DS108 Python and Relational DB

Ming-Ling Lo
20191002

Python’s DB-API

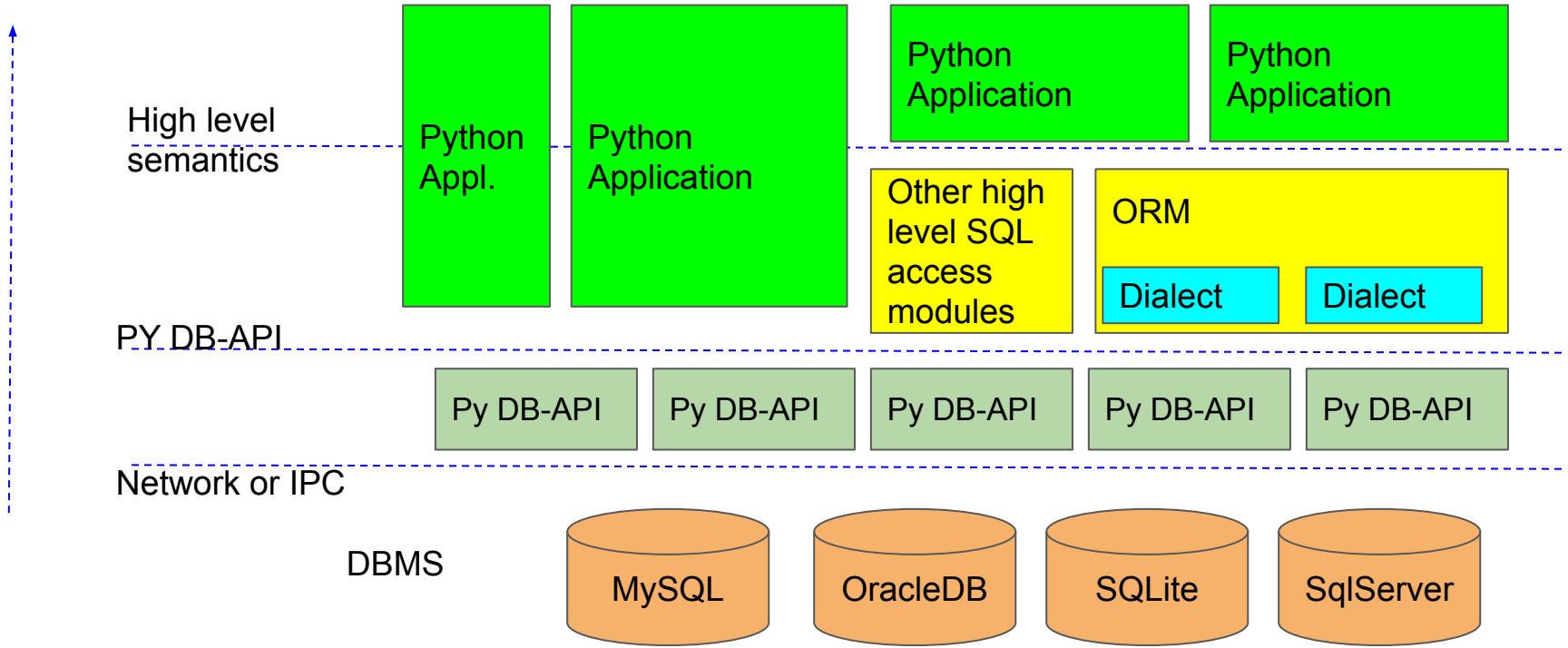
- The Python Database API (DB-API) defines a standard interface for Python database access modules.
  - It’s documented in **PEP 249**.
  - Nearly all Python database modules conform to this interface
    - E.g.: sqlite3, psycopg, mysql-python
  - Tutorials
    - https://halfcooked.com/presentations/osdc2006/python_databases.html
    - http://www.amk.ca/python/writing/DB-API.html

Python and SQL Access Software

- In order to access SQL databases, a Python application can:
  - Talk to a software module that implement DB-API
  - Talk to a high level software module that contains and encapsulates DB-API

- ORM (Object-relational mapper)
  - A type of high-level software module that hide the details of SQL databases
  - Let’s directly “store objects” into databases
  - The module maps your objects into rows and columns of underlying relational database

Python and RDB Architecture



Most Popular SQL Modules

- Pyodbc
  - A Python DB API 2 module for ODBC.
  - This project provides an up-to-date, convenient interface to ODBC using native data types like datetime and decimal.
- SQLAlchemy
  - Both an ORM and a DB-API connector.
- Mysql.connector
  - Connect to mySQL, support by mySQL community
- Pymysql
  - Another way to connect to mySQL

Python SQL Modules

- **SQLAlchemy**
  - SQLAlchemy is a commonly used database toolkit. Unlike many database libraries it not only provides an ORM layer but also a generalized API for writing database-agnostic code without SQL.
- Records
  - Records is minimalist SQL library, designed for sending raw SQL queries to various databases. Data can be used programmatically or exported to a number of useful data formats.
- PugSQL
  - PugSQL is a simple Python interface for organizing and using parameterized, handwritten SQL. It is an anti-ORM that is philosophically lo-fi, but it still presents a clean interface in Python.

Python SQL Modules

- **Django** ORM
  - The Django ORM is the interface used by Django to provide database access.
  - based on **models**, an abstraction that makes it easier to manipulate data in Python.
    - Each model is a Python class that subclasses django.db.models.Model.
    - Each attribute of the model represents a database field.
    - Django gives you an automatically-generated database-access API;
- Peewee
  - another ORM with a focus on being lightweight with support for Python 2.6+/3.2+
  - supports SQLite, MySQL, and PostgreSQL by default.
  - The model layer is similar to that of the Django ORM and it has SQL-like methods to query data.
  - SQLite, MySQL, and PostgreSQL are supported out-of-the-box
    - there is a collection of add-ons available.

Python SQL Modules

- PonyORM
  - PonyORM is an ORM that takes a different approach to querying the database.
  - Instead of writing an SQL-like language or boolean expressions, Python’s generator syntax is used.
  - There’s also a graphical schema editor that can generate PonyORM entities for you.
  - supports Python 2.6+/3.3+ and can connect to SQLite, MySQL, PostgreSQL, and Oracle.
- SQLObject
  - SQLObject is yet another ORM.
  - It supports a wide variety of databases: common database systems like MySQL, PostgreSQL, and SQLite and more exotic systems like SAP DB, SyBase, and Microsoft SQL Server. It only supports Python 2 from Python 2.6 upwards.

List of ORMs (Object Relational Mappers)

- Axiom: MIT-licensed, SQLite-based
- Bazaar ORM: Easy to use and powerful abstraction layer between relational database and object oriented application.
- dal.py: a DB Abstraction Layer (DAL); API that maps Python objects into DB objects such as queries, tables, and records.
- DbObj: ORM
- libschevo: Next-generation OO DBMS.
- MiddleKit: ORM
- Modeling Object-Relational Bridge: ORM and schema design with Zope integration
- Object Relational Membrame: ORM
- PyDo: ORM
- quick_orm: ORM on top of SQLAlchemy to make things simple
- SQLObject: ORM
- SQLAlchemy: SQL Toolkit and ORM
- Storm: Clean and powerful ORM by Canonical.

List of SQL Wrappers & Generators

- Db_row: SQL result wrapper
- pySQLFace: SQL interface over DBAPI2. It provides a database specific API to retreive and save data by creating SQL/DML command objects from a configuration file.
- PyTable
- The PythonWebModules web.**database** module - Database abstraction layer to make it possible to run the same SQL on different databases without changing your code.
- The PythonWebModules web.database.**object** module - ORM - treat an SQL database like python objects for easy programming, the SQL is done behind the scenes.
- QLime (Freshmeat entry): Easy to use, transparent data access to relational databases or other data sources.
- simpleQL: SQL generator using live translation of generator expressions

Relational Python

- Dee: A proposal to supersede SQL and the need for database sub-languages. Adds truly relational capabilities to Python (no wrappers, no mappers).

Python DB API 2

Python DB API 2 (PEP249)

- Framework
  - **Connection objects**
    - Input connection parameters
    - Return connection objects
  - **Cursor objects**
  - Type objects
  - Global values
  - Exception
- The framework is written for people who want to implement a DB-API software module

Module Globals

- These module globals must be defined:
- **Apilevel**
  - String constant stating the supported DB API level.
  - Currently only the strings "1.0" and "2.0" are allowed. If not given, a DB-API 1.0 level interface should be assumed.
- **Threadsafety**
  - **level of thread safety** the interface

threadsafety	Meaning
0	Threads may not share the module.
1	Threads may share the module, but not connections.
2	Threads may share the module and connections.
3	Threads may share the module, connections and cursors.

- Paramstyle
  - String constant stating the type of parameter marker formatting expected by the interface. Possible values are [2]:

paramstyle	Meaning
qmark	Question mark style, e.g. ...WHERE name=?
numeric	Numeric, positional style, e.g. ...WHERE name=:1
named	Named style, e.g. ...WHERE name=:name
format	ANSI C printf format codes, e.g. ...WHERE name=%s
pyformat	Python extended format codes, e.g. ...WHERE name=%(name)s

Exceptions

```
StandardError
|__Warning
|__Error
   |__InterfaceError
   |__DatabaseError
      |__DataError
      |__OperationalError
      |__IntegrityError
      |__InternalError
      |__ProgrammingError
        |__NotSupportedError
```

- The module should make all error information available through exceptions
- **Warning**
  - Exception raised for important warnings like data truncations while inserting, etc.
- **Error**
  - Exception that is the base class of all other error exceptions. You can use this to catch all errors with one single except statement.

Exception Listing

- InterfaceError
  - related to the database interface rather than the database itself.
- DatabaseError
  - errors related to the database.
- DataError
  - errors due to data processing like division by zero, numeric value out of range, etc.
- OperationalError
  - errors related to DB operation, not necessarily under control of programmer
  - e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred, etc.
- IntegrityError
  - The relational integrity of the database is affected, e.g. a foreign key check fails.
- InternalError
  - The database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc. .
- ProgrammingError
  - Programming errors, e.g. table not found, syntax error in the SQL statement, wrong number of parameters specified, etc.
- NotSupportedError
  - A DB function API was used which is not supported by the DB, e.g. requesting a rollback() on a DB that does not support transaction or has transactions turned off.

Connection Object Methods

- Constructor: get a connection
- .**close**()
  - Close the connection now (rather than whenever .__del__() is called).
  - Note that closing a connection without committing the changes first will cause an implicit rollback to be performed.
- .commit()
  - Commit any pending transaction to the database.
  - Note that auto-commit must be turned off for commit() to work.
  - Database modules that do not support transactions should implement this method with void functionality.
- .rollback()
  - Optional, since not all databases provide transaction support.
  - Causes the database to roll back to the start of any pending transaction.
  - Closing a connection without committing the changes first will cause an implicit rollback to be performed.
- .**cursor**()
  - **Return a new Cursor Object** using the connection.
  - If the database does not provide a direct cursor concept, the module will have to emulate cursors using other means.

Cursor Object Attributes

- Cursor Objects
  - Represent a database cursor, used to manage the context of an SQL command
  - Cursors created from the same connection are not isolated, i.e., any changes done to the database by a cursor are immediately visible by the other cursors.
  - Cursors created from different connections can or can not be isolated, depending on how the transaction support is implemented (see also the connection's .rollback() and .commit() methods).
- Cursor attributes **.rowcounts**
  - Read-only attribute
  - the number of rows that the last .execute*() read, inserted or updated
- Cursor attributes **.description**
  - Read-only
  - A sequence of 7-item sequences.
  - Each of these sequences contains information describing one result column:
    - Name (must)
    - Type_code (must)
    - Display_size (operational)
    - Internal_size (operational)
    - Precision (operational)
    - Scale (operational)
    - Null_ok (operational)
  - Will be None for operations that do not return rows or if the cursor has not execute any operation yet.
  - Type_code: see Type Objects specification

Cursor Object Methods

- .callproc(procname [, parameters])
  - (optional) Call a stored database procedure with the given name.
  - The sequence of parameters must contain one entry for each argument that the procedure expects.
  - The result of the call is returned as modified copy of the input sequence. Input parameters are left untouched, output and input/output parameters replaced with possibly new values.
  - The procedure may also provide a **result set** as output. This must then be made available through the standard .fetch*() methods.

- .**close**()
  - Close the cursor now (rather than whenever __del__ is called).
  - The cursor will be unusable from this point forward; an Error (or subclass) exception will be raised if any operation is attempted with the cursor.

Cursor Methods (2)

- .**execute**(operation [, parameters])
  - Prepare and execute a database operation (query or command).
  - Parameters may be provided as sequence or mapping and will be bound to variables in the operation.
  - Variables are specified in a database-specific notation (see the module's paramstyle attribute for details).
  - A reference to the operation will be retained by the cursor. If the same operation object is passed in again, then the cursor can optimize its behavior.
  - This is most effective for algorithms where the same operation is used, but different parameters are bound to it (many times).
  - For maximum efficiency when reusing an operation, it is best to use the .setinputsizes() method to specify the parameter types and sizes ahead of time. It is legal for a parameter to not match the predefined information; the implementation should compensate, possibly with a loss of efficiency.
  - The parameters may also be specified as list of tuples to e.g. insert multiple rows in a single operation, but this kind of usage is deprecated: .executemany() should be used instead.
  - Return values are not defined.

Cursor Methods (3)

- .executemany(operation, seq_of_parameters)
  - Prepare a database operation (query or command) and then execute it against all parameter sequences or mappings found in the sequence seq_of_parameters.
  - Modules are free to implement this method using multiple calls to the .execute() method or by using array operations to have the database process the sequence as a whole in one call.
  - Use of this method for an operation which produces one or more result sets constitutes undefined behavior.
  - The implementation is permitted (but not required) to raise an exception when it detects that a result set has been created by an invocation of the operation.
  - The same comments as for .execute() also apply accordingly to this method.
  - Return values are not defined.
- .**fetchone**()
  - Fetch the next row of a query result set, returning a single sequence, or None when no more data is available. [6]
  - An Error (or subclass) exception is raised if the previous call to .execute*() did not produce any result set or no call was issued yet.

Cursor Methods (4)

- .fetchmany([size=cursor.arraysize])
  - Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a list of tuples).
  - An empty sequence is returned when no more rows are available.
  - The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's arraysize determines the number of rows to be fetched.
  - The method should try to fetch as many rows as indicated by the size parameter.
  - An Error (or subclass) exception is raised if the previous call to .execute*() did not produce any result set or no call was issued yet.
  - Performance considerations: for optimal performance, it is usually best to use the .arraysize attribute. If the size parameter is used, then it is best for it to retain the same value from one .fetchmany() call to the next.
- .**fetchall**()
  - Fetch all (remaining) rows of a query result, returning them as a sequence of sequences (e.g. a list of tuples). Note that the cursor's arraysize attribute can affect the performance of this operation.
  - An Error (or subclass) exception is raised if the previous call to .execute*() did not produce any result set or no call was issued yet.

Cursor Methods (5)

- .nextset()
  - (This method is optional since not all databases support multiple result sets. [3])
  - This method will make the cursor skip to the next available set, discarding any remaining rows from the current set.
  - If there are no more sets, the method returns None. Otherwise, it returns a true value and subsequent calls to the .fetch*() methods will return rows from the next result set.
  - An Error (or subclass) exception is raised if the previous call to .execute*() did not produce any result set or no call was issued yet.
- .arraysize
  - This read/write attribute specifies the number of rows to fetch at a time with .fetchmany(). It defaults to 1 meaning to fetch a single row at a time.
  - Implementations must observe this value with respect to the .fetchmany() method, but are free to interact with the database a single row at a time. It may also be used in the implementation of .executemany().
- .setinputsizes(sizes)
  - This can be used before a call to .execute*() to predefine memory areas for the operation's parameters.

Cursor Methods (6)

- sizes is specified as a sequence — one item for each input parameter. The item should be a Type Object that corresponds to the input that will be used, or it should be an integer specifying the maximum length of a string parameter. If the item is None, then no predefined memory area will be reserved for that column (this is useful to avoid predefined areas for large inputs).
- This method would be used before the .execute*() method is invoked.
- Implementations are free to have this method do nothing and users are free to not use it.

- .setoutputsize(size [, column])
  - Set a column buffer size for fetches of large columns (e.g. LONGs, BLOBs, etc.). The column is specified as an index into the result sequence. Not specifying the column will set the default size for all large columns in the cursor.
  - This method would be used before the .execute*() method is invoked.
  - Implementations are free to have this method do nothing and users are free to not use it.