Écrire des scripts Python avec DB-API

Paul Dubois
paul@kitebird.com

Cet article est une traduction de l'article Writing MySQL Scripts with Python DB-API en version 1.02 (du 2006-09-17), par Paul DuBois (paul@kitebird.com), traduction de Michel Casabianca (michel.casabianca@gmail.com).

Introduction

Python est un des langages Open Source les plus populaires, du fait de sa propre expressivité mais aussi de la variété des modules disponibles pour étendre ses capacités. Un des ces modules est DB-API qui, comme son nom l'indique, fournit une interface de programmation d'applications de bases de données. DB-API a été conçu pour être relativement indépendant des spécificités de chacun des moteurs de base de données, de manière à vous aider à écrire des scripts qui soient portables entre ces moteurs.

La conception de la DB-API est similaire de celle des modules DBI de Perl et Ruby, de la classe PHP PEAR DB et de l'API Java JDBC : elle utilise une architecture à deux couches dans laquelle le niveau supérieur fournit une interface abstraite, qui est similaire pour tous les moteurs de base de données supportés, et un niveau inférieur constitué de pilotes pour les différents moteurs qui gèrent les détails d'implémentation. Cela signifie, bien sûr, que pour utiliser DB-API afin d'écrire des scripts Python, il vous faut un pilote pour votre système de gestion de base de données. Pour MySQL, DB-API fournit un accès à la base de données au moyen du pilote MySQLdb. Ce document commence par expliciter l'installation du pilote (au cas où vous n'auriez pas déjà installé MySQLdb), puis détaille l'écriture de scripts DB-API.

Installation de MySQLdb

Pour écrire des scripts utilisant DB-API, Python lui-même doit être installé. C'est probablement déjà le cas si vous utilisez Linux, mais moins probable pour Windows. Des installeurs pour les deux plates-formes peuvent être trouvés sur le site web de Python (voir la section "Ressources" à la fin de ce document).

Vérifiez que votre version de Python soit *2.3.4* ou ultérieure et que le module MySQLdb soit installé. Vous pouvez vérifier ces deux pré-requis en lançant Python en mode interactif en ligne de commande (quelque chose comme `%` sous Unix ou `C:\>` sous Windows) :

```
% python
Python 2.4.3 (#1, Aug 29 2006, 14:45:33)
[GCC 3.4.6 (Gentoo 3.4.6-r1, ssp-3.4.5-1.0, pie-8.7.9)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import MySQLdb
```

Si votre version de Python est suffisamment récente et qu'aucune erreur ne survient lorsque vous exécutez l'instruction `import MySQLdb`, alors vous êtes prêt pour commencer à écrire des scripts d'accès à des bases de données et vous pouvez passer la section suivante. Cependant, si vous obtenez l'erreur suivante, vous devez d'abord vous procurer et installer MySQLdb :

```
>>> import MySQLdb
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ImportError: No module named MySQLdb
```

Pour obtenir MySQLdb, faites un tour dans la section "Ressources" pour savoir où vous procurer une distribution appropriée à votre système. Des binaires pré-compilés peuvent être disponibles pour votre plate-forme ou vous pouvez installer à partir des sources. Si vous utilisez une distribution binaire, installez la avec la procédure habituelle d'installation de binaires sur votre système. Pour compiler et installer à partir des sources, placez vous dans le répertoire de la distribution de MySQLdb et tapez la commande suivante (sous Unix, il probable que vous ayez à lance la deuxième commande en tant que `root` de manière à ce que les fichiers du pilote puissent être copiés dans votre installation de Python) :

```
% python setup.py build
% python setup.py install
```

Si vous rencontrez un problème, consultez le fichier *README* inclus dans la distribution de MySQLdb.

Un court script DB-API

Les scripts qui accèdent à MySQL au travers de DB-API en utilisant MySQLdb réalisent en général les étapes suivantes :

- Import du module MySQLdb.
- Ouverture d'une connexion au serveur MySQL.
- Exécution de requêtes et récupération des résultats.
- Fermeture de la connexion au serveur.

La suite de cette section présente un court script DB-API qui illustre ces étapes. Les sections suivantes présentent plus en détails les différents aspects de l'écriture de scripts.

Écriture du script

Utilisez un éditeur de texte pour créer un fichier *server_version.py* qui contient le script suivant. Ce script utilise MySQLdb pour interagir avec le serveur MySQL, bien que de manière bien rudimentaire; il se contente en effet de demander au serveur sa version sous forme d'une chaîne de caractères.

```
# server_version.py - récupère et affiche la version du serveur

import MySQLdb

conn = MySQLdb.connect (host = "localhost",
                        user = "testuser",
                        passwd = "testpass",
                        db = "test")
cursor = conn.cursor ()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone ()
print "server version:", row[0]
cursor.close ()
conn.close ()
```

L'instruction `import` indique à Python que le script a besoin d'utiliser le code du module MySQLdb. Cette instruction doit précéder toute tentative de connexion au serveur MySQL. Puis la connexion est établie en invoquant la méthode `connect()` du pilote MySQLdb avec les paramètres de connexion appropriés. Cela inclue le nom d'hôte du serveur, le nom d'utilisateur et le mot de passe de votre compte MySQL ainsi que le nom de la base de donnée à laquelle vous voulez vous connecter. La syntaxe de la liste d'arguments de la méthode `connect()` varie selon les pilotes. En ce qui concerne MySQLdb, il est possible d'utiliser le format `nom = valeur` qui permet de passer les arguments dans n'importe quel ordre. Le script *server_version.py* se connecte au serveur MySQL local pour accéder à la base de données `test` avec `testuser` comme nom d'utilisateur et `testpass` comme mot de passe :

```
conn = MySQLdb.connect (host = "localhost",
                        user = "testuser",
                        passwd = "testpass",
                        db = "test")
```

Si l'appel à `connect()` réussit, il renvoie un objet connexion qui sert pour les futures interactions avec MySQL. Si l'appel échoue, il lève une exception. Le script *server_version.py* ne gère pas l'exception, donc une erreur termine le script. La gestion des erreurs est couverte plus loin dans ce texte.

Après obtention de l'objet connexion, *server_version.py* invoque sa méthode `cursor()` pour créer un objet curseur pour traiter les requêtes. Le script utilise le curseur pour exécuter la requête `SELECT VERSION()` qui renvoie une chaîne contenant la version du serveur :

```
cursor = conn.cursor ()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone ()
print "version du serveur :", row[0]
cursor.close ()
```

La méthode `execute()` du curseur envoie la requête au serveur et `fetchone()` récupère une ligne sous forme d'un tuple. Pour la requête considérée, le tuple ne contient qu'une seule valeur affichée par le script. Si aucune ligne n'est disponible, la méthode `fetchone()` renvoie la valeur

`None`; *server_version.py* suppose allègrement que cela n'arrivera pas, une supposition que vous ne devriez pas faire. Nous gérerons ce cas dans des exemples de la suite du texte. Les objets curseur peuvent être utilisés pour exécuter plusieurs requêtes, mais *server_version.py* n'a plus besoin de `cursor` après avoir récupéré la version, donc il le ferme.

Pour finir, le script appelle la méthode `close()` de l'objet connexion pour fermer la connexion au serveur :

```
conn.close ()
```

Après cela, `conn` devient inopérant et ne devrait plus être utilisé pour accéder au serveur.

Exécution du script

Pour exécuter le script *server_version.py*, il vous faut invoquer Python depuis la ligne de commande et lui passer le nom du script. Vous devriez obtenir un résultat qui ressemble à cela :

```
% python server_version.py
version du serveur : 5.1.12-beta-log
```

Ceci indique que la version du serveur MySQL est 5.1.12, les suffixes `-beta` et `-log` nous renseignent sur la stabilité de la distribution et signalent que la journalisation des requêtes est activée. Vous pouvez rencontrer d'autres suffixes; par exemple, si vous le débogage est activé, vous verrez le suffixe `-debug`.

Il est possible de faire en sorte que le script soit lancé par son nom sans avoir à invoquer Python explicitement. Sous Unix, ajoutez une ligne `#!` au script pour indiquer le chemin absolu de l'interpréteur Python. Ceci indique au système quel programme doit exécuter le script. Par exemple, si dans votre système Python se trouve dans */usr/bin/python*, ajoutez la ligne suivante comme première ligne de votre script :

```
#!/usr/bin/python
```

Ensuite, utilisez le commande `chmod` pour rendre le script exécutable, vous pourrez ainsi lancer le script directement :

```
% chmod +x server_version.py
% ./server_version.py
```

Le `./` du début de ligne indique à votre interpréteur de commande que le script se trouve dans le répertoire courant. La plupart des comptes Unix sont configurés de manière à ne pas chercher les commandes dans le répertoire courant.

Sous Windows, la ligne `#!` n'est pas nécessaire (cependant, elle ne fait pas de mal, de sorte qu'il n'est pas nécessaire de l'enlever si vous écrivez un script sous Unix puis le déplacez sur une machine

Windows). À la place, vous pouvez configurer une association de fichier de manière à ce que les fichiers *.py* soient associés à Python. Au lieu d'utiliser `chmod` pour rendre le fichier exécutable, ouvrez l'item **Options des Dossiers** dans le panneau de contrôle et sélectionnez son onglet **Type de Fichier**. Les types de fichier vous permettent de configurer une association entre les fichiers les terminant par *.py* de manière à ce que Windows les exécute avec Python. Vous pouvez alors invoquer les scripts par leur nom :

```
C:\> server_version.py
```

Si vous installez ActiveState Python sous Windows, l'installeur ActiveState configure l'association automatiquement lors du processus d'installation.

Un script DB-API plus complet

Le script *server_version.py* présente un certain nombre de lacunes. Par exemple, il ne gère pas les exceptions et n'indique pas ce qui n'a pas fonctionné en cas d'erreur; de plus, il n'envisage pas la possibilité que la requête exécutée ne renvoie aucun résultat. Cette section présente comment gérer ces problèmes dans un script plus élaboré *animal.py*, qui utilise une table contenant des noms d'animaux et leur genre :

```
CREATE TABLE animal
(
  name     CHAR(40),
  category CHAR(40)
)
```

Si vous avez lu le document sur PEAR DB que l'on peut trouver sur le site Kitebird (voir la section ressources ci-dessous), vous avez peut être reconnu cette table et certaines des requêtes exécutées par *animal.py*; elles étaient utilisées dans ce document aussi.

Le script *animal.py* commence comme suit (la ligne `#!` permet d'exécuter ce script sous Unix) :

```
#!/usr/bin/python
# animal.py - crée la table animal et y récupère des informations

import sys
import MySQLdb
```

Comme *server_version.py*, le script importe `MySQLdb`, mais il importe aussi le module `sys` pour gérer les erreurs. En effet, *animal.py* utilise `sys.exit()` pour renvoyer `1` qui indique une issue anormale en cas d'erreur.

Gestion des erreurs

Après importation des modules nécessaires, *animal.py* établit une connexion au serveur en utilisant l'appel `connect()`. Pour gérer le cas d'une erreur de connexion (et afficher par exemple la cause

de l'échec), il est nécessaire d'intercepter les exceptions. Pour gérer les exceptions en Python, placez votre code entre une instruction `try` et un `except` qui contiendra le code de gestion des erreurs. La séquence résultante ressemble à cela :

```
try:
  conn = MySQLdb.connect (host = "localhost",
                          user = "testuser",
                          passwd = "testpass",
                          db = "test")
except MySQLdb.Error, e:
  print "Erreur %d : %s" % (e.args[0], e.args[1])
  sys.exit (1)
```

L'instruction `except` indique la classe d'une exception (`MySQLdb.Error` dans cet exemple) pour obtenir les informations spécifiques à l'erreur de la base de données que MySQL peut fournir, ainsi qu'une variable `e` dans laquelle enregistrer l'information. Si une erreur survient, MySQLdb rend cette informations disponible dans `e.args`, un tuple de deux éléments contenant le code numérique de l'erreur ainsi qu'une chaîne de caractères décrivant l'erreur. La clause `except` présentée dans cet exemple affiche ces deux valeurs puis termine le programme.

Toutes les requêtes de base de données peuvent être placées dans un bloc `try/except` semblable de manière à intercepter et afficher les erreurs. Pour rester concise, la discussion suivante ne montre pas le code de gestion des exceptions (le code complet de *animal.py* est présenté en appendice).

Méthodes pour exécuter des requêtes

La section suivante de *animal.py* crée un objet curseur et l'utilise pour exécuter des requêtes qui initialisent et peuplent la table `animal` :

```
cursor = conn.cursor ()
cursor.execute ("DROP TABLE IF EXISTS animal")
cursor.execute ("""
    CREATE TABLE animal
    (
      name     CHAR(40),
      category CHAR(40)
    )
  """)
cursor.execute ("""
    INSERT INTO animal (name, category)
    VALUES
      ('serpent', 'reptile'),
      ('grenouille', 'amphibien'),
      ('thon', 'poisson'),
      ('raton laveur', 'mammifère')
  """)
print "Nombre de lignes insérées : %d" % cursor.rowcount
```

A noter que ce code n'effectue aucune gestion d'erreur (on rappellera que le code sera placé dans une instruction `try`; les erreurs produiront des exceptions qui seront interceptées et traitées dans la clause `except` correspondante, ce qui permet de rendre le code principal plus lisible). Les requêtes

réalisent les actions suivantes :

- Efface la table `animal` si elle existe déjà, de manière à nettoyer le terrain.
- Crée la table `animal`.
- Insère des données dans la table et affiche le nombre de lignes ajoutées.

Chaque requête est exécutée par appel de la méthode `execute()` de l'objet curseur. Les deux premières requêtes ne produisent aucun résultat, mais la troisième renvoie le nombre de lignes insérées. Cette valeur est disponible dans le champ `rowcount` de l'objet curseur. Certaines interfaces de bases de données renvoient cette valeur en retour de l'appel de l'exécution de la requête, mais ce n'est pas le cas de DB-API.

La table `animal` est maintenant préparée, nous pouvons donc exécuter des requêtes de type `SELECT` pour en extraire des informations. Comme pour les requêtes précédentes, les `SELECT` sont exécutés par appel à la méthode `execute()`. Cependant, contrairement à des requêtes `DROP` ou `INSERT`, `SELECT` renvoie un jeu de résultats que l'on doit récupérer. Cependant, `execute()` ne fait que réaliser la requête mais ne renvoie pas encore le résultat. Vous pouvez utiliser `fetchone()` pour récupérer les lignes une à une ou `fetchall()` pour les récupérer toutes en une seule fois. Le script *animal.py* utilise les deux approches. Voici comment utiliser `fetchone()` pour la récupération des lignes une à une :

```
cursor.execute ("SELECT name, category FROM animal")
while (1):
  row = cursor.fetchone ()
  if row == None:
    break
  print "%s, %s" % (row[0], row[1])
print "Nombre de lignes renvoyées : %d" % cursor.rowcount
```

`fetchone()` renvoie la ligne suivante du résultat sous forme d'un tuple, ou `None` si aucune ligne n'est plus disponible. La boucle vérifie cela et quitte lorsque le jeu de résultats a été épuisé. Pour chaque ligne renvoyée, le tuple contient deux valeurs (celles la même qui ont été réclamées par la clause `SELECT`) affichées par *animal.py*. L'instruction `print` ci-dessous accède aux éléments du tuple. Cependant, parce qu'elles sont utilisées dans l'ordre d'occurrence dans le tuple, l'instruction `print` aurait put avoir été écrite comme suit :

```
print "%s, %s" % row
```

Après avoir affiché le résultat de la requête, le script affiche ensuite le nombre de lignes renvoyées (disponible dans la valeur de l'attribut `rowcount`).

`fetchall()` renvoie le jeu de résultats entier, en une seule fois, sous la forme d'un tuple de tuples, ou d'un tuple vide si le jeu de résultats est vide. Pour accéder à chaque ligne sous forme d'un tuple, il faut itérer sur les lignes renvoyées par `fetchall()` :

```
cursor.execute ("SELECT name, category FROM animal")
rows = cursor.fetchall ()
for row in rows:
```

```
  print "%s, %s" % (row[0], row[1])
print "Nombre de lignes renvoyées : %d" % cursor.rowcount
```

Ce code affiche le nombre de lignes en accédant à `rowcount`, comme dans la boucle utilisant `fetchone()`. Une autre manière de déterminer le nombre de lignes lorsqu'on utilise `fetchall()` est de prendre la longueur de la valeur renvoyée :

```
print "%d lignes ont été renvoyées" % len (rows)
```

Les boucles de récupération vues jusqu'à présent récupèrent les lignes sous forme d'un tuple. Il est aussi possible de les récupérer sous forme d'un dictionnaire, ce qui permet d'accéder aux valeurs des colonnes par leur nom. Le code qui suit nous montre comment le faire. Il faut noter que l'accès à l'aide de dictionnaires requiert un autre type de curseur, donc l'exemple ferme le curseur et en obtient un nouveau qui est d'un autre type :

```
cursor.close ()
cursor = conn.cursor (MySQLdb.cursors.DictCursor)
cursor.execute ("SELECT name, category FROM animal")
result_set = cursor.fetchall ()
for row in result_set:
  print "%s, %s" % (row["name"], row["category"])
print "Nombre de lignes renvoyées : %d" % cursor.rowcount
```

Les valeurs `NULL` dans les jeux de résultats sont renvoyées sous forme de `None` à votre programme.

MySQLdb fournit un mécanisme de paramètres qui permet de remplacer des balises dans la requête par des valeurs. Ceci permet de ne pas avoir à insérer directement les valeurs dans la requête. Ce mécanisme gère l'ajout de guillemets autour des données et il traite les caractères spéciaux contenus dans les valeurs. L'exemple suivant présente une requête `UPDATE` qui remplace les serpents par des tortues en utilisant les valeurs littérales puis réalise l'opération inverse avec des paramètres. La requête avec les valeurs littérales ressemble à cela :

```
cursor.execute ("""
      UPDATE animal SET name = 'tortue'
      WHERE name = 'serpent'
    """)
print "Nombre de lignes mises à jour : %d" % cursor.rowcount
```

On peut aussi exécuter la même requête en utilisant la balise `%s` et en y associant les valeurs appropriées :

```
cursor.execute ("""
      UPDATE animal SET name = %s
      WHERE name = %s
    """, ("serpent", "tortue"))
print "Nombre de lignes mises à jour : %d" % cursor.rowcount
```

On notera les points suivants concernant l'appel à `execute()` :

- La balise `%s` doit apparaître une fois pour chaque valeur à insérer dans la requête.
- Aucun guillemet ne doit être placé autour de la balise `%s`, MySQLdb s'en charge.
- Dans l'appel à `execute()`, on passera les valeurs des paramètres sous la forme d'un tuple, dans l'ordre dans lequel ils apparaissent dans la requête. Si une seule valeur `x` doit être passée, on la passera sous la forme d'un tuple à un seul élément `(x,)`.
- Passer la valeur Python `None` en paramètre pour insérer un `NULL` SQL dans la requête.

Après avoir exécuté les requêtes, le script *animal.py* ferme les curseurs, effectue un commit des modifications et déconnecte du serveur :

```
cursor.close ()
conn.commit ()
conn.close ()
```

L'appel à la méthode `commit()` de l'objet connexion valide toutes les modifications en attente dans la transaction en cours et les rend permanentes dans la base de données. Dans DB-API, les connections sont crées avec le mode autocommit désactivé, donc vous devez appeler `commit()` avant de vous déconnecter à la base sans quoi les modifications peuvent être perdues.

Si la table `animal` est de type MyISAM, `commit` n'a aucun effet : MyISAM est un moteur de stockage non transactionnel, donc les modifications aux tables MyISAM sont prises en compte immédiatement sans tenir compte du mode autocommit. Cependant, si la table `animal` utilise un moteur de stockage transactionnel comme InnoDB, ne pas appeler `commit()` résulte en un rollback implicite lorsqu'on se déconnecte. Par exemple, si vous ajoutez `ENGINE=InnoDB` à la fin de la requête `CREATE TABLE` et enlevez l'appel à `commit()` à la fin du script, vous constaterez que la table `animal` est vide après l'exécution du script.

Pour les scripts qui ne font que récupérer des données, aucune modification n'a besoin d'être confirmée et les appels à `commit()` ne sont pas nécessaires.

Notes sur la portabilité

Si vous souhaitez porter un script utilisant DB-API basé sur MySQLdb vers une autre base de données, les obstacles à la portabilité surgissent partout où apparaît le nom du pilote :

- La clause `import` qui importe le module du pilote. Elle doit être modifiée de manière à importer un autre pilote.
- L'appel à `connect()` pour se connecter au serveur de la base de données. La méthode `connect()` est accédée en passant par le nom du module du pilote, donc le nom du pilote doit être changé. De plus, la syntaxe des arguments à `connect()` peut varier suivant les pilotes.
- Gestion des exceptions. Le nom de la classe dans les instructions `except` est référencée par le nom du pilote.

Une autre source d'incompatibilités qui n'est pas liée au nom du pilote est due à l'utilisation des

paramètres. Les spécifications de DB-API autorisent plusieurs syntaxes pour les paramètres et certains pilotes utilisent une syntaxe différente de celle de MySQLdb.

Ressources

Les cripts utilisés comme exemples dans ce document peuvent être téléchargés à l'adresse suivante : http://www.kitebird.com/articles/.

Vous pourriez trouver les ressources suivantes utiles pour utiliser DB-API et le pilote MySQLdb :

- Andy Dustman, l'auteur du module MySQLdb a un site à l'adresse : http://dustman.net/andy/python/. Ce site est le meilleur endroit pour lire la documentation et la FAQ de MySQLdb. On y trouve aussi des liens vers des archives binaires pour la distribution Debian et Windows. Pour obtenir les codes sources ou des RPMs Linux, visitez le repository SourceForge à : http://sourceforge.net/projects/mysql-python.
- Le livre *MySQL Cookbook* inclue DB-API parmi les interfaces de programmation traitées : http://www.kitebird.com/mysql-cookbook ou http://www.oreilly.com/catalog/mysqlckbk/.
- Le site Python met à disposition des installeurs pour l'interpréteur Python, si vous utilisez un système qui ne l'embarque pas déjà : http://www.python.org/.
- Le SIG (special interest group) des bases de données sur le site Python contient d'autres informations à propos de DB-API : http://www.python.org/sigs/db-sig/.
- La table `animal` utilisée par le script *animal.py* est aussi utilisée par le document traitant de PEAR DB sur le site Kitebird : http://www.kitebird.com/articles/. Vous pourriez trouver intéressant de comparer ces documents afin de voir les similitudes et différences de leurs approches concernant l'accès de la base de données.

Appendice

Voici le source complet du script *animal.py* :

```
#!/usr/bin/python
# animal.py - crée la table animal et y récupère des informations

import sys
import MySQLdb

# connexion au serveur MySQL

try:
  conn = MySQLdb.connect (host = "localhost",
                          user = "testuser",
                          passwd = "testpass",
                          db = "test")
except MySQLdb.Error, e:
  print "Erreur %d : %s" % (e.args[0], e.args[1])
  sys.exit (1)

# crée la table animal et la peuple

try:
```

```
  cursor = conn.cursor ()
  cursor.execute ("DROP TABLE IF EXISTS animal")
  cursor.execute ("""
      CREATE TABLE animal
      (
        name     CHAR(40),
        category CHAR(40)
      )
    """)
  cursor.execute ("""
      INSERT INTO animal (name, category)
      VALUES
        ('serpent', 'reptile'),
        ('grenouille', 'amphibien'),
        ('thon', 'poisson'),
        ('raton laveur', 'mammifère')
    """)
  print "Nombre de lignes insérées : %d" % cursor.rowcount

# boucle de récupération utilisant fetchone()

  cursor.execute ("SELECT name, category FROM animal")
  while (1):
    row = cursor.fetchone ()
    if row == None:
      break
    print "%s, %s" % (row[0], row[1])
  print "Nombre de lignes renvoyées : %d" % cursor.rowcount

# boucle de récupération utilisant fetchall()

  cursor.execute ("SELECT name, category FROM animal")
  rows = cursor.fetchall ()
  for row in rows:
    print "%s, %s" % (row[0], row[1])
  print "Nombre de lignes renvoyées : %d" % cursor.rowcount

# exécution d'une requête qui change le nom en incluant les données
# littérales dans la chaîne de la requête, puis remplace le nom
# en utilisant les paramètres

  cursor.execute ("""
        UPDATE animal SET name = 'tortue'
        WHERE name = 'serpent'
      """)
  print "Nombre de lignes mises à jour : %d" % cursor.rowcount

  cursor.execute ("""
        UPDATE animal SET name = %s
        WHERE name = %s
      """, ("serpent", "tortue"))
  print "Nombre de lignes mises à jour : %d" % cursor.rowcount

# crée un curseur dictionnaire de manière à ce que
# les valeurs des colonnes puissent être accédées par
# nom au lieu de la position

  cursor.close ()
```

```
    cursor = conn.cursor (MySQLdb.cursors.DictCursor)
    cursor.execute ("SELECT name, category FROM animal")
    result_set = cursor.fetchall ()
    for row in result_set:
      print "%s, %s" % (row["name"], row["category"])
    print "Nombre de lignes renvoyées : %d" % cursor.rowcount

    cursor.close ()

  except MySQLdb.Error, e:
    print "Erreur %d : %s" % (e.args[0], e.args[1])
    sys.exit (1)

  conn.commit ()
  conn.close ()
```

Remerciements

La première version de ce document a été écrite pour NuSphere Corporation. La version actuelle est une mise à jour de ce premier document.

Historique des révisions

- 1.00 Première version.
- 1.01 2003-01-24 Révisions mineures.
- 1.02 2006-09-17 Mise à jour pour MySQLdb 1.2.1, ajouté des informations à propos du mode autocommit et de la méthode `commit()`.