Python Database APIs

Accessing databases is easy in Python

EuroPython Conference 2002
Charleroi, Belgium

Marc-André Lemburg

EGENIX.COM Software GmbH
Germany

Python Database APIs: Overview

1. Background
2. Basics
3. Advanced Usage
4. Examples and Conclusion

Python Database APIs: Part 1

1. Background
2. Basics
3. Advanced Usage
4. Examples and Conclusion

Background: Why 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
- Things that it is not...
  - a single interface with pluggable drivers
  - a software package you can download
  - application abstraction layers provide these

Background: History

- 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
- DB API 1.0 provided a good start for coding database modules but had some caveats
- After 2-3 years a new effort was started to solve most of the caveats
  - Result: DB API 2.0

    (after long discussions on the Python Database SIG mailing list and solved most of these problems)
- The DB API still has a few caveats and many database modules provide extensions to the API standard
  - Result: a set of standard extensions were defined

Python Database APIs: Part 2

1. Background
2. Basics
3. Advanced Usage
4. Examples and Conclusion

Basics: Connection Objects

- Connection objects logically wrap a database connections

  Example:

  conn = Connect(Datasourcename, Username, Password)

- You can't execute statements on connection objects
  - Database cursors are needed to do this
- Connection objects provide the means to handle transactions
  - Transactions are logical groups of statements executed on a connection.
  - The main benefit is that you can undo changes very easily.
  - Watch out: Not all databases provide transactions !

Basics: Cursor Objects

- Cursor objects provide a way to "talk" to the database

  Example:

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

- Cursor objects can have state: e.g. they hold the query data after a statement was executed

  Example:

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

Basics: Cursor Objects

- Passing data from Python to the database
  - First option:
    Quoting values and sending plain SQL to the database
  - Second option (preferred):
    Using binding parameters in the SQL ('?' for ODBC) and passing in the data using Python lists and tuples
  - Problem: both are database backend dependent

Example:

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

Basics: Cursor Objects

- Sample session (mxODBC using MS Access on Windows)

```
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')]
```

Python Database APIs: Part 3

1. Background
2. Basics
3. Advanced Usage
4. Examples and Conclusion

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:
  - .commit()
    - Write all changes of the last transaction block to the database
  - .rollback()
    - Undo all changes applied to the database on the connection.
- Transaction Isolation: who will see my changes ?
  - Database dependent
  - Sometimes configurable per connection (e.g. mxODBC supports this if the underlying database does)

Advanced Usage: Schema Introspection

- Finding the column types of an existing table:
  - Standard trick:
    1. execute SELECT * FROM TESTTABLE WHERE 1=0
    2. look at the .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)]
- Caveat: only available in mxODBC

Python Database APIs: Part 4

1. Background
2. Basics
3. Advanced Usage
4. Examples and Conclusion

Conclusion:

The Python Database API is easy,

yet powerful !

... use it :-)

Questions...

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/