Ginnastica psycotica o estensioni alla ~~sbarra~~ DBAPI-2.0

di

Federico Di Gregorio
<fog@initd.org>

- PyConUno – Firenze, 9/6/2007

Alcune semplici regole

Nel 1999 iniziammo a scrivere psycopg seguendo 3 semplici regole:

- «Tienilo semplice, stupido!»;oppure
- Generalizzazione non è sinonimo di eleganza
- L'eleganza e inutile se porta ad inefficienza e malfunzionamenti
- DBAPI, DBAPI e ancora DBAPI

Dopo 8 anni psycopg 2.0 è compatibile al 95% con psycopg 0.4, nonostante le innumerevoli migliorie.

Estensione != rivoluzione!

Alcune lezioni imparate estendendo la DBAPI per venire incontro alle caratteristiche di PostgreSQL ed alle necessità degli utenti:

- Pensa due volte, scrivi una volta sola
- Usa gli errori a tuo vantaggio
- Se l'utente lo vuole, convincilo a scriverselo
- Sfrutta gli standard; ovvero: non sei l'unico programmatore con un cervello!

Se solo...

«Se solo ci avessi pensato prima psycopg 2.0 oggi sarebbe...»

```
import psycopg
conn = psycopg.connect(“dbname=test”)
conn.autocommit()
curs = conn.cursor()
curs.execute(“SELECT blah FROM boh”)
print curs.fetchall()
```

...compatibile al 100%

...compatibile al 100% con quello che era psycopg 1.

```
import psycopg2
import psycopg2.extensions as pe
conn = psycopg2.connect("dbname=test")
conn.set_isolation_level(
    pe.ISOLATION_LEVEL_AUTOCOMMIT)
curs.execute("SELECT blah FROM boh")
print curs.fetchall()
```

SDI

Ogni programmatore almeno una volta nella vita viene colpito dalla Sindrome di Deficienza Impulsiva. «Perché non mi sono posto prima queste domande?»

- non è che oltre ad AUTOCOMMIT e SERIALIZABLE PostgreSQL supporterà altro?
- autocommit() senza parametro cosa accidenti significa?
- se l'SQL lo chiama *isolation level* perché io sto per chiamarlo *autocommit*?

Gli errori degli “altri”

Chi di voi abbia mai usato codice come questo alzi la mano!

```
import psycopg2
from psycopg2 import NUMBER, STRING, ...
curs.execute(“SELECT .. WHERE nick = 'C8E'”)
if curs.description[0][1] == NUMBER:
    # Forse nel campo 0 c'è il numero di
    # ricette che C8E conosce?
```

Da PostgreSQL a Python

In psycopg i TypeObject della DBAPI sono responsabili delle conversioni dei tipi da PostgreSQL a Python.

```
# No import? No party!
def magritte(data, curs):
    return “Ceçi n'est pas un(e) “ + data
NESTPAS = pe.new_type(
    (textoid,), “NESTPAS”, magritte)
pe.register_type(NESTPAS)
```

Uroboro

Ma dove lo prendiamo l'identificatore del tipo da passare a `new_type()`? Proprio da quella `.description` con la quale potremo confrontare poi il nostro nuovo “tipo”:

```
# No import? No party!
curs.execute(“SELECT 'xxx'::text AS uroboro”)
textoid = curs.description[0][1]
# Poi, al secondo passaggio...
curs.description[0][1] == NESTPAS -> True
```

«Ciao, sono M.P. e...»

Spiegare allo Joe User di turno, sulla ML, come fare quello che vuole utilizzando quanto già disponibile... 80€, con PsycoCard.

Implementare quello che Joe User chiede in ML perché in effetti non c'è modo di farlo altrimenti... 400€, con PsycoCard.

Convincere Joe User che implementarlo da se è semplice e veloce... non ha prezzo!

Su{b,r}-classare

```
# No import? No party!
class DictCursor(psycopg2.cursor):
    “A cursor that index columns by name.”
    # Seguono circa 30-40 righe abbastanza
    # banali per l'implementazione.

curs = conn.cursor(factory=DictCursor)
conn = psycopg.connect(database=”test”,
    cursor_factory=DictCursor)
curs = conn.cursor()
```

Da Python a PostgreSQL

La conversione di un arbitrario tipo Python ad un formato comprensibile a PostgreSQL non è banale.

- I tipi Python sono tantissimi e PostgreSQL supporta nuovi tipi definiti dagli utenti
- Le conversioni utilizzate di default da psycopg non vanno bene per tutti
- L'override di __str__ non è poi una grande idea (fog colpito da SDI nel 2000 circa)

Sfruttare le idee degli altri

Il PEP 246 spiega come “adattare” un tipo a supportare una certa interfaccia.

Se l'interfaccia è «Ti spiace metterti in formato comprensibile all'interprete, possibilmente evitando SQL-injection?» il problema di convertire i tipi da Python a PostgreSQL è già risolto.

Basta scrivere `adapt()` ed un registry degli *adapter* in 100 righe di C o poco più..

Ma? Perché?

Questa slide è lasciata intenzionalmente vuota, in attesa di ricevere le due (ovvia e meno ovvia) domande dalla platea...

Le domande

Ma non potevi usare una delle implementazioni già esistenti?

Perché hai scelto di farlo in C? Questo è il classico caso in cui il Python è meglio e del resto psycopg già chiama codice Python, no?

Le risposte

Non sempre un'implementazione esistente è la scelta migliore:

- aggiunge una dipendenza
- può essere lenta o contenere errori o non essere adeguata anche se scritta benissimo
- se l'API è ben progettata è sempre e comunque possibile utilizzare una differente implementazione in futuro
- implementare il PEP 246 in 100 righe di C è comunque interessante

Un esempio

```
class SQL_IN(object):
    def __init__(self, seq):
        self._seq = seq
    def prepare(self, conn):
        self._conn = conn
    def getquoted(self):
        po = [adapt(o) for o in self._seq]
        for o in po: o.prepare(self._conn)
        qo = [o.getquoted() for o in po]
        return '('+', '.join(qo)+')'
```

Riferimenti

psycopg 2.0.6 “PyConUno”
è stato rilasciato questa notte ;)

- http://initd.org/pub/software/psycopg
- http://initd.org/tracker/psycopg
- http://lists.initd.org/
- fog@initd.org