

Python Datenbank- programmierung

Einführung in das Python DB-API 2.0

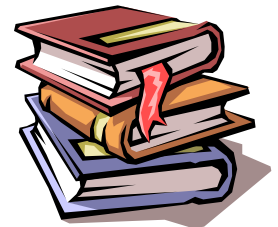
DZUG Tagung 2010
Dresden

Marc-André Lemburg

EGENIX.COM Software GmbH
Langenfeld

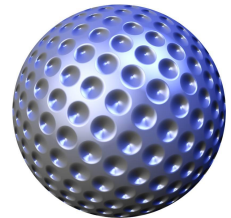
Marc-André Lemburg

- Geschäftsführer der eGenix.com GmbH
 - Mehr als 20 Jahre IT-Erfahrung
 - Diplom in Mathematik
 - Experte in Python, Application Design, Web Technologies, Unicode
 - Python Core Developer (seit 2000)
 - Python Software Foundation Vorstandsmitglied (2002-2004, seit 2010)
 - EMail: mal@egenix.com
- eGenix.com Software GmbH, Germany
 - Gründung in 2000
 - Kernbereiche:
 - **Kundenprojekte:** Schwerpunkte Python und Datenbanken
 - **Produkte:** Python mx Extensions für Python/Plone/Zope/Django (mxODBC, mxDateTime, mxTextTools, etc.)
 - Internationaler Kundenstamm



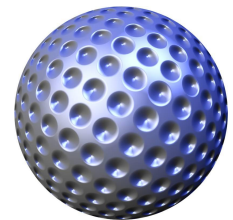
Agenda

1. Einleitung
2. Grundkonzepte
3. Fortgeschrittene Anwendung
4. Diskussion



Einleitung

1. Einleitung
2. Grundkonzepte
3. Fortgeschrittene Anwendung
4. Diskussion



Motivation für das Python Datenbank API (DB-API)

- Bereitstellung einer **standardisierten Schnittstelle** zwischen Python und Datenbanken
- Sollte **einfach umzusetzen und zu verstehen sein**
 - viele Datenbankmodule
 - Module von hoher Qualität
 - viele unterstützte Datenbankprodukte

Häufige Missverständnisse

- Das Python DB-API ist nicht ...
 - ein einziges Modul mit Plugins für Datenbanken
 - ein Python Paket, das man installieren kann
 - ein unflexibler und wenig anpassbarer Standard
 - alt und angestaubt
- Für mehr Komfort sollte man ein **Datenbank-abstraktionspaket** nutzen:
 - SQL Wrappers (nehmen das Schreiben von SQL ab)
 - Object Relational Mappers (ORM; bilden Tabellen auf Objekte ab)
 - Object Database Wrappers (speichern Objekte in Datenbanken)

Entwicklung des Python DB-API: Version 1.0 (1996)

- Die US-Firma eShop (Greg Stein, Bill Tutt) schrieb ein ODBC Model und startete dazu eine Diskussion auf der Python DB-SIG Mailing Liste
 - Ergebnis: DB API 1.0 und das win32 odbc Modul
- Das **DB API 1.0** bot eine gute Grundlage für Datenbankmodule, hatte aber noch einige Nachteile
- Diese Version ist noch als **PEP 248** verfügbar

Entwicklung des Python DB-API: Version 2.0 (1999)

- Nach 2-3 Jahren wurde ein Anlauf unternommen, die Unstimmigkeiten zu beseitigen
 - nach langen Diskussionen auf der Python Database SIG Mailing Liste wurden die meisten Probleme gelöst ...
 - Resultat: DB API 2.0
 - mxODBC war eines der ersten Module, das den neuen Standard umsetzte
- **DB API 2.0 ist (immer noch) die aktuelle DB API Version !**
- Nachzulesen im [PEP 249](#)

Python DB-API: Ein Blick in die Zukunft

- Viele Datenbankmodule haben Erweiterungen des API Standards implementiert
- Das DB-API ist als Konsequenz schrittweise um optionale **standardisierte Erweiterungen** ergänzt worden
- Es gibt immer noch einige Bereiche, die verbessert werden könnten.

Version 3.0 wird diese Punkte adressieren

- flexiblere Datentypkonvertierungen

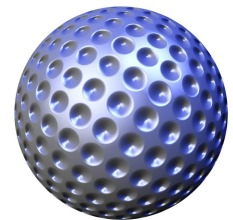
Python DB-API 2.0 kompatible Module

- MySQLdb (MySQL)
- psycopg2 (PostgreSQL)
- cx_Oracle (Oracle)

- mxODBC (SQL Server, DB2, Sybase, Oracle, etc.)
- mxODBC Connect (Client-Server Variante von mxODBC)

Grundkonzepte

1. Einleitung
2. Grundkonzepte
3. Fortgeschrittene Anwendung
4. Diskussion



Verbindungen und Abfragen

- Das DB API basiert auf **zwei wichtigen Objekten**, die den Datenbankzugriff ermöglichen:
- Datenbankverbindungen: **Connection Objects**
 - Netzwerk/RPC Verbindung zur Datenbank
 - Transaktionsmechanismus
- Abfragen: **Cursor Objects**
 - Ausführen von SQL Abfragen
 - Zugriff auf die Abfrageergebnisse

Connection Objects

- Connection Objects bieten eine **logische Sicht auf eine Datenbankverbindung**
- Stellen den physischen Kontakt zur Datenbank her
- Beispiele:

```
conn = connect(Datasourcenname, Username, Password)  
conn = DriverConnect("DSN=test;UID=test;PWD=test")
```
- Connection Objects **dienen *nicht* der SQL-Abfrage**

Connection Objects: Transaktionen

- Connection Objects bieten eine zweite wichtige Funktion: **Datenbanktransaktionen**
- Transaktionen sind logische Gruppen von SQL Anweisungen, die auf einer Verbindung ausgeführt werden
- Hauptvorteil:
man kann Änderungen **leicht rückgängig machen**
 - Ermöglicht Datenkonsistenz - auch im Fehlerfall
 - Leider unterstützen nicht alle Datenbanken Transaktionen !

Connection Objects: API

- Verbindungsaufbau:

```
conn = connect(Datasourcename, Username, Password)
conn = DriverConnect("DSN=test;UID=test;PWD=test")
```

- Transaktionen:

- Starten implizit beim Verbindungsaufbau !
- aktuelle Transaktion rückgängig machen und eine neue starten:
`conn.rollback()`
- aktuelle Transaktion speichern und eine neue starten:
`conn.commit()`

- Verbindung schließen:

- impliziert ein `conn.rollback()`:
`conn.close()`

Cursor Objects

- Werden über die Connection Object Methode `conn.cursor()` erzeugt und sind an dieses gebunden
- Cursor Objects ermöglichen:
 - direkten Zugang zu SQL
 - Erzeugen/Löschen/Verändern von Objekten in der Datenbank
 - Abfrage der Datenbankinhalte
- Beispiel:

```
cursor = conn.cursor()
cursor.execute('create table testtable (id int, name varchar(254))')
```

Cursor Objects: Ausführen von SQL Anweisungen

- Die wichtigste Methode des Cursor Objects ist `cursor.execute()`, mit der das Programm SQL Anweisungen auf dem zugeordneten Connection Object ausführen kann
- Zur Parametrisierung der Anweisungen können der Methode `Parameter` mitgegeben werden

- Beispiele:

```
cursor.execute('create table testtable (id int, name varchar(254))')
```

```
cursor.execute('insert into testtable values (?, ?)', (1, 'Peter'))
```

```
cursor.execute('select * from testtable')
```

Cursor Objects: Datenübergabe

- Datenübergabe von Python an die Datenbank
 - Erste Methode (**nicht empfohlen**):
Werte in SQL Notation umsetzen (Quoting) und direkt in der SQL Anweisung einbetten
 - Zweite Methode (empfohlen):
Platzhalter (binding parameters) im SQL nutzen ('?' for ODBC) und die Werte als Python Parameter an die Datenbank übergeben
- Beispiel:

```
cursor = conn.cursor()
cursor.execute("insert into testtable values (2, 'Fred')")
cursor.execute("insert into testtable value (?,?)", (2, 'Fred'))
```

Cursor Objects: Datenbankabfrage

- Cursor objects dienen auch dazu, die Abfragetabelle verfügbar zu machen: das **Result Set**
- Der Zugriff erfolgt zeilenweise über **cursor.fetch*()** Methoden
- Beispiel:

```
cursor.execute('select * from testtable')
erste_Zeile = cursor.fetchone()
naechsten_10_Zeilen = cursor.fetchmany(10)
alle_uebrigen_Zeilen = cursors.fetchall()
```

Cursor Objects: API

- Erzeugen:

```
cursor = conn.cursor()
```

- Anweisungen/Abfragen durchführen:

- Einfachoperation:

```
cursor.execute(sql, parameter)
```

- Mehrfachoperation:

```
cursor.executemany(sql, list_of_parameters)
```

- Abfragedaten abholen:

- `cursor.fetchone()`, `cursor.fetchmany(number_of_rows)`,
`cursor.fetchall()`

- Schließen:

```
cursor.close()
```

Beispiel: Testanwendung

- Ohne Veränderung der Datenbank:

```
from mx.ODBC.Windows import *
conn = Connect('test', 'test', 'test')
c = conn.cursor()
c.execute('create table testtable (id int, name varchar(254))')
c.execute('insert into testtable values (?, ?)', (1, 'Marc'))
c.execute('insert into testtable values (?, ?)', (2, 'Fred'))
c.execute('insert into testtable values (?, ?)', (3, 'Tim'))
c.execute('insert into testtable values (?, ?)', (4, 'Peter'))
c.execute('select * from testtable')
rows = c.fetchall()
# rows ... [(1, 'Marc'), (2, 'Fred'), (3, 'Tim'), (4, 'Peter')]
```

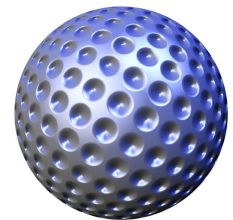
Beispiel: Produktionsanwendung

- Mit Veränderung der Datenbank:

```
from mx.ODBC.Windows import *
conn = Connect('test', 'test', 'test')
c = conn.cursor()
c.execute('create table testtable (id int, name varchar(254))')
conn.commit()
c.execute('insert into testtable values (?, ?)', (1, 'Marc'))
c.execute('insert into testtable values (?, ?)', (2, 'Fred'))
conn.commit()
c.execute('select * from testtable')
rows = c.fetchall()
# rows ... [(1, 'Marc'), (2, 'Fred')]
conn.close()
```

Fortgeschrittene Anwendung

1. Einleitung
2. Grundkonzepte
3. Fortgeschrittene Anwendung
4. Diskussion



Fortgeschrittene Anwendung: Transaktionen

- Transaktionen verbinden mehrere SQL Anweisungen zu einem logischen Block einer Datenbankveränderung
 - Vorteil: **Man kann Veränderungen leicht verwerfen**
- Das Python DB API unterstützt Transaktionen über die beiden Connection Object Methoden:
- **conn.commit()**
 - Fixiert alle Veränderungen seit dem Start der Transaktion
- **conn.rollback()**
 - Verwirft alle Veränderungen seit dem Start der Transaktion

Fortgeschrittene Anwendung: Probleme mit Transaktionen

- **Transaktionsunabhängigkeit (Transaction Isolation):**
Wer wird wann meine Änderungen sehen ?
 - Wiederholbarkeit von Abfragen
 - Einfluß auf offene Abfragentabellen (Result Sets)
 - datenbankabhängige Umsetzung;
manchmal beim Öffnen der Verbindung konfigurierbar
- **Datenbanksperren (Database Locks)**
 - Mehrere Prozesse greifen gleichzeitig auf eine Tabelle zu
 - Meistens implizit von der Datenbank angelegt und schwierig zu umgehen
 - Verhindern Dateninkonsistenzen

Fortgeschrittene Anwendung: Two Phase Commit

- Datenbankübergreifende Transaktionen
- Anwendungsfall:
 - Eine Kontobuchung soll in **zwei Datenbanken** geschrieben werden, von der ersten wird **abgebucht**, in die zweite wird **gutgeschrieben**
 - Natürlich soll die Abbuchung nur dann erfolgen, wenn die Gutschrift durchgeführt werden kann.
 - Ebenso kann die Gutschrift nur dann erfolgen, wenn die Abbuchung gelingt.
- Lösung:
 - **Erste Phase**: Commits werden **vorbereitet**
... sofern die erste Phase insgesamt erfolgreich war:
 - **Zweite Phase**: Commits werden **umgesetzt**

Fortgeschrittene Anwendung: Two Phase Commit

- Das Konzept kann auch auf andere Ressourcen erweitert werden, wie z.B.
 - RPCs (Remote Procedure Calls)
 - Dateien
 - Queues (MQ Series)
 - etc.
- Hierzu werden die Ressourcen über einen **Transaktionsmanager** verbunden, der die beiden Commit-Phasen koordiniert.
- Im Python DB-API gibt es hierzu eine **neue API Erweiterung**
 - **.tpc_*()** APIs (dem XA API nachempfunden)
 - noch sehr neu - wird nur von wenigen Datenbankmodulen

Fortgeschrittene Anwendung: Datenbankschemata

- Anwendungsfall:
 - Eine Anwendung soll Daten aus einer fremden Datenbank lesen, auswerten und anzeigen
- Hierzu muß die Anwendung die **Datenbankstruktur** (das Schema) auslesen können
- Einfache Lösung bei Kenntnis der Tabellennamen:
 - Leerabfrage auf alle Spalten einer Tabelle:
`cursor.execute('select * from testtable where 1=0')`
 - **`cursor.description`** liefert nun einen Einblick in die Datentypen der Tabellenspalten (näheres hierzu in PEP 249)

Fortgeschrittene Anwendung: Datenbankschemata

- Fortgeschrittene Methode:
 - Auslesen des Schemas aus **Systemtabellen der Datenbank**
 - bei den meisten Datenbanken möglich
 - Nachteil: datenbankspezifisch
- mxODBC liefert hierzu datenbankunabhängige Cursor Object Katalogmethoden:

```
cursor.columns(table='testtable')
```

```
rows = cursor.fetchall()
```

```
# rows ... [('D:\\tmp\\test', None, 'testtable', 'id', 4, 'INTEGER', 10, 4, 0, 10, 1, None, None, 4, None, None, 1, 'YES', 1), ('D:\\tmp\\test', None, 'testtable', 'name', 12, 'VARCHAR', 254, 508, None, None, 1, None, None, 12, None, 508, 2, 'YES', 2)]
```

Fortgeschrittene Anwendung: Mehrere Result Sets

- Anwendungsfall:
 - Eine Datenbankprozedur muß Listen- und skalare Daten zurückliefern, z.B.
 - eine Liste von gefundenen Datensatz-IDs und
 - ein Satz von Statistikdaten zu den entsprechenden Datensätzen
- Lösung:
 - Die Datenbankprozedur erzeugt mehrere Result Sets
 - Die Anwendung fragt diese dann sukzessive mit der Python DB-API Methode `cursor.next()` ab

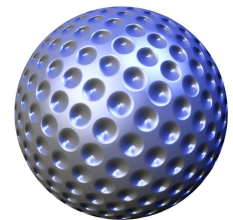
Fortgeschrittene Anwendung: Mehrere Result Sets

- Beispiel:

```
# Datenbankprozedur aufrufen
cursor.execute('myStatisticsStoredProcedure()')
# erstes Result Set einlesen
first_result_set = cursor.fetchall()
# [(1,), (2,), ...] in eine einfache Liste umformatieren
list_of_record_ids = [id for (id,) in first_result_set]
# nächstes Result Set aufrufen
cursor.next()
# Zeile mit den Statistikwerten abholen
average, median, min, max = cursor.fetchone()
# Ressourcen freigeben
cursor.close()
```

Diskussion

1. Einleitung
2. Grundkonzepte
3. Fortgeschrittene Anwendung
4. Diskussion



Zusammenfassung

Das Python Database API ist

einfach anzuwenden und

trotzdem mächtig !



Python Datenbankprogrammierung



Zeit für die Kaffeepause ...



Vielen Dank für Ihre Aufmerksamkeit !

Kontakt

eGenix.com Software, Skills and Services GmbH

Marc-André Lemburg

Pastor-Löh-Str. 48

D-40764 Langenfeld

Germany

eMail: mal@egenix.com

Phone: +49 211 9304112

Fax: +49 211 3005250

Web: <http://www.egenix.com/>