Basi di Dati
Modulo Laboratorio

Lezione 9: Accesso a una base di dati PostgreSQL da Python

DR. SARA MIGLIORINI

Introduzione

- DB-API v2.0 è la Application Program Interface (API) ufficiale (Python Enhancement Proposals (PEP) 249,
- https://www.python.org/dev/peps/pep-0249) che descrive come un modulo Python deve accedere a una base di dati esterna.
- Diversi gruppi di sviluppo hanno reso disponibili moduli (librerie) DB-API per diversi tipi di DBMS.
- Alla pagina https://wiki.python.org/moin/DatabaseInterfaces c'è l'elenco aggiornato dei moduli disponibili.
- Per PostgreSQL ci sono più di 10 (2022-05-26) implementazioni diverse di DB-API v2.0.
- In questa lezione si introduce DB-API v2.0 considerando l'implementazione psycopg2 (http://initd.org/psycopg/).

Introduzione a psycopg2

- psycopg2 è una libreria scritta quasi completamente in C e implementa DB-API v2.0 mascherando la libreria C ufficiale libpq del gruppo di PostgreSQL (https://www.postgresql.org/docs/current/libpq.html)
- Implementa i cursori lato client e lato server, comunicazione asincrone, le notifiche e il comando COPY
- Molti tipi di dati Python sono supportati e mappati nei tipi di dati PostgreSQL.
- La mappatura dei tipi di dati può essere personalizzata in modo semplice.

Fondamenti di DB-API v2.0: Connection

- L'accesso a un database avviene tramite un oggetto di tipo Connection. Il metodo connect(...) accetta i parametri necessari per la connessione e ritorna un oggetto Connection.

```
connector = psycopg2.connect( host="dbserver.scienze.univr.it", database="db0", \
user="user0", password="xxx" )
```

Fondamenti di DB-API v2.0: Connection

Classe Connection – metodi principali:

1. cursor(): ritorna un cursore della base di dati. Un oggetto cursore permette di inviare comandi SQL al DBMS e di accedere al risultato del comando restituito dal DMBS.
2. commit(): registra la transazione corrente. Attenzione! Normalmente una connessione apre una transazione al primo invio di comandi. Se non si esegue un commit() prima di chiudere, tutte le eventuali modifiche/inserimenti vengono persi.
3. rollback(): abortisce la transazione corrente.
4. close(): chiude la connessione corrente. Implica un rollback() automatico delle operazioni non registrate.

Fondamenti di DB-API v2.0: Connection

5. autocommit: proprietà r/w. Se True, ogni comando inviato è una transazione isolata. Se False (default) il primo comando inviato inizia una transazione, che deve essere chiusa con commit() o rollback() .
6. readonly: proprietà r/w. Se True, nella sessione non si possono inviare comandi di modifica dati. Il default è False.
7. isolation_level: proprietà r/w. Modifica il livello di isolamento per la prossima transazione. Valori leciti: 'READ UNCOMMITTED', 'READ COMMITTED', 'REPEATABLE READ', 'SERIALIZABLE', 'DEFAULT'. Meglio assegnare questa variabile subito dopo la creazione della connessione.

```
connector = psycopg2.connect(...)
connector.isolation_level = 'REPEATABLE READ'
```

Fondamenti di DB-API v2.0: Cursore

- Un cursore gestisce l'interazione con la base di dati: mediante un cursore è possibile inviare una comando SQL e accedere all'esito e ai dati di risposta del comando.

Classe Cursor – metodi principali:

1. execute(comando, parametri): prepara ed esegue un 'comando' SQL usando i 'parametri'. Parametri devono essere passati come tupla o come dict. Il comando ritorna None. Eventuali risultati di query si devono recuperare con il fetch*().

```
cur.execute( "CREATE TABLE test( id SERIAL PRIMARY KEY, num integer, data varchar)" )
cur.execute( "INSERT INTO test (num, data) VALUES (%s, %s)", (100, "abc'def"))
# psyconpg2 fa le conversioni!
```

Fondamenti di DB-API v2.0: Cursore

2. executemany(comando, parametri): prepara ed esegue un 'comando' SQL per ciascun valore presente nella lista 'parametri'.

```
cur. executemany( "INSERT INTO test (num, data ) VALUES (%s, %s)",
    [(100, "abc'def"), (None , 'dada'), (42 , 'bar')]
)
```

NOTA

Per come è attualmente implementato, executemany() è meno efficiente di un ciclo for con execute() o, meglio ancora, di un unico insert con più tuple:

```
cur.execute( "INSERT INTO test (num, data) VALUES (%s, %s), (%s, %s), (%s, %s)",
    (100, "abc'def", None, 'dada', 42, 'bar'))
```

Fondamenti di DB-API v2.0: Cursore

cursor.execute* accetta %s come indicatore di posizione parametro. La conversione dal tipo Python al dominio SQL è automatica per tutti i tipi fondamentali.

```
cur. execute( "INSERT INTO test1 (id , date_val , item ) VALUES \
(%s, %s, %s)", (42 , datetime.date(2005 , 11, 18), "O'Reilly" ))

è convertita in SQL
INSERT INTO test1( id , date_val , item ) VALUES ( 42 , '2005 -11 -18 ', 'O''Reilly' );
```

Fondamenti di DB-API v2.0: Cursore

Python	PostgreSQL	Python	PostgreSQL	Python	PostgreSQL
None	NULL	bool	BOOL	float	REAL DOUBLE
int	SMALLINT INTEGER bigint	Decimal	NUMERIC	str	VARCHAR TEXT
date	DATE	time	TIME timez	datetime	TIMESTAMP timestamptz

Maggiori dettagli: http://initd.org/psycopg/docs/usage.html

Fondamenti di DB-API v2.0: Cursore

Tipico errore da principiante: SQL Injection

Alcuni pensano sia più efficiente e facile scrivere

```
cur.execute("SELECT 1 FROM users WHERE name='" + user + "' AND pw='" + pw + "')
```

Anziché

```
cur.execute("SELECT 1 FROM users WHERE name=%s AND pw=%s", (user, pw))
```

dove user e pw sono letti da input.
Altri, più intelligenti, possono sfruttare questa ingenuità per fare altro:

Fondamenti di DB-API v2.0: Cursore

Tipico errore da principiante: SQL Injection

Assegnando:

- ' OR TRUE -- a user
- un qualsiasi carattere a pw

il risultato è

SELECT 1 FROM users WHERE name='' OR TRUE --' AND pw='a'

Fondamenti di DB-API v2.0: Cursore

Tipico errore da principiante: SQL Injection

```
cur.execute("SELECT 1 FROM users WHERE name='" + user + "' AND pw='" + pw + "'")
```

Assegnando:

- `'; DROP TABLE users CASCADE; --` a user
- un qualsiasi carattere a pw

il risultato è

```
SELECT 1 FROM users WHERE name='';
DROP TABLE users CASCADE; --' AND pw='a'
```

Fondamenti di DB-API v2.0: Cursore

3. fetchone(): ritorna una tupla della tabella risultato. Si può usare dopo un execute("SELECT ..."). Se non ci sono tuple, ritorna None.

```
>>> cur. execute( "SELECT * FROM test WHERE id = %s", (3, ) )
>>> cur. fetchone()
(3, 42, 'bar')
```

Fondamenti di DB-API v2.0: Cursore

4. fetchmany(<numero>) : ritorna una lista di tuple della tabella risultato di lunghezza max <numero>. Si può usare dopo un execute("SELECT ..."). Se non ci sono tuple, ritorna una lista vuota.

```
>>> cur.execute( "SELECT * FROM test WHERE id < %s", (4 ,) )
>>> cur. fetchmany(3)
[(1 , 100 , " abc 'def "), (2, None , 'dada '), (3, 42, 'bar ')]
>>> cur. fetchmany(2)
[]
```

Fondamenti di DB-API v2.0: Cursore

5. Dopo un execute("SELECT ..."), il cursore è un iterabile sulla tabella risultato. È possibile quindi accedere alle tuple del risultato anche con un ciclo.

```
>>> cur. execute( "SELECT * FROM test WHERE id < %s", (4, ) )
>>> for record in cur:
... print (record, end=", ")
(1, 100 , " abc 'def "), (2, None , 'dada '), (3, 42, 'bar '),
```

Fondamenti di DB-API v2.0: Cursore

6. rowcount: di sola lettura, = numero di righe prodotte da ultimo comando. -1 indica che non è possibile determinare il valore.
7. statusmessage: di sola lettura, = messaggio ritornato dall'ultimo comando eseguito.

```
>>> cur. execute( "INSERT INTO test (num , data ) VALUES (%s, %s)", (42 , 'bar'))
>>> cur. statusmessage
'INSERT 0 1'
```

Schema per usare psycopg2

- psycopg2 nons fa parte della distribuzione standard.
- Python3 ha un meccanismo di installazione semplice (pip) dei moduli che sono registrati presso Python Packaging Index.
- pip può installare a livello di sistema (richiede diritti amministratore) o a livello utente. Dettagli: https://docs.python.org/3/installing.
- Installazione Psyconpg2 su Ubuntu Desktop:

1. Si assume che Python3 sia già installato come pure il programma pip3. (Dalla versione Ptyhon 3.5, pip è già presente nella distribuzione).
2. Da una shell, si avvia l'installazione di Psycopg2 con il comando: pip3 install --user psycopg2 psycopg2-binary
3. Dopo qualche compilazione, il modulo viene salvato in ~/.local/lib/python3.x/site-packages/ ed è disponibile automaticamente in ogni sessione Python3.

Schema per usare Psycopg2

- Lo schema tipico di un modulo Python che comunica con un DBMS PostgreSQL via Psycopg2 deve:

1. aprire una connessione tramite conn = psycopg2.connect(...),
2. eventualmente modificare le proprietà di livello di isolamento, autocommit, readonly.
3. creare un cursore tramite cur = conn.cursor(),
4. eseguire le operazioni previste,
5. se la sessione non è in autocommit, eseguire un conn.commit() se si sono dati comandi SQL di aggiornamento (registra le modifiche) (o conn.rollback() per annullare),
6. chiudere il cursore, cur.close(), e la connessione, conn.close().

Schema per usare Psycopg2

- Dalla versione 2.5 della libreria, la gestione dei close e dei commit è semplificata se si usa il costrutto with:
- Quando si usa una connessione con il with, all'uscita del blocco viene fatto un commit automatico e la connessione non viene chiusa.
- Quando si usa/crea un cursore con il with, all'uscita del blocco viene fatto un close automatico del cursore.

Schema per usare Psycopg2

1. conn = psycopg2.connect (...)
2. with conn:
3. __with conn.cursor () as cur1:
4. __ __...
5. __print (" Qui cur1 è stato chiuso ")
6. print ("Qui è stato fatto solo un commit : conn è ancora aperta!")
7. with conn :
8. __with conn . cursor () as cur2 :
9. __ __...
10. __print (" Qui cur2 è stato chiuso ")
11. print("Qui è stato fatto un commit: conn è ancora aperta!")
12. conn.close()

Connessioni e Cursori

- Si deve porre attenzioni alla combinazione di cursori sulla medesima connessione
- Aprire una connessione costa in tempo (e spazio). Meglio aprire/chiudere poche connessioni in un'esecuzione.
- Con un oggetto connessione si possono creare più cursori. Questi cursori condividono la connessione.
- Psycopg2 garantisce solo che le istruzioni inviate dai cursori vengono sequenzializzate.
- Quindi non si possono gestire transazioni concorrenti usando diversi cursori sulla medesima connessione.
- Regola pratica: usare più cursori sulla medesima connessione quando si fanno transazioni in auto-commit o solo transazioni di sola lettura.

Schema per usare Psycopg2

Esempio di modulo basato su Psycopg2 (1/5)

```
"""
Gestione semplice tabella Spese su PostgreSQL .
Manca la gestione delle eccezioni !
"""


from datetime import date
from decimal import Decimal

import psycopg2

# Il seguente import definisce delle variabili con dati sensibili come le
password. \
Metodo semplice per isolare in un solo file tali parametri
from myAppConfig import myHost, myDatabase, myUser, myPas
```

Schema per usare Psycopg2

Esempio di modulo basato su Psycopg2 (2/5)

```
connessione = psycopg2.connect ( host=myHost , \
database=myDatabase, user=myUser, password=myPasswd )
with connessione :
__with connessione.cursor () as cursore :
__ __cursore.execute (
__ __ __"""CREATE TABLE IF NOT EXISTS Spese (
__ __ __id SERIAL PRIMARY KEY,
__ __ __data DATE NOT NULL,
__ __ __voce VARCHAR NOT NULL,
__ __ __importo NUMERIC NOT NULL
__ __ __)"""
__ __ )
print( 'Esito della creazione della tabella Spese : {:s}\n Eventuali notifiche : \
{:s}'.format( cursore . statusmessage , connessione . notices [ -1]))
```

Schema per usare Psycopg2

Esempio di modulo basato su Psycopg2 (3/5)

```
__ __cursore.execute( """SELECT count (*) FROM Spese""" )
__ __numeroRighe = cursore.fetchone()[0]
__ __if numeroRighe == 0:
__ __ __cursore.execute( """INSERT INTO \
Spese(data, voce, importo ) VALUES (%s, %s, %s),
__ __ __ __(%s, %s, %s),
__ __ __ __(%s, %s, %s),
__ __ __ __(%s, %s, %s)""" ,
__ __ __ __( date(2016, 2, 24), "Stipendio", Decimal( "0.1" ),
__ __ __ __date(2016, 2, 24), "Stipendio 'Bis'", Decimal( "0.1" ),
__ __ __ __date(2016, 2, 24), "Stipendio 'Tris'", Decimal( "0.1" ),
__ __ __ __date (2016 ,2 ,27) , " Affitto ", Decimal (" -0.3"))
__ __ __)
__ __ __print ("Esito dell' inserimento delle 4 tuple : \
{:s}".format( cursore.statusmessage ))
```

Schema per usare Psycopg2

Esempio di modulo basato su Psycopg2 (4/5)

```
__ __else :
__ __ __print ( "La tabella è già presente con delle tuple e \
quindi nessuna tupla è stata aggiunta.“ )
__connessione.commit () # Tutti i dati sono salvati

__with connessione.cursor () as lettore:
__ __lettore.execute( """SELECT id, data, voce, importo \
FROM Spese""“ )
__ __print( "Esito della selezione di tutte le tuple : \
{:s}".format( cursore.statusmessage ))
__ __print( '=' * 55 )
__ __patternRiga = "| {: >2s} | {:10 s} | {: <20s} | {: >10s} |"
```

Schema per usare Psycopg2

Esempio di modulo basato su Psycopg2 (5/5)

```
___ ___print( patternRiga.format( "N", "Data", "Voce", "Importo“ ))
___ ___print ('-' * 55)
___ ___tot = Decimal ("0")
___ ___patternRiga = "| {: >2d} | {:10 s} | {: <20s} | {: >10.2 f} |"
___ ___for tupla in lettore:
___ ___ ___print( patternRiga.format( tupla[0], tupla[1].isoformat(), tupla[2],
tupla[3] ))
___ ___ ___tot += tupla [3]

___ ___print( '-' * 55 ) # A questo punto la tabella è stampata in output
connessione.close()
print( "{: >40s} {:10.2 f}".format( "Totale", tot ))
```

Schema per usare Psycopg2

Esempio di modulo basato su Psycopg2 - Esecuzione

Esito di una esecuzione:
Esito creazione tabella : CREATE TABLE
Esito inserimento tabella : INSERT 0 4

```
============================================
| N | Data          | Voce              | Importo |
--------------------------------------------
| 1 | 2016 -02 -24 | Stipendio         |    0.10 |
| 2 | 2016 -02 -24 | Stipendio "Bis"   |    0.10 |
| 3 | 2016 -02 -24 | Stipendio "Tris" |    0.10 |
| 4 | 2016 -02 -27 | Affitto           |   -0.30 |
--------------------------------------------
Totale 0.00
```