PYTHON PROGRAMMING

Presented By
POTU NARAYANA
Research Scholar
Osmania University
Dept. of CSE
B.Tech, M.Tech, M.A, PGDHR, MIAENG, MIS,(Ph.D)
Email : pnarayana@osmania.ac.in
Contact : +91 9704868721

UNIT 5
Database Programming

- In this we discuss how to communicate with databases from Python. Earlier, we discussed simplistic persistent storage, but in many cases, a full-fledged relational database management system (RDBMS) is required for your application.

Introduction

Persistent Storage

- In any application, there is a need for persistent storage. Generally, there are three basic storage mechanisms: files, a relational database system (RDBMS), or some sort of hybrid, i.e., an API (application programmer interface) that "sits on top of" one of those existing systems, an object relational mapper (ORM), file manager, spreadsheet, configuration file, etc.

- In an earlier chapter, we discussed persistent storage using both plain file access as well as a Python and DBM overlay on top of files, i.e., *dbm, dbhash/bsddb files, shelve (combination of pickle and DBM), and using their dictionary-like object interface. This chapter will focus on using RDBMSs for the times when files or writing your own system does not suffice for larger projects.

Basic Database Operations and SQL

- Before we dig into databases and how to use them with Python, we want to present a quick introduction (or review if you have some experience) to some elementary database concepts and the Structured Query Language (SQL).
- **Underlying Storage**
- Databases usually have a fundamental persistent storage using the file system, i.e., normal operating system files, special operating system files, and even raw disk partitions.
- **User Interface**
- Most database systems provide a command-line tool with which to issue SQL commands or queries. There are also some GUI tools that use the command-line clients or the database client library, giving users a much nicer interface.
- **Databases**
- An RDBMS can usually manage multiple databases, e.g., sales, marketing, customer support, etc., all on the same server (if the RDBMS is server-based; simpler systems are usually not). In the examples we will look at in this chapter, MySQL is an example of a server-based RDBMS because there is a server process running continuously waiting for commands while neither SQLite nor Gadfly have running servers.

- **Components**
- The *table is the storage abstraction for databases. Each row of data will have fields that correspond to* database *columns. The set of table definitions of columns and data types per table all put together* define the database *schema.*
- Databases are *created and dropped. The same is true for tables. Adding new rows to a database is* called *inserting, changing existing rows in a table is called updating, and removing existing rows in a* table is called *deleting. These actions are usually referred to as database commands or operations.* Requesting rows from a database with optional criteria is called *querying* .When you query a database, you can *fetch all of the results (rows) at once, or just iterate slowly over* each resulting row. Some databases use the concept of a *cursor for issuing SQL commands, queries,*
- and grabbing results, either all at once or one row at a time.

- **SQL**
- Database commands and queries are given to a database by SQL. Not all databases use SQL, but the majority of relational databases do. Here are some examples of SQL commands. Most databases are configured to be case-insensitive, especially database commands. The accepted style is to use CAPS for database keywords. Most command-line programs require a trailing semicolon (;) to terminate a SQL statement.
- ***Creating a Database***
- CREATE DATABASE test;
- GRANT ALL ON test.* to *user(s);*

- The first line creates a database named "test," and assuming that you are a database administrator, the second line can be used to grant permissions to specific users (or all of them) so that they can perform the database operations below.

- ***Using a Database***

  USE test;

- If you logged into a database system without choosing which database you want to use, this simple statement allows you to specify one with which to perform database operations.
- ***Dropping a Database***

  DROP DATABASE test;

- This simple statement removes all the tables and data from the database and deletes it from the system.
- ***Creating a Table***

  CREATE TABLE users (login VARCHAR(8), uid INT, prid INT);

- This statement creates a new table with a string column login and a pair of integer fields uid and prid.
- ***Dropping a Table***

  DROP TABLE users;

- This simple statement drops a database table along with all its data.

- ***Inserting a Row***

  INSERT INTO users VALUES('leanna', 311, 1);

- You can insert a new row in a database with the INSERT statement. Specify the table and the values that go into each field. For our example, the string 'leanna' goes into the login field, and 311 and 1 to uid and prid, respectively.
- ***Updating a Row***

  UPDATE users SET prid=4 WHERE prid=2;

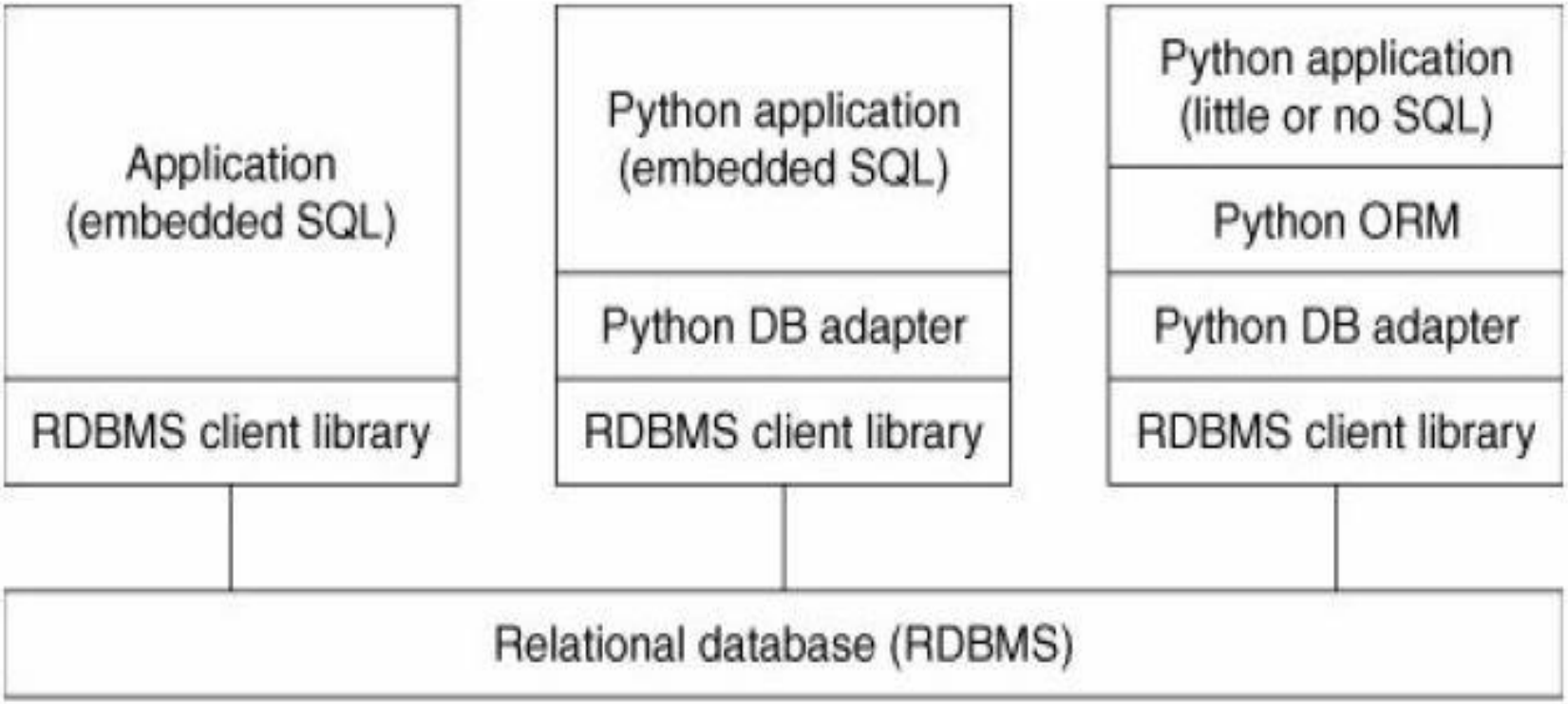
  UPDATE users SET prid=1 WHERE uid=311;

- To change existing table rows, you use the UPDATE statement. Use SET for the columns that are changing and provide any criteria for determining which rows should change. In the first example, all users with a "project ID" or prid of 2 will be moved to project #4. In the second example, we take one user (with a UID of 311) and move them to project #1.
- ***Deleting a Row***

  DELETE FROM users WHERE prid=%d;

  DELETE FROM users;

- To delete a table row, use the DELETE FROM command, give the table you want to delete rows from, and any optional criteria. Without it, as in the second example, all rows will be deleted.



Multitiered communication between application and database.
The first box is generally a C/C++ program while DB-API compliant adapters let you program applications in Python. ORMs can simplify an application by handling all of the database-specific details.

Python Database Application Programmer's Interface (DB-API)

- Where can one find the interfaces necessary to talk to a database? Simple. Just go to the database topics section at the main Python Web site. There you will find links to the full and current DB-API (version 2.0), existing database modules, documentation, the special interest group, etc. Since its inception, the DB-API has been moved into PEP 249. (This PEP obsoletes the old DB-API 1.0 specification which is PEP 248.) What is the DB-API?
- The API is a specification that states a set of required objects and database access mechanisms to provide consistent access across the various database adapters and underlying database systems. Like most community-based efforts, the API was driven by strong need.

- In the "old days," we had a scenario of many databases and many people implementing their own database adapters. It was a wheel that was being reinvented over and over again. These databases and adapters were implemented at different times by different people without any consistency of functionality. Unfortunately, this meant that application code using such interfaces also had to be customized to which database module they chose to use, and any changes to that interface also meant updates were needed in the application code.
- A special interest group (SIG) for Python database connectivity was formed, and eventually, an API was born ... the DB-API version 1.0. The API provides for a consistent interface to a variety of relational databases, and porting code between different databases is much simpler, usually only requiring tweaking several lines of code.

Module Attributes

- The DB-API specification mandates that the features and attributes listed below must be supplied. A DB-API compliant module must define the global attributes as shown in below table.

DB-API Module Attributes	
Attribute	***Description***
apilevel	Version of DB-API module is compliant with
threadsafety	Level of thread safety of this module
paramstyle	SQL statement parameter style of this module
connect()	Connect() function

- **Data Attributes**
- ***Apilevel***
- This string (not float) indicates the highest version of the DB-API the module is compliant with, i.e.,"1.0", "2.0", etc. If absent, "1.0" should be assumed as the default value.
- ***threadsafety***
- This an integer with these possible values:

  0: Not threadsafe, so threads should not share the module at all

  1: Minimally threadsafe: threads can share the module but not connections

  2: Moderately threadsafe: threads can share the module and connections but not cursors

  3: Fully threadsafe: threads can share the module, connections, and cursors

- ***paramstyle***
- The API supports a variety of ways to indicate how parameters should be integrated into an SQL statement that is eventually sent to the server for execution. This argument is just a string that specifies the form of string substitution you will use when building rows for a query or command.

paramstyle Database Parameter Styles		
Parameter Style	***Description***	***Example***
numeric	Numeric positional style	WHERE name=:1
named	Named style	WHERE name=:name
pyformat	Python dictionary printf() format conversion	WHERE name=%(name)s
qmark	Question mark style	WHERE name=?
format	ANSI C printf() format conversion	WHERE name=%s

- **Function Attribute(s)**
- connect() Function access to the database is made available through Connection objects. A compliant module has to implement a connect() function, which creates and returns a Connection object.

connect() Function Attributes	
Parameter	***Description***
user	Username
password	Password
host	Hostname
database	Database name
dsn	Data source name

```
connect(dsn='myhost:MYDB',user='guido',password='234$')
```

- **Exceptions**
- Exceptions that should also be included in the compliant module as globals are shown in the table.

DB-API Exception Classes	
Exception	***Description***
Warning	Root warning exception class
Error	Root error exception class
InterfaceError	Database interface (not database) error
DatabaseError	Database error
DataError	Problems with the processed data
OperationalError	Error during database operation execution
IntegrityError	Database relational integrity error
InternalError	Error that occurs within the database

DB-API Exception Classes	
Exception	***Description***
ProgrammingError	SQL command failed
NotSupportedError	Unsupported operation occurred

Connection Objects

- Connections are how your application gets to talk to the database. They represent the fundamental communication mechanism by which commands are sent to the server and results returned. Once a connection has been established (or a pool of connections), you create cursors to send requests to and receive replies from the database.

Connection Object Methods	
Method Name	***Description***
close()	Close database connection
commit()	Commit current transaction
rollback()	Cancel current transaction
cursor()	Create (and return) a cursor or cursor-like object using this connection
errorhandler(cxn, cur, errcls, errval)	Serves as a handler for given connection cursor

Cursor Objects

- Once you have a connection, you can start talking to the database. As we mentioned above in the introductory section, a cursor lets a user issue database commands and retrieve rows resulting from queries. A Python DB-API cursor object functions as a cursor for you, even if cursors are not supported in the database. In this case, the database adapter creator must implement CURSOR objects so that they act like cursors. This keeps your Python code consistent when you switch between database systems that have or do not have cursor support.

Cursor Object Attributes	
Object Attribute	***Description***
Arraysize	Number of rows to fetch at a time with fetch many(); defaults to 1
Connection	Connection that created this cursor (optional)
Description	Returns cursor activity (7-item tuples): (name, type_code, display_size, internal_ size, precision, scale, null_ok); only name and type_code are required
lastrowid	Row ID of last modified row (optional; if row IDs not supported, default to None)
rowcount	Number of rows that the last execute*() produced or affected
callproc(*func[, args])*	Call a stored procedure
close()	Close cursor

Cursor Object Attributes	
Object Attribute	***Description***
execute(*op[, args])*	Execute a database query or command
executemany(*op, args)*	Like execute() and map() combined; prepare and execute a database query or command over given arguments
fetchone()	Fetch next row of query result
fetchmany ([*size=cursor.arraysize])*	Fetch next size rows of query result
fetchall()	Fetch all (remaining) rows of a query result
__iter__()	Create iterator object from this cursor (optional; also see next ())
messages	List of messages (set of tuples) received from the database for cursor execution (optional)
next()	Used by iterator to fetch next row of query result (optional;like fetchone(), also see __iter__())

Cursor Object Attributes	
Object Attribute	***Description***
nextset()	Move to next results set (if supported)
Rownumber	Index of cursor (by row, 0-based) in current result set (optional)
setinput-sizes(*sizes)*	Set maximum input-size allowed (required but implementation optional)
setoutput size(*size[, col])*	Set maximum buffer size for large column fetches (required but implementation optional)

Type Objects and Constructors

- Oftentimes, the interface between two different systems are the most fragile. This is seen when converting Python objects to C types and vice versa. Similarly, there is also a fine line between Python objects and native database objects. As a programmer writing to Python's DB-API, the parameters you
- send to a database are given as strings, but the database may need to convert it to a variety of different, supported data types that are correct for any particular query.

Type Objects and Constructors	
Type Object	***Description***
Date(*yr, mo, dy)*	Object for a date value
Time(*hr, min, sec)*	Object for a time value
Timestamp(*yr, mo, dy, hr, min, sec)*	Object for a timestamp value
DateFromTicks(*ticks)*	Date object given number of seconds since the epoch
TimeFromTicks(*ticks)*	Time object given number of seconds since the epoch
TimestampFromTicks(*ticks)*	Timestamp object given number of seconds since the epoch
Binary(*string)*	Object for a binary (long) string value
STRING	Object describing string-based columns, e.g., VARCHAR
BINARY	Object describing (long) binary columns, i.e., RAW, BLOB

Type Objects and Constructors	
Type Object	***Description***
NUMBER	Object describing numeric columns
DATETIME	Object describing date/time columns
ROWID	Object describing "row ID" columns

Relational Databases

- ***Commercial RDBMSs***

  Informix

  Sybase

  Oracle

  MS SQL Server

  DB/2

  SAP

  Interbase

  Ingres

Open Source RDBMSs

MySQL

PostgreSQL

SQLite

Gadfly

Database APIs

JDBC

ODBC

Object-Relational Managers (ORMs)

- **Think Objects, Not SQL**
- Creators of these systems have abstracted away much of the pure SQL layer and implemented objects in Python that you can manipulate to accomplish the same tasks without having to generate the required lines of SQL. Some systems allow for more flexibility if you do have to slip in a few lines of SQL,but for the most part, you can avoid almost all the general SQL required.
- Database tables are magically converted to Python classes with columns and features as attributes and methods responsible for database operations. Setting up your application to an ORM is somewhat similar to that of a standard database adapter. Because of the amount of work that ORMs perform on your behalf, some things are actually more complex or require more lines of code than using an adapter directly. Hopefully, the gains you achieve in productivity make up for a little bit of extra work.

Python and ORMs

- The most well-known Python ORMs today are SQLAlchemy and SQLObject. We will give you examples of SQLAlchemy and SQLObject because the systems are somewhat disparate due to different philosophies,but once you figure these out, moving on to other ORMs is much simpler.
- Some other Python ORMs include PyDO/PyDO2, PDO, Dejavu, PDO, Durus, QLime, and ForgetSQL.Larger Web-based systems can also have their own ORM component, i.e., WebWare MiddleKit and Django's Database API. Note that "well-known" does not mean "best for your application." Although these others were not included in our discussion, that does not mean that they would not be right for your application.

Related Modules

Database-Related Modules and Websites

Name	*Online Reference or Description*
Gadfly	http://gadfly.sf.net
MySQL	http://mysql.com or http://mysql.org
MySQLdb a.k.a. MySQL-python	http://sf.net/projects/mysql-python
PostgreSQL	http://postgresql.org
Psycopg	http://initd.org/projects/psycopg1
Psycopg2	http://initd.org/software/initd/psycopg/
PyPgSQL	http://pypgsql.sf.net
PyGreSQL	http://pygresql.org

THANK YOU