Introduction to Python Database Programming

Python DB-API 2.0

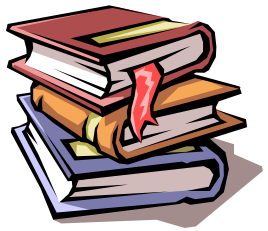
EuroPython 2011
Florence, Italy

Marc-André Lemburg

EGENIX.COM Software GmbH
Langenfeld

Speaker: Marc-André Lemburg

- CEO eGenix.com and Consultant
  - More than 20 years software development experience
  - Diploma in Mathematics
  - Expert in Python, Application Design, Web Technologies and Unicode
  - Python Core Developer (since 2000)
  - Python Software Foundation Board Member (2002-2004, since 2010)
  - Contact: mal@egenix.com
- eGenix.com Software GmbH, Germany
  - Founded in 2000
  - Core business:
    - Projects: implementing custom solutions using Python
    - Products: professional quality Python/Plone/Zope/Django developer tools (mxODBC, mxDateTime, mxTextTools, etc.)
  - International customer base



(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Agenda

1. Introduction
2. Basic Concepts
3. Advanced Techniques
4. Discussion

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Introduction

1. Introduction
2. Basic Concepts
3. Advanced Techniques
4. Discussion

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Motivation for the Python DB API

- Provide a standard interface from Python to databases
- Standard should be easy to implement and understand
  - more database modules
  - higher quality modules
  - more supported backends

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Common misunderstandings

- Things the Python DB-API is not...
  - a single interface with pluggable drivers
  - a software package you can download and install
  - an inflexible and hard-coded standard
  - old cruft from the past

- Use database abstraction layers for higher level database interfacing
  - SQL Wrappers (help write SQL)
  - Object Relational Mappers (ORM; map tables to objects)
  - Object Database Wrappers (store objects in databases)

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

History of the Python DB-API: Version 1.0 (1996)

- US company eShop (Greg Stein, Bill Tutt) wrote an ODBC module and started discussing a standard database Python API on the newsgroup/mailing list
  - Result: DB API 1.0 and the win32 odbc module
- DB-API 1.0 provided a good start for coding database modules but had some caveats
- Now referenced as PEP 248

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

History of the Python DB-API: Version 2.0 (1999)

- After 2-3 years a new effort was started to solve most of the caveats
  - after long discussions on the Python Database SIG mailing list and solved most of these problems …
  - Result: DB-API 2.0
  - mxODBC was among the first modules to implement DB API 2.0
- DB-API 2.0 is (still) the current DB-API version !
- Now referenced as PEP 249

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Python DB-API: The Future

- Many database modules provide extensions to the API standard
- The DB API 2.0 addressed this by defining a set of standard extensions
- The DB API still has a few caveats, which should be fixed. Version 3.0 will address these.
  - Flexible data type mappings
  - Integration of a few extensions into the standard
  - Adjustments to better support Python 3.x

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Python DB-API 2.0: Compatible Modules

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

- mxODBC (SQL Server, DB2, Sybase, Oracle, etc.)
- mxODBC Connect (client server version of mxODBC)

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Basic Concepts

1. Introduction
2. Basic Concepts
3. Advanced Techniques
4. Discussion

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Connections and Queries

- The DB-API uses two important objects for processing database queries:

- Database Connections: Connection objects
  - Network/RPC connection to the database
  - Transactions

- Database Queries: Cursor objects
  - SQL statement execution
  - Access to the results

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Connection Objects

- Connection objects logically wrap a database connection
- Provide physical network/RPC access to the database
- Examples:

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

- Connection objects are *not used* for running SQL statements. Cursor objects provide this functionality.

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Connection Objects: Transactions

- Connection objects also provide a second important feature: the means to handle database transactions
- Transactions are logical groups of statements run on a connection (units of recovery)
- Main benefit: you can undo changes very easily
  - Ensures data consistency at all times
  - Not all databases provide transactions !

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Connection Objects: API

- Establish a database connection:

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

- Transactions:
  - Implicitly started on connect
  - Undo current transaction changes and start a new one:
    conn.rollback()
  - Apply current transaction changes permanently and start a new one:
    conn.commit()

- Close a database connection:

  conn.close()

  (implicitly calls conn.rollback() prior to closing the connection)

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Cursor Objects

- Created by calling the conn.cursor() method on Connection Objects and are bound to these
- Cursor Objects provide:
  - direct access to SQL
  - ways to add/modify/delete database objects (schemas, tables, views, indexes, rows, etc.)
  - ways to query the database contents and its meta-data
- Examples:

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


(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Cursor Objects: Execute SQL statements

- Execute SQL statements on the associated Connection Object: cursor.execute(sql)

- The method also accepts a sequence of parameters to customize the SQL statement: cursor.execute(sql, params)

- Examples:

```
cursor.execute('create table testtable (id int, name varchar(254))')
cursor.execute('insert into testtable values (?, ?)', (1, 'Peter'))
cursor.execute('select * from testtable where id=?', (1,))
```


(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Cursor Objects: Passing data to the database

- Passing data from Python to the database
  - First option (discouraged):
    Quoting values and sending plain SQL to the database
  - Second option (preferred):
    Using placeholders (binding parameters) in the SQL ('?' for mxODBC) and passing the data using Python sequences
- Example:

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


(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Cursor Objects: Accessing query data

- Cursor objects also make the query data available after a statement has been executed: the Result Set.
- Read access to the result set rows is provided via the cursor.fetch*() methods
- Example:

```
cursor.execute('select * from testtable')
first_row = cursor.fetchone()
next_10_rows = cursor.fetchmany(10)
all_remaining_rows = cursors.fetchall()
```


(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Cursor Objects: API

- Create a cursor object:

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

- Execute SQL statements:
  - One-time execution:

    ```
    cursor.execute(sql, parameter)
    ```
  - Multi-row execution:

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

- Access result sets:
  - cursor.fetchone(), cursor.fetchmany(number_of_rows), cursor.fetchall()

- Close a cursor object:
  - cursor.close()

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Example Session: Testing only

- Without changing the database:

```
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')]
# no conn.commit(), so no permanent database changes !
```


(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Example Session: Production use

- With permanent changes to the database:

```
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()
```


(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques

1. Introduction
2. Basic Concepts
3. Advanced Techniques
4. Discussion

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques: Transactions

- Transactions logically wrap multiple SQL statements into blocks of recovery
  - Benefit: You can easily undo changes
- The Python DB-API supports transactions using these two methods on the Connection Object:
- conn.commit()
  - Make all changes applied in the current transaction block permanent.
- conn.rollback()
  - Undo all changes applied since the start of the transaction.

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques: Transaction Isolation

- Transaction Isolation:
  Who will see my changes and when ?
- Repeatability of queries
  - Will the same query return the same result set,
    even if another process has added new data to a table ?
- Changes to open result sets
  - Will new rows appear in an already open result set
    while fetching rows ?
- Database dependent
  - sometimes configurable per connection

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques: Database Locks

- Database Locks:
  Needed to prevent inconsistencies in multi-process setups
- Multiple processes accessing the same table
  - Two processes trying to update the same row
  - One process trying to read a table,
    while another process writes to it
  - One process trying to delete rows from a table,
    while another process reads it
- Locks are usually implicitly set by the database and difficult to work around
- Effect: errors, timeouts and even dead locks

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques: Two Phase Commit

- Distributed Transactions (across multiple databases)
- Use Case:
  - A bank wants to execute an account transfer using two databases.
  - The first account database needs to get the debit booking, the second the credit booking.
    - The credit booking should only succeed, if the debit booking can be applied (there's enough money available on the account)
    - The debit booking should only succeed, if the credit booking can be applied (the target account exists and can be written to).
- Solution: Two-Phase Commits
  - First Phase: Commits are prepared

    … if this first phase is successful:
  - Second Phase: Prepared commits are executed

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques: Two Phase Commit APIs

- The concept can be extended to other resources as well:
  - RPCs (Remote Procedure Calls)
  - Files
  - Queues (MQ Series)
  - FTP Uploads
  - etc.
- A transaction manager (TM) is used to connect the resources and manage and coordinate the two-phase commit transactions.
  - X/Open XA compatible TMs: MQ Servies, J2EE, MSDTC, DB2, Oracle, etc.
- The Python DB-API provides a new API extension for these:
  - .tpc_*() APIs (designed after the X/Open XA API)
  - still very new and only few database interfaces support it

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques: Schema Introspection

- Use Case:
  - An application is supposed to interface to an external database, read data from it, analyze and display this data in a GUI
- The application will need to query the database structure (schema) for this to work
- Simple solution in case the table names are known:
  - Empty query on all table columns:

    cursor.execute('select * from testtable where 1=0')
  - cursor.description will then provide information about the data types, column names, etc.;
    see the DB-API 2.0 (PEP 249)for more details.

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques: Schema Introspection

- Advanced method:
  - Read the schema from the database system tables
    - Possible with most databases
    - Problem: database specific

- mxODBC has database independent Cursor Object catalog methods to access this data:

```
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)]
```


(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques: Multiple Result Sets

- Use Case:
  - A database procedure needs to return two sets of data, e.g.
    - A list of database row IDs with all rows matching a query, and
    - A list of statistical data for those rows
- Solution:
  - The database procedure creates two result sets
  - The application then reads these one-by-one using the Python DB-API method cursor.next() to skip to the next result set

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Advanced Techniques: Multiple Result Sets

- Example:

```
# call database procedure
cursor.execute('myStatisticsStoredProcedure()')
# read first result set
first_result_set = cursor.fetchall()
# gives: [(1,), (2,), ...]; convert to a simple list of IDs
list_of_record_ids = [id for (id,) in first_result_set]
# skip to next result set
cursor.next()
# fetch statistics data row
average, median, min, max = cursor.fetchone()
# free resources used by the result sets
cursor.close()
```


(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Discussion

1. Introduction
2. Basic Concepts
3. Advanced Techniques
4. Discussion

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Conclusion

Das Python Database API ist

easy to use,

yet powerful !

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Python Database Programming

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Time for coffee ...

Thank you for your time.

(c) 2011 EGENIX.COM Software GmbH, info@egenix.com

Contact

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/