:

```
>> def f():
>>     "funzione inutile"
>>     pass
>> print f.__doc__
funzione inutile
```

Come vedete la stringa "funzione inutile" inserita all'inizio della funzione viene usata come **docstring** in `__doc__`

Ora la funzione `help()` ha anche altri pregi, ad es. quello di poter visualizzare la documentazione di un modulo senza importarlo, racchiudendo il nome del modulo tra gli apici: `help('math')`.

Infine, mettiamo che abbiate importato un modulo, chissà, mettiamo `math`, e ipotizziamo che non ricordiate quella data funzione di `math` che calcola il logaritmo di un numero, e supponiamo pure che non vi vada di cercare nella documentazione (sfaticati)... Bene se da riga di comando digitate:

```
>> import math
>> dir(math)
```

Vi verranno elencate tutte le funzioni del modulo... mica male eh?

2.2 Il tipo di dato char (carattere)

by Lethalman

Vi sarete chiesti come mai in Python non vi è il tipo di dato built-in `char`! A volte può servire nel momento in cui si ha un carattere e lo si vuole aumentare, ad es. portare 's' a 't' facendo 's'+1, ma questo non è possibile. Bisogna ricorrere alle seguenti operazioni:

```
>> char = 't'
>> print chr(ord(char)+1)
```

Con **ord** si trasforma il carattere nel suo corrispettivo numero, lo si aumenta,

ed in seguito lo si trasforma nuovamente in carattere con `chr`. Una classe potrebbe aiutarci nell'intento:

```
>>> class char:
>>>     def __init__(self, c):
>>>         if type(c) is int: self.__char = c
>>>         else: self.__char = ord(c)
>>>     def __add__(self, i):
>>>         return char(self.__char+i)
>>>     def __sub__(self, i):
>>>         return char(self.__char-i)
>>>     def __str__(self, i):
>>>         return chr(self.__char)
>>>     def __repr__(self, i):
>>>         return "'%c'" % self.__char
>>>     def __int__(self, i):
>>>         return self.__char
```

All'inizio viene salvato il carattere in modo numerico, con `__add__` e `__sub__` specifichiamo l'operazione da eseguire rispettivamente in caso di `+` o `-`. Con `__str__` viene ritornata la stringa in caso si voglia usare la funzione `str()`, con `__int__` il numero del carattere usando `int()` e `__repr__` che ritorna normalmente il valore in stringa.

Avendo la suddetta base, possiamo effettuare qualche test con il nostro carattere:

```
>>> c = char('s')
>>> print c+1, c-1, c-2+3, int(c)
t r t 115
```

2.3 Numeri e valori booleani

2.3.1 Cosa é 0, cosa 1

by Fred1

A dispetto del titolo (che é chiaro indice della mia salute mentale), voglio spiegare cosa, in Python, é vero o falso... Tutto ciò che é **vuoto**, é falso.

Ma come fare a verificarlo? Si utilizza la funzione built-in `bool` (che non si legge 'boooooooooool' con una quantità pressoché infinita di 'o'):

```
>> bool(' ')
True
```

Notate anche che ' ' non é vuota, perché é comunque uno spazio. Quindi ecco un esempio *False*:

```
>> bool([])
False
```

Visto? La lista vuota é falsa. Vorrei farvi notare che *False* é uguale a 0... Perché nel codice binario (l'unico linguaggio che il computer comprende ed elabora) si utilizzano due simboli: 1 e 0; 1 sta per 'acceso', mentre '0' per 'spento' (corso di informatica per brocchi?)... Quindi:

```
>> bool(0)
False
```

Ed é per questo che per creare un loop infinito (che é il top, se volete la CPU un tantino occupata - quanto basta per farvi perdere tutto il lavoro non salvato...) si usa:

```
>> while 1: print 'a'
```

Che tradotto é: *finché 1 stampa la stringa 'a'...* ovvero: *finché 1 risulta vero stampa la stringa 'a'*. Infatti il codice sopra potrebbe essere scritto cosí:

```
>> while 1 == True: print 'a'
```

(Ok, nel caso in cui abbiate disgraziatamente digitato il codice INUTILE riportato sopra citato dovete premere contemporaneamente ctrl + c per bloccare immediatamente il ciclo). Quindi se dovete sapere se la variabile, lista, tupla, ecc. non é vuota, basta fare: *[if—while] var*. Comodo, vero? ;)

2.3.2 Potenza di un numero

by **The_One**

Per chi muove i primi passi nulla é scontato...

Chi di voi non si é mai trovato di fronte al dilemma:

”COME SI CALCOLA LA POTENZA ENNESIMA DI UN NUMERO?”

Sebbene Python nel modulo **math** implementi una funzione dedicata:

```
double math.pow(b, exp)
```

con le dovute restrizioni matematiche per la base (b) e l'esponente (exp), esiste un mezzo molto piú rapido che non va a scomodare moduli particolari:

```
b**exp
```

dove b sta per la base e exp per l'esponente. Prendiamo in considerazione il seguente esempio:

```
>> import math
>> math.pow(2,4)
16.0
>> 2**4
16
```

Da notare che in questo caso, a differenza di `math.pow`, abbiamo un numero intero e non un *double*, con tutti i vantaggi e le conseguenze derivanti.

L'operatore `**` restituisce numeri reali solo se necessario:

```
>> (1.3)**4
2.8561000000000005
```

2.3.3 Come creare un numero random

by Fred1

In Python, tra gli infiniti moduli, ne esiste uno con un mucchio di funzioni (la maggior parte delle quali incomprensibili dall'uomo medio :)) per creare numeri a caso: **random**. Si puó decidere di utilizzare il metodo `random()`, che crea un float a caso fra 0 e 1:

```
>> import random
>> random.random()
0.86693578658824788
```

Io direi che é totalmente inutile, no?

Alcuni programmi, come i giochi, richiedono dei numeri random interi e tra

un inizio e una fine..... Facciamo un esempio: vogliamo che un mostro salti fuori a n secondi dall'inizio del gioco. Che senso avrebbe farlo spuntare a 0.86693578658824788 secondi? Praticamente salterebbe fuori sempre allo stesso momento (a meno che voi non riusciate a distinguere i decimi/centesimi di secondo :))...

Quindi é meglio usare numeri interi.

Ci sono due modi per creare numeri interi a caso: *randint()* e *randrange()*.

```
>> import random
>> random.randint(0,10)
9
```

randint() permette di creare un intero a caso includendo l'inizio e la fine. *randrange()*, invece, da in output un intero appoggiandosi alla funzione built-in *range()*.

```
>> import random
>> random.randrange(10)
6
```

Notate che *randint()* pretende sia l'inizio che la fine, mentre *randrange()*, proprio come *range()*, se presente un solo numero, immette come inizio 0.

2.3.4 Approssimare un numero

by Luigi Pantano aka bornFreeThinker

Questo semplice codice é nato dall'esigenza di approssimare in modo 'esatto' un numero in python ad una determinata cifra decimale. La funzione accetta 2 paramentri:

- il primo paramentro **num** é il valore numerico che si vuole approssimare;
- il secondo parametro **k** rappresenta il grado di approssimazione, che per default é stato imposto a 4.

Per meglio comprendere il codice mi sembra opportuno dare una rinfrenscata al concetto di approssimazione con un breve esempio, ovviamente daró per scontato che conosciate almeno il concetto di approssimazione.

Dato $x = 1.48794$ vogliamo approssimarlo alla 4 cifra decimale, ciò significa che tutte le crifre fino alla 3 saranno esatte, cioè non risentiranno del grado di approssimazione; invece la 4 cifra sará inesatta perché approssimata. Ergo

se volessi ottenere un numero che presenti fino alla 5 cifra decimale esatta dovrei approssimare alla 6 cifra decimale.

Per approssimare x alla 4 cifra decimale analizziamo prima di tutto il valore della 5 cifra decimale che nella fattispecie vale 4, essendo il valore inferiore a 5 il numero si tronca, ottenendo $x = 1.4879$.

Per approssimare x alla 3 cifra decimale analizziamo il valore della 4 cifra decimale che vale 9, essendo il valore maggiore a 5 il numero si approssima per eccesso ottenendo $x = 1.488$.

Il codice da me proposto contempla anche il caso in cui il valore da approssimare sia uguale a 5. Esempio: $x = 1.4565$ da approssimare alla 3 cifra decimale. La 4 cifra vale 5, in questo caso si procede contemplando la 3 cifra decimale, quindi se la 3 cifra decimale é dispari si approssima per eccesso, altresí per difetto.

```
>>> def approssima(num, k=4):
>>>     b = str(num - int(num))[2:2+1+k]
>>>     h = int(b[-1])
>>>     if h > 5 or (h == 5 and divmod(int(b[-2]), 2)[1] != 0):
>>>         incremento = (10 - h) * pow(10, -(k+1))
>>>         num = num + incremento
>>>     return str(num).split('.')[0] + '.' + str(num).split('.')[1][:k]
```

Spulciando tra le numerose librerie del python ho trovato la libreria **fpformat** che esegue esattamente la stessa operazione della funzione "approssima" con l'unica differenza che la funzione "approssima" contempla anche il caso in cui la cifra da approssimare é pari a 5.

Riporto un semplice esempio per capire l'uso e la differenza con la funzione "approssima":

```
>>> import fpformat
>>> fpformat.fix(1.345, 2)
'1.35'
>>> approssima(1.345, 2)
'1.34'
```

Se la precisione é ciò che vi interessa vi consiglio di utilizzare la funzione da me scritta anziché la libreria fpformat.

Un'altro metodo, recentemente scoperto dal nostro amico **FraFra**, permetterebbe di effettuare in un altro modo una approssimazione esatta, il tutto con una

sola riga di programma!

```
>> # Numero da arrotondare
>> numero = 12.3456789
>> # Utilizzo: arrotondiamo alla 3 cifra decimale
>> print '%.3f' % numero
12.346
```

Da questo tips si evincono principalmente due concetti importantissimi:

1. Nel mondo della programmazione l'unico limite per la risoluzione di un problema é la vostra fantasia.
2. Nel mondo della programmazione open source l'unico limite per la risoluzione di un problema é la fantasia della collettività!

2.3.5 True o not(True)

by Luigi Pantano aka bornFreeThinker

Leggendo l'illuminante tips "Cosa e' 0, cosa 1" di *Fred1* mi é venuto spontaneo scrivere questo tips che metterà in luce un aspetto poco noto del comportamento dell'istruzione condizionale **if**.

Tale comportamento che a breve illustreró, tecnicamente chiamato "short circuit" dei costrutti condizionali e di iterazione, é una importante caratteristica ereditata dal C che invece é assente in altri linguaggi di programmazione come ad esempio il Visual Basic che usa la valutazione "complete".

Questo tips illustrerá solo il particolare comportamento dell'istruzione **if**, per tale ragione daró per scontata la conoscenza delle operazioni booleane elementari.

Presentiamo da subito il codice che sará alla base del mio tips:

```
>> def a(k):
>>     print 'funzione A', 'return', k
>>     return k
>> def b(k):
>>     print 'funzione B', 'return', k
>>     return k
>> def c(k):
>>     print 'funzione C', 'return', k
>>     return k
>> if a(True) and b(True) and c(True): # riga PIPPO
```

```
>> print 'la condizione if risulta VERA'  
>> else:  
>> print 'la condizione if risulta FALSA'
```

Il codice in questione é semplice e non necessita grandi spiegazioni di sorta. Abbiamo tre funzioni identiche (per comoditá ne ho prese soltanto tre ma potevo prenderne molte di piú) che accettano come parametro un valore booleano e come valore di return ritornano il parametro accettato.

Quindi con il comando $a(True)$ ottengo come risultato True. Vi prego di non sottovalutare la banalitá del codice che una volta compreso nel suo reale utilizzo verranno generalizzati dei concetti particolarmente importanti.

Se eseguiamo l'esempio cosí come'é otteniamo giustamente il seguente output:

```
funzione A return True  
funzione B return True  
funzione C return True  
la condizione if risulta VERA
```

"Evviva abbiamo scoperto l'acqua calda!". Era un risultato perfettamente attendibile e prevedibile, sicuramente non siete rimasti stupiti di ció.

Modifichiamo la riga incriminata PIPPO con la seguente:

```
>> if a(False) and b(True) and c(True):
```

rieseguiamo il codice ed otteniamo come output:

```
funzione A return False  
la condizione if risulta FALSA
```

Notiamo subito che sia la funzione $b()$ che $c()$ non sono state eseguite, ma di ció non dovremo preoccuparci perché il comportamento del programma é perfettamente logico. La funzione if esegue il salto condizionale solo se la condizione risulta True.

Nel primo caso tutte e 3 le funzioni essendo vere ed essendo messe in relazione booleana con un *and* danno come risultato una condizione True.

Nel secondo caso invece la funzione $a()$ restituisce un valore False quindi essendo le tre funzioni correlate da un *and* e sapendo che *and* restituisce True **SOLO** quando tutti i termini di confronto sono veri, il programma non ha avuto bisogno di eseguire il resto delle funzioni $b()$ e $c()$ perché ai fini della verifica della condizione risultavano, in questo specifico caso, ininfluenti i loro valori.

Nota: nel caso in cui non fosse molto chiaro il concetto, consiglio innanzi tutto di "giocare" un pó con la riga PIPPO modificando i valori passati alle funzioni oppure cambiando il tipo di relazione da *and* a *or* (rimarrete stupiti dal risultato!) ed inoltre di proseguire con la lettura della guida.

Questo risultato ottenuto é eccezionale e mostra un comportamento intelligente da parte dell'interprete Python che evita di sprecare cicli macchina e quindi tempo nell'esecuzione di istruzioni non necessarie.

Adesso é arrivato il momento di generalizzare un concetto abbastanza fondamentale nella vita di ogni buon programmatore che passa gran parte della sua vita non a scrivere i programmi, ma ad eseguire debug e soprattutto ad ottimizzare la velocità di esecuzione del codice.

Supponiamo che le funzioni $a()$, $b()$, $c()$ siano molto piú complesse di quella dell'esempio e che siano strutturate in modo tale che alla fine diano un risultato dipendente dal parametro immesso ma che ciò non sia cosí banale come nell'esempio. Una maggiore complessità della funzione comporta maggiori cicli macchina per eseguirla, quindi un tempo di esecuzione piú lungo.

Supponiamo che le funzioni impieghino rispettivamente i tempi: $\mathbf{T_a}$, $\mathbf{T_b}$, $\mathbf{T_c}$ e che tale tempi siano posti in relazione $T_a > T_b > T_c$. Quindi la riga PIPPO affinché venga eseguita avrà un tempo sicuramente maggiore della somma dei tempi di esecuzione delle tre funzioni: $T_{pippo} > T_a + T_b + T_c$. Ho supposto che il tempo sia maggiore data la presenza dell'if e dell'and che sicuramente impiegheranno un loro tempo di esecuzione.

Nel primo caso da noi analizzato il tempo T_{pippo} era il piú alto avendo eseguito tutte e tre le funzioni invece nel secondo caso abbiamo ottenuto sicuramente un tempo inferiore rispetto a quanto preventivato, ma lo si può ridurre ancora con questo semplice trucco.

Sotto la precedente ipotesi $T_a > T_b > T_c$ per accorciare i tempi di esecuzione basterá modificare la riga PIPPO come segue:

```
>>> if c(valore_C) and b(valore_B) and a(valore_A):
```

in questo modo se la funzione $c()$ risultasse falsa otterrei il tempo T_{pippo} piú basso in assoluto!

Volendo procedere in un altro modo si potrebbe verificare quale funzione da piú velocemente un risultato falso e porla come prima funzione da analizzare.

Benvenuti nel mondo dell'ottimizzazione del codice, un mondo in cui l'unico

limite é' la vostra fantasia ed il vostro ingegno.

2.3.6 Operazioni con i numeri complessi

by Luigi Pantano aka bornFreeThinker

Ho trovato particolarmente interessante constatare che python puó effettuare conteggi sui numeri complessi in modo nativo, senza l'utilizzo di particolari accorgimenti.

Forse chi non ha affrontato studi universitari di tipo ingegneristico o matematico/fisico sconosce questo insieme numerico, ma vi assicuro che nella teoria circuitale, e non solo, é utilizzatissimo (http://it.wikipedia.org/wiki/Numero_complesso).

Comunque ecco come si crea in python un numero immaginario ed alcuni semplici operazioni algebriche:

```
>>> a = complex(1, -4)
>>> b = complex(-8, 5)
>>> a + b
(-7+1j)
>>> a * b
(12+37j)
>>> a.imag
-4.0
>>> a.real
1.0
```

2.4 Stringhe

2.4.1 Template di stringhe

by Lethalman

Vi é mai capitato di dover inserire dei valori in una stringa con il normale formatting delle stringhe builtin in Python 'formato' % (dati)?

C'é un'altro metodo, che in base alle proprie esigenze puó anche rivelarsi piú semplice, si tratta di string.Template:

```
>>> from string import Template
>>> s = Template('Ciao $nome')
```

```
>>> print s.substitute(nome='Luca')
Ciao Luca
```

Come vedete, si crea un'istanza `Template` col formato della stringa mettendo `$` davanti alle variabili da sostituire.

Ma cosa succede se attaccato al nome della variabile volessimo mettere qualcos'altro? In questo caso basta mettere agli estremi `{ e }`:

```
>>> from string import Template
>>> s = Template('Ciao ${nome}s')
>>> print s.substitute(nome='Luca')
Ciao Lucas
```

Per scrivere il carattere `$` senza prenderlo come template basta scrivere `$$`. In caso si usa un template inesistente, `substitute` dar  un errore per non aver trovato la variabile, in questo caso si pu  usare `safe_substitute`:

```
>>> from string import Template
>>> s = Template('Ciao $nome $cognome, hai in tasca $$100')
>>> print s.safe_substitute(nome='Luca')
Ciao Luca $cognome, hai in tasca $100
```

In questo caso non avevamo specificato `$cognome`, e nonostante tutto non ha dato errore ed ha sorvolato il templating.

2.4.2 Sostituire una stringa con un'altra

by Fred1

Good morning mister → *Good night mister*

In questo esempio ho sostituito *morning* con *night*. In python ci sono molteplici modi per farlo, ma il pi  semplice   `replace()`.

```
>>> print 'ciao ciao'.replace('ao', 'uf')
'ciuf ciuf'
```

Come vedete, `replace()` necessita di 2 argomenti, ma ce n'  uno opzionale: *maxsplit* (ovvero il numero massimo di sostituzioni). Eccone un esempio:

```
>>> print 'ciao ciao'.replace('ao', 'uf', 1)
'ciuf ciao'
```

Come possiamo vedere, anche se si riscontrano 2 possibili sostituzioni, ne viene effettuata solo 1.

Si possono concatenare anche piú `replace()`, visto che comunque restituiscono una stringa:

```
>> a = 'To be or not to be?'
>> print a.replace(' ', '_').replace('o', '0').replace('i', '|')
'T0_be_0r_n0t_t0_be?'
```

2.4.3 Pulizia stringhe

by Luigi Pantano aka bornFreeThinker

Essendo una persona estremamente sbadata dimentico spesso di inserire nel codice e nei commenti che posto anziché il carattere accentato da tastiera l'apostrofo per l'accentazione. Per tale ragione ho scritto questo semplicissimo codice che dimostrerà ai novizi del python la flessibilità che il linguaggio offre. Inoltre per completezza ho aggiunto del codice per ottenere lettere maiuscole dopo ogni punto.

Dopo aver aperto il file assegnamo il contenuto letto alla variabile *testo* e procediamo con la sostituzione delle lettere accentate. Per rendere il codice maggiormente leggibile ho preferito usare un dizionario nel quale ad ogni lettera accentata corrisponde il suo equivalente con apostrofo. Tramite il ciclo `for` e la funzione **replace** ottengo la ripetuta sostituzione di tutte le lettere interessate.

La seconda parte del codice non fa altro che "formattare" il testo. Dopo aver diviso il testo in paragrafi, ciò lo si ottiene splittato il testo ad ogni '.' (punto), tramite il ciclo `for` ad ogni paragrafo vengono eliminati gli spazi iniziali con la funzione **lstrip** e viene "capitalizzato il testo" tramite la funzione **capitalize** (la prima lettera del paragrafo diverrá maiuscola!). Infine ogni paragrafo viene aggiunto alla variabile *testo_formattato* seguito da un punto piú uno spazio. Scrittura del file e print del *testo_formattato* ed il gioco é fatto. Niente di piú semplice. :-D

```
>> def ordina(filename):
>>     f = open(filename)
>>     testo = f.read()
>>     f.close()
>>     # lettere da sostituire
```

```

>> d = {"é": "e'", "á": "a'", "ú": "u'", "ó": "o'"}
>> for carattere in d:
>>     testo = testo.replace(carattere, d[carattere])
>> # formatta il testo
>> testo_formattato = ''
>> paragrafi = testo.split('.')
>> for x in xrange(len(paragrafi)):
>>     testo_formattato += paragrafi[x].rstrip().capitalize() +
', '
>> f = open(filename, 'w')
>> f.write(testo_formattato)
>> f.close()
>> print testo_formattato
>> ordina('esempio_banale.txt')

```

2.4.4 Invertire una stringa

by Luigi Pantano aka bornFreeThinker

Spulciando in internet ho trovato questo simpatico trucchetto che permette di invertire il contenuto di una stringa.

Codice valido per tutte le versioni di Python:

```

>> str = 'Hello, world!'
>> revstr = str[::-1]
>> print revstr
!dlrow ,olleH

```

Codice specifico per la versione Python 2.4:

```

>> str = 'Hello, world!'
>> print ''.join(reversed(str))
!dlrow ,olleH

```

Nel caso usiate python 2.4 potrete usare indiscriminatamente l'uno o l'altro codice.

Ecco vi illustro un possibile uso di questo codice. Questo esempio permette di ottenere, dato un percorso, il nome del file:

```
>> fullpath = '/home/bornFreeThinker/Music/prova.txt'
>> print fullpath[::-1].split('/')[0][::-1]
prova.txt
```

Questo codice può sembrare un pó crittico inizialmente ma se lo osservate con attenzione é molto semplice.

Per chi ama invece divertirsi con le frasi bifronte (<http://it.wikipedia.org/wiki/Bifronte>) troverá grazioso il seguente esempio:

```
>> str = 'ANNA AMA BOB'
>> print str[::-1]
BOB AMA ANNA
```

2.5 Liste

2.5.1 Iterare all'incontrario

Una nuova feature in Python 2.4: una funzione che stampa all'incontrario una lista! Prima del 2.4, per stampare una lista all'incontrario, dovevamo fare:

```
>> a = ['a', 'b', 'c', 8, 'foo']
>> a[::-1]
['foo', 8, 'c', 'b', 'a']
```

Mentre adesso possiamo usare la funzione built-in **reversed()**, che agisce solo sulle iterazioni (con ovvio risparmio della memoria):

```
>> a = ['a', 'b', 'c', 8, 'foo']
>> print reversed(a)
<listreverseiterator object at 0x0117BF30>
>> for i in reversed(a): print i,
foo 8 c b a
>> b = [f for f in reversed(a)]
>> print b
['foo', 8, 'c', 'b', 'a']
```

2.6 Funzioni

2.6.1 Creare generatori con yield

by Lethalman

Nelle ultime versioni del Python si stanno sempre di piú affermando i generatori che permettono di avere maggiori performance e controllo nelle proprie funzioni:

```
>>> gen = (i for i in 'abc')
>>> print gen.next(), gen.next(), gen.next()
a b c
>>> gen.next()
Traceback (most recent call last):
File "<stdin>", line 1, in ?
StopIteration
```

Abbiamo appena utilizzato una *generator expression* per creare un generatore. Quindi un generatore non é altro che un contenitore bufferizzato che ritorna di volta in volta gli oggetti registrati. Come le liste, ha anche una funzione che permette l'iterazione con il for per averne un utilizzo piú semplice:

```
>>> gen = (i for i in 'abc')
>>> for word in gen: print word,
a b c
>>> print gen.next()
Traceback (most recent call last):
File "<stdin>", line 1, in ?
StopIteration
```

Il risultato sará identico, come lo é anche l'errore alla fine. Il generatore, una volta terminata la sua iterazione, non ha piú elementi da ritornare. Lo si puó capire meglio utilizzando lo statment **yield** all'interno di una funzione:

```
>>> def fgen():
>>>     yield 'a'
>>>     yield 'b'
>>>     yield 'c'
>>> gen = fgen()
>>> print gen.next(), gen.next(), gen.next()
a b c
```

Come si può notare, *yield* è simile a *return*. Nel momento in cui si chiama la funzione, non viene eseguito nulla al suo interno e viene solo ritornato un generatore. Infatti quando chiama *next()* viene eseguito il codice all'interno della funzione fino a che non incontra *yield*, bloccandosi temporaneamente fino all'esecuzione nel prossimo *next()*. E così via fino all'ultimo *yield*, il quale codice che lo sussegue verrà eseguito con *next()* che darà errore poiché non vi sono presenti ulteriori *yield*:

```
>> def fgen():
>>     for word in 'abc':
>>         print 'Ritorno '+word
>>         yield word
>>     print 'Generatore terminato'
>> gen = fgen()
>> for word in gen: print word
Ritorno a
a
Ritorno b
b
Ritorno c
c
>> gen.next()
Generatore terminato
Traceback (most recent call last):
File "<stdin>", line 1, in ?
StopIteration
```

Come vedete, il funzionamento dei generatori non è così complicato, e in alcuni casi possono essere d'aiuto.

2.6.2 Funzioni semplici al volo: Lambdas

by Lethalman

Avete mai avuto il bisogno di creare delle semplicissime funzioni al volo? In questi casi viene in soccorso **lambda**: uno statment che permette di creare una funzione completa di argomenti in cui la prima espressione caratterizza il valore di ritorno. Ammettiamo di avere questa funzione e di seguito una lambda:

```

>> def func(): return 'test'
>> lfunc = lambda: 'test'
>> print func(), lfunc()
test test

```

Come vediamo, lambda restituisce una funzione che viene inserita nella variabile *lfunc*. Non abbiamo avuto bisogno di specificare il return, visto che la lambda accetta un'unica espressione di ritorno. Vediamo ora come specificare degli argomenti:

```

>> lfunc = lambda nick, nome='Luca': 'Nome: %s - Nick: %s' %
(nome, nick)
>> print lfunc('Lethalman')
Nome: Luca - Nick: Lethalman

```

Avrete notato che nelle lambda é anche possibile utilizzare le keyword arguments, come anche gli altri tipi di argomenti passabili ad una normale funzione.

Bisogna stare attenti a non abusare delle lambda, poiché essendo cosí ristrette, possono essere illeggibili. Inoltre, il fatto che non si può specificare la documentazione ad esse pertinenti (eccetto scriverla `...doc_`), non é un punto a favore nei propri programmi sotto l'aspetto professionale.

Un esempio pratico dell'uso delle lambda viene nel momento in cui si ha bisogno di creare una semplice funzione al volo da passare ad un'altra funzione:

```

>> lista = ['a', '', 'b', 'c']
>> print filter(lambda x: len(x)>0, lista)
['a', 'b', 'c']

```

filter é una funzione che necessita di una funzione come primo argomento, che viene chiamata per ogni elemento della lista specificata nel secondo argomento, e che toglie gli elementi con i quali la funzione ritorna False.

In questo caso, se la lunghezza dell'elemento *x* non é maggiore di 0, ritorna falso e toglie quindi i caratteri vuoti dalla lista.

2.7 Classi, variabili e moduli

2.7.1 Variabili globali

by The_One

Premesso che la vita di una variabile é legata all'esecuzione di una funzione, cioé:

```
>> a = 1
>> def f():
>>     a=5
>>     print a
1
>> f()
>>     print a
1
```

Stampa due volte 1 (e non 1 e 5) perché f() modifica il valore della variabile *a* **locale** ad essa stessa, la cui vita termina al terminare appunto di f(), e quindi non modifica quella globale.

É quindi importante conoscere l'esistenza delle variabili ”**globali**”!

La vita di una variabile definita globale all'interno di una funzione non dipende dall'esecuzione della funzione stessa e non termina con essa.

```
>> a = 1
>> def f():
>>     global a
>>     print a
>>     a = 4
>> f()
1
>>     print a
4
```

Il primo print dentro f() visualizza 1 (ma questo é ovvio, ed avverrebbe anche se *a* non fosse globale) Il secondo print invece stampa 4. Il valore di *a* stato modificato da f() perché era stata dichiarata globale.

2.7.2 Accesso dinamico al nome delle variabili

by **The_One**

Ipotizziamo che abbiate un pó di variabili, i namespace di python sono implementati tramite dizionari, questo significa che esiste un dizionario che memorizza tutte le nostre variabili/funzioni ed in corrispondenza i relativi valori.

Prendiamo in considerazione il seguente esempio:

```
>> a = 1
>> b = 3
```

Il dizionario avrà un contenuto simile:

```
{ 'a' : 1, 'b' : 3, ... }
```

Per accedere a questo dizionario si usa **locals()** oppure **globals()** a seconda del tipo di variabili che ci interessano, ovvero del contesto in cui ci troviamo. Piú precisamente, riferendoci all'esempio di prima possiamo stampare il valore di *a* in questo modo:

```
>> print locals()['a']
3
```

Come possiamo vedere viene stampato 3 che é proprio il valore di *a* locale alla sessione in cui stiamo lavorando.

Naturalmente con lo stesso meccanismo é possibile fare di tutto con le nostre variabili, assegnazioni comprese:

```
>> locals()['a'] = 7
>> print a
7
```

Dopo aver capito questo semplice meccanismo cerchiamo di inturne un possibile utilizzo.

All'interno delle parentesi quadre é infatti possibile anche concatenare piú stringhe per far riferimento ad una variabile:

```
>> abc = 9
>> print locals()['a'+ 'b'+ 'c']
```

Incredibile ma vero!

Morale della favola, utilizzando in generale la sintassi

```
locals()['partefissa'+'parteDaAggiungere']
```

é possibile assegnare ad esempio un valore ad una grande quantità di variabili che differiscono per una parte del nome:

```
>> suffissi = ['a','b','c','d']
>> prefissoa=5
>> prefissob=4
>> prefissoc=3
>> prefissod=0
>> for s in suffissi:
>>     print s, locals()['prefisso'+s]
a 5
b 4
c 3
d 0
```

Oltre a `locals()` vi é anche `vars()` che in assenza di argomenti si comporta come `locals()` mentre con un argomento permette di ispezionare il dizionario dell'argomento stesso:

```
>> class c: pass
>> o = c()
>> o.x = 42
>> o.y = 'test'
>> vars(o)
{'y': 'test', 'x': 42}
```

Naturalmente é possibile anche utilizzare lo stesso meccanismo per inizializzare molte variabili simili, oltre che per accedervi.

2.7.3 Inheritance dinamica di classi

by Lethalman

Vi é mai capitato di dover creare una classe che ne acquisisce altre due in runtime senza saperlo al momento della scrittura del codice?

Ammettiamo di avere le classi **A**, **B** e **C**, e per alcune ragioni abbiamo bisogno di una classe che acquisisce A, B e C e possiamo saperlo solo in runtime. A questo punto dobbiamo trasformare una classe in una *class factory*, la quale restituisce un'istanza non di essa ma di una A+B+C.

Questo é come lo faremmo se sapessimo cosa acquisire:

```
>> class A:
>>     a = 'sono A'
>> class B:
>>     b = 'sono B'
>> class C:
>>     c = 'sono C'
>> class X(A, B, C): pass
>> instance = X()
>> print instance.a, instance.b, instance.c
sono A sono B sono C
```

Ma come fare quando sappiamo in runtime di dover acquisire A, B e C? Ricorriamo quindi al metodo `__new__` acquisendo la classe `object`, che ci permette di poter ritornare ciò che vogliamo.

Prendiamo in esempio questo codice:

```
>> class X(object):
>>     def __new__(cls):
>>         print 'Chiamato __new__'
>>         return 'test'
>>     def __init__(self): print 'Chiamato __init__'
>> instance = X()
Chiamato __new__
>> print instance
test
```

Incredibile! Come avete visto noi chiamiamo `X()` aspettandoci un'istanza della classe `X` e invece otteniamo "test", e di conseguenza possiamo osservare la sequenza delle operazioni che vengono effettuate chiamando `X()`, ovvero viene chiamato `__new__(cls)`¹.

Come mai invece non viene invece chiamato `__init__(self)`? Il motivo é semplice: 'test' non é un'istanza di `X` e sappiamo che `__init__` viene chiamato per

¹`cls` é la stessa classe **X**. Come per tutti gli altri metodi *self* é l'istanza, in questo caso é la classe.

inizializzare un'istanza. Alché possiamo permetterci di scrivere questo:

```
>> class X(object):
>>     def __new__(cls, inherit1, inherit2, inherit3):
>>         class nuovaclasse(inherit1, inherit2, inherit3): pass
>>         return nuovaclasse()
>> instance = X(A, B, C)
>> print isinstance(instance, A), isinstance(instance, B), isinstance(instance,
C)
True True True
```

Come possiamo vedere, la richiesta dell'istanza X ritorna invece una di una nuova classe creata in un runtime che acquisisce A, B, e C su nostra richiesta. Come fare invece per far si che la **stessa classe** acquisisca altre classi?

```
>> class X(object):
>>     def __new__(cls, inherit1, inherit2, inherit3):
>>         class X(cls, inherit1, inherit2, inherit3): pass
>>         return object.__new__(X)
>> instance = X(A, B, C)
>> print isinstance(instance, X), isinstance(instance, A), isinstance(instance,
B), isinstance(instance, C)
True True True True
```

Rispetto a prima abbiamo utilizzato `object.__new__` poiché facendo `X()` veniva rieseguito il `__new__` della nostra classe e quindi un loop infinito. `object.__new__` inoltre é richiamabile visto e considerato che `cls` (ovvero X) acquisisce a sua volta `object`.

In questo modo abbiamo esteso una classe in runtime! Questa tecnica é solitamente inutilizzata ma a volte progetti grossi e molto dinamici possono averne bisogno.

2.7.4 Importare moduli dinamicamente

by Lethalman

In progetti molto dinamici avrete avuto il bisogno di importare dei moduli senza che voi ne sapeste il nome, e magari in contesti diversi.

Bisogna sapere che `import nomemodulo` non fa altro che importare tutti gli oggetti contenuti nel file nella variabile `nomemodulo` nel contesto locale,

ed essendo quindi un comune oggetto variabile é modificabile come tale. Eccovene una prova:

```
>> import os
>> print os.getcwd()
/home
>> os = 'test'
>> print os.getcwd()
Traceback (most recent call last):
File "<stdin>", line 1, in ?
AttributeError: 'str' object has no attribute 'getcwd'
```

Mettiamo caso che noi volessimo importare un modulo e ne conoscessimo il nome in una stringa, possiamo ricorrere a `__import__()`:

```
>> modulo = __import__('os')
>> print modulo.getcwd()
/home
```

Vi sará capitato anche di dover ricaricare un modulo, magari lo avete modificato e adesso bisogna ricaricarlo per aggiornarne il contenuto, infatti se si modifica un modulo e lo si reimporta, il contenuto risulta identico. Si puó quindi risolvere il problema usando `reload()`:

```
>> modulo = __import__('os')
>> modulo = reload(modulo)
```

In questo modo il modulo `os` risulterà aggiornato. Questo a volte puó servire nel momento in cui si vogliono aggiornare dei moduli creati personalmente durante il runtime di un programma.

2.7.5 Estendere l'assegnazione, lettura e cancellazione di un oggetto

by Lethalman

Per chi é abituato alla pulizia e alla sistematica del proprio codice, nelle classi utilizza sicuramente delle funzioni tipo `setVar`, `getVar` invece di darle un accesso diretto, anche perché si possono eseguire ulteriori operazioni oltre alla normale lettura/assegnazione.

In questi casi si può ricorrere a **property**, un metodo semplice e veloce per estendere l'assegnazione, lettura e cancellazione di un oggetto in modo del tutto naturale:

```
>>> class X(object):
>>>     def __init__(self):
>>>         self.__var = None
>>>     def getVar(self):
>>>         print 'Sto ritornando __var'
>>>         return self.__var
>>>     def setVar(self, value):
>>>         print 'Sto assegnando value a __var'
>>>         self.__var = value
>>>     def delVar(self):
>>>         print 'Sto cancellando __var. Adesso non potrai piú accedervi...'
>>>         del self.__var
>>>     var = property(getVar, setVar, delVar, 'Documentazione variabile')
>>> x = X()
>>> x.var = test'
Sto assegnando value a __var
>>> print x.var
Sto ritornando __var
test
>>> del x.var
Sto cancellando __var. Adesso non potrai piú accedervi...
```

Vediamo che *var* é divenuta ai nostri occhi una normale variabile, però in realtà é una classe che nel momento in cui le viene assegnata qualcosa, letta o rimossa, vengono eseguite le funzioni specificate. Da notare che si può utilizzare in una classe che acquisisce *object*, poiché dá la possibilità di sfruttare alcune funzioni speciali.

L'unica differenza in questo metodo sta nell'utilizzare le normali operazioni builtin al posto di chiamare delle apposite funzioni.

Capitolo 3

Console e I/O

3.1 Come togliere la newline dalla funzione `print()`

by Fred1

Molti di voi si saranno posti il problema: come faccio, in Python, a togliere l'odioso `\n` (newline) che viene aggiunto in automatico alla funzione `print()`? Ci sono principalmente due modi:

1. aggiungere `'` a fine comando
2. utilizzare il modulo `sys`

La prima é molto semplice, ovvero basta digitare `print 'ciao'`, e poi provare a fare un `print 'mondo!'` che la shell ti stampa `"ciao mondo"`. Un problema di questo metodo é che ti aggiunge uno spazio che magari tu non vuoi. Facciamo finta che tu voglia scrivere dei numeri random... ovviamente non li vuoi vedere cosí: `1 9 4 3`, ma `1943!` Utilizzando il primo modo, potresti creare un codice del tipo:

```
>> import random
>> for i in range(10):
>>     print random.randint(0,9),
7 6 4 9 9 9 2 0 4
```

E se invece li volessi attaccati? Beh, basta importare il modulo `sys` e poi digitare: `sys.stdout.write('testo che vuoi')`; notare che non vuole `print()`. Quindi il codice di prima diventerá:

```
>> import random, sys
>> for i in range(0,9): sys.stdout.write(str(random.randint(0,9)))
096867479
```

3.2 Come catturare l'output di un comando avviato con python

by [slash]

LICENSE: All the Python code contained in this recipe is released under GNU General Public License version 2. You can find a copy of the license here:

<http://www.gnu.org/licenses/gpl.html>

In un programma con interfaccia grafica (ma forse anche senza interfaccia grafica) può capitare di dover eseguire un comando esterno e conoscerne l'output in 'tempo reale' cioè riga per riga durante l'esecuzione del comando (se non si ha questa ultima necessità si può usare il modulo `commands`), ad esempio per usare una barra di avanzamento, o avere una console con l'output del comando avviato.

Per realizzare questo, semplicemente, basta lanciare il comando ed aprire una pipe da quel processo (realizzato tramite `popen` che restituisce un oggetto file collegato alla pipe); poi tramite un semplice ciclo bisogna leggere una riga per volta dall'oggetto file collegato alla pipe, controllare che non sia terminato il contenuto dell'oggetto file (e in tal caso uscire dal ciclo), ed infine usare la riga in base alle proprie necessità; una volta terminato il ciclo è possibile conoscere lo "stato di uscita" del comando tramite il valore restituito alla chiusura dell'oggetto file che si è usato.

Il codice:

```
>> import os
>> command = 'wget http://kernel.org/pub/linux/kernel/v2.6/patch-2.6.11.11.bz2'
>> stout = os.popen('%s 2>&1' % command)
>> while 1:
>>     line = stout.readline()
>>     if not line:
>>         break
>>     print line,
```

```
>> status = stout.close()
```

Nel caso l'output sia statico ad esempio un comando tipo 'netstat', 'ls', ecc. . .

```
>> import os
>> output = os.popen('ls').read()
>> print output
```

3.3 Misurare un file "a crudo"

by **The_One**

Vi sarete chiesti: "Ma possibile che per sapere quanto sia grande un file si debba importare per forza OS?"

Prima o poi qualcuno mi dirá che é facilissimo e si fa in una riga. . . nel frattempo io vi voglio far vedere un piccolo *stratagemma*.

In effetti esiste un modo abbastanza sbrigativo per aggirare il problema e consiste nell'utilizzo di metodi propri dell'oggetto **file**: questi metodi sono per l'appunto **seek()** e **tell()**.

In questo esempio apriamo il file, dopo di posizioniamo alla fine del buffer ottenendo il numero di bytes totali.

```
>> a = open('file.txt', 'r')
>> a.seek(0,2)
>> print a.tell()
66749
```

Ci dice che il puntatore si trova al 66749esimo byte, ma essendo questo l'ultimo per ipotesi, il teorema é dimostrato e il file é dunque lungo 66749 bytes.

NOTA: Per evitare problemi successivi con la lettura del file conviene rimettere le cose apposto, cioè il puntatore all'inizio del file.

```
...
>> a.seek(0,0)
```

Capitolo 4

Sockets

4.1 Stream sockets

4.1.1 Listen su piú porte e un unico obiettivo

by Lethalman

Non vi é mai capitato di aver bisogno di mettere un server in ascolto su piú porte e far sí che i client connessi vengano elaborati nella stessa maniera? Ho avuto questo problema con il mio server IRC, e dovendo effettuare questa operazione ho dovuto ricorrere al *select* delle socket.

Questa tecnica consiste nel controllare l'input di ogni server socket messa in ascolto sulle varie porte e quindi accettare il client.

Ci viene quindi in aiuto il modulo `socket` che contiene `poll`, una classe in cui é possibile registrare i vari buffer. Il `poll` controlla automaticamente se vi sono nuovi dati da leggere e ritorna i fileno dei buffer registrati.

```
>> import socket, select
>> socketpoll = select.poll()
>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>> sock.bind(('localhost', 12345))
>> sock.listen(1000)
>> socketpoll.register(sock)
>> waiting = socketpoll.poll()
>> print waiting
[(3,1)]
```

Cosa é successo? All'inizio abbiamo creato un `poll`, e dopo una normale

socket. Dopodiché abbiamo registrato sock nel poll e lo abbiamo avviato. Nel momento in cui un buffer ha nuovi dati (in questo caso la connessione di un client) la funzione ritorna una lista contenente delle tuple, formate dal fileno e da flags.

Per conoscere il fileno della socket, basta fare `sock.fileno()`. Per le flags (premettendo che voi conosciate il funzionamento degli operatori bitwise) vi rimando alla documentazione select del Python.

Ora noi dobbiamo metterci in ascolto su piú porte, questo comporta la creazione di piú socket, e fin qui niente di strano. Il problema é che `poll()` restituisce i fileno, quindi abbiamo bisogno di un dict di questo tipo **{fileno: socket}** per averne un accesso diretto e immediato.

```
>>> import socket, select
>>> sockets = {}
>>> socketpoll = select.poll()
>>> for port in 12345, 12346, 12347:
>>>     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>>     sock.bind(('localhost', port))
>>>     sock.listen(1000)
>>>     sockets[sock.fileno()] = sock
>>>     socketpoll.register(sock)
>>> while 1:
>>>     waiting = socketpoll.poll()
>>>     for sock in waiting:
>>>         if not sock[1] & select.POLLIN:
>>>             del sockets[sock[0]]
>>>             socketpoll.unregister(sock[0])
>>>         else:
>>>             clientsock, clientaddr = sockets[sock[0]].accept()
>>>         ...
```

In questo codice abbiamo salvato ogni socket creata in sockets con key `.fileno()` e value la stessa socket. Dopo si entra nel ciclo e si comincia a controllare le socket. Nel momento in cui avviene qualcosa, la funzione si sblocca e quindi controlliamo tuple per tuple (fileno, flags) di waiting. Se la socket non ha dati in entrata, significa che qualcosa é andato storto, poiché una server socket deve per forza avere un input prima di tutto, quindi la cancelliamo da sockets e la deregistriamo dal poll. Altrimenti, otteniamo la socket da sockets specificando come key il fileno e accettiamo la connessione del client.

4.1.2 Indirizzo in uso e socket chiusa

by Lethalman

Vi é mai capitato di riavviare un server e dover aspettare qualche minuto prima di poter riattivare la socket poiché diceva che l'indirizzo sul quale andava in ascolto era in uso? É abbastanza strana questa situazione, ma viene a verificarsi nel momento in cui un client si collega e la socket viene chiusa, rimanendo quindi l'indirizzo in uso per operazioni ancora in sospeso.

Questo problema é risolvibile specificando l'opzione REUSEADDR nella socket:

```
>> import socket
>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>> sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Adesso il problema non verrà piú a verificarsi!

4.1.3 IPC con le socket in dominio UNIX

by Lethalman

La comunicazione tra i processi su un sistema UNIX ha un'importanza fondamentale. Essa può essere attuata in diversi modi: via pipe, via fifo, ecc., ma una delle migliori soluzioni é sicuramente usare le socket in dominio UNIX (**AF_UNIX**). Ciò che contraddistingue questo metodo dagli altri é che le socket sono specializzate nell'I/O non bufferizzato e nell'handling dei clients da parte del server.

Le socket AF_UNIX funzionano allo stesso modo di quelle AF_INET con la differenza che sono locali, cioè l'indirizzo a cui collegarsi é un file. Vediamo ora il codice del server:

```
>> import socket
>> sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
>> sock.bind('/tmp/test.sock')
>> sock.listen(1)
>> while 1:
>>     client = sock.accept()[0]
>>     print client.recv(1024)
>>     client.close()
```

E quello del client:

```
>> import socket
>> sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
>> sock.connect('/tmp/test.sock')
>> sock.send('Test')
>> sock.close()
```

Se tutto é andato bene vedremo il server stampare a video "Test". Il codice del server però accetta solo un client alla volta e nel momento in cui un client sia collegato un altro non potrà collegarsi, ma come ben sapete questo problema é risolvibile grazie ai threads.

Come fare invece per ricevere dati da un client alla volta e non perdere connessioni in contemporanea? Ricorriamo dunque a questo stratagemma:

```
>> import socket, thread
>> from Queue import Queue
>> def newclient(queue):
>>     while 1:
>>         client = queue.get(1)
>>         print client.recv(1024)
>>         client.close()
>> sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
>> sock.bind('/tmp/test.sock')
>> sock.listen(1)
>> queue = Queue()
>> thread.start_new_thread(newclient, (queue,))
>> while 1: queue.put(sock.accept()[0])
```

Abbiamo implementato una queue contenente tutti i nuovi client che tentano di collegarsi proprio nel momento in cui ve ne sia già un altro a comunicare con il server. La funzione *newclient()* viene avviata come thread altrimenti l'handling dei dati bloccherebbe le nuove connessioni.

Ogni qualvolta si collega un client, esso viene inserito in queue in modo che il thread, non appena vi sia un nuovo client in lista d'attesa, possa comunicare con il client successivo.

4.2 World Wide Web

4.2.1 Webserver senza files

by Lethalman

Avete mai avuto la voglia di creare un web server magari per dare un'interfaccia grafica ad un vostro programma senza utilizzare dei files come output web? In questi casi bisogna crearsi un proprio handler per il web server in modo da redirezionare le richieste del client verso funzioni personalizzate. Il risultato che sia ha solitamente con i files si può rappresentare con il seguente esempio:

Richiesta /index → *Server* → *Output file index*

Con il nostro metodo verrà invece a verificarsi:

Richiesta /index → *Server* → *Output personalizzato interno*

Bene, innanzitutto per il web server ci viene in aiuto **BaseHTTPServer**, dal quale verrà utilizzato **HTTPServer** per creare il server, e **BaseHTTPRequestHandler** da acquisire in una nostra classe per crearci un nostro handler personalizzato.

Partiamo quindi da una base molto elementare:

```
>> from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler
>> server = HTTPServer(('', 8080), BaseHTTPRequestHandler)
>> server.serve_forever()
```

Eseguendo questo codice, e provando ad accedere alla porta 8080, non vi sarà possibile utilizzare nessun comando (es. *GET*, *POST*, ecc.) del protocollo HTTP, poiché il nostro handler é veramente elementare.

I comandi HTTP vengono specificati dalle funzioni di tipo *do_COMANDO*, nel caso di GET sarà *do_GET*. Quindi ora creiamoci una classe che acquisisce *BaseHTTPRequestHandler* e che aggiunge la nostra funzione:

```
>> from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler
>> class Handler(BaseHTTPRequestHandler):
>>     def do_GET(self):
>>         response = 'Metodo GET'
>>         self.wfile.write(response)
>>         self.wfile.flush()
```

```

>> self.connection.shutdown(1)
>> server = HTTPServer('', 8080), Handler)
>> server.serve_forever()

```

Guardando la documentazione, si evince che *wfile* é il descrittore che permette di scrivere l'output che verrà visualizzato dal client. Successivamente lo flusha, in modo da farlo visualizzare anche prima di chiudere il descrittore, e infine viene chiusa la socket in modo write. Se proviamo ad effettuare un GET collegandoci al server, potremo notare la mancanza totale di headers e il messaggio "Metodo GET".

Quello di cui abbiamo bisogno ora é cogliere i dati passati dal client, come ad esempio le variabili dalla barra indirizzi. L'indirizzo (path) richiesto dal client é contenuto in *self.path*, e possiamo quindi sfruttarlo per crearne un dizionario facilmente accessibile:

```

>> from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler
>> from urllib import unquote
>> class Handler(BaseHTTPRequestHandler):
>>     def args(self, data):
>>         ret = {}
>>         for val in data.split('&'):
>>             val = val.split('=', 1)
>>             k = val[0]
>>             if len(val) > 1: v = val[1]
>>             else: v = ''
>>             ret[k] = unquote(v)
>>         return ret
>>     def do_GET(self):
>>         path = self.path.split('?', 1)
>>         if len(path) > 1: self.getvars = self.args(path[1])
>>         else: self.getvars = {}
>>         response = ""Metodo GET<br>
Path: %s<br>
GETVars: %s"" % (self.path, self.getvars)
>>         self.wfile.write(response)
>>         self.wfile.flush()
>>         self.connection.shutdown(1)
>> server = HTTPServer('', 8080), Handler)
>> server.serve_forever()

```

Puntando il nostro browser al link </index?a=test&b> verrà visualizzato il seguente risultato:

```
Metodo GET<br>
Path: /index?a=test&b<br>
GETVars: {'a': 'test', 'b': ''}
```

Come possiamo vedere, siamo riusciti ad ottenere le variabili passate dal client con il metodo GET in un dizionario.

Adesso ciò che ci serve é aggiungere anche il metodo POST, che é molto simile al GET ma le variabili vengono passate in un modo piú sicuro. Questo perché con il POST, oltre a passare le variabili direttamente dal path, le passa in input con un file descriptor e vengono automaticamente encodeate. Inoltre cominciamo ad analizzare il path per estrarne la pagina richiesta:

```
>> from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler
>> from urllib import unquote
>> class Handler(BaseHTTPRequestHandler):
>>     def args(self, data):
>>         ret = {}
>>         for val in data.split('&'):
>>             val = val.split('=', 1)
>>             k = val[0]
>>             if len(val) > 1: v = val[1]
>>             else: v = ''
>>             ret[k] = unquote(v)
>>         return ret
>>     def do(self):
>>         path = self.path.split('?', 1)
>>         if len(path) > 1: self.getvars = self.args(path[1])
>>         else: self.getvars = {}
>>         if self.command == 'POST':
>>             self.postvars = self.args(
self.rfile.read(int(self.headers['content-length'])))
>>         else: self.postvars = {}
>>         response = """Pagina: %s<br>
Path: %s<br>
GETVars: %s - POSTVars: %s
<form action=page method=POST><input type=text name=text>
<input type=submit value='Prova POST'>""" % (path[0], self.path,
self.getvars, self.postvars)
```

```

>> self.send_response(200)
>> self.send_header('Content-type', 'text/html')
>> self.send_header('Content-length', str(len(response)))
>> self.end_headers() >> self.wfile.write(response)
>> self.wfile.flush()
>> self.connection.shutdown(1)
>> server = HTTPServer(('', 8080), Handler)
>> server.serve_forever()

```

Ho inserito gli header e i response per far funzionare il codice HTML e rendere piú completo l'handling, ma che non tratteró perché non sono argomento principale del tip.

Ora tocca a voi sfruttare questo esempio per i vostri obiettivi!

4.2.2 Ricavare il nome di un webserver al volo

by mozako & blackbird

Sovente capita di dover scrivere qualche programmino che richiami il nome del webserver caricato su un determinato host: vediamo come fare.

1. Importiamo la libreria socket:

```
>> import socket
```

2. Definiamo due variabili: HOST e PORT, all'interno delle quali definiremo rispettivamente l'hostname e la porta cui vogliamo vagliare il check:

```
>> HOST = '127.0.0.1'
>> PORT = 80
```

3. Connettiamoci e inviamo una richiesta di GET al server:

```
>> connessione = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>> connessione.connect((HOST, PORT))
>> connessione.send('HEAD / HTTP/1.0 \n\n')
>> data = connessione.recv(1024)
```

4. Creo l'espressione regolare che mi cattura la stringa **Server** dalle intestazioni HTTP:

```
>> import re
>> match = re.search(r'Server: (.*)', data, re.I)
```

5. Stampiamo a video il risultato e chiudiamo la connessione:

```
>> if match:
>>     server = match.group(1)
>>     print server
>> else:
>>     print 'Web server non specificato'
>>     connessione.close()
```

L'output che verrà visualizzato conterrà il nome del web server utilizzato dal determinato host.

4.2.3 Download con urllib

by Luigi Pantano aka bornFreeThinker

L'altro giorno mi sono imbattuto in questo semplice problema su come sia possibile, utilizzando python, effettuare un download. In prima analisi ho provato a risolvere il problema con il seguente codice consigliatomi da Fred1:

```
>> import urllib
>> url = 'http://download.gna.org/py-tips-tricks/Python_Tips_Tricks.pdf'
>> a = urllib.urlopen(url)
>> f = open('Python_Tips_Tricks.pdf', 'w')
>> f.write(a.read())
>> f.close()
```

Purtroppo oltre ad essere una soluzione poco elegante ha dato qualche piccolo problema sui sistemi operativi windows, sui quali alla fine del download il file risultava corrotto.

Dopo una attenta analisi della libreria urllib ho trovato la reale soluzione ai miei problemi:

```
>> import urllib
>> url = 'http://download.gna.org/py-tips-tricks/Python_Tips_Tricks.pdf'
>> urllib.urlretrieve(url, 'Python_Tips_Tricks.pdf')
```

dove in generale il primo argomento della funzione rappresenta il percorso del download ed il secondo invece è il nome con il quale il file verrà salvato in locale.

Nota: il seguente codice può essere utilizzato per effettuare indistintamente download sia dal protocollo *http* che dal protocollo *ftp*.

Capitolo 5

DataBases

5.1 Accesso immediato e niente SQL

by Lethalman

Vi sarete sicuramente imbattuti in questo tipo di dilemma: *”Come posso creare un database su un file, accederci in modo dinamico ed immediato senza l’uso di query SQL?”*

Ebbene, il modulo **shelve** offre proprio la possibilità di creare dei database su file in cui salvare qualsiasi oggetto in Python accedendovi senza SQL e in modo del tutto naturale!

Shelve si interfaccia a **pickle**, un modulo in grado di serializzare oggetti Python, il resto consiste nel salvataggio dei dati sul file, usandolo come un semplice dizionario.

```
>>> import shelve
>>> db = shelve.open('file.db', writeback=True)
>>> db['chiave'] = 'valore'
>>> db.close()
>>> db = shelve.open('file.db', writeback=True)
>>> print db['chiave']
valore
>>> db.close()
```

In questo modo abbiamo creato il database **file.db**. L’opzione `writeback=True` è necessaria nel momento in cui si salvano degli oggetti (come un dict) di cui shelve non si accorge che siano state apportate modifiche.

La chiusura del database è necessaria al fine di salvare i dati nel database,

infatti l'unico punto debole di questo metodo é che il buffer non viene scritto finquando non viene chiuso il database. Di conseguenza non é possibile aprire piú volte lo stesso database contemporaneamente e apportare in modo sicuro le modifiche, poiché verrà scritto il buffer dell'ultimo database aperto, anche se volendo, questo problema é risolvibile con l'uso dei lock.

Ora diamo un'occhiata su come cancellare delle chiavi (in questo caso quella sopra creata) e fare una semplice rubrica:

```
>>> import shelve
>>> db = shelve.open('file.db', writeback=True)
>>> del db['chiave']
>>> db['rubrica'] = {}
>>> db['rubrica']['Luca'] = 123456
>>> db['rubrica']['Mario'] = 321654
>>> db.close()
>>> db = shelve.open('file.db', writeback=True)
>>> print db['rubrica']
{'Mario': 321654, 'Luca': 123456}
>>> db.close()
```

Come abbiamo visto, shelve ha il pregio di creare in modo semplice e veloce dei database con accesso immediato e naturale. Ma per via dei suoi difetti, é utilizzabile in applicazioni non multiutente con diversi accessi temporanei, come ad es. dei siti web.

Capitolo 6

Interfaccia grafica

6.1 GTK

6.1.1 Utilizzo dei threads

by Lethalman

In questo semplice tip spiegheremo in pochi passi come utilizzare i threads mediante la creazione di una GUI con le GTK.

Ammettiamo di dover realizzare un programma con una *Entry* che contiene un numero in continua progressione con un loop infinito. Senza utilizzare i threads, tutto ciò bloccherrebbe totalmente gli eventi, poiché si rimarrebbe bloccati al loop che risiede all'interno di un evento (ad es. il click di un bottone che avvia il loop).

Un'ulteriore soluzione potrebbe essere quella di utilizzare i threads come li si usa di consuetudine, ma in questo caso non si avrebbero comunque i risultati desiderati, poiché l'unico thread che detiene il lock é quello principale.

Per questi scopi esistono le seguenti funzioni:

- **gtk.threads_init()** → Preparazione della gestione dei threads
- **gtk.threads_enter()** → Acquisizione del lock per il thread corrente
- **gtk.threads_leave()** → Rilascio del lock per il thread corrente

Ebbene, ecco il risultato finale che piú avanti commenteremo:

```

>> import gtk, thread
>> gtk.threads_init()
>> def run():
>>     while 1:
>>         gtk.threads_enter()
>>         time.set_text(str(int(time.get_text()+1))
>>         gtk.threads_leave()
>> def azzera(*w):
>>     time.set_text('0')
>> def avvia(w):
>>     w.set_sensitive(0)
>>     azzera()
>>     thread.start_new_thread(run, ())
>> win = gtk.Window()
>> win.connect('destroy', lambda w: gtk.main_quit())
>> vbox = gtk.VBox()
>> hbox = gtk.HBox()
>> hbox.pack_start(gtk.Label('Tempo:'))
>> time = gtk.Entry()
>> azzera()
>> hbox.pack_start(time)
>> vbox.pack_start(hbox)
>> hbox = gtk.HBox()
>> btn = gtk.Button('Avvia')
>> btn.connect('clicked', avvia)
>> hbox.pack_start(btn)
>> btn = gtk.Button('Azzera')
>> btn.connect('clicked', azzera)
>> hbox.pack_start(btn)
>> vbox.pack_start(hbox)
>> win.add(vbox)
>> win.show_all()
>> gtk.threads_enter()
>> gtk.main()
>> gtk.threads_leave()

```

Come possiamo vedere, all'inizio inizializziamo la gestione del threading. Dopodiché creiamo la GUI e avviamo il mainloop acquisendone il lock, affinché si possa interagire con la GUI.

Nel momento in cui si preme il bottone di avvio, la funzione avvia un thread, nel quale vi é un ciclo che di v olta in volta ottiene e rilascia il

lock aumentando di 1 il valore contenuto nella *Entry*.

Come potrete constatare, se tutto é andato a buon fine, durante il loop avrete la possibilità di clickare sul bottone di azzeramento. Si può quindi desumere che nel momento in cui si ha bisogno di interagire con la GUI all'interno di un thread, bisogna prima acquisire e dopo rilasciare il lock, stando attenti a non acquisirlo 2 volte all'interno dello stesso.

6.1.2 Bottone con immagine

by Lethalman

Spiegheró brevemente come inserire una immagine in un bottone utilizzando le librerie grafiche GTK. Cominciamo da una piccola premessa sulla struttura grafica dei bottoni.

Un bottone non é altro che un contenitore, dove all'interno possono essere quindi inseriti altri oggetti. Creiamo un normale bottone con:

```
>> import gtk
>> ...
>> gtk.Button('Test')
>> ...
```

Verrá automaticamente creato il bottone, all'interno un widget **Alignment** con un allineamento centrato che a sua volta conterrá la Label "Ciao". Questo é possibile farlo manualmente nel seguente modo:

```
>> import gtk
>> ...
>> button = gtk.Button()
>> align = gtk.Alignment(0.5, 0.5)
>> align.add(gtk.Label('Test'))
>> button.add(align)
>> ...
```

In questo modo avremo il nostro bottone. Non abbiamo fatto altro che crearne uno vuoto, dopodiché creare un allineamento centrato contenente la Label. Infine abbiamo inserito l'allineamento nel bottone. Senza l'allineamento, ovvero inserendo direttamente la Label, la scritta non sarebbe stata centrata.

Ora, prendendo per esempio uno stock button, esso non é altro che l'insieme

di una Label ed una Image, quindi nell'allineamento si può benissimo inserire una HBox formata appunto dalla Label e dall'Image:

```
>> import gtk
>> ...
>> button = gtk.Button()
>> align = gtk.Alignment(0.5, 0.5)
>> hbox = gtk.HBox(spacing=2)
>> image = gtk.Image()
>> ...
>> hbox.pack_start(image, 0)
>> hbox.pack_start(gtk.Label('Test'), 0)
>> align.add(hbox)
>> button.add(align)
>> ...
```

Abbiamo settato lo spacing della HBox a 2 in modo da distaccare la Label dalla Image, dopodiché abbiamo inserito prima l'Image e dopo la Label con *expand* disabilitato.

Questo è un piccolo esempio di come possono essere sfruttati i widget contenitori in GTK sottolineando la grande flessibilità di questo rinomato toolkit grafico.

6.1.3 Come compilare le pygtk con py2exe

by Luigi Pantano aka bornFreeThinker

LICENSE: All the Python code contained in this recipe is released under GNU General Public License version 2. You can find a copy of the license here:

<http://www.gnu.org/licenses/gpl.html>

Ecco il file setup.py:

```
>> from distutils.core import setup
>> import py2exe
>> import glob
>> opts = {
>>     'py2exe': {
>>         'includes': 'pango,atk,gobject',
>>         'dll_excludes': [
>>             'iconv.dll', 'intl.dll', 'libatk-1.0-0.dll',
```

```

>>     'libgdk_pixbuf-2.0-0.dll', 'libgdk-win32-2.0-0.dll',
>>     'libglib-2.0-0.dll', 'libgmodule-2.0-0.dll',
>>     'libgobject-2.0-0.dll', 'libgthread-2.0-0.dll',
>>     'libgtk-win32-2.0-0.dll', 'libpango-1.0-0.dll',
>>     'libpangowin32-1.0-0.dll'],
>> }
>> }
>> setup(
>>     name = 'nome programma',
>>     description = 'descrizione',
>>     version = 'versione',
>>     windows = [
>>         {'script': 'main.py',
>>          'icon_resources': [(1, 'pixmaps/icona.ico)]
>>         }
>>     ],
>>     options=opts,
>>     data_files=[('readme.txt'), ('pixmaps', glob.glob('pixmaps/icona.ico'))],
>>     ],
>> )

```

Dove:

main.py é il file del vostro progetto

readme.txt é il vostro personale readme legato al progetto

pixmaps/icona.ico é l'icona del vostro programma

Capitolo 7

Audio

7.1 Beep dall'altoparlante

by The_One & scitrek

Puó capitare a volte di dover inserire in un programma una notifica sonora, un semplice "beep". In genere é sufficiente effettuare questa operazione:

```
>> print '\a'
```

e l'altoparlantino di sistema emetterá un breve segnale. Ma se si volesse ottenere un segnale dal suono diverso, o di diversa durata? Sotto windows viene in aiuto la libreria **winsound**, ed in particolare la funzione:

```
Beep(frequency, duration)
```

Il primo parametro indica la frequenza del suono da emettere, in un range da 37 e 32767 Hz.

Il secondo, la durata del suono in millisecondi. Ad esempio, il seguente codice:

```
>> from winsound import Beep
>> Beep(131, 1000)
```

produrrá un suono a 131 Hz (*DO* della seconda ottava) della durata di un secondo.

É importante notare che, poiché il suono viene comunque emesso dall'altoparlante di sistema, il rendimento é piuttosto scarso. Tuttavia questa funzione puó

essere utilizzata per comporre brevi musiche introduttive, conferme, suoni di errore, ...

Capitolo 8

Compressione

8.1 Comprimere con bz2

by Luigi Pantano aka bornFreeThinker

Questa libreria é molto semplice da usare soprattutto se già avete familiarità con le operazioni di IO del python o di linguaggi simili. Quindi in questa brevissima "guida" daró per scontato che già sappiate eseguire le elementari procedure di I/O che per tale ragione non tratteró.

Gli stessi comandi quali *open*, *read*, *write*, *seek*, *readline*, ecc., si applicano al medesimo modo per quanto riguarda la libreria **bz2**.

Niente di piú semplice ed immediato! :-D

Commentare il banale esempio qui di seguito riportato sarebbe un'esagerazione ed un sacrilegio. Tengo solo a far notare che la peculiarità di questa libreria consiste nel fatto che la compressione/decompressione dei dati é totalmente trasparente all'utente, infatti quest'ultimo utilizzerá il file come se dovesse effettuare delle normali operazioni di scrittura/lettura dati:

```
>>> import bz2
>>> f = bz2.BZ2File('esempio.txt.bz2', 'w')
>>> f.write('tutto funziona correttamente')
>>> f.close()
>>> f = bz2.BZ2File('esempio.txt.bz2', 'r')
>>> print f.read()
tutto funziona correttamente
>>> f.close()
```

Per chi non avesse familiarità con il formato bz2, quest'ultimo é un formato di compressione molto usato in ambiente *nix ed un pó diverso dal "classico" zip utilizzato in modo massiccio in ambiente windows. Infatti il bz2 puó compressare solo un file, quindi l'archivio di "esempio.txt.bz2" conterrà al suo interno soltanto il file *esempio.txt* e nient'altro. Inoltre attenzione a rinominare l'archivio! Infatti cambiando il nome dell'archivio automaticamente rinominere anche il nome del file in esso contenuto!

8.2 Comprimere con zlib

by Luigi Pantano aka bornFreeThinker

La libreria **zlib** insieme alla **bz2** sono le librerie open source piú usate nell'ambito della compressione/decompressione dati nel mondo open source in generale ma principalmente nei sistemi Unix like, qual'é Linux.

Se proprio dovessimo fare un confronto tra le due librerie in questione, potremmo sicuramente affermare che la migliore dal punto di vista del livello massimo di compressione é sicuramente bz2 invece per quanto riguarda la velocità di compressione la zlib risulta migliore.

La scelta dipende dalle proprie esigenze o dalla propria comodità, se volete un mio parere preferisco la bz2 per la semplicità di utilizzo della libreria in python. Ovviamente il mio modesto parere non vi deve in alcun modo influenzare senza aver prima provato queste ottime librerie.

Per qualunque dubbio che la mia guida non riuscisse a colmare sull'utilizzo di questa ottima libreria rimando alla reference manual di python.

Procediamo nell'analisi delle funzioni contenute nella classe zipfile.

La funzione *_compress*, come del resto fa intendere perfettamente il nome, permette tramite il passaggio dei parametri "file_name" "compression_level", che rispettivamente sono il nome del file da compressare ed il livello di compressione che si vuole raggiungere, di compressare un file. Banalmente il file in questione viene aperto ed il suo contenuto viene compressato ed assegnato alla variabile "data_compressed", infine viene aperto in scrittura un file avente lo stesso nome del file da compressare con in piú il suffisso '.gz' e su quest'ultimo vengo scritti i dati contenuti dalla variabile.

Nota: il livello di compressione é un intero da 1 a 9 che esprime il livello di compressione; 1 é il piú veloce e produce la compressione minore, 9 é il piú lento e produce la massima compressione. Il valore predefinito é 6.

La funzione `_decompress` esegue esattamente il procedimento inverso di quello della funzione appena trattata. Dopo aver aperto il file compresso, ne legge il contenuto decomprimendolo ed assegnandolo alla variabile "data_decompressed", il cui contenuto verrà scritto sul file di destinazione avente lo stesso nome del file compresso esente però dal suffisso '.gz'.

```
>> import zlib
>> class zipfile:
>>     def _compress(self, file_name, compression_level=6):
>>         # lettura e compressione della sorgente
>>         f = open(file_name, 'r')
>>         data_compressed = zlib.compress(f.read(), compression_level)
>>         f.close()
>>         # scrittura dei dati
>>         f = open(file_name + '.gz', 'w')
>>         f.write(data_compressed)
>>         f.close()
>>     def _decompress(self, file_name):
>>         # lettura del file compresso
>>         f = open(file_name, 'r')
>>         data_decompressed = zlib.decompress(f.read())
>>         f.close()
>>         # scrittura dei dati
>>         f = open(file_name[:-3], 'w')
>>         f.write(data_decompressed)
>>         f.close()
>> # Esempio
>> z = zipfile()
>> z._compress('prova.txt')
>> z._decompress('prova.txt.gz')
```

Capitolo 9

Crittografia

9.1 Uso della libreria Crypto

by Luigi Pantano aka bornFreeThinker

LICENSE: All the Python code contained in this recipe is released under GNU General Public License version 2. You can find a copy of the license here:

<http://www.gnu.org/licenses/gpl.html>

Prima di iniziare ad entrare nel vivo del tips, riporto il sito da cui é possibile scaricare la libreria Crypto:

<http://www.amk.ca/python/code/crypto>

Features della libreria:

- Hash functions: MD2, MD4, RIPEMD, SHA256.
- Block encryption algorithms: AES, ARC2, Blowfish, CAST, DES, Triple-DES, IDEA, RC5.
- Stream encryption algorithms: ARC4, simple XOR.
- Public-key algorithms: RSA, DSA, ElGamal, qNEW.
- Protocols: All-or-nothing transforms, chaffing/winnowing.
- Miscellaneous: RFC1751 module for converting 128-key keys into a set of English words, primality testing.

Con questo semplice esempio mi prefiggo di semplificare l'utilizzo della ottima libreria di criptaggio pycrypt.

Dopo aver importato il modulo ed impostato la password con la quale criptare i dati decidiamo il tipo di codifica da effettuare nel nostro caso DES3.MODE_ECB (consultando la guida relativa alla libreria ne troverete altri, sta a voi la scelta).

Nota: vi sono algoritmi efficaci ma lenti, ed altri molto veloci ma poco sicuri. La scelta del rapporto sicurezza/velocità dipende da voi e dall'uso che intendiate farne nel vostro programma. Nel caso del mio Password Manager ho optato per l'algoritmo DES3 estremamente sicuro ed in generale molto lento, però tenendo in considerazione che la mole di dati da salvare per ogni password è di molto inferiore ad 1KB, il programma non risente minimamente della lentezza del DES3.

Analizziamo prima di tutto la funzione **complete** che non fa altro che controllare che il numero di caratteri presenti in *string* sia un multiplo di *number*, il cui valore di default è 24, altresì incrementa *string* con un certo numero di caratteri *D1* in modo tale da ottenere alla fine una stringa multipla di 24.

Nota: Nel caso avessimo scelto un'altro algoritmo di criptaggio diverso, tipo DES o AES, sarebbe stato necessario che la stringa da cifrare come del resto la password cifrante fosse multipla di 8 o 16. In generale tale valore dipende strettamente dall'algoritmo usato, quindi leggete attentamente le specifiche dell'algoritmo che avete intenzione di usare.

Se fin qui è stato tutto chiaro risulterà banale comprendere le funzioni **encrypt_data** e **decrypt_data**. La prima accetta come parametro la stringa da cifrare (che deve essere come abbiamo visto un multiplo di 24) e ritorna come valore una stringa cifrata.

La seconda funzione invece svolge esattamente il compito inverso della precedente accettando come parametro una stringa cifrata e restituendo come risultato la stringa non cifrata, ovviamente prima di fare ciò eliminerà gli eventuali caratteri aggiuntivi.

```
>> from Crypto.Cipher import DES3
>> from random import randint
>> password = 'prova'
>> crypt_type = DES3.MODE_ECB
```

```

>> def complete(string, number=24):
>>     # verifica se la stringa é un multiplo della variabile number
>>     altresí incrementa la stringa con caratteri casuali
>>     h = divmod(len(string), number)
>>     if h[-1] != 0:
>>         string += '\021'
>>         if (number-h[1]) > 1:
>>             # Genera la sequenza di caratteri casuali
>>             for x in range(1, number-h[1]):
>>                 string += chr(randint(0, 255))
>>     return string
>> def encrypt_data(string):
>>     obj = DES3.new(password, crypt_type)
>>     string = complete(string)
>>     return obj.encrypt(string)
>> def decrypt_data(string):
>>     obj = DES3.new(password, crypt_type)
>>     return obj.decrypt(string).split('\021')[0]
>> if __name__ == '__main__':
>>     password = complete(password)
>>     a = encrypt_data('TUTTO FUNZIONA CORRETTAMENTE')
>>     b = encrypt_data('TUTTO FUNZIONA CORRETTAMENTE')
>>     print 'Le stringhe crittate sono uguali?', a == b
Le stringhe crittate sono uguali? False
>>     print decrypt_data(a), decrypt_data(b)
TUTTO FUNZIONA CORRETTAMENTE TUTTO FUNZIONA CORRETTAMENTE

```

Capitolo 10

Programmazione su Windows

10.1 Arrestare Windows XP

by Fred1

Come fare ad arrestare Windows XP? L'altro giorno in chat stato chiesto se in Python esiste un modulo sperduto che permette di arrestare/riavviare Windows. Beh, la risposta é no.

Peró possiamo lanciare un file, che Windows XP (e non so se esiste anche nelle versioni precedenti) usa sempre per arrestare il computer.

Per raggiungere questo obiettivo, ci possiamo appellare al modulo `os`, che ha quella simpatica funzioncina chiamata `system` che permette di eseguire il comando (che poi é una stringa) in una sottoshell (*"Execute the command (a string) in a subshell"* come dice la documentazione di IDLE).

Quindi per arrestare Windows:

```
>> import os
>> os.system('C:\windows\system32\shutdown.exe -s -t 00')
```

Ma si può fare di meglio! Nei sistemi *nix esiste la possibilità di spegnere il computer ad una data ora, mentre in Windows no... Perché non farci il programma da noi?

```
>> def arresta():
>>     from time import localtime, sleep
>>     from sys import argv
>>     from os import system
>>     minut, hour = {}, {}
```

```

>> lista = argv[2].split(':')
>> for k in range(1,60): minut[k] = i*60
>> for k in range(1,25): hour[k] = i*60*60
>> if argv[1] == 'time' and len(lista) == 3:
>>     print 'Il computer verrà riavviato alle ore', argv[2]
>>     for i in lista:
>>         if i == lista[0]:
>>             try: oreinsec = hour[int(i)-localtime()[3]]
>>             except: oreinsec = 0
>>         elif i == lista[1]:
>>             try: mininsec = minut[int(i)-time.localtime()[4]]
>>             except: mininsec = 0
>>         else: break
>>     sleep(oreinsec+mininsec+int(lista[2]))
>>     system('C:\windows\system32\shutdown.exe -s -t 00')
>> elif argv[1] == 'sec' and len(lista) == 1:
>>     print 'Il computer verrà riavviato fra', argv[2], 'secondi'
>>     sleep(int(argv[2]))
>>     system('C:\windows\system32\shutdown.exe -s -t 00')
>> else: print 'Hai sbagliato qualcosa'

```

Il codice non dovrebbe presentare troppi problemi. Ovviamente può essere migliorato, ma questo compito lo lascio a voi :)

Comunque per far partire il programma, andare dal Prompt dei comandi (una volta chiamato MSdos) e digitare `.\nomefile.py <time|sec> <tempo>`. Ad esempio, per far arrestare il sistema alle 24:00 precise:

```
C:\Documents and Settings\percorso> nomefile.py time 24:00:00
```

Non ho fatto in modo che la funzione guardi che il tempo dato debba essere maggiore di quello attuale, ma di solito chi usa Windows non pu (o non riesce per colpa dello stesso :)) lasciare acceso il computer di notte, quindi non ci sono problemi...

Invece, per arrestare il sistema fra 20 secondi:

```
C:\Documents and Settings\percorso> nomefile.py sec 20
```

10.2 No more shell

by scitrek

Quando si crea una qualsiasi applicazione sotto Windows, all'apertura del programma viene aperta anche una finestra di prompt dell'MS-Dos. Questo perché l'interprete esegue il programma sotto console.

Fin qua tutto normale... il problema viene a verificarsi nel momento in cui si creano applicazioni che non necessitano della shell, come ad esempio programmi grafici e server, per i quali ciò non rappresenta altro che un fastidio.

Per eliminarla, basta modificare l'estensione dello script python da **.py** → **.pyw**. Così facendo il sistema capisce che non deve aprire una finestra di shell.

Capitolo 11

Programmazione su UNIX

11.1 Demonizzazione

by mozako

Se programmate applicativi su sistemi UNIX-like talvolta dovrete ricorrere alla demonizzazione di una funzione o dell'intero programma. Ecco come potete fare:

```
>> import os
>> def demone():
>>     if os.fork():
>>         os._exit(0)
>>     os.setsid()
>>     if os.fork():
>>         os._exit(0)
>>     os.umask(077)
>>     null = os.open('/dev/null', os.O_RDWR)
>>     for i in range(3):
>>         try:
>>             os.dup2(null, i)
>>         except OSError, e:
>>             if e.errno != errno.EBADF:
>>                 raise
>>     os.close(null)
```

Questo stratagemma vi permette, richiamando la funzione `demone()`, di demonizzare il vostro software o parti di esso (utile soprattutto per la gestione delle socket

e/o argomenti annessi alla programmazione di rete).

Capitolo 12

Ottimizzazione, statistica e debug

12.1 Libreria profile

by Luigi Pantano aka bornFreeThinker

Quante volte si é presentata ad ogni programmatore la necessità di ottimizzare un proprio programma per renderlo piú veloce ed efficiente possibile.

Il concetto chiave dell'ottimizzazione consiste nel trovare i cosí detti "colli di bottiglia", che non sono altro che delle righe di codice (o generalmente funzioni) che rallentano particolarmente l'utilizzo del programma.

Ovviamente questo tip non si soffermerá sulle tecniche di ottimizzazione ma solo sull'utilizzo della libreria profile che permette appunto di valutare il tempo complessivo (e non solo questo) che una funzione impiega per essere eseguita. Supponiamo di voler calcolare il tempo impiegato dalla funzione fattoriale presente in questo esempio didattico, basterá richiamarla semplicemente con il comando **profile.run**. Alla fine dell'esecuzione ci verrà riportato un dettagliato report nel quale oltre ad essere evidenziato il tempo complessivo di esecuzione, che nel nostro caso ammonta a 0.003 sec, troveremo altre importanti informazioni (per maggiori dettagli leggete il reference manual di python).

```
>> import profile
>> def fattoriale(n):
>>     fatt = 1
>>     for x in range(1, n+1):
>>         fatt = fatt*x
>>     print '!'+str(x)+'\t=\t', fatt
```

```

>> return fatt
>> profile.run('fattoriale(10)')
!1 = 1
!2 = 2
!3 = 6
!4 = 24
!5 = 120
!6 = 720
!7 = 5040
!8 = 40320
!9 = 362880
!10 = 3628800
      5 function calls in 0.003 CPU seconds

```

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.000	0.000	:0(range)
1	0.002	0.002	0.002	0.002	:0(setprofile)
1	0.000	0.000	0.000	0.000	<string>:1(?)
1	0.000	0.000	0.003	0.003	profile:0(fattoriale(10))
0	0.000		0.000		profile:0(profiler)
1	0.000	0.000	0.000	0.000	prova.py:6(fattoriale)

12.2 Controllare ogni operazione

by Lethalman

In Python esiste una funzione nel modulo `sys` che permette di controllare ogni operazione effettuata da uno specifico script all'interno dello stesso. Si tratta di `sys.settrace`: accetta come primo argomento una funzione che verrà chiamata, al momento in cui si fa una qualsiasi operazione, con i seguenti argomenti:

- frame - Informazioni sul frame da cui proviene l'azione (*codice, traceback, variabili, ecc...*)
- event - Il nome dell'evento (*call, exception, ecc...*)
- arg - Ulteriore dato a sostegno dell'evento

Ammettiamo di dover loggare ogni operazione effettuata da un nostro script, piú precisamente vogliamo registrare il nome della funzione che viene chiamata:

```
>>> import sys
>>> def logger(frame, event, arg):
>>>     if event == 'call':
>>>         print 'The function %r has been called' % frame.f_code.co_name
>>> def test(arg):
>>>     print arg
>>> sys.settrace(logger)
>>> test('Log this!')
The function 'test' has been called
Log this!
```

La funzione logger inizialmente controlla che l'evento corrisponda ad una chiamata, dopodiché ottiene il nome del codice del frame. Ma se volessimo avere anche l'argomento passato alla funzione, basta un tocco di fantasia e "smanettamento" con `dir()` e raggiungeremo anche questo obiettivo:

```
>>> import sys
>>> def logger(frame, event, arg):
>>>     args = []
>>>     for data in frame.f_locals.values():
>>>         args.append(data)
>>>     if event == 'call':
>>>         print 'The function %r has been called with the following
arguments: %r' % (frame.f_code.co_name, args)
>>> def test(arg):
>>>     print arg
>>> sys.settrace(logger)
>>> test('Log this!')
The function 'test' has been called with the following arguments:
['Log this!']
Log this!
```

La variabile `f_locals` del frame contiene le variabili locali presenti nel momento in cui la funzione viene chiamata, ovvero gli argomenti.

Capitolo 13

Decorazioni e vari trucchetti utili

13.1 Aggiungere e togliere elementi da un list con << e >>

by Lethalman

A volte vi é capitato di odiare il metodo `.append()` delle liste o magari fare delle funzioni apposite per eliminare degli elementi da una lista in modo continuato.

In questo articolo utilizzeremo << per aggiungere un elemento e >> per togliere tutte le occorrenze di un dato oggetto. Eccone subito un pratico esempio:

```
>> class list(list):
>>     def __lshift__(self, val):
>>         self.append(val)
>> l = list()
>> l << 'a'
>> print l
['a']
```

Inizialmente dichiariamo la classe `list` che acquisisce la sua stessa classe, aggiungendo nuove funzioni.

Visto che in Python non é possibile variare i tipi builtin, dobbiamo ricorrere alla creazione di una lista mediante `list()` invece di `[]` come si fa di solito.

Dopodiché viene chiamato il metodo `__lshift__` utilizzando gli shift (<<) nel seguente modo: `l.__lshift__('a')`.

Infine la funzione non fa altro che aggiungere a se stessa il dato oggetto.

Ora abbiamo intenzione di usare gli rshift (<<) per eliminare totalmente un dato oggetto dalla lista:

```
>> class list(list):
...
>>     def __rshift__(self, val):
>>         try:
>>             while 1: self.remove(val)
>>         except: pass
>> l = list()
>> l << 'a'
>> l << 'b'
>> l << 'a'
>> l << 'c'
>> print l
['a', 'b', 'a', 'c']
>> l >> 'a'
>> print l
['b', 'c']
```

Come possiamo notare, abbiamo tolto dalla lista ogni occorrenza di 'a'. Il suddetto codice sembra veramente dare un aspetto molto semplice e immediato nell'aggiunta/rimozione di elementi in una lista, però la ripetizione continua per l'inserimento di nuovi elementi é abbastanza lunga. Sappiamo che le funzioni possono anche ritornare dei valori, infatti l << 'a' ad esempio ritorna *None*; cosa succederebbe invece se ritornasse la lista stessa? Si potrebbero quindi usare piú volte gli shift:

```
>> class list(list):
>>     def __lshift__(self, val):
>>         self.append(val)
>>         return self
>>     def __rshift__(self, val):
>>         try:
>>             while 1: self.remove(val)
>>         finally: return self
>> l = list()
>> l << 'a' << 'b' << 'a' << 'c'
>> print l
```

```
['a', 'b', 'a', 'c']
>> print l >> 'a' >> 'b'
['c']
```

Ogni volta che si usano gli shift, la funzione ritorna la lista stessa, quindi di volta in volta gli shift operano sulla lista.

13.2 Da stringa a comando py: eval()

by **The_One**

Mettiamo di avere due funzioni $f()$ e $g()$ così definite:

```
>> def f():
>>     print 'ciao sono la funzione f'
>> def g():
>>     print 'ciao sono la funzione g'
```

Ci poniamo il problema di far eseguire al nostro programma una sola di queste due funzioni e vogliamo che sia l'utente del programma a scegliere quale: possiamo utilizzare qualcosa del tipo:

```
...
>> func = raw_input('Quale funzione? ')
>> if func == 'f()': ...
>> elif func == 'g()': ...
```

In questo modo usiamo una normale condizione ed eseguire di conseguenza la funzione scelta dall'utente. Il problema viene a verificarsi nel momento in cui, per un qualsiasi motivo (solitamente in progetti molto dinamici), dobbiamo tramutare una stringa in codice python. In questo caso ci viene in aiuto la funzione `eval()`:

```
...
>> eval(func)
ciao sono la funzione f
```

In questo caso é evidente che abbiamo specificato in input `"f()"` e come possiamo vedere `eval()` ha correttamente eseguito l'operazione data in input

dall'utente. Le funzioni sono un esempio pratico ma sono possibili eseguire anche le altre istruzioni.

13.3 Coding delle funzioni

by Lethalman

Beh questo tip servirá a ben poco probabilmente, ma é sempre cultura che é alla base del Python. Potrete sicuramente notare moduli come **code**, **dis**, e altri che servono ad interagire con il codice Python, ma vi chiederete: come? quando? perché? É semplice, visto che in Python tutto é un oggetto, ebbene anche i pezzi di codice sono degli oggetti!

Come sapete, i files *.pyc* sono dei bytecode di codice Python, già compilati e pronti all'esecuzione senza bisogno di reinterpretare il codice. Nel caso delle funzioni, esse hanno il proprio bytecode contenuto in una variabile. Proviamo dunque a creare una funzione e a leggerne il suo bytecode:

```
>>> def test(): print 'test'
>>> print test.func_code.co_code
'd\x01\x00GHd\x00\x00S'
```

Ecco il nostro bytecode, formato da caratteri come potete vedere il piú delle volte in esadecimale, poiché non sono rappresentabili sullo schermo. *test.func_code* é un code object, creabile tramite il modulo `code`:

```
>>> import code
>>> func_code = code.compile_command("print 'test'")
>>> print func_code.co_code
'd\x01\x00GHd\x00\x00S'
```

Come possiamo notare questo bytecode non ha nulla di diverso con il precedente. Ora proviamo invece a modificare il codice di una funzione:

```
>>> import code
>>> def test(): pass
>>> func_code = code.compile_command("print 'test'")
>>> test.func_code = func_code
>>> test()
test
```

Come per magia, vediamo stampato "test". Ora la modifica del bytecode delle funzioni dipende dalle esigenze del programmatore, e la difficoltà dell'operazione pu'oraggiungere il suo culmine nel momento in cui bisogna inserire del codice in un preciso punto della funzione. . . ma ribadisco che tutto ciò é solo a scopo educativo e difficilmente viene messo in pratica.

Ora vediamo come disassemblare un code object, e trarne il suo codice in un modo un pó piú leggibile dall'occhio umano:

```
>>> import code, dis
>>> func_code = code.compile_command("print 'test'")
>>> def test(): print 'test'
>>> print dis.dis(func_code)
1 0 LOAD_CONST 1 ('test')
3 PRINT_ITEM
4 PRINT_NEWLINE
5 LOAD_CONST 0 (None)
8 RETURN_VALUE
>>> print dis.dis(test)
1 0 LOAD_CONST 1 ('test')
3 PRINT_ITEM
4 PRINT_NEWLINE
5 LOAD_CONST 0 (None)
8 RETURN_VALUE
```

Verrá visualizzato il codice disassemblato in modo totalmente identico. Adesso tentiamo di creare una funzione avendo già a disposizione un code object:

```
>>> import code
>>> from new import function
>>> func_code = code.compile_command("print 'test'")
>>> test = function(func_code, globals())
>>> test()
test
```

Con la funzione function é possibile anche specificare gli argomenti ed altri tipi di dati necessari ad una funzione, ma l'approfondimento spetta a voi.

13.4 For, non fermiamoci alle apparenze

by Fred1

Tutti, immagino, sapete come funziona un for in Python. Anche un perfetto newbie sa come funziona. Il problema é che molte persone non hanno capito la base, il funzionamento del ciclo for.

Conosco molti utenti che usano un ciclo apparentemente in modo normale:

```
>> for i in range(10): print i
```

peró, se gli chiedi di iterare una lista, fanno cosí:

```
>> for k in range(10): print list[k]
```

Ciò non é sbagliatissimo ma piú che altro non é appropriato.

Non ho ancora trovato una guida che spiega il funzionamento del for in modo da imprimerlo per sempre nella mente del lettore. Quindi capiamoci: il for itera una lista, ovvero esegue un procedimento nel quale prende un elemento per volta della lista in questione ed assegna alla variabile i il valore dell'elemento (in questo contesto é cos, ma una definizione di for piú ampia é: **il for itera tutto ciò che é iterabile**: liste, tuple, stringhe e - solo dalla versione 2.3 in poi - anche generatori.). É un pó ingarbugliato, quindi andiamo alla pratica:

```
>> for i in range(3): print i,
```

Vi restituirá: "0 1 2". Però, quando vedi **range()** nelle guide senza che essa sia spiegata (e anche se lo é, non cambia molto), molte persone (per non dire tutte) la utilizzano senza realmente sapere cosa fa. range() restituisce una lista di numeri, niente di piú.

```
>> for i in [0,1,2]: print i,
```

Restituirá la stessa cosa di prima.

Quindi, tornando all'interrogativo posto in partenza:

```
>> for i in list: print i,
```

é molto piú professionale e intelligente che

```
>> for k in range(len(list)): print list[k],
```

13.5 ConfigParser per i propri config files

by Luigi Pantano aka bornFreeThinker

LICENSE: All the Python code contained in this recipe is released under GNU General Public License version 2. You can find a copy of the license here:

<http://www.gnu.org/licenses/gpl.html>

Avendo la necessità di utilizzare tale libreria per il mio Password Manager ho creato una semplice classe che permette di scrivere e di leggere le opzioni del mio programma da un file apposito. Premetto che la libreria ha molti più comandi di quelli che ho utilizzato quindi consiglio di approfondire l'argomento leggendo la reference manual di python.

Dopo questa breve ma doverosa premessa mettiamo in luce le funzionalità della classe in questione.

La funzione `__init__` ha il compito di inizializzare e settare la variabile `self.config_file_name` che conterrà il nome del file di configurazione e l'oggetto `self.cfg`, inoltre ha il compito di leggere nel caso esista il contenuto del file.

Nota: se non leggesi inizialmente il contenuto del file, la classe sovrascriverebbe ogni volta il file con nuovi dati perdendo tutte le precedenti opzioni settate.

Nella funzione `setValue` inizialmente verifico l'esistenza della sezione "section" poi creo la chiave "key" di valore "value" ed infine salvo i dati all'interno del file.

Nota: avendo constatato che la libreria è case-sensitive (sensibile al caso) per quanto riguarda la lettura del nome delle sezioni ho preferito "rimpicciolire il carattere" della "section" per evitare futuri problemi. Vediamo inoltre che ho impostato su string i valori di "key" e "value" ciò permetterà di avere, nel caso ve ne sia l'esigenza, come nomi di chiavi anche numeri.

La funzione `getValue` se quanto detto sopra risulta chiaro è assolutamente banale il suo utilizzo. Tengo solo a far notare la possibilità di ottenere vari tipi di risultati: stringa, integer, boolean...

Spero che qualunque dubbio abbiate fin ora venga perfettamente colmato dal piccolo esempio esplicativo che ho allegato.

```

>> import ConfigParser
>> class ConfigFile:
>>     def __init__(self, file_name):
>>         '''Inizializza la classe settando il nome de file che conterrà
la configurazione del programma'''
>>         self.config_file_name = file_name
>>         self.cfg = ConfigParser.ConfigParser()
>>         self.cfg.read(self.config_file_name) # legge il file nel
caso esista
>>     def setValue(self, section, key, value):
>>         '''Aggiunge ad una sezione section una chiave key di valore
value'''
>>         section = section.lower()
>>         if not self.cfg.has_section(section): self.cfg.add_section(section)
>>         self.cfg.set(section, str(key), str(value))
>>         self.cfg.write(open(self.config_file_name, 'w'))
>>     def getValue(self, section, key, value_type=0):
>>         '''Restituisce il valore di una chiave key contenuta nella
sezione section value_type: 0=String (default), 1=Int , 2=Float,
3=Boolean'''
>>         section = section.lower()
>>         key = str(key)
>>         if value_type == 0: return self.cfg.get(section, key)
>>         elif value_type == 1: return self.cfg.getint(section, key)
>>         elif value_type == 2: return self.cfg.getfloat(section,
key)
>>         else: return self.cfg.getboolean(section, key)
>> # Esempio
>> c = ConfigFile('esempio.cfg')
>> c.setValue('setup', 'path', '/home/bornfreethinker/')
>> c.setValue('setup', 'run_on_startup', 1)
>> c.setValue('window', '1', 1.4)
>> print 'Verifica dati:'
>> print 'PATH =', c.getValue('setup', 'path')
>> print 'RunOnStarup =', c.getValue('setup', 'run_on_startup',
3)
>> print '1 =', c.getValue('window', '1', 2)

```

13.6 Come identificare il Sistema Operativo

by Luigi Pantano aka bornFreeThinker

Che Python sia un linguaggio multiplatforma é piú che risaputo, infatti é proprio questo uno dei suoi tanti vantaggi che rendono questo linguaggio di scripting ottimo per ogni situazione.

Vista la portabilitá del codice a volte si presenta la necessita di effettuare degli adattamenti ai nostri script in base all'OS sul quale lo si utilizza, adattamenti dovuti al tipo di versione del sistema operativo che usiamo Windows 98, XP, Linux 2.6, Linux 2.4, Mac OSX, ecc... oppure dovuti alla versione di Python che é installata sulla macchina. Quindi é molto importante sapere questo tipo di informazioni.

Per risolvere questi problemi basterá utilizzare la libreria **platform** nel seguente modo:

```
>> from platform import *
>> print 'Architettura:', architecture()[0]
Architettura: 32bit
>> print 'Tipo macchina:', machine()
Tipo macchina: i686
>> print 'Nome di rete del computer:', node()
Nome di rete del computer: localhost
>> print 'Specifica:', platform()
Specifica: Linux-2.6.8-1-386-i686-with-debian-3.1
>> print 'Sistema operativo:', system()
Sistema operativo: Linux
>> print 'Versione:', release(), version()
Versione: 2.6.8-1-386 #1 Thu Nov 25 04:24:08 UTC 2004
>> print 'Versione Python:', python_version()
Versione Python: 2.3.5
```

Nota: i comandi sopra usati sono multiplatforma ma nel caso in cui avessimo necessita di informazioni piú specifiche su un dato sistema operativo troveremo nella guida del Python comandi appositi quale ad esempio "win32_ver" che permette di ricavare informazioni supplementari sulla versione dal Windows Registry.

13.7 Commenti su piú righe

by Luigi Pantano aka bornFreeThinker

A volte si presenta la necessità di inserire un commento su piú di una riga il problema si risolve utilizzando il triplice apice per l'apertura e la chiusura del commento.

```
>>> '''Questo é
>>> un
>>> commento
>>> multiriga'''
>>> print 'funziona!'
funziona!
```

Nota: Questo trucchetto può essere anche usato per creare stringhe su piú righe oppure un altro utilizzo particolarmente utile é quello di inserire un commento di questo tipo subito dopo la dichiarazione di una vostra funzione in modo tale che questa venga commentata durante l'operazione di autocompletamento che troviamo di solito in ambienti di sviluppo avanzati.

13.8 Questione di switch

by Luigi Pantano aka bornFreeThinker

L'altro giorno parlando con un amico si lamentava del fatto che in Python non esiste il classico costrutto swich come in altri linguaggi come C, Pascal, Visual Basic, ecc. . .

Effettivamente devo dargli ragione; il costrutto non esiste ma in compenso esiste uno strumento che ne é l'equivalente: i dizionari.

Ecco come al solito l'esempio (puramente didattico) chiarificatore:

```
>>> def a():
>>>     return 'la funzione a() é stata eseguita'
>>> def b():
>>>     return 'la funzione b() é stata eseguita'
>>> # definiamo lo swich con un dizionario
>>> c = {1231: a, 54: b}
>>> # choise rappresenta il termine da confrontare nel vostro switch
>>> choise = 54
>>> print c[choise]()
```

la funzione `b()` é stata eseguita

Come volevasi dimostrare il costrutto `switch` esiste anche in Python, solo che é piú semplice ed elegante.

Capitolo 14

Note finali

Python Tips & Tricks é stato offerto da *Italian Python User Group*, un gruppo italiano di programmatori Python.

Speriamo che questo progetto non sia stato vano e che abbia aiutato in qualche modo il lettore.

Per dubbi, suggerimenti e segnalazioni di vario tipo riguardo il CookBook e/o l'IPUG, visitate il sito <http://www.italianpug.org>.

Capitolo 15

Autori e contribuenti

Ecco qui di seguito in ordine rigorosamente alfabetico tutti coloro che hanno partecipato alla realizzazione di questo cookbook:

Blackbird

Luigi Pantano aka bornFreeThinker

Fred1

Lethalman

Mozako

scitrek

slash

The_One

In ordine alfabetico i vari contributi offerti a sostegno del progetto:

Enrico "Gabubbi" Manfredi . . . Licenza del documento

Capitolo 16

Indice analitico

Di seguito verrà riportato l'**indice analitico**, ovvero un insieme di parole utilizzate all'interno del cookbook che potranno essere facilmente rintracciate all'interno dello stesso.

In questo documento la maggior parte degli argomenti ha delle parole chiave su cui vengono basati gli stessi tips (ad es. nomi di librerie, keywords, funzioni, ecc...), le quali parole vengono inserite nell'indice analitico con a fianco la referenza della pagina in cui sono collocate.

L'indice analitico può quindi essere utilizzato come metodo di ricerca per accedere ai tips in modo sistematico e diretto ai fini di trovare gli argomenti di cui si vuole venire a conoscenza.

Indice analitico

`**`, 11
`.pyw`, 62
`__add__`, 9
`__doc__`, 7
`__import__`, 30
`__lshift__`, 68
`__new__`, 28
`__sub__`, 9
`\a`, 52
approssimare, 12
BaseHTTPServer, 39
bool, 9
bz2, 54
chr, 9
class factory, 28
code, 71
ConfigParser, 74
console, 32
cookbook, 5
Crypto, 57
daemon, 63
`dir()`, 7
dis, 71
docstring, 8
`eval()`, 70
`file.seek()`, 34
`file.tell()`, 34
filter, 24
for, 73
fpformat, 13
generator expression, 22
generatore, 22
global, 25
`globals()`, 26
GTK threads, 47
`gtk.Alignment`, 49
`gtk.threads_enter()`, 47
`gtk.threads_init()`, 47
`gtk.threads_leave()`, 47
`help()`, 7
inheritance, 27
ipug, 5
lambda, 23
`locals()`, 26
`math.pow()`, 11
ord, 8
`os.fork()`, 63
`os.popen()`, 33
platform, 76
poll, 35
profile, 65
`profile.run()`, 65
`property()`, 31
Queue, 38
random, 11, 32
`random.randint()`, 12

random.random(), 11
random.randrange(), 12
reload, 30
reversed(), 20, 21

select, 35
select flags, 36
shelve, 45
socket.AF_UNIX, 37
string.replace(), 18
string.Template, 17
switch, 77
sys.settrace(), 66
sys.stdout.write, 32

template, 17

urllib.urlretrieve(), 43

vars(), 27

winsound, 52

yield, 22

zlib, 55

Appendice A

The GNU General Public License

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.

51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions

translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.
You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
 - (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
 - (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
 - (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy

the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

one line to give the program’s name and a brief idea of what it does.

Copyright (C) yyyy name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail. If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) yyyy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.

This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the
program
'Gnomovision' (which makes passes at compilers) written by James
Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.