Chapter 21. Database Programming

Chapter Topics

In this chapter, 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.

21.1. Introduction

21.1.1. 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.

21.1.2. 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.

Now that you are up to speed on basic database concepts, it should make following the rest of the chapter and its examples much easier. If you need additional help, there are plenty of database books out in the market that you can check out.

21.1.3. Databases and Python

We are going to cover the Python database API and look at how to access relational databases from Python, either directly through a database interface, or via an ORM, and how you can accomplish the same task but without necessarily having to give explicitly commands in SQL.

Topics such as database principles, concurrency, schema, atomicity, integrity, recovery, proper complex left JOINs, triggers, query optimization, transactions, stored procedures, etc., are all outside the scope of this text, and we will not be discussing these in this chapter other than direct use from a Python application. There are plenty of resources you can refer to for general information. Rather, we will present how to store and retrieve data to/from RDBMSs while playing within a Python framework. You can then decide which is best for your current project or application and be able to study sample code that can get you started instantly. The goal is to get you up to speed as quickly as possible if you need to integrate your Python application with some sort of database system.

We are also breaking out of our mode of covering only the "batteries included" features of the Python standard library. While our original goal was to play only in that arena, it has become clear that being able to work with databases is really a core component of everyday application development in the

Python world.

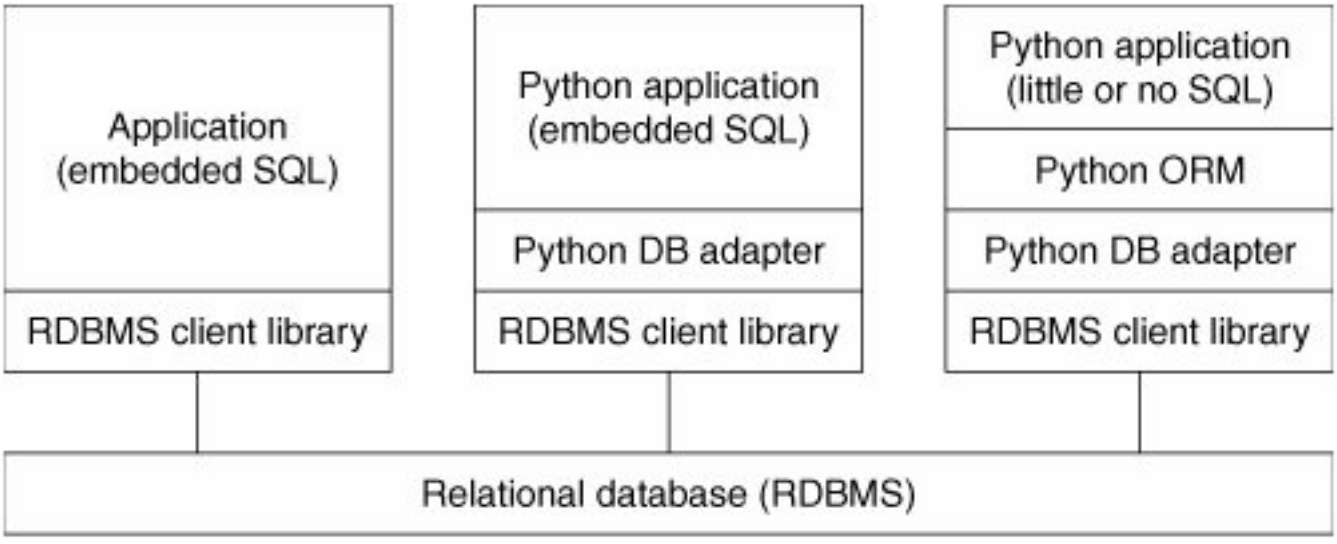
As a software engineer, you can probably only make it so far in your career without having to learn something about databases: how to use one (command-line and/or GUI interfaces), how to pull data out of one using the Structured Query Language (SQL), perhaps how to add or update information in a database, etc. If Python is your programming tool, then a lot of the hard work has already been done for you as you add database access to your Python universe. We first describe what the Python "DB-API" is, then give examples of database interfaces that conform to this standard.

We will give some examples using popular open source relational database management systems (RDBMSs). However, we will not include discussions of open source vs. commercial products, etc. Adapting to those other RDBMS systems should be fairly straightforward. A special mention will be given to Aaron Watters's Gadfly database, a simple RDBMS written completely in Python.

The way to access a database from Python is via an *adapter*. An adapter is basically a Python module that allows you to interface to a relational database's client library, usually in C. It is recommended that all Python adapters conform to the Python DB-SIG's Application Programmer Interface (API). This is the first major topic of this chapter.

Figure 21.1 illustrates the layers involved in writing a Python database application, with and without an ORM. As you can see, the DB-API is your interface to the C libraries of the database client.

Figure 21-1. 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.



PREV NEXT

21.2. 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. You will see an example of this later on in this chapter.

21.2.1. 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 Table 21.1.

Table 21.1. 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
(Various exceptions)	(*See Table 21.4*)

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

If a resource is shared, a synchronization primitive such as a spin lock or semaphore is required for atomic-locking purposes. Disk files and global variables are not reliable for this purpose and may interfere with standard mutex operation. See the threading module or the chapter on multithreaded programming (Chapter 16) on how to use a lock.

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 (see Table 21.2).

Table 21.2. `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. Table 21.3 shows the arguments to `connect()`.

Table 21.3. `connect()` Function Attributes

Parameter	*Description*
`user`	Username
`password`	Password
`host`	Hostname
`database`	Database name
`dsn`	Data source name

You can pass in database connection information as a string with multiple parameters (DSN) or individual parameters passed as positional arguments (if you know the exact order), or more likely, keyworded arguments. Here is an example of using `connect()` from PEP 249:

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

The use of DSN versus individual parameters is based primarily on the system you are connecting to. For example, if you are using an API like ODBC or JDBC, you would likely be using a DSN, whereas if you are working directly with a database, then you are more likely to issue separate login parameters. Another reason for this is that most database adapters have not implemented support for DSN. Below are some examples of non-DSN `connect()` calls. Note that not all adapters have implemented the specification exactly, e.g., `MySQLdb` uses `db` instead of `database`.

- `MySQLdb.connect(host='dbserv', db='inv', user='smith')`
- `PgSQL.connect(database='sales')`
- `psycopg.connect(database='template1', user='pgsql')`
- `gadfly.dbapi20.connect('csrDB', '/usr/local/database')`
- `sqlite3.connect('marketing/test')`

Exceptions

Exceptions that should also be included in the compliant module as globals are shown in Table 21.4.

Table 21.4. 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
`ProgrammingError`	SQL command failed
`NotSupportedError`	Unsupported operation occurred

21.2.2. `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.

Methods

`Connection` objects are not required to have any data attributes but should define the methods shown in Table 21.5.

Table 21.5. `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

When `close()` is used, the same connection cannot be used again without running into an exception.

The `commit()` method is irrelevant if the database does not support transactions or if it has an auto-commit feature that has been enabled. You can implement separate methods to turn auto-commit off or on if you wish. Since this method is required as part of the API, databases that do not have the concept of transactions should just implement "pass" for this method.

Like `commit(), rollback()` only makes sense if transactions are supported in the database. After execution, `rollback()` should leave the database in the same state as it was when the transaction began. According to PEP 249, "Closing a connection without committing the changes first will cause an implicit rollback to be performed."

If the RDBMS does not support cursors, `cursor()` should still return an object that faithfully emulates or imitates a real cursor object. These are just the minimum requirements. Each individual adapter developer can always add special attributes specifically for their interface or database.

It is also recommended but not required for adapter writers to make all database module exceptions (see above) available via a connection. If not, then it is assumed that `Connection` objects will throw the corresponding module-level exception. Once you have completed using your connection and cursors closed, you should `commit()` any operations and `close()` your connection.

21.2.3. `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.

Once you have created a cursor, you can execute a query or command (or multiple queries and commands) and retrieve one or more rows from the results set. Table 21.6 shows data attributes and methods that cursor objects have.

Table 21.6. `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
`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__`())
`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)

The most critical attributes of cursor objects are the `execute*()` and the `fetch*()` methods ... all the service requests to the database are performed by these. The `arraysize` data attribute is useful in setting a default size for `fetchmany()`. Of course, closing the cursor is a good thing, and if your database supports stored procedures, then you will be using `callproc()`.

21.2.4. 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.

For example, should the Python string be converted to a VARCHAR, a TEXT, a BLOB, or a raw BINARY object, or perhaps a DATE or TIME object if that is what the string is supposed to be? Care must be taken to provide database input in the expected format, so because of this another requirement of the DB-API is to create constructors that build special objects that can easily be converted to the appropriate database objects. Table 21.7 describes classes that can be used for this purpose. SQL NULL values are mapped to and from Python's NULL object, `None`.

Table 21.7. 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
`NUMBER`	Object describing numeric columns
`DATETIME`	Object describing date/time columns
`ROWID`	Object describing "row ID" columns

Changes to API Between Versions

Several important changes were made when the DB-API was revised from version 1.0 (1996) to 2.0 (1999):

- Required `dbi` module removed from API
- Type objects were updated
- New attributes added to provide better database bindings
- `callproc()` semantics and return value of `execute()` redefined
- Conversion to class-based exceptions

Since version 2.0 was published, some of the additional optional DB-API extensions that you read about above were added in 2002. There have been no other significant changes to the API since it was published. Continuing discussions of the API occur on the DB-SIG mailing list. Among the topics brought up over the last 5 years include the possibilities for the next version of the DB-API, tentatively named DB-API 3.0. These include the following:

- Better return value for `nextset()` when there is a new result set
- Switch from `float` to `Decimal`
- Improved flexibility and support for parameter styles
- Prepared statements or statement caching
- Refine the transaction model
- State the role of API with respect to portability
- Add unit testing

If you have strong feelings about the API, feel free to participate and join in the discussion. Here are some references you may find handy.

- http://python.org/topics/database
- http://www.linuxjournal.com/article/2605
- http://wiki.python.org/moin/DbApi3

21.2.5. Relational Databases

So, you are now ready to go. A burning question must be, "Interfaces to which database systems are available to me in Python?" That inquiry is similar to, "Which platforms is Python available for?" The answer is, "Pretty much all of them." Following is a list that is comprehensive but not exhaustive:

Commercial RDBMSs

- Informix
- Sybase
- Oracle
- MS SQL Server
- DB/2
- SAP
- Interbase
- Ingres

Open Source RDBMSs

- MySQL
- PostgreSQL
- SQLite
- Gadfly

Database APIs

- JDBC
- ODBC

To find a current list of what databases are supported, check out:

http://python.org/topics/database/modules.html

21.2.6. Databases and Python: Adapters

For each of the databases supported, there exists one or more adapters that let you connect to the target database system from Python. Some databases, such as Sybase, SAP, Oracle, and SQLServer, have more than one adapter available. The best thing to do is to find out which ones fit your needs best. Your questions for each candidate may include: how good its performance is, how useful is its documentation and/or Web site, whether it has an active community or not, what the overall quality and stability of the driver is, etc. You have to keep in mind that most adapters provide just the basic necessities to get you connected to the database. It is the extras that you may be looking for. Keep in mind that you are responsible for higher-level code like threading and thread management as well as management of database connection pools, etc.

If you are squeamish and want less hands-onfor example, if you wish to do as little SQL or database administration as much as possiblethen you may wish to consider object-relational mappers, covered

later on in this chapter.

Let us now look at some examples of how to use an adapter module to talk to a relational database. The real secret is in setting up the connection. Once you have this and use the DB-API objects, attributes, and object methods, your core code should be pretty much the same regardless of which adapter and RDBMS you use.

21.2.7. Examples of Using Database Adapters

First, let us look at a some sample code, from creating a database to creating a table and using it. We present examples using MySQL, PostgreSQL, and SQLite.

MySQL

We will use MySQL as the example here, along with the only MySQL Python adapter: `MySQLdb`, aka MySQL-python. In the various bits of code, we will also show you (deliberately) examples of error situations so that you have an idea of what to expect, and what you may wish to create handlers for.

We first log in as an administrator to create a database and grant permissions, then log back in as a normal client.

```
>>> import MySQLdb
>>> cxn = MySQLdb.connect(user='root')
>>> cxn.query('DROP DATABASE test')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
_mysql_exceptions.OperationalError: (1008, "Can't drop
database 'test'; database doesn't exist")
>>> cxn.query('CREATE DATABASE test')
>>> cxn.query("GRANT ALL ON test.* to ''@'localhost'")
>>> cxn.commit()
>>> cxn.close()
```

In the code above, we did not use a cursor. Some adapters have `Connection` objects, which can execute SQL queries with the `query()` method, but not all. We recommend you either not use it or check your adapter to make sure it is available.

The `commit()` was optional for us as auto-commit is turned on by default in MySQL. We then connect back to the new database as a regular user, create a table, and perform the usual queries and commands using SQL to get our job done via Python. This time we use cursors and their `execute()` method.

The next set of interactions shows us creating a table. An attempt to create it again (without first dropping it) results in an error.

```
>>> cxn = MySQLdb.connect(db='test')
>>> cur = cxn.cursor()
>>> cur.execute('CREATE TABLE users(login VARCHAR(8), uid INT)')
0L
```

Now we will insert a few rows into the database and query them out.

```
>>> cur.execute("INSERT INTO users VALUES('john', 7000)")
1L
>>> cur.execute("INSERT INTO users VALUES('jane', 7001)")
1L
>>> cur.execute("INSERT INTO users VALUES('bob', 7200)")
1L
>>> cur.execute("SELECT * FROM users WHERE login LIKE 'j%'")
2L
>>> for data in cur.fetchall():
... print '%s\t%s' % data
...
john   7000
jane   7001
```

The last bit features updating the table, either updating or deleting rows.

```
>>> cur.execute("UPDATE users SET uid=7100 WHERE uid=7001")
1L
>>> cur.execute("SELECT * FROM users")
3L
>>> for data in cur.fetchall():
...  print '%s\t%s' % data
...
john    7000
jane    7100
bob     7200
>>> cur.execute('DELETE FROM users WHERE login="bob"')
1L
>>> cur.execute('DROP TABLE users')
0L
>>> cur.close()
>>> cxn.commit()
>>> cxn.close()
```

MySQL is one of the most popular open source databases in the world, and it is no surprise that a Python adapter is available for it. Keep in mind that no database modules are available in the Python standard libraryall adapters are third-party packages that have to be downloaded and installed separately from Python. Please see the References section toward the end of the chapter to find out how to download it.

PostgreSQL

Another popular open source database is PostgreSQL. Unlike MySQL, there are no less than three current Python adapters available for Postgres: `psycopg`, PyPgSQL, and PyGreSQL. A fourth, PoPy, is now defunct, having contributed its project to combine with that of PyGreSQL back in 2003. Each of the three remaining adapters has its own characteristics, strengths, and weaknesses, so it would be a good idea to practice due diligence to determine which is right for you.

The good news is that the interfaces are similar enough that you can create an application that, say, measures the performance between all three (if that is a metric that is important to you). Here we show you the setup code to get a `Connection` object for each:

psycopg

```
>>> import psycopg
>>> cxn = psycopg.connect(user='pgsql')
```

PyPgSQL

```
>>> from pyPgSQL import PgSQL
>>> cxn = PgSQL.connect(user='pgsql')
```

PyGreSQL

```
>>> import pgdb
>>> cxn = pgdb.connect(user='pgsql')
```

Now comes some generic code that will work for all three adapters.

```
>>> cur = cxn.cursor()
>>> cur.execute('SELECT * FROM pg_database')
>>> rows = cur.fetchall()
>>> for i in rows:
...  print i
>>> cur.close()
>>> cxn.commit()
>>> cxn.close()
```

Finally, you can see how their outputs are slightly different from one another.

PyPgSQL

```
sales
template1
template0
```

psycopg

```
('sales', 1, 0, 0, 1, 17140, '140626', '3221366099',
'', None, None)
('template1', 1, 0, 1, 1, 17140, '462', '462', '', None,
'{pgsql=C*T*/pgsql}')
('template0', 1, 0, 1, 0, 17140, '462', '462', '', None,
'{pgsql=C*T*/pgsql}')
```

PyGreSQL

```
['sales', 1, 0, False, True, 17140L, '140626',
'3221366099', '', None, None]
['template1', 1, 0, True, True, 17140L, '462', '462',
```

```
'', None, '{pgsql=C*T*/pgsql}']
['template0', 1, 0, True, False, 17140L, '462',
'462', '', None, '{pgsql=C*T*/pgsql}']
```

SQLite

For extremely simple applications, using files for persistent storage usually suffices, but the most complex and data-driven applications demand a full relational database. SQLite targets the intermediate systems and indeed is a hybrid of the two. It is extremely lightweight and fast, plus it is serverless and requires little or no administration.

SQLite has seen a rapid growth in popularity, and it is available on many platforms. With the introduction of the `pysqlite` database adapter in Python 2.5 as the `sqlite3` module, this marks the first time that the Python standard library has featured a database adapter in any release.

2.5

It was bundled with Python not because it was favored over other databases and adapters, but because it is simple, uses files (or memory) as its backend store like the DBM modules do, does not require a server, and does not have licensing issues. It is simply an alternative to other similar persistent storage solutions included with Python but which happens to have a SQL interface.

Having a module like this in the standard library allows users to develop rapidly in Python using SQLite, then migrate to a more powerful RDBMS such as MySQL, PostgreSQL, Oracle, or SQL Server for production purposes if this is their intention. Otherwise, it makes a great solution to stay with for those who do not need all that horsepower.

Although the database adapter is now provided in the standard library, you still have to download the actual database software yourself. However, once you have installed it, all you need to do is start up Python (and import the adapter) to gain immediate access:

```
>>> import sqlite3
>>> cxn = sqlite3.connect('sqlite_test/test')
>>> cur = cxn.cursor()
>>> cur.execute('CREATE TABLE users(login VARCHAR(8), uid
      INTEGER)')
>>> cur.execute('INSERT INTO users VALUES("john", 100)')
>>> cur.execute('INSERT INTO users VALUES("jane", 110)')
>>> cur.execute('SELECT * FROM users')
>>> for eachUser in cur.fetchall():
...     print eachUser
...
(u'john', 100)
(u'jane', 110)
>>> cur.execute('DROP TABLE users')
<sqlite3.Cursor object at 0x3d4320>
>>> cur.close()
>>> cxn.commit()
>>> cxn.close()
```

Okay, enough of the small examples. Next, we look at an application similar to our earlier example with

MySQL, but which does a few more things:

- Creates a database (if necessary)
- Creates a table
- Inserts rows into the table
- Updates rows in the table
- Deletes rows from the table
- Drops the table

For this example, we will use two other open source databases. SQLite has become quite popular of late. It is very small, lightweight, and extremely fast for all the most common database functions. Another database involved in this example is Gadfly, a mostly SQL-compliant RDBMS written entirely in Python. (Some of the key data structures have a C module available, but Gadfly can run without it [slower, of course].)

Some notes before we get to the code. Both SQLite and Gadfly require the user to give the location to store database files (while MySQL has a default area and does not require this information from the use). The most current incarnation of Gadfly is not yet fully DB-API 2.0 compliant, and as a result, is missing some functionality, most notably the cursor attribute `rowcount` in our example.

Database Adapter Example Application

In the example below, we want to demonstrate how to use Python to access a database. In fact, for variety, we added support for three different database systems: Gadfly, SQLite, and MySQL. We are going to create a database (if one does not already exist), then run through various database operations such as creating and dropping tables, and inserting, updating, and deleting rows. Example 21.1 will be duplicated for the upcoming section on ORMs as well.

Example 21.1. Database Adapter Example (`ushuffle_db.py`)

This script performs some basic operations using a variety of databases (MySQL, SQLite, Gadfly) and a corresponding Python database adapter.

```
1   #!/usr/bin/env python
2
3   import os
4   from random import randrange as rrange
5
6   COLSIZ = 10
7   RDBMSs = {'s': 'sqlite', 'm': 'mysql', 'g': 'gadfly'}
8   DB_EXC = None
9
10  def setup():
11      return RDBMSs[raw_input('''
12  Choose a database system:
13
14 (M)ySQL
15 (G)adfly
16 (S)QLite
17
18  Enter choice: ''').strip().lower()[0]]
19
20  def connect(db, dbName):
21      global DB_EXC
22      dbDir = '%s_%s' % (db, dbName)
```

```
23
24      if db == 'sqlite':
25          try:
26              import sqlite3
27          except ImportError, e:
28               try:
29                   from pysqlite2 import dbapi2 as sqlite3
30               except ImportError, e:
31                   return None
32
33          DB_EXC = sqlite3
34          if not os.path.isdir(dbDir):
35              os.mkdir(dbDir)
36          cxn = sqlite.connect(os.path.join(dbDir, dbName))
37
38      elif db == 'mysql':
39          try:
40               import MySQLdb
41               import _mysql_exceptions as DB_EXC
42          except ImportError, e:
43               return None
44
45          try:
46               cxn = MySQLdb.connect(db=dbName)
47          except _mysql_exceptions.OperationalError, e:
48          cxn = MySQLdb.connect(user='root')
49               try:
50                     cxn.query('DROP DATABASE %s' % dbName)
51               except DB_EXC.OperationalError, e:
52                    pass
53               cxn.query('CREATE DATABASE %s' % dbName)
54               cxn.query("GRANT ALL ON %s.* to ''@'localhost'" % dbName)
55           cxn.commit()
56               cxn.close()
57               cxn = MySQLdb.connect(db=dbName)
58
59      elif db == 'gadfly':
60           try:
61                 from gadfly import gadfly
62                 DB_EXC = gadfly
63           except ImportError, e:
64                 return None
65
66           try:
67                 cxn = gadfly(dbName, dbDir)
68           except IOError, e:
69                 cxn = gadfly()
70                 if not os.path.isdir(dbDir):
71                     os.mkdir(dbDir)
72                 cxn.startup(dbName, dbDir)
73      else:
74           return None
75      return cxn
76
77  def create(cur):
78      try
79          cur.execute('''
80            CREATE TABLE users (
81              login VARCHAR(8),
```

```
82              uid INTEGER,
83              prid INTEGER)
84          ''')
85      except DB_EXC.OperationalError, e:
86          drop(cur)
87          create(cur)
88
89  drop = lambda cur: cur.execute('DROP TABLE users')
90
91  NAMES = (
92      ('aaron', 8312), ('angela', 7603), ('dave', 7306),
93      ('davina',7902), ('elliot', 7911), ('ernie', 7410),
94      ('jess', 7912), ('jim', 7512), ('larry', 7311),
95      ('leslie', 7808), ('melissa', 8602), ('pat', 7711),
96      ('serena', 7003), ('stan', 7607), ('faye', 6812),
97      ('amy', 7209),
98  )
99
100 def randName():
101     pick = list(NAMES)
102     while len(pick) > 0:
103         yield pick.pop(rrange(len(pick)))
104
105 def insert(cur, db):
106     if db == 'sqlite':
107         cur.executemany("INSERT INTO users VALUES(?, ?, ?)",
108         [(who, uid, rrange(1,5)) for who, uid in randName()])
109     elif db == 'gadfly':
110          for who, uid in randName():
111            cur.execute("INSERT INTO users VALUES(?, ?, ?)",
112               (who, uid, rrange(1,5)))
113     elif db == 'mysql':
114          cur.executemany("INSERT INTO users VALUES(%s, %s, %s)",
115          [(who, uid, rrange(1,5)) for who, uid in randName()])
116
117 getRC = lambda cur: cur.rowcount if hasattr(cur,
    'rowcount') else -1
118
119 def update(cur):
120     fr = rrange(1,5)
121     to = rrange(1,5)
122     cur.execute(
123         "UPDATE users SET prid=%d WHERE prid=%d" % (to, fr))
124     return fr, to, getRC(cur)
125
126 def delete(cur):
127     rm = rrange(1,5)
128     cur.execute('DELETE FROM users WHERE prid=%d' % rm)
129     return rm, getRC(cur)
130
131 def dbDump(cur):
132     cur.execute('SELECT * FROM users')
133     print '\n%s%s%s' % ('LOGIN'.ljust(COLSIZ),
134         'USERID'.ljust(COLSIZ), 'PROJ#'.ljust(COLSIZ))
135     for data in cur.fetchall():
136          print '%s%s%s' % tuple([str(s).title().ljust(COLSIZ) \
137              for s in data])
138
139 def main():
```

```
140      db = setup()
141      print '*** Connecting to %r database' % db
142      cxn = connect(db, 'test')
143      if not cxn:
144           print 'ERROR: %r not supported, exiting' % db
145           return
146      cur = cxn.cursor()
147
148      print '\n*** Creating users table'
149         create(cur)
150
151      print '\n*** Inserting names into table'
152      insert(cur, db)
153      dbDump(cur)
154
155      print '\n*** Randomly moving folks',
156      fr, to, num = update(cur)
157      print 'from one group (%d) to another (%d)' % (fr, to)
158      print '\t(%d users moved)' % num
159      dbDump(cur)
160
161      print '\n*** Randomly choosing group',
162      rm, num = delete(cur)
163      print '(%d) to delete' % rm
164      print '\t(%d users removed)' % num
165      dbDump(cur)
166
167      print '\n*** Dropping users table'
168      drop(cur)
169      cur.close()
170      cxn.commit()
171      cxn.close()
172
173  if__name__ == '__main__':
174     main()
```

Line-by-Line Explanation

Lines 118

The first part of this script imports the necessary modules, creates some global "constants" (the column size for display and the set of databases we are supporting), and features the `setup()` function, which prompts the user to select the RDBMS to use for any particular execution of this script.

The most notable constant here is `DB_EXC`, which stands for DataBase EXCeption. This variable will eventually be assigned the database exception module for the specific database system that the users chooses to use to run this application with. In other words, if users choose MySQL, `DB_EXC` will be `_mysql_exceptions`, etc. If we developed this application in more of an object-oriented fashion, this would simply be an instance attribute, i.e., `self.db_exc_module` or something like that.

Lines 2075

The guts of consistent database access happens here in the `connect()` function. At the beginning of each section, we attempt to load the requested database modules. If a suitable one is not found, `None` is returned to indicate that the database system is not supported.

Once a connection is made, then all other code is database and adapter independent and should work across all connections. (The only exception in our script is `insert().`) In all three subsections of this set of code, you will notice that a valid connection should be passed back as `cxn`.

If SQLite is chosen (lines 24-36), we attempt to load a database adapter. We first try to load the standard library's `sqlite3` module (Python 2.5+). If that fails, we look for the third-party `pysqlite2` package. This is to support 2.4.x and older systems with the pysqlite adapter installed. If a suitable adapter is found, we then check to ensure that the directory exists because the database is file based. (You may also choose to create an in-memory database.) When the `connect()` call is made to SQLite, it will either use one that already exists or make a new one using that path if it does not.

MySQL (lines 38-57) uses a default area for its database files and does not require this to come from the user. Our code attempts to connect to the specified database. If an error occurs, it could mean either that the database does not exist or that it does exist but we do not have permission to see it. Since this is just a test application, we elect to drop the database altogether (ignoring any error if the database does not exist), and re-create it, granting all permissions after that.

The last database supported by our application is Gadfly (lines 59-75). (At the time of writing, this database is mostly but not fully DB-API-compliant, and you will see this in this application.) It uses a startup mechanism similar to that of SQLite: it starts up with the directory where the database files should be. If it is there, fine, but if not, you have to take a roundabout way to start up a new database. (Why this is, we are not sure. We believe that the `startup()` functionality should be merged into that of the constructor `gadfly.gadfly().`)

Lines 7789

The `create()` function creates a new users table in our database. If there is an error, that is almost always because the table already exists. If this is the case, drop the table and re-create it by recursively calling this function again. This code is dangerous in that if the recreation of the table still fails, you will have infinite recursion until your application runs out of memory. You will fix this problem in one of the exercises at the end of the chapter.

The table is dropped from the database with the one-liner `drop()`.

Lines 91103

This is probably the most interesting part of the code outside of database activity. It consists of a constant set of names and user IDs followed by the generator `randName()` whose code can be found in Chapter 11 (Functions) in Section 11.10. The `NAMES` constant is a tuple that must be converted to a list for use with `randName()` because we alter it in the generator, randomly removing one name at a time until the list is exhausted. Well, if `NAMES` was a list, we would only use it once. Instead, we make it a tuple and copy it to a list to be destroyed each time the generator is used.

Lines 105115

The `insert()` function is the only other place where database-dependent code lives, and the reason is that each database is slightly different in one way or another. For example, both the adapters for SQLite and MySQL are DB-API-compliant, so both of their cursor objects have an `executemany()` function,

whereas Gadfly does not, so rows have to be inserted one at a time.

Another quirk is that both SQLite and Gadfly use the `qmark` parameter style while MySQL uses `format`. Because of this, the format strings are different. If you look carefully, however, you will see that the arguments themselves are created in a very similar fashion.

What the code does is this: for each name-userID pair, it assigns that individual to a project group (given by its project ID or `prid`). The project ID is chosen randomly out of four different groups `(randrange(1,5))`.

Line 117

This single line represents a conditional expression (read as: Python ternary operator) that returns the rowcount of the last operation (in terms of rows altered), or if the cursor object does not support this attribute (meaning it is not DB-API-compliant), it returns -1.

Conditional expressions were added in Python 2.5, so if you are using 2.4.x or older, you will need to convert it back to the "old-style" way of doing it:

```
getRC = lambda cur: (hasattr(cur, 'rowcount') \
    and [cur.rowcount] or [-1])[0]
```

If you are confused by this line of code, don't worry about it. Check the FAQ to see why this is, and get a taste of why conditional expressions were finally added to Python in 2.5. If you *are* able to figure it out, then you have developed a solid understanding of Python objects and their Boolean values.

2.5

Lines 119129

The `update()` and `delete()` functions randomly choose folks from one group. If the operation is update, move them from their current group to another (also randomly chosen); if it is delete, remove them altogether.

Lines 131137

The `dbDump()` function pulls all rows from the database, formats them for printing, and displays them to the user. The **`print`** statement to display each user is the most obfuscated, so let us take it apart.

First, you should see that the data were extracted after the SELECT by the `fetchall()` method. So as we iterate each user, take the three columns (`login, uid, prid`), convert them to strings (if they are not already), titlecase it, and format the complete string to be `COLSIZ` columns left-justified (right-hand space padding). Since the code to generate these three strings is a list (via the list comprehension), we need to convert it to a tuple for the format operator (%).

Lines 139174

The director of this movie is `main()`. It makes the individual functions to each function described above that defines how this script works (assuming that it does not exit due to either not finding a database adapter or not being able to obtain a connection [lines 143-145]). The bulk of it should be fairly self-explanatory given the proximity of the **`print`** statements. The last bits of `main()` close the cursor, and commit and close the connection. The final lines of the script are the usual to start the script.

PREV NEXT

21.3. Object-Relational Managers (ORMs)

As seen in the previous section, a variety of different database systems are available today, and most of them have Python interfaces to allow you to harness their power. The only drawback to those systems is the need to know SQL. If you are a programmer who feels more comfortable with manipulating Python objects instead of SQL queries, yet still want to use a relational database as your data backend, then you are a great candidate to be a user of ORMs.

21.3.1. 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.

21.3.2. 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.

21.3.3. Employee Role Database Example

We will port our user shuffle application `ushuffle_db.py` to both SQLAlchemy and SQLObject below. MySQL will be the backend database server for both. You will note that we implement these as classes because there is more of an object "feel" to using ORMs as opposed to using raw SQL in a database adapter. Both examples import the set of `NAMES` and the random name chooser from `ushuffle_db.py`. This is to avoid copying-and-pasting the same code everywhere as code reuse is a good thing.

SQLAlchemy

We start with SQLAlchemy because its interface is somewhat closer to SQL than SQLObject's interface. SQLAlchemy abstracts really well to the object world but does give you more flexibility in issuing SQL if you have to. You will find both of these ORMs (Examples 21.2 and 21.3) very similar in terms of setup and access, as well as being of similar size, and both shorter than `ushuffle_db.py` (including the sharing of the names list and generator used to randomly iterate through that list).

Example 21.2. SQLAlchemy ORM Example (`ushuffle_sa.py`)

This "user shuffle" application features SQLAlchemy paired up with the MySQL database as its backend.

```
#!/usr/bin/env python

import os
from random import randrange as rrange
from sqlalchemy import *
from ushuffle_db import NAMES, randName

FIELDS = ('login', 'uid', 'prid')
DBNAME = 'test'
COLSIZ = 10

class MySQLAlchemy(object):
    def __init__(self, db, dbName):
        import MySQLdb
        import _mysql_exceptions
        MySQLdb = pool.manage(MySQLdb)
        url = 'mysql://db=%s' % DBNAME
        eng = create_engine(url)
        try:
            cxn = eng.connection()
        except _mysql_exceptions.OperationalError, e:
            eng1 = create_engine('mysql://user=root')
            try:
                 eng1.execute('DROP DATABASE %s' % DBNAME)
            except _mysql_exceptions.OperationalError, e:
                 pass
            eng1.execute('CREATE DATABASE %s' % DBNAME)
            eng1.execute(
            "GRANT ALL ON %s.* TO ''@'localhost'" % DBNAME)
            eng1.commit()
            cxn = eng.connection()

        try:
            users = Table('users', eng, autoload=True)
        except exceptions.SQLError, e:
            users = Table('users', eng,
                Column('login', String(8)),
                Column('uid', Integer),
                Column('prid', Integer),
                redefine=True)

        self.eng = eng
        self.cxn = cxn
        self.users = users

    def create(self):
        users = self.users
           try:
            users.drop()
        except exceptions.SQLError, e:
            pass
         users.create()

```

```
 54     def insert(self):
 55         d = [dict(zip(FIELDS,
 56         [who, uid, rrange(1,5)])) for who, uid in randName()]
 57         return self.users.insert().execute(*d).rowcount
 58 
 59     def update(self):
 60         users = self.users
 61         fr = rrange(1,5)
 62         to = rrange(1,5)
 63         return fr, to, \
 64     users.update(users.c.prid==fr).execute(prid=to).rowcount
 65 
 66     def delete(self):
 67         users = self.users
 68         rm = rrange(1,5)
 69         return rm, \
 70     users.delete(users.c.prid==rm).execute().rowcount
 71 
 72     def dbDump(self):
 73         res = self.users.select().execute()
 74         print '\n%s%s%s' % ('LOGIN'.ljust(COLSIZ),
 75             'USERID'.ljust(COLSIZ), 'PROJ#'.ljust(COLSIZ))
 76         for data in res.fetchall():
 77             print '%s%s%s' % tuple([str(s).title().ljust
(COLSIZ) for s in data])
 78 
 79     def __getattr__(self, attr):
 80         return getattr(self.users, attr)
 81 
 82     def finish(self):
 83         self.cxn.commit()
 84         self.eng.commit()
 85 
 86  def main():
 87     print '*** Connecting to %r database' % DBNAME
 88     orm = MySQLAlchemy('mysql', DBNAME)
 89 
 90     print '\n*** Creating users table'
 91     orm.create()
 92 
 93     print '\n*** Inserting names into table'
 94     orm.insert()
 95     orm.dbDump()
 96 
 97        print '\n*** Randomly moving folks',
 98     fr, to, num = orm.update()
 99     print 'from one group (%d) to another (%d)' % (fr, to)
100     print '\t(%d users moved)' % num
101     orm.dbDump()
102 
103     print '\n*** Randomly choosing group',
104     rm, num = orm.delete()
105     print '(%d) to delete' % rm
106     print '\t(%d users removed)' % num
107     orm.dbDump()
108 
109     print '\n*** Dropping users table'
110     orm.drop()
111     orm.finish()
```

```
112
113  if__name__ == '__main__':
114    main()
```

Example 21.3. SQLObject ORM Example (ushuffle_so.py)

This "user shuffle" application features SQLObject paired up with the MySQL database as its backend.

```
1   #!/usr/bin/env python
2
3   import os
4   from random import randrange as rrange
5   from sqlobject import *
6   from ushuffle_db import NAMES, randName
7
8   DBNAME = 'test'
9   COLSIZ = 10
10  FIELDS = ('login', 'uid', 'prid')
11
12  class MySQLObject(object):
13      def __init__(self, db, dbName):
14           import MySQLdb
15           import _mysql_exceptions
16           url = 'mysql://localhost/%s' % DBNAME
17
18           while True:
19               cxn = connectionForURI(url)
20               sqlhub.processConnection = cxn
21               #cxn.debug = True
22              try:
23                   class Users(SQLObject):
24                       class sqlmeta:
25                            fromDatabase = True
26                       login = StringCol(length=8)
27                       uid = IntCol()
28                       prid = IntCol()
29                   break
30           except _mysql_exceptions.ProgrammingError, e:
31                class Users(SQLObject):
32                    login = StringCol(length=8)
33                    uid = IntCol()
34                    prid = IntCol()
35                break
36           except _mysql_exceptions.OperationalError, e:
37                  cxn1 = sqlhub.processConnection=
connectionForURI('mysql://root@localhost')
38               cxn1.query("CREATE DATABASE %s" % DBNAME)
39               cxn1.query("GRANT ALL ON %s.* TO ''@'
  localhost'" % DBNAME)
40               cxn1.close()
41          self.users = Users
42          self.cxn = cxn
43
```

```
44   def create(self):
45        Users = self.users
46         Users.dropTable(True)
47      Users.createTable()
48
49   def insert(self):
50      for who, uid in randName():
51          self.users(**dict(zip(FIELDS,
52               [who, uid, rrange(1,5)])))
53
54   def update(self):
55      fr = rrange(1,5)
56      to = rrange(1,5)
57      users = self.users.selectBy(prid=fr)
58      for i, user in enumerate(users):
59          user.prid = to
60      return fr, to, i+1
61
62   def delete(self):
63      rm = rrange(1,5)
64      users = self.users.selectBy(prid=rm)
65      for i, user in enumerate(users):
66          user.destroySelf()
67      return rm, i+1
68
69   def dbDump(self):
70      print '\n%s%s%s' % ('LOGIN'.ljust(COLSIZ),
71           'USERID'.ljust(COLSIZ), 'PROJ#'.ljust(COLSIZ))
72      for usr in self.users.select():
73          print '%s%s%s' % (tuple([str(getattr(usr,
74              field)).title().ljust(COLSIZ) \
75              for field in FIELDS]))
76
77   drop = lambda self: self.users.dropTable()
78   finish = lambda self: self.cxn.close()
79
80 def main():
81    print '*** Connecting to %r database' % DBNAME
82    orm = MySQLObject('mysql', DBNAME)
83
84    print '\n*** Creating users table'
85    orm.create()
86
87    print '\n*** Inserting names into table'
88    orm.insert()
89    orm.dbDump()
90
91    print '\n*** Randomly moving folks',
92    fr, to, num = orm.update()
93    print 'from one group (%d) to another (%d)' % (fr, to)
94    print '\t(%d users moved)' % num
95    orm.dbDump()
96
97    print '\n*** Randomly choosing group',
98    rm, num = orm.delete()
99    print '(%d) to delete' % rm
100   print '\t(%d users removed)' % num
101   orm.dbDump()
102
```

```
103   print '\n*** Dropping users table'
104   orm.drop()
105   orm.finish()
106
107 if__name__ == '__main__':
108   main()
```

Line-by-Line Explanation

Lines 110

As expected, we begin with module imports and constants. We follow the suggested style guideline of importing Python Standard Library modules first, followed by third-party or external modules, and finally, local modules to our application. The constants should be fairly self-explanatory.

Lines 1231

The constructor for our class, like `ushuffle_db.connect()`, does everything it can to make sure that there is a database available and returns a connection to it (lines 18-31). This is the only place you will see real SQL, as such activity is typically an operational task, not application-oriented.

Lines 3344

The `TRy-except` clause (lines 33-40) is used to reload an existing table or make a new one if it does not exist yet. Finally, we attach the relevant objects to our instance.

Lines 4670

These next four methods represent the core database functionality of table creation (lines 46-52), insertion (lines 54-57), update (lines 59-64), and deletion (lines 66-70). We should also have a method for dropping the table:

```
  def drop(self):
      self.users.drop()
```

or

```
   drop = lambda self: self.users.drop()
```

However, we made a decision to give another demonstration of *delegation* (as introduced in Chapter 13, Object-Oriented Programming). Delegation is where missing functionality (method call) is passed to another object in our instance which has it. See the explanation of lines 79-80.

Lines 7277

The responsibility of displaying proper output to the screen belongs to the `dbDump()` method. It extracts

the rows from the database and pretty-prints the data just like its equivalent in `ushuffle_db.py`. In fact, they are nearly identical.

Lines 7980

We deliberately avoided creating a `drop()` method for the table since it would just call the table's drop method anyway. Also, there is no added functionality, so why create yet another function to have to maintain? The `__getattr__()` special method is called whenever an attribute lookup fails.

If our object calls `orm.drop()` and finds no such method, `getattr (orm, 'drop')` is invoked. When that happens, `__getattr__()` is called and delegates the attribute name to `self.users`. The interpreter will find that self.users has a `drop` attribute and pass that method call to it: `self. users.drop()!`

Lines 8284

The last method is `finish()`, which commits the transaction.

Lines 86114

The `main()` function drives our application. It creates a `MySQLAlchemy` object and uses that for all database operations. The script is the same as for our original application, `ushuffle_db.py`. You will notice that the database parameter `db` is optional and does not serve any purpose here in `ushuffle_sa.py` or the upcoming SQLobject version `ushuffle_so.py`. This is a placeholder for you to add support for other RDBMSs in these applications (see Exercises at the end of the chapter).

Upon running this script, you may get output that looks like this:

```
$ ushuffle_sa.py
*** Connecting to 'test' database

*** Creating users table

*** Inserting names into table

LOGIN     USERID    PROJ#
Serena    7003      4
Faye      6812      4
Leslie    7808      3
Ernie     7410      1
Dave      7306      2
Melissa   8602      1
Amy       7209      3
Angela    7603      4
Jess      7912      2
Larry     7311      1
Jim       7512      2
Davina    7902      3
Stan      7607      4
Pat       7711      2
Aaron     8312      2
Elliot    7911      3

*** Randomly moving folks from one group (1) to another (3)
```

```
        (3 users moved)

LOGIN     USERID    PROJ#
Serena    7003      4
Faye      6812      4
Leslie    7808      3
Ernie     7410      3
Dave      7306      2
Melissa   8602      3
Amy       7209      3
Angela    7603      4
Jess      7912      2
Larry     7311      3
Jim       7512      2
Davina    7902      3
Stan      7607      4
Pat       7711      2
Aaron     8312      2
Elliot    7911      3
*** Randomly choosing group (2) to delete
        (5 users removed)

LOGIN     USERID    PROJ#
Serena    7003      4
Faye      6812      4
Leslie    7808      3
Ernie     7410      3
Melissa   8602      3
Amy       7209      3
Angela    7603      4
Larry     7311      3
Davina    7902      3
Stan      7607      4
Elliot    7911      3

*** Dropping users table
$
```

Line-by-Line Explanation

Lines 110

This modules imports and constant declarations are practically identical to those of `ushuffle_sa.py` except that we are using SQLObject instead of SQLAlchemy.

Lines 1242

The constructor for our class does everything it can to make sure that there is a database available and returns a connection to it, just like our SQLAlchemy example. Similarly, this is the only place you will see real SQL. Our application, as coded here, will result in an infinite loop if for some reason a Users table cannot be created in SQLObject.

We are trying to be clever in handling errors by fixing the problem and retrying the table (re)create. Since SQLobject uses metaclasses, we know that special magic is happening under the covers, so we have to define two different classesone for if the table already exists and another if it does not.

The code works something like this:

1. Try and establish a connection to an existing table; if it works, we are done (lines 23-29)
2. Otherwise, create the class from scratch for the table; if so, we are done (lines 31-36)
3. Otherwise, we have a database issue, so try and make a new database (lines 37-40)
4. Loop back up and try all this again

Hopefully it (eventually) succeeds in one of the first two places. When the loop is terminated, we attach the relevant objects to our instance as we did in `ushuffle_sa.py`.

Lines 4467, 7778

The database operations happen in these lines. We have table create (lines 44-47) and drop (line 77), insert (lines 49-52), update (lines 54-60), and delete (lines 62-67). The `finish()` method on line 78 is to close the connection. We could not use delegation for table drop like we did for the SQLAlchemy example because the would-be delegated method for it is called `dropTable()` not `drop()`.

Lines 6975

This is the same and expected `dbDump()` method, which pulls the rows from the database and displays things nicely to the screen.

Lines 80108

This is the `main()` function again. It works just like the one in `ushuffle_sa.py`. Also, the `db` argument to the constructor is a placeholder for you to add support for other RDBMSs in these applications (see Exercises at the end of the chapter).

Here is what your output may look like if you run this script:

```
$ ushuffle_so.py
*** Connecting to 'test' database
*** Creating users table
*** Inserting names into table
LOGIN     USERID    PROJ#
```

```
Jess      7912      1
Amy       7209      4
Melissa   8602      2
Dave      7306      4
Angela    7603      4
Serena    7003      2
Aaron     8312      1
Leslie    7808      1
Stan      7607      3
Pat       7711      3
Jim       7512      4
Larry     7311      3
Ernie     7410      2
Faye      6812      4
Davina    7902      1
Elliot    7911      4

*** Randomly moving folks from one group (2) to another (3)
        (3 users moved)

LOGIN     USERID    PROJ#
Jess      7912      1
Amy       7209      4
Melissa   8602      3
Dave      7306      4
Angela    7603      4
Serena    7003      3
Aaron     8312      1
Leslie    7808      1
Stan      7607      3
Pat       7711      3
Jim       7512      4
Larry     7311      3
Ernie     7410      3
Faye      6812      4
Davina    7902      1
Elliot    7911      4

*** Randomly choosing group (3) to delete
        (6 users removed)

LOGIN     USERID    PROJ#
Jess      7912      1
Amy       7209      4
Dave      7306      4
Angela    7603      4
Aaron     8312      1
Leslie    7808      1
Jim       7512      4
Faye      6812      4
Davina    7902      1
Elliot    7911      4

*** Dropping users table
$
```

21.3.4. Summary

We hope that we have provided you with a good introduction to using relational databases with Python. When your application's needs go beyond those offered by plain files, or specialized files like DBM, pickled, etc., you have many options. There are a good number of RDBMSs out there, not to mention one completely implemented in Python, freeing one from having to install, maintain, or administer a real database system. Below, you will find information on many of the Python adapters plus database and ORM systems out there. We also suggest checking out the DB-SIG pages as well as the Web pages and mailing lists of all systems of interest. Like all other areas of software development, Python makes things easy to learn and simple to experiment with.

PREV NEXT

21.4. Related Modules

Table 21.8 lists most of the common databases out there along with working Python modules and packages that serve as adapters to those database systems. Note that not all adapters are DB-API-compliant.

Table 21.8. Database-Related Modules and Websites

Name	*Online Reference or Description*
Databases	
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
PoPy	Deprecated; merged into PyGreSQL project
SQLite	http://sqlite.org
pysqlite	http://initd.org/projects/pysqlite
`sqlite3`[a]	`pysqlite` integrated into Python Standard Library; use this one unless you want to download the latest patch
APSW	http://rogerbinns.com/apsw.html
MaxDB (SAP)	http://mysql.com/products/maxdb
sdb	http://dev.mysql.com/downloads/maxdb/7.6.00.html#Python
sapdb	http://sapdb.org/sapdbPython.html
Firebird (InterBase)	http://firebird.sf.net
KInterbasDB	http://kinterbasdb.sf.net
SQL Server	http://microsoft.com/sql
pymssql	http://pymssql.sf.net (requires FreeTDS [http://freetds.org])

adodbapi	http://adodbapi.sf.net
Sybase	http://sybase.com
sybase	http://object-craft.com.au/projects/sybase
Oracle	http://oracle.com
cx_Oracle	http://starship.python.net/crew/atuining/cx_Oracle
DCOracle2	http://zope.org/Members/matt/dco2 (older, for Oracle8 only)
Ingres	http://ingres.com
Ingres DBI	http://ingres.com/products/ Prod_Download_Python_DBI.html
ingmod	http://www.informatik.uni-rostock.de/~hme/software/
ORMs	
SQLObject	http://sqlobject.org
SQLAlchemy	http://sqlalchemy.org
PyDO/PyDO2	http://skunkweb.sf.net/pydo.html

[a] pysqlite added to Python 2.5 as sqlite3 module.

PREV NEXT

21.5. Exercises

21-1. *Database API*. What is the Python DB-API? Is it a good thing? Why (or why not)?

21-2. *Database API*. Describe the differences between the database module parameter styles (see the `paramstyle` module attribute).

21-3. *Cursor Objects*. What are the differences between the cursor `execute*()` methods?

21-4. *Cursor Objects*. What are the differences between the cursor `fetch*()` methods?

21-5. *Database Adapters*. Research your RDBMS and its Python module. Is it DB-API compliant? What additional features are available for that module that are extras not required by the API?

21-6. *Type Objects*. Study using Type objects for your database and DB-API adapter and write a small script that uses at least one of those objects.

21-7. *Refactoring*. In the `create()` function of Example 21.1 (`ushuffle_db.py`), a table that already exists is dropped and re-created by recursively calling `create()` again. This is dangerous in case the re-creation of the table fails (again) because you will then have infinite recursion. Fix this problem by creating a more practical solution that does not involve copying the create query (`cur.execute()`) again in the exception handler. Extra Credit: Try to recreate the table a maximum of three times before returning failure back to the caller.

21-8. *Database and HTML*. Take any existing database table, and use the knowledge you developed from Chapter 20 and output the contents of a database table into an HTML table.

21-9. *Web Programming and Databases*. Take our "user shuffle" example (`ushuffle_db.py`), and create a Web interface for it.

21-10. *GUI Programming and Databases*. Take our "user shuffle" example (`ushuffle_db.py`), and throw a GUI for it.

21-11. *Stock Portfolio Class*. Update the stock database example from Chapter 13 to use a relational database.

21-12. *Switching ORMs to a Different RDBMS*. Take either the SQLAlchemy (`ushuffle_sa.py`) or SQLObject (`ushuffle_so.py`) application and swap out MySQL as the back-end RDBMS for another one of your choice.