Database Access

Python 3.x 3.0–3.6 , Python 2.x 2.7

Improvements requested:

Examples

SQLite

SQLite is a lightweight, disk-based database. Since it does not require a separate database server, it is often used for prototyping or for small applications that are often used by a single user or by one user at a given time.

```
import sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

c.execute("CREATE TABLE user (name text, age integer)")

c.execute("INSERT INTO user VALUES ('User A', 42)")
c.execute("INSERT INTO user VALUES ('User B', 43)")

conn.commit()

c.execute("SELECT * FROM user")
print(c.fetchall())

conn.close()
```

The code above connects to the database stored in the file named `users.db`, creating the file first if it doesn't already exist. You can interact with the database via SQL statements.

The result of this example should be:

```
[(u'User A', 42), (u'User B', 43)]
```

To avoid SQL injection, it is advised to use the following format when inserting values (especially if handled by user) into the database.

```
c.execute("INSERT INTO user VALUES(?, ?)", ('User A', 42))
```

Accessing MySQL database using MySQLdb

The first thing you need to do is create a connection to the database using the connect method. After that, you will need a cursor that will operate with that connection.

Use the execute method of the cursor to interact with the database, and every once in a while, commit the changes using the commit method of the connection object.

Once everything is done, don't forget to close the cursor and the connection.

Here is a Dbconnect class with everything you'll need.

```
import MySQLdb

class Dbconnect(object):

    def __init__(self):

        self.dbconection = MySQLdb.connect(host='host_example',
                                           port=int('port_example'),
                                           user='user_example',
                                           passwd='pass_example',
                                           db='schema_example')
        self.dbcursor = self.dbconection.cursor()

    def commit_db(self):
        self.dbconection.commit()

    def close_db(self):
        self.dbcursor.close()
        self.dbconection.close()
```

Interacting with the database is simple. After creating the object, just use the execute method.

```
db = Dbconnect()
```

```
db.dbcursor.execute('SELECT * FROM %s' % 'table_example')
```

If you want to call a stored procedure, use the following syntax. Note that the parameters list is optional.

```
db = Dbconnect()
db.callproc('stored_procedure_name', [parameters] )
```

After the query is done, you can access the results multiple ways. The cursor object is a generator that can fetch all the results or be looped.

```
results = db.dbcursor.fetchall()
for individual_row in results:
    first_field = individual_row[0]
```

If you want a loop using directly the generator:

```
for individual_row in db.dbcursor:
    first_field = individual_row[0]
```

If you want to commit changes to the database:

```
db.commit_db()
```

If you want to close the cursor and the connection:

```
db.close_db()
```

Connection

Creating a connection

According to PEP 249, the connection to a database should be established using a `connect()` constructor, which returns a `Connection` object. The arguments for this constructor are database dependent. Refer to the database specific topics for the relevant arguments.

```
import MyDBAPI

con = MyDBAPI.connect(*database_dependent_args)
```

This connection object has four methods:

1: close

```
con.close()
```

Closes the connection instantly. Note that the connection is automatically closed if the `Connection.__del___` method is called. Any pending transactions will implicitely be rolled back.

2: commit

```
con.commit()
```

Commits any pending transaction the to database.

3: rollback

```
con.rollback()
```

Rolls back to the start of any pending transaction. In other words: this cancels any non-committed transaction to the database.

4: cursor

```
cur = con.cursor()
```

Returns a `Cursor` object. This is used to do transactions on the database.

Oracle database

Pre-requisites:

- cx_Oracle package - See here for all versions

- cx_Oracle package - See here for all versions
- Oracle instant client - For Windows x64 , Linux x64

Setup:

- Install the cx_Oracle package as:

  `sudo rpm -i <YOUR_PACKAGE_FILENAME>`

- Extract the Oracle instant client and set environment variables as:

```
ORACLE_HOME=<PATH_TO_INSTANTCLIENT>
PATH=$ORACLE_HOME:$PATH
LD_LIBRARY_PATH=<PATH_TO_INSTANTCLIENT>:$LD_LIBRARY_PATH
```

Creating a connection:

```
import cx_Oracle

class OraExec(object):
    _db_connection = None
    _db_cur = None

    def __init__(self):
        self._db_connection =
            cx_Oracle.connect('<USERNAME>/<PASSWORD>@<HOSTNAME>:<PORT>/<SERVICE_NAME>')
        self._db_cur = self._db_connection.cursor()
```

Get database version:

```
ver = con.version.split(".")
print ver
```

Sample Output: ['12', '1', '0', '2', '0']

Execute query: SELECT

```
_db_cur.execute("select * from employees order by emp_id")
for result in _db_cur:
    print result
```

Output will be in Python tuples:

(10, 'SYSADMIN', 'IT-INFRA', 7)

(23, 'HR ASSOCIATE', 'HUMAN RESOURCES', 6)

Execute query: INSERT

```
_db_cur.execute("insert into employees(emp_id, title, dept, grade)
                values (31, 'MTS', 'ENGINEERING', 7)
_db_connection.commit()
```

When you perform insert/update/delete operations in an Oracle Database, the changes are only available within your session until `commit` is issued. When the updated data is committed to the database, it is then available to other users and sessions.

Execute query: INSERT using Bind variables

Reference

Bind variables enable you to re-execute statements with new values, without the overhead of re-parsing the statement. Bind variables improve code re-usability, and can reduce the risk of SQL Injection attacks.

```
rows = [ (1, "First" ),
     (2, "Second" ),
     (3, "Third" ) ]
_db_cur.bindarraysize = 3
_db_cur.setinputsizes(int, 10)
_db_cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)
_db_connection.commit()
```

Close connection:

```
_db_connection.close()
```

The close() method closes the connection. Any connections not explicitly closed will be automatically released when the script ends.

PostgreSQL Database access using psycopg2

psycopg2 is the most popular PostgreSQL database adapter that is both lightweight and efficient. It is the current implementation of the PostgreSQL adapter.

> Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection)

Establishing a connection to the database and creating a table

```
import psycopg2

# Establish a connection to the database.
# Replace parameter values with database credentials.
conn = pyscopg2.connect(database="testpython",
                        user="postgres",
                        host="localhost",
                        password="abc123",
                        port="5432")

# Create a cursor. The cursor allows you to execute database queries.
cur = conn.cursor()

# Create a table. Initialise the table name, the column names and data type.
cur.execute("""CREATE TABLE FRUITS (
                    id          INT ,
                    fruit_name  TEXT,
                    color       TEXT,
                    price       REAL
            )""")
conn.commit()
conn.close()
```

Inserting data into the table:

```
# After creating the table as shown above, insert values into it.
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Apples', 'green', 1.00)""")

cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
               VALUES (1, 'Bananas', 'yellow', 0.80)""")
```

Retrieving table data:

```
# Set up a query and execute it
cur.execute("""SELECT id, fruit_name, color, price
             FROM fruits""")

# Fetch the data
rows = cur.fetchall()

# Do stuff with the data
for row in rows:
    print "ID = {} ".format(row[0])
    print "FRUIT NAME = {}".format(row[1])
    print("COLOR = {}".format(row[2]))
    print("PRICE = {}".format(row[3]))
```

The output of the above would be:

```
ID = 1
NAME = Apples
COLOR = green
PRICE = 1.0

ID = 2
NAME = Bananas
COLOR = yellow
PRICE = 0.8
```

And so, there you go, you now know half of all you need to know about ***psycopg2*** ! :)

Syntax

Parameters

Remarks

Python can handle many different types of databases. For each of these types a different API exists. So encourage similarity between those different API's, PEP 249 has been introduced.

> This API has been defined to encourage similarity between the Python modules that are used to access databases. By doing this, we hope to achieve a consistency leading to more easily understood modules, code that is generally more portable across databases, and a broader reach of database connectivity from Python.
> PEP-249