The following is from the 3rd Edition, and pertains to the Python 2.X version of the msql-python interface to MySql. Its code was changed to 3.X form, but was not tested due to the lack of a 3.X mysql-python. Other database are already available for 3.X: see the web.

SQL Database Interfaces

The `shelve` module and ZODB package of the prior sections are powerful tools. Both allow scripts to throw nearly arbitrary Python objects on a keyed-access file and load them back later—in a single step for shelves and with a small amount of administrative code for ZODB. Especially for applications that record highly structured data, object databases can be convenient and efficient—there is no need to split and later join together the parts of large objects, and stored data is processed with normal Python syntax because it is normal Python objects.

Shelves and ZODB aren't relational database systems, though; objects (records) are accessed with a single key, and there is no notion of SQL queries. Shelves, for instance, are essentially databases with a single index and no other query-processing support. Although it's possible to build a multiple-index interface to store data with multiple shelves, it's not a trivial task and requires manually coded extensions.

ZODB supports some types of searching beyond shelve (e.g., its cataloging feature), and persistent objects may be traversed with all the power of the Python language. However, neither shelves nor ZODB object-oriented databases provide the full generality of SQL queries. Moreover, especially for data that has a naturally tabular structure, relational databases may sometimes be a better fit.

For programs that can benefit from the power of SQL, Python also supports relational database systems. Relational databases are not necessarily mutually exclusive with the object persistence topics we studied earlier in this chapter—it is possible, for example, to store the serialized string representation of a Python object produced by pickling in a relational database. ZODB also supports the notion of mapping an object database to a relational storage medium.

The databases we'll meet in this section, though, are structured and processed in very different ways:

- They store data in related tables of columns (rather than in persistent dictionaries of arbitrarily structured persistent Python objects).
- They support the SQL query language for accessing data and exploiting relationships among it (instead of Python object traversals).

For some applications, the end result can be a potent combination. Moreover, some SQL-based database systems provide industrial-strength persistence support.

Today, there are freely available interfaces that let Python scripts utilize all common relational database systems, both free and commercial: MySQL, Oracle, Sybase, Informix, InterBase, PostgreSQL (Postgres), SQLite,[2] ODBC, and more. In addition, the Python community has defined a database API specification that works portably with a variety of underlying database packages. Scripts written for this API can be migrated to different database vendor packages, with minimal or no source code changes.

[2] Late-breaking news: Python 2.5 will likely include support for the SQLite relational database system as part of its standard library. For more on the cutting edge, see also the popular SQLObject third-party Object Relational Manage, which grafts an object interface onto your database, with tables as classes, rows as instances, and columns as attributes.

SQL Interface Overview

Like ZODB, and unlike the pickle and shelve persistence modules presented earlier, SQL databases are optional extensions that are not part of Python itself. Moreover, you need to know SQL to fully understand their interfaces. Because we don't have space to teach SQL in this text, this section gives a brief overview of the API; please consult other SQL references and the database API resources mentioned in the next section for more details.

The good news is that you can access SQL databases from Python, through a straightforward and portable model. The Python database API specification defines an interface for communicating with underlying database systems from Python scripts. Vendor-specific database interfaces for Python may or may not conform to this API completely, but all database extensions for Python seem minor variations on a theme. SQL databases in Python are grounded on a few concepts:

Connection objects

Represent a connection to a database, are the interface to rollback and commit operations, and generate cursor objects.

Cursor objects

Represent an SQL statement submitted as a string and can be used to step through SQL statement results.

Query results of SQL `select` *statements*

Are returned to scripts as Python sequences of sequences (e.g., a list of tuples), representing database tables of rows. Within these row sequences, column field values are normal Python objects such as strings, integers, and floats (e.g., `[('bob',38), ('emily',37)]`). Column values may also be special types that encapsulate things such as date and time, and database NULL values are returned as the Python `None` object.

Beyond this, the API defines a standard set of database exception types, special database type object constructors, and informational calls.

For instance, to establish a database connection under the Python API-compliant Oracle interface available from Digital Creations, install the extension and Oracle, and then run a line of this form:

```
connobj = Connect("user/password@system")
```

The string argument's contents may vary per database and vendor, but they generally contain what you provide to log in to your database system. Once you have a connection object, there a variety of things you can do with it, including:

```
connobj.close()                     close connection now (not at object __del__ time)
connobj.commit()                    commit any pending transactions to the database
connobj.rollback()                  roll database back to start of pending transactions
connobj.callproce(proc, params)     fetch stored procedure's code
connobj.getSource(proc)             fetch stored procedure's code
```

But one of the most useful things to do with a connection object is to generate a cursor object:

```
cursobj = connobj.cursor()          return a new cursor object for running SQL
```

Cursor objects have a set of methods too (e.g., `close` to close the cursor before its destructor runs), but the most important may be this one:

```
cursobj.execute(sqlstring [, parameters])   run SQL query or command string
```

Parameters are passed in as a sequence or mapping of values, and are substituted into the SQL statement string according to the interface module's replacement target conventions. The `execute` method can be used to run a variety of SQL statement strings:

- DDL definition statements (e.g., `CREATE TABLE`)

- DML modification statements (e.g., `UPDATE` or `INSERT`)
- DQL query statements (e.g., `SELECT`)

After running an SQL statement, the cursor's `rowcount` attribute gives the number of rows changed (for DML) or fetched (for DQL); `execute` also returns the number of rows affected or fetched in the most vendor interfaces. For DQL query statements, you must call one of the `fetch` methods to complete the operation:

```
tuple       = cursobj.fetchone()          fetch next row of a query result
listoftuple = cursobj.fetchmany([size])   fetch next set of rows of query result
listoftuple = cursobj.fetchall()          fetch all remaining rows of the result
```

And once you've received fetch method results, table information is processed using normal Python sequence operations (e.g., you can step through the tuples in a `fetchall` result list with a simple `for` loop). Most Python database interfaces also allow you to provide values to be passed to SQL statement strings, by providing targets and a tuple of parameters. For instance:

```
query = 'SELECT name, shoesize FROM spam WHERE job = ? AND age = ?'
cursobj.execute(query, (value1, value2))
results = cursobj.fetchall()
for row in results:...
```

In this event, the database interface utilizes prepared statements (an optimization and convenience) and correctly passes the parameters to the database regardless of their Python types. The notation used to code targets in the query string may vary in some database interfaces (e.g., `:p1` and `:p2` rather than the two `?`s used by the Oracle interface); in any event, this is not the same as Python's `%` string formatting operator.

Finally, if your database supports stored procedures, you can generally call them with the `callproc` method or by passing an SQL `CALL` or `EXEC` statement string to the `execute` method. `callproc` may generate a result table retrieved with a `fetch` variant, and returns a modified copy of the input sequence—input parameters are left untouched, and output and input/output parameters are replaced with possibly new values. Additional API features, including support for database blobs, is described in the API's documentation. For now, let's move on to do some real SQL processing in Python.

An SQL Database API Tutorial

We don't have space to provide an exhaustive reference for the database API in this book. To sample the flavor of the interface, though, let's step through a few simple examples. We'll use the MySQL database system for this tutorial. Thanks to Python's portable database API, other popular database packages such as PostgreSQL, SQLite, and Oracle are used almost identically, but the initial call to log in to the database will generally require different argument values.

The MySQL system

With a reported 8 million installations and support for more than 20 platforms, MySQL is by most accounts the most popular open source relational database system today. It is a powerful, fast, and full-featured SQL database system that serves as the storage mechanism for many of the sites you may visit on the Web.

MySQL consists of a database server, as well as a collection of clients. Technically, its SQL engine is a multithreaded server, which uses some of the same threaded socket server techniques we met in Chapter 12. It listens for requests on a socket and port, can be run either on a remote machine or on your local computer, and handles clients in parallel threads for efficiency and responsiveness.

On Windows, the MySQL server may be run automatically as a Windows service, so it is always available; on Unix-like machines, it runs as a perpetual demon process. In either case, the server can accept requests over a network or simply run on your machine to provide access to locally stored databases. Ultimately, your databases take the form of a set of files, stored in the server's "data" directory and represented as B-tree disk tables. MySQL handles concurrent updates by automatically locking tables when they are written by client conversation threads.

The MySQL server is available both as a separate program for use in a client/server networked environment, and as a library that can be linked into standalone applications. Clients can submit queries to the server over a TCP/IP socket on any platform; as usual with sockets, use the machine name "localhost" if the server is running locally. Besides sockets, the database server also supports connections using named pipes on Windows NT-based platforms (NT, 2000, XP, and so on); Unix domain socket files on Unix; shared memory on Windows; as well as ODBC, JDBC, and ADO.NET.

For our purposes, the main thing to note is that the standard MySQL interface for Python is compliant with the current version of the database API (2.0). Because of that, most of the code we'll see here will also work unchanged on other database systems, as long as their interfaces also support the portable database API. If you use the PostgreSQL database, for instance, the PyGreSQL open source Python extension provides DB-API 2.0-compliant interfaces that largely work the same way.

Installation

Before we can start coding, we need to install both MySQL itself, as well as the Python MySQL interface module. The MySQL system implements a language-neutral database server; the Python interface module maps calls in our script to the database server's interfaces. This is a straightforward process, but here are a few quick notes:

MySQL

> At this writing, MySQL can be found on the Web at http://dev.mysql.com. It's available in two flavors—a community version, which is open source (under the GNU license), as well as a commercial version, which is not free, but is relatively inexpensive and removes the restrictions of the GNU license (you won't have to make all your source code available, for instance). See the MySQL web site for more on which version may be right for you; the open source package was used for this book. MySQL installs itself today in *C:\Program Files\MySQL* on Windows. It includes the database server program, command-line tools, and more.

Python MySQL interface

> The primary DB API-compliant MySQL interface for Python was `mysql-python` when I wrote this (but run a web search on "Python MySQL" for current information). You may also find links to this package at the http://www.python.org page for the database Special Interest Group (SIG), as well as at the PyPI web site. Both `mysql-python`, as well as MySQL itself, are simple self-installing executables on Windows. On Windows, like most third-party packages, the Python MySQL interface shows up in the Python install tree, in the *site-packages* subdirectory of the standard library: *C:\Python24\Lib\site-packages\MySQLdb*. As usual, because this directory is automatically added to the module import search path, no path configuration is required.

Getting started

Time to write some code; this isn't a book on either MySQL or the SQL language, so we'll defer to other resources for details on the commands we'll be running here (O'Reilly has a suite of books on both topics). In fact, the databases we'll use are tiny, and the commands we'll use are deliberately simple as SQL goes—you'll want to extrapolate from what you see here to the more realistic tasks you face. This section is just a brief look at how to use the Python language in conjunction with an SQL database.

The basic SQL interface in Python is very simple, though. In fact, it's hardly object-oriented at all—queries and other database commands are sent as strings of SQL. If you know SQL, you already have most of what you need in Python.

The first thing we want to do is open a connection to the database and create a table for storing records:

```
C:\...\PP4E\Dbase> python
>>> import MySQLdb
>>> conn = MySQLdb.connect(host='localhost', user='root', passwd='python')
```

We start out by importing the Python MySQL interface here—it's a package directory called `MySQLdb` to our scripts that looks like a simple module. Next we create a connection object, passing in the items our database requires for a login—here, the name of the machine the server is running on, along with a user and

password. This first API tends to vary per database system, because each has unique login requirements. After you've connected, though, the rest of the API is largely the same for any database.

As usual, the server name "localhost" means the local computer, but this could also be any TCP/IP server name if we're using a database on a remote machine. You normally would log in with a username created by your database administrator, but we'll use "root" here to keep this simple (the root user is created automatically when MySQL is installed, and I gave it a password of "python" during installation).

Making databases and tables

Next, make a cursor for submitting SQL statements to the database server:

```
>>> curs = conn.cursor()
>>> curs.execute('create database peopledb')
1L
>>> curs.execute('use peopledb')
0L
>>> tblcmd = 'create table people (name char(30), job char(10), pay int(4))'
>>> curs.execute(tblcmd)
0L
```

The first command here makes a database called *peopledb*. We can make more than one—for instance, one for test and one for production, or one per developer—but most SQL statements are relative to a particular database. We can tell the server which database to use with the `use` SQL statement or by passing in a `db` keyword argument when calling `connect` to make the initial connection. In MySQL, we can also qualify table names with their databases in SQL statements (`database.table`), but it's usually more convenient to select the database and make it implied.

Finally, the last command creates the table called "people" within the *peopledb* database; the name, job, and pay information specifies the columns in this table, as well as their datatypes using a "type(size)" syntax—two strings and an integer. Datatypes can be more sophisticated than ours, but we'll ignore such details here (see SQL references).

Adding records

So far, we've logged in and created a database and table. Now let's start a new Python session and create some records. There are three basic statement-based approaches we can use here: inserting one row at a time, or inserting multiple rows with a single call statement or a Python loop. Here is the simple case:

```
C:\...\PP4E\Dbase> python
>>> import MySQLdb
>>> conn = MySQLdb.connect(host='localhost', db='peopledb',
...                        user='root', passwd='python')
>>> curs = conn.cursor()
>>> curs.execute('insert people values (%s, %s, %s)', ('Bob', 'dev', 5000))
1L
>>> curs.rowcount
1L
>>> MySQLdb.paramstyle
'format'
```

Create a cursor object to submit SQL statements to the database server as before. The SQL `insert` command adds a single row to the table. After an execute call, the cursor's `rowcount` attribute gives the number of rows produced or affected by the last statement run. This is also available as the return value of an `execute` call in the MySQL module, but this is not defined in the database API specification (in other words, don't depend on it if you want your database scripts to work on other database systems).

Parameters to substitute into the SQL statement string are passed in as a sequence (e.g., list or tuple). Notice the module's `paramstyle`—this tells us what style it uses for substitution targets in the statement string. Here, `format` means this module accepts string formatting-style targets; `%s` means a string-based substitution, just as in Python `%` string expressions. Other database modules might use styles such as `qmark` (a `?` target), or numeric indexes or mapping keys (see the DB API for details).

To insert multiple rows with a single statement, use the `executemany` method and a sequence of row sequences (e.g., a list of lists). This call is like calling `execute` once for each row sequence in the argument, and in fact may be implemented as such; database interfaces may use database-specific techniques to make this run quicker, though:

```
>>> curs.executemany('insert people values (%s, %s, %s)',
...           [ ('Sue', 'mus', '70000'),
...             ('Ann', 'mus', '60000')])
2L
>>> curs.rowcount
2L
```

We inserted two rows at once in the last statement. It's hardly any more work to achieve the same result by inserting one row at a time with a Python loop:

```
>>> rows = [['Tom', 'mgr', 100000],
...         ['Kim', 'adm', 30000],
...         ['pat', 'dev', 90000]]

>>> for row in rows:
...     curs.execute('insert people values (%s , %s, %s)', row)
...
1L
1L
1L
>>> conn.commit()
```

Blending Python and SQL like this starts to open up all sorts of interesting possibilities. Notice the last command; we always need to call the connection's `commit` method to write our changes out to the database. Otherwise, when the connection is closed, our changes will be lost. In fact, if you quit Python without calling the `commit` method, none of your inserts will be retained.

Technically, the connection object automatically calls its `rollback` method to back out changes that have not yet been committed, when it is closed (which happens manually when its `close` method is called or automatically when the connection object is about to be garbage-collected). For database systems that don't support transaction commit and rollback operations, these calls may do nothing.

Running queries

OK, we've now added six records to our database table. Let's run an SQL query to see how we did:

```
>>> curs.execute('select * from people')
6L
>>> curs.fetchall()
(('Bob', 'dev', 5000L), ('Sue', 'mus', 70000L), ('Ann', 'mus', 60000L), ('Tom',
'mgr', 100000L), ('Kim', 'adm', 30000L), ('pat', 'dev', 90000L))
```

Run an SQL `select` statement with a cursor object to grab all rows and call the cursor's `fetchall` to retrieve them. They come back to our script as a sequence of sequences. In this module, it's a tuple of tuples—the outer tuple represents the result table, the nested tuples are that table's rows, and the nested tuple's contents are the column data. Because it's all Python data, once we get the query result, we process it with normal Python code. For example, to make the display a bit more coherent, loop through the query result:

```
>>> curs.execute('select * from people')
6L
>>> for row in curs.fetchall():
...     print(row)
...
('Bob', 'dev', 5000L)
('Sue', 'mus', 70000L)
('Ann', 'mus', 60000L)
('Tom', 'mgr', 100000L)
('Kim', 'adm', 30000L)
```

```
('pat', 'dev', 90000L)
```

Tuple unpacking comes in handy in loops here too, to pick out column values as we go; here's a simple formatted display of two of the columns' values:

```
>>> curs.execute('select * from people')
6L
>>> for (name, job, pay) in curs.fetchall():
...     print(name, ':', pay)
...
Bob : 5000
Sue : 70000
Ann : 60000
Tom : 100000
Kim : 30000
pat : 90000
```

Because the query result is a sequence, we can use Python's powerful sequence tools to process it. For instance, to select just the name column values, we can run a more specific SQL query and get a tuple of tuples:

```
>>> curs.execute('select name from people')
6L
>>> names = curs.fetchall()
>>> names
(('Bob',), ('Sue',), ('Ann',), ('Tom',), ('Kim',), ('pat',))
```

Or we can use a Python list comprehension to pick out the fields we want—by using Python code, we have more control over the data's content and format:

```
>>> curs.execute('select * from people')
6L
>>> names = [rec[0] for rec in curs.fetchall()]
>>> names
['Bob', 'Sue', 'Ann', 'Tom', 'Kim', 'pat']
```

The `fetchall` call we've used so far fetches the entire query result table all at once, as a single sequence (an empty sequence comes back, if the result is empty). That's convenient, but it may be slow enough to block the caller temporarily for large result tables or generate substantial network traffic if the server is running remotely. To avoid such a bottleneck, we can also grab just one row, or a bunch of rows, at a time with `fetchone` and `fetchmany`. The `fetchone` call returns the next result row or a `None` false value at the end of the table:

```
>>> curs.execute('select * from people')
6L
>>> while True:
...     row = curs.fetchone()
...     if not row: break
...     print(row)
...
('Bob', 'dev', 5000L)
('Sue', 'mus', 70000L)
('Ann', 'mus', 60000L)
('Tom', 'mgr', 100000L)
('Kim', 'adm', 30000L)
('pat', 'dev', 90000L)
```

The `fetchmany` call returns a sequence of rows from the result, but not the entire table; you can specify how many rows to grab each time with a parameter or fall back on the setting of the cursor's `arraysize` attribute. Each call gets at most that many more rows from the result or an empty sequence at the end of the table:

```
>>> curs.execute('select * from people')
6L
>>> while True:
```

```
...     rows = curs.fetchmany()           # size=N optional argument
...     if not rows: break
...     for row in rows:
...         print(row)
...
('Bob', 'dev', 5000L)
('Sue', 'mus', 70000L)
('Ann', 'mus', 60000L)
('Tom', 'mgr', 100000L)
('Kim', 'adm', 30000L)
('pat', 'dev', 90000L)
```

For this module at least, the result table is exhausted after a `fetchone` or `fetchmany` returns a `False`, though `fetchall` continues to return the whole table. The DB API says that `fetchall` returns "all (remaining) rows," so you may want to call `execute` anyhow to regenerate results before fetching, for portability:

```
>>> curs.fetchone()
>>> curs.fetchmany()
()
>>> curs.fetchall()
(('Bob', 'dev', 5000L), ('Sue', 'mus', 70000L), ('Ann', 'mus', 60000L), ('Tom',
'mgr', 100000L), ('Kim', 'adm', 30000L), ('pat', 'dev', 90000L))
```

Naturally, we can do more than fetch an entire table; the full power of the SQL language is at your disposal in Python:

```
>>> curs.execute('select name, job from people where pay > 60000')
3L
>>> curs.fetchall()
(('Sue', 'mus'), ('Tom', 'mgr'), ('pat', 'dev'))
```

The last query fetches names and pay fields for people who earn more than $60,000. The next is similar, but passes in the selection value as a parameter and orders the result table:

```
>>> query = 'select name, job from people where pay >= %s order by name'
>>> curs.execute(query, [60000])
4L
>>> for row in curs.fetchall(): print(row)
...
('Ann', 'mus')
('pat', 'dev')
('Sue', 'mus')
('Tom', 'mgr')
```

Running updates

Cursor objects also are used to submit SQL update statements to the database server—updates, deletes, and inserts. We've already seen the `insert` statement at work. Let's start a new session to perform some other kinds of updates:

```
C:\...\PP4E\Dbase> python
>>> import MySQLdb
>>> conn = MySQLdb.connect(host='localhost', user='root', passwd='python')
>>> curs = conn.cursor()
>>> curs.execute('use peopledb')
>>> curs.execute('select * from people')
6L
>>> curs.fetchall()
(('Bob', 'dev', 5000L), ('Sue', 'mus', 70000L), ('Ann', 'mus', 60000L), ('Tom',
'mgr', 100000L), ('Kim', 'adm', 30000L), ('pat', 'dev', 90000L))
```

The SQL `update` statement changes records—the following changes three records' pay column values to `65000` (Bob, Ann, and Kim), because their pay was no more than $60,000. As usual, the cursor's `rowcount` gives the number of records changed:

```
>>> curs.execute('update people set pay=%s where pay <= %s', [65000, 60000])
3L
>>> curs.rowcount
3L
>>> curs.execute('select * from people')
6L
>>> curs.fetchall()
(('Bob', 'dev', 65000L), ('Sue', 'mus', 70000L), ('Ann', 'mus', 65000L), ('Tom',
 'mgr', 100000L), ('Kim', 'adm', 65000L), ('pat', 'dev', 90000L))
```

The SQL `delete` statement removes records, optionally according to a condition (to delete all records, omit the condition). In the following, we delete Bob's record, as well as any record with a pay that is at least $90,000:

```
>>> curs.execute('delete from people where name = %s', ['Bob'])
1L
>>> curs.execute('delete from people where pay >= %s',(90000,))
2L
>>> curs.execute('select * from people')
3L
>>> curs.fetchall()
(('Sue', 'mus', 70000L), ('Ann', 'mus', 65000L), ('Kim', 'adm', 65000L))

>>> conn.commit()
```

Finally, remember to commit your changes to the database before exiting Python, assuming you wish to keep them. Without a commit, a connection rollback or close call, as well as the connection's `__del__` deletion method, will back out uncommitted changes. Connection objects are automatically closed if they are still open when they are garbage-collected, which in turn triggers a `__del__` and a rollback; garbage collection happens automatically on program exit, if not sooner.

Building Record Dictionaries

Now that we've seen the basics in action, let's move on and apply them to a few large tasks. The SQL API defines query results to be sequences of sequences. One of the more common features that people seem to miss from the API is the ability to get records back as something more structured—a dictionary or class instance, for example, with keys or attributes giving column names. Because this is Python, it's easy to code this kind of transformation, and the API already gives us the tools we need.

Using table descriptions

For example, after a query execute call, the DB API specifies that the cursor's `description` attribute gives the names and types of the columns in the result table (we're continuing with the database in the state in which we left it in the prior section):

```
>>> curs.execute('select * from people')
3L
>>> curs.description
(('name', 254, 3, 30, 30, 0, 1), ('job', 254, 3, 10, 10, 0, 1), ('pay', 3, 5, 4,
 4, 0, 1))
>>> curs.fetchall()
(('Sue', 'mus', 70000L), ('Ann', 'mus', 65000L), ('Kim', 'adm', 65000L))
```

Formally, the description is a sequence of column-description sequences, each of the following form (see the DB API for more on the meaning of the type code slot—it maps to objects at the top level of the `MySQLdb` module):

```
(name, type_code, display_size, internal_size, precision, scale, null_ok)
```

Now, we can use this metadata anytime we want to label the columns—for instance, in a formatted records display:

```
>>> colnames = [desc[0] for desc in curs.description]
>>> colnames
['name', 'job', 'pay']

>>> for row in curs.fetchall():
...     for name, value in zip(colnames, row):
...         print(name, '\t=>', value)
...     print()
...
name    => Sue
job     => mus
pay     => 70000

name    => Ann
job     => mus
pay     => 65000

name    => Kim
job     => adm
pay     => 65000
```

Notice how a tab character is used to try to make this output align; a better approach might be to determine the maximum field name length (we'll see how in a later example).

Record dictionaries

It's a minor extension of our formatted display code to create a dictionary for each record, with field names for keys—we just need to fill in the dictionary as we go:

```
>>> curs.execute('select * from people')
3L
>>> colnames = [desc[0] for desc in curs.description]
>>> rowdicts = []
>>> for row in curs.fetchall():
...     newdict = {}
...     for name, val in zip(colnames, row):
...         newdict[name] = val
...     rowdicts.append(newdict)
...
>>> for row in rowdicts: print(row)
...
{'pay': 70000L, 'job': 'mus', 'name': 'Sue'}
{'pay': 65000L, 'job': 'mus', 'name': 'Ann'}
{'pay': 65000L, 'job': 'adm', 'name': 'Kim'}
```

Because this is Python, though, there are more powerful ways to build up these record dictionaries. For instance, the dictionary constructor call accepts the zipped name/value pairs to fill out the dictionaries for us:

```
>>> curs.execute('select * from people')
3L
>>> colnames = [desc[0] for desc in curs.description]
>>> rowdicts = []
>>> for row in curs.fetchall():
...     rowdicts.append( dict(zip(colnames, row)) )
...
>>> rowdicts[0]
{'pay': 70000L, 'job': 'mus', 'name': 'Sue'}
```

And finally, a list comprehension will do the job of collecting the dictionaries into a list—not only is this less to type, but it probably runs quicker than the original version:

```
>>> curs.execute('select * from people')
3L
>>> colnames = [desc[0] for desc in curs.description]
>>> rowdicts = [dict(zip(colnames, row)) for row in curs.fetchall()]

>>> rowdicts[0]
{'pay': 70000L, 'job': 'mus', 'name': 'Sue'}
```

One of the things we lose when moving to dictionaries is record field order—if you look back at the raw result of `fetchall`, you'll notice that record fields are in the name, job, and pay order in which they were stored. Our dictionary's fields come back in the pseudorandom order of Python mappings. As long as we fetch fields by key, this is irrelevant to our script. Tables still maintain their order, and dictionary construction works fine because the description result tuple is in the same order as the fields in row tuples returned by queries.

We'll leave the task of translating record tuples into class instances as a suggested exercise, except for the small hint that we can access fields as attributes rather than as keys, by simply creating an empty class instance and assigning to attributes with the Python `setattr` function. Classes would also provide a natural place to code inheritable tools such as standard display methods.

Automating with scripts and modules

Up to this point, we've essentially used Python as a command-line SQL client—our queries have been typed and run interactively. All the kinds of code we've run, though, can be used as the basis of database access in script files. Working interactively requires retyping things such as logic calls, which can become tedious. With scripts, we can automate our work.

To demonstrate, let's make the last section's prior example into a utility module—Example 17-9 is a reusable module that knows how to translate the result of a query from row tuples to row dictionaries.

*Example **Error! No text of specified style in document.**-1. PP4E\Dbase\SQL\makedicts.py*

```
"""
convert list of row tuples to list of row dicts with field name keys
this is not a command-line utility: hardcoded self-test if run
"""

def makedicts(cursor, query, params=()):
    cursor.execute(query, params)
    colnames = [desc[0] for desc in cursor.description]
    rowdicts = [dict(zip(colnames, row)) for row in cursor.fetchall()]
    return rowdicts

if __name__ == '__main__':   # self test
    import MySQLdb
    conn = MySQLdb.connect(host='localhost', user='root', passwd='python')
    cursor = conn.cursor()
    cursor.execute('use peopledb')
    query  = 'select name, pay from people where pay < %s'
    lowpay = makedicts(cursor, query, [70000])
    for rec in lowpay: print(rec)
```

As usual, we can run this file from the system command line as a script to invoke its self-test code:

```
...\PP4E\Dbase\SQL> makedicts.py
{'pay': 65000L, 'name': 'Ann'}
{'pay': 65000L, 'name': 'Kim'}
```

Or we can import it as a module and call its function from another context, like the interactive prompt. Because it is a module, it has become a reusable database tool:

```
...\PP4E\Dbase\SQL> python
>>> from makedicts import makedicts
>>> from MySQLdb import connect
>>> conn = connect(host='localhost', user='root',passwd='python', db='peopledb')
```

```
>>> curs = conn.cursor()
>>> curs.execute('select * from people')
3L
>>> curs.fetchall()
(('Sue', 'mus', 70000L), ('Ann', 'mus', 65000L), ('Kim', 'adm', 65000L))

>>> rows = makedicts(curs, "select name from people where job = 'mus'")
>>> rows
[{'name': 'Sue'}, {'name': 'Ann'}]
```

Our utility handles arbitrarily complex queries—they are simply passed through the DB API to the database server. The `order by` clause here sorts the result on the name field:

```
>>> query = 'select name, pay from people where job = %s order by name'
>>> musicians = makedicts(curs, query, ['mus'])
>>> for row in musicians: print(row)
...
{'pay': 65000L, 'name': 'Ann'}
{'pay': 70000L, 'name': 'Sue'}
```

Tying the Pieces Together

So far, we've learned how to makes databases and tables, insert records into tables, query table contents, and extract column names. For reference, and to show how these techniques are combined, Example 17-10 collects them into a single script.

*Example **Error! No text of specified style in document.**-2. PP4E\Dbase\SQL\testdb.py*

```
from MySQLdb import Connect
conn = Connect(host='localhost', user='root', passwd='python')
curs = conn.cursor()
try:
    curs.execute('drop database testpeopledb')
except:
    pass  # did not exist

curs.execute('create database testpeopledb')
curs.execute('use testpeopledb')
curs.execute('create table people (name char(30), job char(10), pay int(4))')

curs.execute('insert people values (%s, %s, %s)', ('Bob', 'dev', 50000))
curs.execute('insert people values (%s, %s, %s)', ('Sue', 'dev', 60000))

curs.execute('select * from people')
for row in curs.fetchall():
    print(row)

curs.execute('select * from people')
colnames = [desc[0] for desc in curs.description]
while True:
    print('-' * 30)
    row = curs.fetchone()
    if not row: break
    for (name, value) in zip(colnames, row):
        print('%s => %s' % (name, value))

conn.commit()   # save inserted records
```

Refer to prior sections in this tutorial if any of the code in this script is unclear. When run, it creates a two-record database and lists its content to the standard output stream:

```
('Bob', 'dev', 50000L)
('Sue', 'dev', 60000L)
------------------------------
```

```
name => Bob
job => dev
pay => 50000
------------------------------
name => Sue
job => dev
pay => 60000
------------------------------
```

As is, this example is really just meant to demonstrate the database API. It hardcodes database names, and it re-creates the database from scratch each time. We could turn this code into generally useful tools by refactoring it into reusable parts, as we'll see later in this section. First, though, let's explore techniques for getting data into our databases.

Loading Database Tables from Files

One of the nice things about using Python in the database domain is that you can combine the power of the SQL query language, with the power of the Python general-purpose programming language. They naturally compliment each other.

Loading with SQL and Python

Suppose, for example, that you want to load a database table from a flat file, where each line in the file represents a database row, with individual field values separated by commas. Examples 17-11 and 17-12 list two such datafiles we're going to be using here.

*Example **Error! No text of specified style in document.**-3. PP4E\Dbase\SQL\data.txt*

```
bob,devel,50000
sue,music,60000
ann,devel,40000
tim,admin,30000
kim,devel,60000
```

*Example **Error! No text of specified style in document.**-4. PP4E\Dbase\SQL\data2.txt*

```
bob,developer,80000
sue,music,90000
ann,manager,80000
```

Now, MySQL has a handy SQL statement for loading such a table quickly. Its `load data` statement parses and loads data from a text file, located on either the client or the server machine. In the following, the first command deletes all records in the table, and we're using the fact that Python automatically concatenates adjacent string literals to split the SQL statement over multiple lines:

```
>>> curs.execute('delete from people')                # all records
3L
>>> curs.execute(
...     "load data local infile 'data.txt' "
...           "into table people fields terminated by ','")
5L
>>> curs.execute('select * from people')
5L
>>> for row in curs.fetchall(): print(row)
...
('bob', 'devel', 50000L) ('sue', 'music', 60000L)
('ann', 'devel', 40000L)
('tim', 'admin', 30000L)
('kim', 'devel', 60000L)

>>> conn.commit()
```

This works as expected. What if you might someday wish to use your script on a system without this SQL statement, though? Perhaps you just need to do something more custom than this MySQL statement allows.

Not to worry—a small amount of simple Python code can easily accomplish the same result (some irrelevant output lines are omitted here):

```
>>> curs.execute('delete from people')
>>> file = open('data.txt')
>>> rows = [line.split(',') for line in file]
>>> for rec in rows:
...     curs.execute('insert people values (%s, %s, %s)', rec)
...
>>> curs.execute('select * from people')
>>> for rec in curs.fetchall(): print(rec)
...
('bob', 'devel', 50000L)
('sue', 'music', 60000L)
('ann', 'devel', 40000L)
('tim', 'admin', 30000L)
('kim', 'devel', 60000L)
```

This code makes use of a list comprehension to collect string split results for all lines in the file, and file iterators to step through the file line by line. Its Python loop does the same work as the MySQL `load` statement, and it will work on more database types. We can get the same result from an `executemany` DB API call shown earlier as well, but the Python `for` loop here has the potential to be more general.

Python versus SQL

In fact, you have the entire Python language at your disposal for processing database results, and a little Python can often duplicate or go beyond SQL syntax. For instance, SQL has special aggregate function syntax for computing things such as sums and averages:

```
>>> curs.execute("select sum(pay), avg(pay) from people where job = 'devel'")
1L
>>> curs.fetchall()
((150000.0, 50000.0),)
```

By shifting the processing to Python, we can sometimes simplify and do more than SQL's syntax allows (albeit sacrificing any query performance optimizations the database may perform). Computing pay sums and averages with Python can be accomplished with a simple loop:

```
>>> curs.execute("select name, pay from people where job = 'devel'")
3L
>>> result = curs.fetchall()
>>> result
(('bob', 50000L), ('ann', 40000L), ('kim', 60000L))

>>> tot = 0
>>> for (name, pay) in result: tot += pay
...
>>> print('total:', tot, 'average:', tot / len(result))
total: 150000 average: 50000
```

Or we can use more advanced tools such as comprehensions and generator expressions to calculate sums, averages, maximums, and the like:

```
>>> print(sum(rec[1] for rec in result))          # 2.4 generator expr
150000
>>> print(sum(rec[1] for rec in result) / len(result))
50000
>>> print(max(rec[1] for rec in result))
60000
```

The Python approach is more general, but it doesn't buy us much until things become more complex. For example, here are a few more advanced comprehensions that collect the names of people whose pay is above and below the average in the query result set:

```
>>> avg = sum(rec[1] for rec in result) / len(result)
```

```
>>> print([rec[0] for rec in result if rec[1] > avg])
['kim']
>>> print([rec[0] for rec in result if rec[1] < avg])
['ann']
```

We may be able to do some of these kinds of tasks with more advanced SQL techniques such as nested queries, but we eventually reach a complexity threshold where Python's general-purpose nature makes it attractive. For comparison, here is the equivalent SQL:

```
>>> query = ("select name from people where job = 'devel' and "
...                 "pay > (select avg(pay) from people where job = 'devel')")
>>> curs.execute(query)
1L
>>> curs.fetchall()
(('kim',),)
>>>
>>> query = ("select name from people where job = 'devel' and "
...                 "pay < (select avg(pay) from people where job = 'devel')")
>>> curs.execute(query)
1L
>>> curs.fetchall()
(('ann',),)
```

This isn't the most complex SQL you're likely to meet, but beyond this point, SQL can become more involved. Moreover, unlike Python, SQL is limited to database-specific tasks by design. Imagine a query that compares a column's values to data fetched off the Web, or from a user in a GUI—simple with Python's Internet and GUI support, but well beyond the scope of a special-purpose language such as SQL. By combining Python and SQL, you get the best of both and can choose which is best suited to your goals.

With Python, you also have access to utilities you've already coded: your database tool set is arbitrarily extensible with functions, modules, and classes. To illustrate, here are some of the same operations coded in a more mnemonic fashion with the dictionary-record module we wrote earlier:

```
>>> from makedicts import makedicts
>>> recs = makedicts(curs, "select * from people where job = 'devel'")
>>> print(len(recs), recs[0])
3 {'pay': 50000L, 'job': 'devel', 'name': 'bob'}

>>> print([rec['name'] for rec in recs])
['bob', 'ann', 'kim']
>>> print(sum(rec['pay'] for rec in recs))
150000

>>> avg = sum(rec['pay'] for rec in recs) / len(recs)
>>> print([rec['name'] for rec in recs if rec['pay'] > avg])
['kim']
>>> print([rec['name'] for rec in recs if rec['pay'] >= avg])
['bob', 'kim']
```

For more advanced database extensions, see the SQL-related tools available for Python in the third-party domain. For example, a variety of packages add an OOP flavor to the DB API—the ORMs discussed near the start of this chapter.

SQL Utility Scripts

At this point in our SQL DB API tour, we've started to stretch the interactive prompt to its breaking point—we wind up retyping the same boilerplate code again every time we start a session and every time we run a test. Moreover, the code we're writing is substantial enough to be reused in other programs. Let's wrap up by transforming our code into reusable scripts that automate tasks and support reuse.

To illustrate more of the power of the Python/SQL mix, this section presents a handful of utility scripts that perform common tasks—the sorts of things you'd otherwise have to recode often during development. As an added bonus, most of these files are both command-line utilities and modules of functions that can be

imported and called from other programs. Most of the scripts in this section also allow a database name to be passed in on the command line; this allows us to use different databases for different purposes during development—changes in one won't impact others.

Table load scripts

Let's take a quick look at code first, before seeing it in action; feel free to skip ahead to correlate the code here with its behavior. As a first step, Example 17-13 shows a simple way to script-ify the table-loading logic of the prior section.

*Example **Error! No text of specified style in document.**-5. PP4E\Dbase\SQL\loaddb1.py*

```
"""
load table from comma-delimited text file; equivalent to executing this SQL:
load data local infile 'data.txt' into table people fields terminated by ','"
"""

import MySQLdb
conn = MySQLdb.connect(host='localhost', user='root', passwd='python')
curs = conn.cursor()
curs.execute('use peopledb')

file = open('data.txt')
rows = [line.split(',') for line in file]
for rec in rows:
    curs.execute('insert people values (%s, %s, %s)', rec)

conn.commit()       # commit changes now, if db supports transactions
conn.close()        # close, __del__ call rollback if changes not committed yet
```

As is, Example 17-13 is a top-level script geared toward one particular case. It's hardly any extra work to generalize this into a function that can be imported and used in a variety of scenarios, as in Example 17-14.

Notice the way this code uses two list comprehensions to build a string of record values for the `insert` statement (see its comments for the transforms applied). We could use an `executemany` call as we did earlier, but we want to be general and avoid hardcoding the fields template.

This file also defines a `login` function to automate the initial connection calls—after retyping this 4-command sequence 10 times, it seemed a prime candidate for a function. In addition, this reduces code redundancy; in the future, things like username and host need to be changed in only a single location, as long as the `login` function is used everywhere. (For an alternative approach to such automation that might encapsulate the connection object, see the class we coded for ZODB connections in the prior section.)

*Example **Error! No text of specified style in document.**-6. PP4E\Dbase\SQL\loaddb.py*

```
"""
like loaddb1, but insert more than one row at once, and reusable function
command-line usage: loaddb.py dbname? datafile? (tablename is implied)
"""

tablename = 'people'   # generalize me

def login(host='localhost', user='root', passwd='python', db=None):
    import MySQLdb
    conn = MySQLdb.connect(host=host, user=user, passwd=passwd)
    curs = conn.cursor()
    if db: curs.execute('use ' + db)
    return conn, curs

def loaddb(cursor, table, datafile='data.txt', conn=None):
    file = open(datafile)                           # x,x,x\nx,x,x\n
    rows = [line.split(',') for line in file]       # [ [x,x,x], [x,x,x] ]
    rows = [str(tuple(rec)) for rec in rows]        # [ "(x,x,x)", "(x,x,x)" ]
    rows = ', '.join(rows)                          # "(x,x,x), (x,x,x)"
```

```
    curs.execute('insert ' + table + ' values ' + rows)
    print(curs.rowcount, 'rows loaded')
    if conn: conn.commit()

if __name__ == '__main__':
    import sys
    database, datafile = 'peopledb', 'data.txt'
    if len(sys.argv) > 1: database = sys.argv[1]
    if len(sys.argv) > 2: datafile = sys.argv[2]
    conn, curs = login(db=database)
    loaddb(curs, tablename, datafile, conn)
```

Table display script

Once we load data, we probably will want to display it. Example 17-15 allows us to display results as we go—it prints an entire table with either a simple display (which could be parsed by other tools), or a formatted display (generated with the dictionary-record utility we wrote earlier). Notice how it computes the maximum field-name size for alignment with a generator expression; the size is passed in to a string formatting expression by specifying an asterisk (*) for the field size in the format string.

*Example **Error! No text of specified style in document.**-7. PP4E\Dbase\SQL\dumpdb.py*

```
"""
display table contents as raw tuples, or formatted with field names
command-line usage: dumpdb.py dbname? [-] (dash for formatted display)
"""

def showformat(recs, sept=('-' * 40)):
    print(len(recs), 'records')
    print(sept)
    for rec in recs:
        maxkey = max(len(key) for key in rec)                # max key len
        for key in rec:                                      # or: \t align
            print('%-*s => %s' % (maxkey, key, rec[key]))    # -ljust, *len
        print(sept)

def dumpdb(cursor, table, format=True):
    if not format:
        cursor.execute('select * from ' + table)
        while True:
            rec = cursor.fetchone()
            if not rec: break
            print(rec)
    else:
        from makedicts import makedicts
        recs = makedicts(cursor, 'select * from ' + table)
        showformat(recs)

if __name__ == '__main__':
    import sys
    dbname, format = 'peopledb', False
    cmdargs = sys.argv[1:]
    if '-' in cmdargs:                      # format if '-' in cmdline args
        format = True                       # dbname if other cmdline arg
        cmdargs.remove('-')
    if cmdargs:
        dbname = cmdargs[0]

    from loaddb import login
    conn, curs = login(db=dbname)
    dumpdb(curs, 'people', format)
```

While we're at it, let's code some utility scripts to initialize and erase the database, so we do not have to type these by hand at the interactive prompt again every time we want to start from scratch. Example 17-16

completely deletes and re-creates the database, to reset it to an initial state (we did this manually at the start of the tutorial).

*Example **Error! No text of specified style in document.**-8. PP4E\Dbase\SQL\makedb.py*

```
"""
physically delete and re-create db files in mysql's data\ directory
usage: makedb.py dbname? (tablename is implied)
"""

import sys
dbname = (len(sys.argv) > 1 and sys.argv[1]) or 'peopledb'

if input('Are you sure?').lower() not in ('y', 'yes'):
    sys.exit()

from loaddb import login
conn, curs = login(db=None)
try:
    curs.execute('drop database ' + dbname)
except:
    print('database did not exist')

curs.execute('create database ' + dbname)        # also: 'drop table tablename'
curs.execute('use ' + dbname)
curs.execute('create table people (name char(30), job char(10), pay int(4))')
conn.commit()   # this seems optional
print('made', dbname)
```

The clear script in Example 17-17 deletes all rows in the table, instead of dropping and re-creating them entirely. For testing purposes, either approach is usually sufficient.

*Example **Error! No text of specified style in document.**-9. PP4E\Dbase\SQL\cleardb.py*

```
"""
delete all rows in table, but don't drop the table or database it is in
usage: cleardb.py dbname? (tablename is implied)
"""

import sys
if input('Are you sure?').lower() not in ('y', 'yes'):
    sys.exit()
dbname = 'peopledb'                             # cleardb.py
if len(sys.argv) > 1: dbname = sys.argv[1]      # cleardb.py testdb

from loaddb import login
conn, curs = login(db=dbname)
curs.execute('delete from people')
conn.commit()                                   # else rows not really deleted
print(curs.rowcount, 'records deleted')         # conn closed by its __del__
```

Finally, Example 17-18 provides a command-line tool that runs a query and prints its result table in formatted style. There's not much to this script; because we've automated most of its tasks already, this is largely just a combination of existing tools. Such is the power of code reuse in Python.

*Example **Error! No text of specified style in document.**-10. PP4E\Dbase\SQL\xd5 uerydb.py*

```
"""
run a query string, display formatted result table
example: querydb.py testdb "select name, job from people where pay > 50000"
"""

import sys
database, query = 'peopledb', 'select * from people'
if len(sys.argv) > 1: database = sys.argv[1]
if len(sys.argv) > 2: query = sys.argv[2]
```

```
from makedicts import makedicts
from dumpdb    import showformat
from loaddb    import login

conn, curs = login(db=database)
rows = makedicts(curs, query)
showformat(rows)
```

Using the scripts

Last but not least, here is a log of a session that makes use of these scripts in command-line mode, to illustrate their operation. Most of the files also have functions that can be imported and called from a different program; the scripts simply map command-line arguments to the functions' arguments when run standalone. The first thing we do is initialize a testing database and load its table from a text file:

```
...\PP4E\Dbase\SQL> makedb.py testdb
Are you sure?y
database did not exist
made testdb

...\PP4E\Dbase\SQL> loaddb.py testdb data2.txt
3 rows loaded
```

Next, let's check our work with the dump utility (use a `-` argument to force a formatted display):

```
...\PP4E\Dbase\SQL> dumpdb.py testdb
('bob', 'developer', 80000L)
('sue', 'music', 90000L)
('ann', 'manager', 80000L)

...\PP4E\Dbase\SQL> dumpdb.py testdb -
3 records
----------------------------------------
pay  => 80000 job  => developer
name => bob
----------------------------------------
pay  => 90000
job  => music
name => sue
----------------------------------------
pay  => 80000
job  => manager
name => ann
----------------------------------------
```

The dump script is an exhaustive display; to be more specific about which records to view, use the query script and pass in a query string on the command line (the command line is wrapped here to fit in this book):

```
...\PP4E\Dbase\SQL> querydb.py testdb
                                  "select name, job from people where pay = 80000"
2 records
----------------------------------------
job  => developer
name => bob
----------------------------------------
job  => manager
name => ann
----------------------------------------

...\PP4E\Dbase\SQL> querydb.py testdb
                                  "select * from people where name = 'sue'"
1 records
----------------------------------------
```

```
pay  => 90000
job  => music
name => sue
----------------------------------------
```

Now, let's erase and start again with a new data set file. The clear script erases all records but doesn't reinitialize the database completely:

```
...\PP4E\Dbase\SQL> cleardb.py testdb
Are you sure?y
3 records deleted

...\PP4E\Dbase\SQL> dumpdb.py testdb -
0 records
----------------------------------------

...\PP4E\Dbase\SQL> loaddb.py testdb data.txt
5 rows loaded

...\PP4E\Dbase\SQL> dumpdb.py testdb
('bob', 'devel', 50000L)
('sue', 'music', 60000L)
('ann', 'devel', 40000L) ('tim', 'admin', 30000L)
('kim', 'devel', 60000L)
```

In closing, here are three queries in action on this new data set: they fetch developers' jobs that pay above an amount and record with a given pay sorted on a field. We could run these at the Python interactive prompt, of course, but we're getting a lot of setup and boilerplate code for free here.

```
...\PP4E\Dbase\SQL> querydb.py testdb
                                  "select name from people where job = 'devel'"
3 records
----------------------------------------
name => bob
----------------------------------------
name => ann
----------------------------------------
name => kim
----------------------------------------

...\PP4E\Dbase\SQL> querydb.py testdb
                                  "select job from people where pay >= 60000"
2 records
----------------------------------------
job => music
----------------------------------------
job => devel
----------------------------------------

...\PP4E\Dbase\SQL> querydb.py testdb
                                 "select * from people where pay >= 60000 order by job"
2 records
----------------------------------------
pay  => 60000
job  => devel
name => kim
----------------------------------------
pay  => 60000
job  => music
name => sue
----------------------------------------
```

Before we move on, a few caveats are worth noting. The scripts in this section illustrate the benefits of code reuse, accomplish their purpose (which was partly demonstrating the SQL API), and serve as a model for

canned database utilities. But they are not as general or powerful as they could be. As is, these scripts allow you to pass in the database name but not much more. For example, we could allow the table name to be passed in on the command line too, support sorting options in the dump script, and so on.

Although we could generalize to support more options, at some point we may need to revert to typing SQL commands in a client—part of the reason SQL is a language is because it must support so much generality. Further extensions to these scripts are left as exercises. Change this code as you like; it's Python, after all.

SQL Resources

Although the examples we've seen in this section are simple, their techniques scale up to much more realistic databases and contexts. The web sites we studied in the prior part of the book, for instance, can make use of systems such as MySQL to store page state information as well as long-lived client information. Because MySQL supports both large databases and concurrent updates, it's a natural for web site implementation.

There is more to database interfaces than we've seen, but additional API documentation is readily available on the Web. To find the full database API specification, search the Web for "Python Database API" at Google.com (or at a similar site). You'll find the formal API definition—really just a text file describing PEP number 249 (the Python Enhancement Proposal under which the API was hashed out).

Perhaps the best resource for additional information about database extensions today is the home page of the Python database SIG. Go to http://www.python.org, click on the SIGs link near the top, and navigate to the database group's page (or go straight to http://www.python.org/sigs/db-sig, the page's current address at the time of this writing). There, you'll find API documentation (this is where it is officially maintained), links to database-vendor-specific extension modules, and more.

While you're at Python.org, be sure to also explore the Gadfly database package—a Python-specific SQL-based database extension, which sports wide portability, socket connections for client/server modes, and more. Gadfly loads data into memory, so it is currently somewhat limited in scope. On the other hand, it is ideal for prototyping database applications—you can postpone cutting a check to a vendor until it's time to scale up for deployment. Moreover, Gadfly is suitable by itself for a variety of applications; not every system needs large data stores, but many can benefit from the power of SQL. And as always, see the PyPI web site and search the web at large for related third-party tools and extensions.