

Using the Python Database API

Using databases from Python is easy

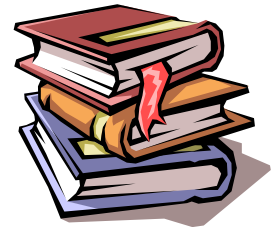
EuroPython Conference 2008
Vilnius, Lithuania

Marc-André Lemburg

EGENIX.COM Software GmbH
Germany

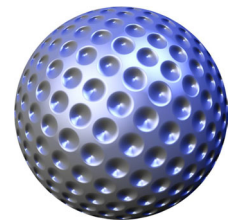
Speaker Introduction: Marc-André Lemburg

- 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)
 - Contact: mal@egenix.com
- eGenix.com Software GmbH, Germany
 - Founded in 2000
 - Core business:
 - **Consulting**: helping companies write successful Python software
 - **Product design**: professional quality Python/Zope developer tools (mxODBC, mxDateTime, mxTextTools, etc.)
 - International customer base



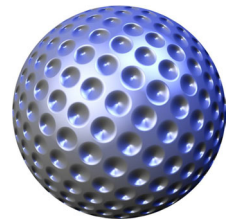
Agenda

1. Introduction
2. Basics
3. Advanced Usage
4. Discussion



Introduction

1. Introduction
2. Basics
3. Advanced Use
4. Discussion



Why a Python DB API specification ?

- 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

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
 - cruft from the past
- Use **database abstraction layers** for higher level database interfacing
 - SQL Wrappers
 - Object Relational Mappers (ORM)

History of the Python DB API: Version 1.0

- 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**

History of the Python DB API: Version 2.0

- 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 the current DB API version !
- Now referenced as [PEP 249](#)

Python DB API: The Future

- Many database modules provide extensions to the API standard
- The DB API addresses this by defining a **set of standard extensions**
- The DB API still has a few caveats: these will get solved in a version 3.0
 - e.g. flexible data type mappings

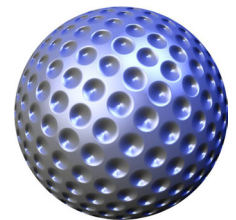
Python DB API 2.0 compatible modules

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

- mxODBC (SQL Server, DB2, Sybase, Oracle, etc.)
- mxODBC Connect (same as mxODBC, but more platforms)

Basics

1. Introduction
2. Basics
3. Advanced Usage
4. Discussion



Concepts

- The DB API uses **two main concepts** for processing database queries:
- **Connection objects**
 - Network/RPC connection to the database
 - Transactions
- **Cursor objects**
 - Statement execution
 - Access to results

Connection Objects

- Connection objects **logically wrap a database connection**
- Provide network/RPC access to the database

- Examples:

```
conn = Connect(Datasourcename, Username, Password)
```

```
conn = DriverConnect("DSN=test;UID=test;PWD=test")
```

- You **can't execute statements** on connection objects

Connection Objects: Transactions

- Connection objects also provide the means to handle **database transactions**
- Transactions are logical groups of statements executed on a connection (units of recovery)
- The main benefit is that **you can undo changes very easily**
 - Ensures data consistency at all times
 - Not all databases provide transactions !

Cursor Objects

- Created on connections using the `conn.cursor()` method
- Cursor objects provide ways to
 - manipulate and
 - query the database

- Example:

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

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

Cursor Objects: Executing SQL statements

- Cursor objects provide a `cursor.execute()` method to execute SQL statements on a connection
- The method also accepts a `sequence of parameters` to customize the SQL statement

- Example:

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

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 binding parameters in the SQL (place holder '?' for mxODBC) and passing in 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'))
```

Cursor Objects: Accessing query data

- Cursor objects also hold the query data after a statement was executed, the “result set”
- Provide `cursor.fetch*()` methods to read data from the result set
- Example:

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

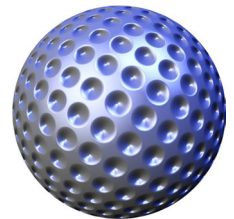
Example Session

- Sample session

```
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')]
```

Advanced Usage

1. Introduction
2. Basics
3. Advanced Usage
4. Discussion



Advanced Usage: Transactions

- Transactions logically wrap (multiple) statements into blocks of execution
 - Benefit: You can undo changes very easily
- The DB API supports transactions (if the database supports them) via methods on the connection object:
- `conn.commit()`
 - Write all changes of the last transaction block to the database
- `conn.rollback()`
 - Undo all changes applied to the database on the connection.

Advanced Usage: Problems with Transactions

- **Transaction Isolation:** Who will see my changes and when ?
 - Database dependent
 - Sometimes configurable per connection
(e.g. mxODBC supports this if the underlying database does)
- **Database Locks**
 - Multiple processes accessing the same table
 - Usually implicit and difficult to get around
 - Needed to prevent inconsistencies

Advanced Usage: Two Phase Commit Transactions

- Enables transactions across a set of resources, e.g.
 - databases
 - file system
 - queues (MQ Series)
 - etc.
- Advantage:
 - If the transaction fails on one resource, all changes to the other resources are rolled back as well
- The DB API defines the `.tpc_*()` API extension for these
 - Warning: Still very new !
 - Only few databases support two phase commit

Advanced Usage: Schema Introspection

- Finding the column types of an existing table:
 - Standard trick:
 - `cursor.execute('select * from testtable where 1=0')`
 - look at the `cursor.description` attribute
- More advanced: use catalog methods from `mxODBC`

```
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)]
```

Advanced Usage: Multiple Result Sets

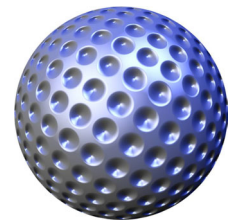
- Stored procedures executed in the database server often create more than one result set.
- The DB API always makes the first result available, but also allows jumping to the next using the `cursor.next()` method

- Example

```
cursor.execute('myProcedure()')
first_result_set = cursor.fetchall()
if cursor.next():
    second_result_set = cursor.fetchall()
```

Discussion

1. Introduction
2. Basics
3. Advanced Usage
4. Discussion



Conclusion

The Python Database API is

easy to use,

yet powerful !



Using the Python Database API



And finally...



Thank you for your time.

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/>