CHAPTER 9

Databases

Databases have traditionally been treated as entities distinct from the larger code base. This is reflected organizationally; a firm wall often exists between developers and database administrators (DBAs); the DBAs work in parallel with the rest of the organization.

This is rooted in the historical nature of the technology. In the past, databases have been expensive both in terms of software and hardware, and this means that there haven't been very many of them. With so few resources, very few people acquired the skills to work with them. Dedicated staff were required to gate access to these limited resources, and to prevent the naive from doing stupid things. Additionally, most products were very hard to configure. Getting acceptable performance required much tuning, and thus even more expertise. The company jewels were often stored in these mines, and had to be protected from fumbling hands and untested scripts.

In recent years, the technical landscape has changed. Over the last ten years, free database implementations have blossomed, as have computing and storage capabilities.

This has given rise to a proliferation of databases. SQL databases have morphed from beasts with complicated interacting processes and dedicated raw filesystem drivers to serverless libraries that can be linked and embedded within shipping code. Examples of these include HSQLDB, embedded MySQL, and SQLite. As of Python 2.5, SQLite even ships in the standard library.

Every developer can now have a database on the desktop (or laptop or palmtop). The consequences of this have been slow to sink in. Agile software development techniques have a long history behind them, but agile database development techniques do not.

A New Religion

The ultimate goal of any development organization is delivering business value. While the proximate goal of development is producing software and the proximate goal of a database is organizing data for retrieval, these are not the ultimate organizational goals. If the CEO could get the same information more cheaply and reliably by calling a televangelist, then she'd be doing it.

Neither the goals of development nor the goals of the DBA organization are meaningful without assistance from the each other. Development wants to use the databases to accomplish meaningful work for the company, and the DBAs want to ensure that the company's data is protected.[1] These two organizations are often at loggerheads when they should be working in concert to meet the overall organization's needs.

Agile development recognizes that different business groups have differing needs and priorities, and that these change over time. Code must change to reflect these realities, and this leads to the need for constant refactoring. The same is true of data models. Like Code, they will rot if they're not regularly maintained.

The database groups need to work closely with their customers to understand these issues. As with development, they need to focus on the issues with the biggest payoffs. The work should be prioritized, and some things will fall by the wayside. There will always be new problems, so the database organization shouldn't try to solve everything. The key is not to try to eliminate and address all problems, but to design a process that addresses new issues as normal occurrences.

Blurring the Boundaries

Only by creating integrated and fully automated processes can an organization meet the rapid turnaround required by short iterations, and this can only be done by integrating the automation into the entire production cycle from start to finish. Agile development breaks down some of the separations between development, operation, and administration. Agile development therefore has strong impacts on the DBA organization:

Database design becomes an *evolutionary process*. Since change is a constant pressure, the database schema is never complete. These changes must be propagated quickly from development through to production. This must be done in such a way that it can be replicated, and it must be done without human intervention.

Databases are improved through *refactorings*. These are changes that improve the structure of the database without altering its function. The need to accommodate live changes imposes certain design constraints not present in code.

Code must be *isolated* from the underlying data model as much as possible. Much is written about an object-relational mismatch. I don't subscribe to that view any more than I subscribe to a view of an object-filesystem mismatch or an object-thread mismatch. Relational databases are complicated, but that doesn't mean that there is a fundamental misfit. It does mean that there is a lot of machinery required to magically unify the two.

Testing must be performed. Changes must be made to the database. Changes must also be made to the code that uses the database. A variety of techniques are used to accomplish these tests. Some require little more than the machinery already discussed in previous chapters, and some require new classes of software.

---

1. The DBAs should ensure that the company's data is available and protect it from loss. Often the first is forgotten, but if nobody is doing useful work with the production databases, then either the databases are superfluous or the company is in dire trouble. Either way, the DBAs are in trouble.

Developers and DBAs both have a role in this, but since many tools reside in the software development process, the DBAs have to learn more about those tools and processes. At the same time, developers will have to learn more about being a DBA. The DBA's job becomes less about adjudicating changes and more about providing expertise and advising against absolute stupidity. Because there is no clear organizational boundary, the DBAs have to work closely with the developers to ensure that proper procedural boundaries are observed.

Concealing Data Access

At some point, your code has to talk to the database. At that point, the code needs to understand the details of the data. It must know how to locate the data source and initiate a conversation. It must know the structure of the data to perform efficient queries. It needs to convert between local types and stored types, and back again, and it must know how and when to write out changes. It must be able to recognize stale results, and it often needs to cache data that is expensive to retrieve from the database.

When the structure of the data changes, the code that accesses that data needs to change. If the data access code is scattered throughout a program, then every change necessitates seeking those points out and rewriting the access code. This is time-consuming and prone to error.

Therefore, code dealing with the database should be in a central location. This layer mediates all access to the database. It can be as simple or as complex as needed. At one end of the spectrum, it might simply be a few methods that read and write strings to a file. At the other end are systems that map between relational databases and classes or objects within a program.

Such libraries are called object-relational mappers (ORMs). These subsystems provide an elaborate framework concealing the details of the underlying query mechanisms. They make it easy to interface with the underlying database systems. With a good ORM, it is easier to write database access code than it is to work with files.

Object-Relational Mappers

ORMs generally have four aspects:

- A description of the database schema
- A mapping between the schema and the application objects
- A way of selecting data
- A mechanism for writing changes

ORMs differ widely in how these are aspects are handled. In some cases, they are manually specified. In others, they are automatically derived from a running system. In some cases, the running system's configuration is derived from the ORM definitions.

I'm going to discuss the two leading Python ORMs: SQLObject and SQLAlchemy. There are three common patterns that are useful when discussing them:

- Active record
- Data mapper
- Unit of work

The Active Record Pattern

The active record pattern describes a simple relationship between a database and the programming language. A database table corresponds to a class, a row in a table corresponds to an instance of the class, and a column corresponds to an attribute.

Queries return objects, and the values are read from the attributes. Writing to an attribute updates the database. Creating an instance inserts a row. Deleting an object deletes the row. Inherent in the active record pattern is the idea that each row has an identity.

This pattern is easy to describe and understand. It combines the steps of describing the database schema and producing a mapping between the schema and application objects. It has the advantage of working very well for small-to-medium-sized cases.

While it easily maps tables, rows, and columns, it doesn't easily map other database objects, such as procedure results, views, joins, column selects, and multitable or multidatabase results.

The biggest problem with the active record pattern is that the resulting code closely mirrors the database schema. When the database structure changes, the code must also change, and these changes are distributed throughout the code. Solving this requires a layer of indirection.

The Data Mapper Pattern

The data mapper pattern maps columns into arbitrary objects. The underlying structure is described, and then the mappings are specified between the storage entities and the application objects.

This indirection separates the database from the application. The storage format can be altered, while the objects remain the same, and vice versa. Changing the database structure no longer necessitates changing the application code, and arbitrary SQL results can be sensibly mapped.

On the other hand, it's a little more complicated to set up. It hides database access and structure by distributing them throughout your code. The relationships between attributes in one place and those in another can be concealed. It's a little harder to understand what is going on in some cases.

The Unit of Work Pattern

In this pattern, the code tracks the changes that have been made and commits them in a single batch within a single transaction.

Talking to the database is expensive. Each batch of changes incurs a significant time lag. Often the majority of an application's time is spent waiting for results from the database. I have personally seen situations in which more than 90 percent of an application's response time was spent waiting on the database. The actual code took microseconds to run, but each

round trip to the database took milliseconds. Committing the changes in a single batch reduces this overhead dramatically.

Since the database transaction is only held for the length of the batched connection, there is less contention between queries and less opportunity for deadlock. The application quickly uses and returns connections, so the running application needs to have fewer open connections to the database in order to achieve the same throughput.

The application is in control of the commits, so it knows when problems occur. The commit points also provide a natural point to handle rollback.

There are disadvantages, though. Control comes at the expense of effort and forethought. Developers must be aware of when changes are committed and how the batches are constructed. Potentially, an application can continue running with uncommitted changes that haven't been rolled back, leading to inconsistent views of the database and possible loss of data. The application may be less responsive. While its overall performance may increase, the lower latency of a do-it-immediately approach may be worth the increase in responsiveness. A straight do-it-now access policy is useful and appropriate for many small applications.

Python ORMs

There are many Python ORMs, but there are two 900-pound gorillas. They are SQLObject and SQLAlchemy. SQLObject has been around quite a bit longer than SQLAlchemy, but the latter is gaining in popularity. Although more complicated for novices, it is far more capable when it comes to real production problems.

SQLObject

SQLObject is based on the active record pattern. It has minimal support for the unit of work pattern, and many people simply write to the database. It has an aggressive caching policy by default, and it uses a simple declarative format to specify both the schema and mappings. It really wants to use numeric keys for database records.

As always, obtaining the package is the first step:

```
$ easy_install -U SQLObject
```

```
Searching for SQLObject
Reading http://pypi.python.org/simple/SQLObject/
...
Processing dependencies for SQLObject
Finished processing dependencies for SQLObject
```

I'm using a classic example—that of students in a school. The `student` table looks like Figure 9-1.

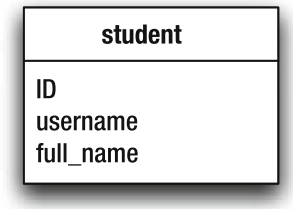

Figure 9-1. *The student table*

The schema for this table might be generated by the following SQL:

```
CREATE TABLE student (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    full_name VARCHAR(64) NOT NULL,
    username VARCHAR(16) NOT NULL
);
```

This table would be described to SQLObject as follows:

```
from sqlobject import SQLObject, StringCol

class Student(SQLObject):
    username = StringCol(length=16)
    fullName = StringCol(length=64)
```

Connecting to the Database

The next step is establishing a connection to the database. SQLObject uses standard connection URI syntax:

```
scheme://[user[:password]@]host[:port]/database[?parameters]
```

Examples include the following:

- `mysql://jeff:myPasswordHere@localhost/test_db`
- `postgres://bob@my.host.com/another_db?debug=1&cache=0`
- `postges:///path/to/socket/db_name`
- `sqlite:///path/to/the/database`

As of version 0.9, the common parameters are as follows:

- `debug`
- `debugOutput`
- `debugThreading`
- `cache`

- autoCommit
- logger
- logLevel

Since SQLite ships with Python, I'll be using it for the examples. The following code fragment sets up a SQLite connection:

```
filename = "test_db"
abs_path = os.path.abspath(filename)
connection_uri = 'sqlite://' + abs_path
connection = sqlobject.connectionForURI(connection_uri)
sqlobject.sqlhub.processConnection = connection
```

You can turn this into the following method:

```
def sqlite_connect(abs_path):
    connection_uri = 'sqlite://' + abs_path
    connection = sqlobject.connectionForURI(connection_uri)
    sqlobject.sqlhub.processConnection = connection
```

The important thing is that you set the `processConnection` variable to the correct connection. If you turn this into a method, the corresponding test is as follows:

```
@use_pymock
def test_sqlite_connect():
    f = '/x'
    uri = 'sqlite:///x'
    connection = dummy()
    override(sqlobject, 'connectionForURI').expects(uri).\
        returns(connection)
    replay()
    sqlite_connect(f)
    assert sqlobject.sqlhub.processConnection is connection
    verify()
```

Creating Rows

New rows are created by instantiating objects. Here's a simple test for this:

```
s1 = Student(username="jeff", fullName="Jeff Younker")
assert s1.username == "jeff"
assert s1.fullName == "Jeff Younker"
```

There's a good deal of setup and tear-down that needs to be done, though. A new database file must be created, and the connection to that database must be initiated. At the end of the test, the file should be removed, the object cache should be cleared to prevent other tests from stomping on yours, and finally the connection should be closed.

■ **Note** The connection hub's caching plays havoc with the SQLite driver, so the test generates a new randomly named connection each time.

```
import random
...
def random_string(length):
    seq = [chr(x) for x in range(ord('a'), ord('z')+1)]
    return ''.join([x for x in random.sample(seq, length)])

def test_creating_student():
    f = os.path.abspath(random_string(8) + '.db')
    if os.path.exists(f):
        os.unlink(f)
    sqlite_connect(f)
    try:
        s1 = Student(username="jeff", fullName="Jeff Younker")
        assert s1.username == "jeff"
        assert s1.fullName == "Jeff Younker"
    finally:
        sqlobject.sqlhub.processConnection.cache.clear()
        sqlobject.sqlhub.processConnection.close()
        del sqlobject.sqlhub.processConnection
        os.unlink(f)
```

When this runs, it gives the following error:

```
Traceback (most recent call last):
  File "/Library/Python/2.5/site-packages/nose-0.10.0-py2.5.egg/nose/case.py",➥
 line 202, in runTest
    self.test(*self.arg)
...
  File "/Users/jeff/Library/Python/2.5/site-packages/SQLObject-0.10.0b2-py2.5.egg/➥
sqlobject/sqlite/sqliteconnection.py", line 177, in _executeRetry
    raise OperationalError(ErrorMessage(e))
OperationalError: no such table: student
```

In other words, the schema has not been defined yet. The tests could create the schema directly, but that ties them to the specific database used for the unit tests. Fortunately, `SQLObject` instances know how to create themselves. One command creates this new table. The revised test method is as follows:

```
def test_creating_student():
    f = os.path.abspath('test_db')
    if os.path.exists(f):
        os.unlink(f)
```

```
    sqlite_connect(f)
    try:
        Student.createTable()
        s1 = Student(username="jeff", fullName="Jeff Younker")
        assert s1.username == "jeff"
        assert s1.fullName == "Jeff Younker"
    finally:
        sqlobject.sqlhub.processConnection.cache.clear()
        sqlobject.sqlhub.processConnection.close()
        del sqlobject.sqlhub.processConnection
        os.unlink(f)
```

The test now runs successfully to conclusion. It's a mess, though, and there are going to be many more of these written. The setup and tear-down can be refactored into a decorator:

```
from decorator import decorator
...
@decorator
def with_sqlobject(tst):
    f = os.path.abspath(random_string(8) + '.db')
    if os.path.exists(f):
        os.unlink(f)
    sqlite_connect(f)
    try:
        Student.createTable()
        tst()
    finally:
        sqlobject.sqlhub.processConnection.cache.clear()
        sqlobject.sqlhub.processConnection.close()
        os.unlink(f)

@with_sqlobject
def test_writing_student():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    assert s1.username == "jeff"
    assert s1.fullName == "Jeff Younker"
```

The resulting test is significantly more concise. The preceding code uses the decorator module, which is a third-party module that simplifies writing decorators. Most decorators usually involve creating at least one closure, and this closure is nearly always the same. Here's a decorator that prints `before` and then executes the wrapped function:

```
def before(f):
    def wrapper(*args, *kw):
         print "before"
         return f(*args, **kw)
    return wrapper
```

The decorator module supplies the necessary closure machinery:

```
from decorator import decorator
...
@decorator
def before(f, *args, **kw)
    print "before"
    return f(*args, **kw)
```

I find the resulting decorators much cleaner and easier to understand.

Putting the Schema Where It Belongs

Right now there is only one table, but eventually there will be many. Every time a new table is added, the schema definition in `with_sqlobject()` will grow. This schema creation information may also be useful in the program itself, particularly when it needs to be installed, so it should go into the file with the schema declarations.

```
from sqlobject_ex import create_schema
...
@decorator
def with_sqlobject(tst):
    f = os.path.abspath(random_string(8) + '.db')
    if os.path.exists(f):
        os.unlink(f)
    sqlite_connect(f)
    try:
        create_schema()
        tst()
    finally:
        sqlobject.sqlhub.processConnection.cache.clear()
        sqlobject.sqlhub.processConnection.close()
        os.unlink(f)
```

And the `create_schema()` method should go into `sqlobject_ex.py`:

```
def create_schema():
    Student.createTable()
```

Attribute Defaults

What happens if one of the student attributes is omitted? For example

```
>>> Student(fullName="Jeff Younker")
```

gives the following error:

```
Traceback (most recent call last):
  ...
ValueError: Unknown SQL builtin type: <type 'classobj'> for <class sqlobject.sql➥
builder.NoDefault at 0xde05d0>
```

All attributes are required unless a default is defined. In other words, all attributes are assumed to be `NOT NULL` unless declared otherwise with the `default` attribute. The following code makes the username optional:

```
class Student(SQLObject):
    username = StringCol(length=16, default=None)
    fullName = StringCol(length=64)
```

Selecting Objects

SQLObject has three methods for retrieving objects from the database. The `get()` method retrieves a single object by its ID. The attribute `id` maps to the field ID. It is transparently managed by the ORM. All mapped tables must have an ID field.

```
@with_sqlobject
def test_get():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student.get(s1.id)
    assert s1 is s2
```

The `select()` class method chooses one or more objects. With no arguments, it returns all instances in the table.

```
from sets import Set
...
@with_sqlobject
def test_select():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    students = list(Student.select())
    assert len(students) == 2
    assert Set(students) == Set([s1, s2])
```

The `select()` method takes a SQLBuilder expression. SQLBuilder is part of SQLObject. You build SQL queries from SQLBuilder calls. The package makes extensive use of operator overloading, so for simple cases, queries look just like normal Python comparison expressions.

```
@with_sqlobject
def test_select_using_full_name():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    unused_s2 = Student(username="doug", fullName="Doug McBride")
    students = Student.select(Student.q.fullName == "Jeff Younker")
    assert list(students) == [s1]
```

The class variable `Student.q` contains column descriptions. These are used in SQLBuilder queries. The preceding expression translates to the following SQL:

```
select * from student where full_name = "Jeff Younker"
```

For simple comparisons, you'll never have to access SQLBuilder directly, but more esoteric expressions require more direct meddling. The following code uses a SQL-like expression to search for all students with a full name containing *ou*.

```
from sqlobject.sqlbuilder import LIKE
...
@with_sqlobject
def test_select_using_partial_name():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    unused_s3 = Student(username="amy", fullName="Amy Woodward")
    students = Student.select(LIKE(Student.q.fullName, '%ou%'))
    assert Set(students) == Set([s1, s2])
```

The `selectBy()` method is a concise method of querying exact column matches. Keywords specify the attributes to be compared, and the values are those to be compared with.

```
@with_sqlobject
def test_selectBy_full_name():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    unused_s3 = Student(username="amy", fullName="Amy Woodward")
    students = Student.selectBy(fullName="Jeff Younker")
    assert list(students) == [s1]
```

Like `select()`, if no arguments are supplied, it returns the entire table:

```
@with_sqlobject
def test_selectBy_all():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    students = Student.selectBy()
    assert Set(students) == Set([s1, s2])
```

Updating Fields

Values are modified via simple assignment:

```
@with_sqlobject
def test_modify_values():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s1.fullName = "Jeff M. Younker"
    students = Student.selectBy(fullName="Jeff M. Younker")
    assert list(students) == [s1]
```

Deleting Rows

All `SQLObject` instances have a `destroySelf()` method. Calling this method deletes the associated row from the database:

```
@with_sqlobject
def test_delete():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s1.destroySelf()
    students = Student.select()
    assert list(students) == []
```

The `destroySelf()` method does not perform cascading deletes, but that can be accomplished by overriding this method.

One-to-Many Relationships

SQLObject specifies joins (specifically inner joins) declaratively. The products of the joins appear as arrays contained in instance variables. To demonstrate, I've expanded the schema to include an e-mail address for each student. Each student may have more than one e-mail address (see Figure 9-2).

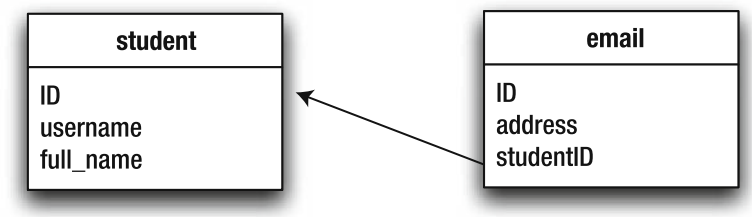

Figure 9-2. *A many-to-one relationship between student and e-mail address*

The corresponding SQL for the new table is as follows:

```
CREATE TABLE email (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    address VARCHAR(255) NOT NULL,
    studentID INTEGER NOT NULL,
    FOREIGN KEY studentID REFERENCES student(id)
);
```

Foreign Keys

The new table is defined in `sqlobject_ex.py` as follows:

```
from sqlobject import ForeignKey, SQLObject, StringCol
...
class Email(SQLObject):
    email = StringCol(length=255)
```

```
    student = ForeignKey('Student')
...
def create_schema():
    Student.createTable()
    Email.createTable()
```

ForeignKey defines the attribute student as a link to the class Student. When this attribute is accessed, the key studentID will be dereferenced and the row will be instantiated. The IDs are handled under the hood.

```
@with_sqlobject
def test_email_creation():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    e1 = Email(address="jeff@not.real.org", student=s1)
    assert e1.student is s1
    e1.student = s2
    assert e1.student is s2
```

SQLObject foreign keys are always expected to end in ID. This is one of the drawbacks of using SQLObject. The underlying foreign key can be accessed directly via the attribute:

```
@with_sqlobject
def test_direct_id_access():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    e1 = Email(address="jeff@not.real.org", student=s1)
    assert e1.studentID == s1.id
    e1.studentID = s2.id
    assert e1.student is s2
```

Multiple Joins

So far, if the code has an Email, it can locate the associated Student, but there is no way to go in the other direction. The MultipleJoin class provides this functionality. You modify the Student class like this:

```
from sqlobject import ForeignKey, MultipleJoin, SQLObject, StringCol
...
class Student(SQLObject):
    fullName = StringCol(length=64)
    username = StringCol(length=16)
    emails = MultipleJoin('Email')
```

Accessing the emails attribute returns a list of associated Email objects:

```
@with_sqlobject
def test_multiple_join():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    e1 = Email(address="jeff@not.real.org", student=s1)
    assert s1.emails == [e1]
```

If there are no objects, then an empty list is returned:

```
@with_sqlobject
def test_multiple_join_empty_returns_empty_list():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    assert s1.emails == []
```

The attribute looks like it should be mutable, but you can't assign to it:

```
@with_sqlobject
def test_multiple_join_cant_assign():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    e1 = Email(address="jeff@not.real.org", student=s1)
    try:
        s1.emails = [e1]
        assert False
    except AttributeError:
        pass
```

`MultipleJoin` attributes are read-only. The only way to alter their contents is by changing the foreign key:

```
@with_sqlobject
def test_changing_a_multiple_join():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    e1 = Email(address="jeff@not.real.org", student=s1)
    e2 = Email(address="doug@not.real.org", student=s2)
    assert s1.emails == [e1]
    e2.student = s1
    assert Set(s1.emails) == Set([e1, e2])
```

Many-to-Many Relationships

Students are in many classes, and classes contain many students. This kind of relationship is referred to as a many-to-many relationship. In relational databases, these are expressed through intermediate tables. Each entry is essentially a double-ended pointer to the tables it relates (see Figure 9-3).

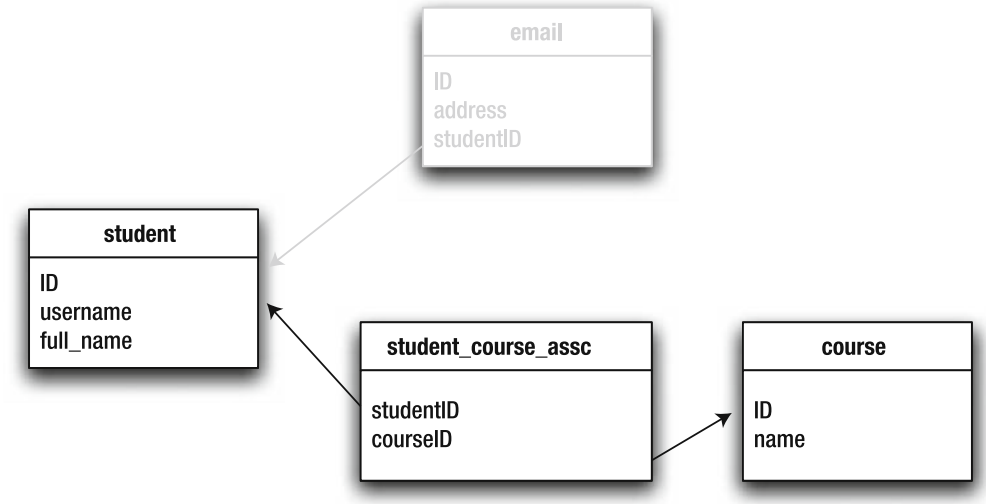

Figure 9-3. *A many-to-many relationship between students and classes*

The SQL defining these tables in SQLite is as follows:

```
CREATE TABLE course (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(64) NOT NULL
);

CREATE TABLE student_course_assc (
    studentID INTEGER NOT NULL,
    courseID INTEGER NOT NULL,
    FOREIGN KEY studentID REFERENCES student(ID),
    FOREIGN KEY courseID REFERENCES course(ID)
);
```

Only the target class is defined. The intermediate table is implicit in the `RelatedJoin` declarations. The components related to the join are shown in bold:

```
from sqlobject import ForeignKey, MultipleJoin, RelatedJoin, \
    SQLObject, RelatedJoin, StringCol
...
def create_schema():
    Student.createTable()
    Email.createTable()
    Course.createTable()
...
```

```
class Student(SQLObject):
    fullName = StringCol(length=64)
    username = StringCol(length=16)
    emails = MultipleJoin('Email')
    courses = RelatedJoin('Course')
...
class Course(SQLObject):
    name = StringCol(length=64)
    students = RelatedJoin('Student')
```

Joining Students and Courses

The join statements create add and remove methods. These are named after the class being joined. In the `Student` class, they would be `addCourse()` and `removeCourse()`. As with a multiple join, the attribute returns a list of associated objects:

```
from sqlobject_ex import Course, Email, sqlite_connect, Student, create_schema
...
@with_sqlobject
def test_related_join_add():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    c2 = Course(name="Biochemistry")
    s1.addCourse(c1)
    s1.addCourse(c2)
    assert Set(s1.courses) == Set([c1, c2])
    assert c1.students == [s1]
    assert c2.students == [s1]
```

The relationship can be established from either end:

```
@with_sqlobject
def test_related_join_add_in_other_order():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    c2 = Course(name="Biochemistry")
    s1.addCourse(c1)
    c2.addStudent(s1)
    assert Set(s1.courses) == Set([c1, c2])
    assert c1.students == [s1]
    assert c2.students == [s1]
```

Relations are removed with the `remove`*`Foo`*`()` method:

```
@with_sqlobject
def test_related_join_remove():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    c2 = Course(name="Biochemistry")
    s1.addCourse(c1)
    s1.addCourse(c2)
    assert Set(s1.courses) == Set([c1, c2])
    c2.removeStudent(s1)
    s1.removeCourse(c1)
    assert s1.courses == []
    assert c1.students == [] and c2.students == []
```

Adds can be performed multiple times, and they result in multiple records:

```
@with_sqlobject
def test_related_join_multiple_adds():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    s1.addCourse(c1)
    s1.addCourse(c1)
    assert s1.courses == [c1, c1]
    assert c1.students == [s1, s1]
```

Removes take away all the duplicates:

```
@with_sqlobject
def test_related_join_removing_multiples():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    s1.addCourse(c1)
    s1.addCourse(c1)
    s1.removeCourse(c1)
    assert s1.courses == []
```

Multiple Relationships

Multiple relationships are frequently created between two tables. For example, a student may be enrolled in a course or have completed a course. A corresponding schema is shown in Figure 9-4.

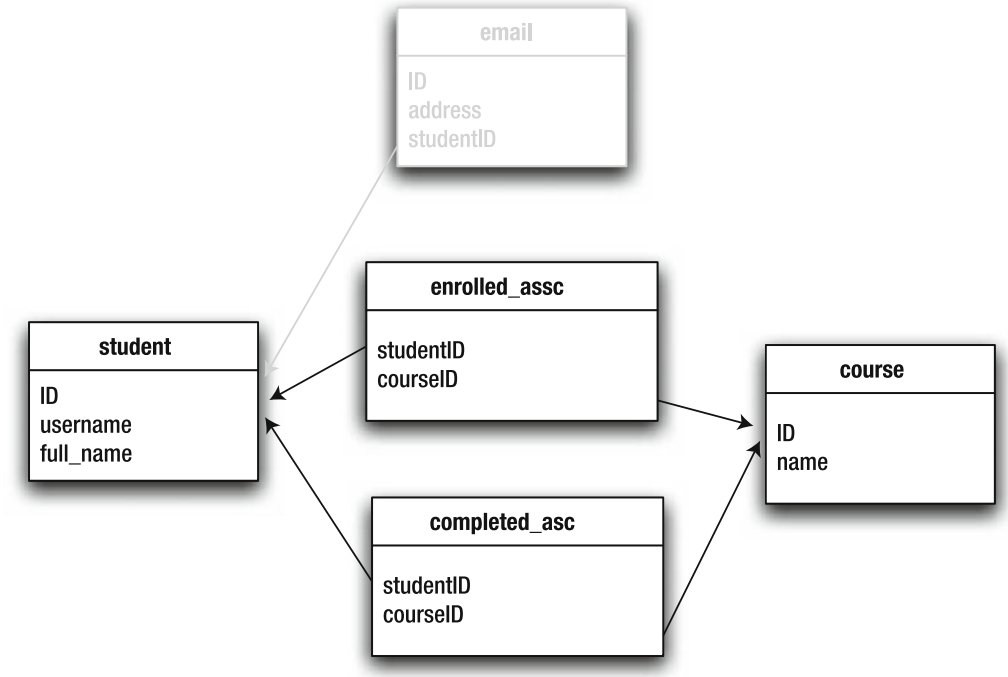

Figure 9-4. *A student can be enrolled in a course or may have completed a course.*

The SQLObject model is modified to reflect this:

```
class Student(SQLObject):
    fullName = StringCol(length=64)
    username = StringCol(length=16)
    emails = MultipleJoin('Email')
    enrolled = RelatedJoin('Course',
                           intermediateTable="enrolled_assc",
                           joinColumn="studentID",
                           otherColumn="courseID",
                           addRemoveName="Enrolled")
    completed = RelatedJoin('Course',
                            intermediateTable="completed_assc",
                            joinColumn="studentID",
                            otherColumn="courseID",
                            addRemoveName="Completed")

class Email(SQLObject):
    address = StringCol(length=255)
    student = ForeignKey('Student')
```

```
class Course(SQLObject):
    name = StringCol(length=64)
    enrolled = RelatedJoin('Student',
                           intermediateTable="enrolled_assc",
                           joinColumn="courseID",
                           otherColumn="studentID",
                           addRemoveName="Enrolled")
    completed = RelatedJoin('Student',
                            intermediateTable="completed_assc",
                            joinColumn="courseID",
                            otherColumn="studentID",
                            addRemoveName="Completed")
```

As with the simple `RelatedJoin`, the first argument is the class being joined with the containing class. The table containing the relation is specified with the `intermediateTable` keyword. The `joinColumn` and `otherColumn` keywords specify column names in the relation table. The join column points back to the containing class, and the other column links to the contained class. If left to its own devices, SQLObject generates the add and remove methods from the class name. The `addRemoveName` keyword supplies a different one.

■ **Note** It is not necessary for these definitions to be symmetrical. If your application will only add courses to students, and you can guarantee that courses will never be queried for the students they contain, then the related join can be omitted from the course.

At this point, all of the tests relating `Course` instances to `Student` instances have become invalid, and they need to be removed. Running the test suite indicates which ones should be removed.

The new `enrolled` relation and its add and remove methods are shown in this test:

```
@with_sqlobject
def test_enrollment_add_and_remove():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    s1.addEnrolled(c1)
    assert s1.enrolled == [c1]
    s1.removeEnrolled(c1)
    assert s1.enrolled == []
```

And it is clear that the two new relations point to separate tables:

```
@with_sqlobject
def test_enrollment_relations_are_separate():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    c2 = Course(name="Biochemistry")
    s1.addEnrolled(c1)
```

```
    s1.addCompleted(c2)
    assert s1.enrolled == [c1]
    assert s1.completed == [c2]
```

SQLAlchemy

SQLObject focuses on making easy things easy, but hard things are still hard. SQLAlchemy requires more configuration, but it pays that back in power. The results of any arbitrary join or select statement may be mapped to objects. The package gives fine-grained control over object graph loading and saving. It provides connection pooling and a low-level database interface. Generated SQL can even be replaced with custom queries when needed.

It is obtained via `easy_install`:

```
$ easy_install SQLAlchemy
```

```
Searching for SQLAlchemy
Reading http://pypi.python.org/simple/SQLAlchemy/
...
Installed /Users/jeff/Library/Python/2.5/site-packages/SQLAlchemy-0.4.3-py2.5.egg
Processing dependencies for SQLAlchemy
Finished processing dependencies for SQLAlchemy
```

As with SQLObject, the examples here will use SQLite. Without SQLObject's caching issues, an in-memory database can be used safely. The application code is defined in `sqlalchemy_ex.py`, and the tests are defined in `test_sqlalchemy_ex.py`.

With SQLAlchemy, setting up a connection is simple enough that it doesn't justify encapsulating the code:

```
from sqlalchemy import create_engine

def test_connection():
    engine = create_engine('sqlite:///:memory:')
```

The schema for the `student` table is pictured in Figure 9-5. The table is defined in `sqlachemy_ex.py`.

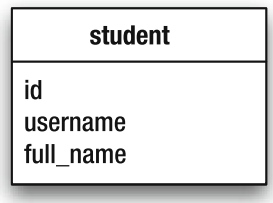

Figure 9-5. *The student table again*

```
from sqlalchemy import Column, Integer, MetaData, Table, String

schema = MetaData()
student_table = Table('student', schema,
                      Column('id', Integer, primary_key=True),
                      Column('username', String(16)),
                      Column('full_name', String(64)),
)
```

The `MetaData` class describes a connection. The `Table()` method describes the table's schema and associates it with the `MetaData` object (called `schema`). The declaration looks very much like a SQL table statement. Unlike SQLObject, the primary key's identity and type is not assumed, and must be declared.

This test creates the schema in the connected database:

```
from sqlalchemy_ex import schema
...
def test_schema_creation():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
```

This creates a schema, but there is no way to alter data yet. Doing this involves two steps. First, a class must be declared, and then the table must be mapped to the class. This linkage occurs through the names of table columns and object attributes.

```
from sqlalchemy import Column, Integer, MetaData, Table, String
from sqlalchemy.orm import mapper

schema = MetaData()
student_table = \
    Table('student', schema,
        Column('id', Integer, primary_key=True, nullable=False),
        Column('username', String(16), nullable=False),
        Column('full_name', String(64), nullable=False),
)

class Student(object):

    def __init__(self, username, full_name):
        self.username = username
        self.full_name = full_name

mapper(Student, student_table)
```

Notice that the primary key `id` is implicitly mapped. SQLAlchemy understands the significance of the primary key, and the mapper automatically manages it for you. SQLAlchemy draws a distinction between creating an object and saving it to the database. Until the object is saved, the `id` is `None`.

While SQLObject columns assume that columns are `IS NOT NULL`, SQLAlchemy columns assume the opposite. The `nullable` keyword allows the code to change this. In the table just defined, each column explicitly sets `nullable` to `False`.

```
def test_create_unsaved_student():
    s1 = Student(username="jeff", full_name="Jeff Younker")
    assert s1.username == "jeff"
    assert s1.full_name == "Jeff Younker"
    assert s1.id is None
```

Manipulating data requires a session. Sessions come from `Session` classes, which are in turn created by the `sessionmaker()` function. The session must be bound to a database engine at some point, which can be done either when it is created or afterward. The following tests demonstrate both methods:

```
from sqlalchemy.orm import sessionmaker
...
def test_getting_a_session():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
    Session = sessionmaker(bind=engine, autoflush=True,
       transactional=True)
    unused_session = Session()

def test_getting_a_session_and_binding_later():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
    Session = sessionmaker(autoflush=True, transactional=True)
    Session.configure(bind=engine)
    unused_session = Session()
```

Saving an object adds it to the session, but the session does not instantly flush the changes to the database. A flush can be manually forced:

```
def test_create_and_save_student():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
    Session = sessionmaker(bind=engine, autoflush=True, \
      transactional=True)
    session = Session()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    assert s1.id is None
    session.flush()
    assert s1.id is not None
```

The test methods are getting a little unwieldy at this point, so refactoring the code is a good idea. The database and session configuration is refactored into a new method:

```
def session_from_new_db():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
    Session = sessionmaker(bind=engine, autoflush=True,
      transactional=True)
    return Session()

def test_create_and_save_and_flush_student():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    assert s1.id is None
    session.flush()
    assert s1.id is not None
```

Flushing is required in the preceding code, but during normal operation, `autoflush` will trigger a flush at appropriate times.

The next test saves a student and then retrieves it:

```
def test_retrieve_from_database():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    f = session.query(Student).filter_by(username="jeff").first()
    assert f is s1
    assert s1.id is not None
```

It is clear that a flush happened immediately before the query, since it found the new record, and because `id` has been set.

When working in transactional mode, it is necessary to commit the changes before they become permanent:

```
def test_commit_changes():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    session.commit()
```

Committing the session flushes all saved changes in a single transaction, closes it, and begins a new one.

As with SQLObject, attributes map through to the underlying columns in the database, and they can be modified as if they were instance variables:

```
def test_set_and_modify_database():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    #
    f = session.query(Student).filter_by(full_name=\
      "Jeff M. Younker").first()
```

```
    assert f is None
    #
    s1.full_name = "Jeff M. Younker"  # flush happens before query
    f = session.query(Student).filter_by(full_name=\
      "Jeff M. Younker").first()
    assert f is s1
```

Queries

All queries begin with the `query()` method; this has been show previously, but not noted. Queries differ in the subsequent filtering commands. With no filtering, a query returns all the rows in a table:

```
def test_query_all_rows():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="doub", full_name="Doug McBride")
    session.save(s1)
    session.save(s2)
    f = session.query(Student)
    assert Set(f) == Set([s1, s2])
```

Choosing Results

A slice of results may be selected. So far, it may appear that the results are returned as lists. This is not the case; they are actually iterable result sets. Subsets may be obtained through slicing. The chosen results are obtained using the SQL `LIMIT` and `OFFSET` directives. This allows a limited subset to be efficiently returned from a large result set, without having to transfer the query.

```
def test_query_slice():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="doub", full_name="Doug McBride")
    s3 = Student(username="amy", full_name="Amy Woodward")
    for s in [s1, s2, s3]:
        session.save(s)
    sliced = session.query(Student)[1:3]
    assert [s2, s3] == list(sliced)
    assert [s2, s3] != sliced
```

Slicing a single element does not return a result set; it immediately returns the requested element:

```
def test_query_results_with_index():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="doub", full_name="Doug McBride")
```

```
    session.save(s1)
    session.save(s2)
    f = session.query(Student)[0]
    assert s1 == f
```

This previous test's data is used repeatedly. Extracting the method `prepare_two_students()` leads to the following code:

```
def prepare_two_students(session):
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="doub", full_name="Doug McBride")
    session.save(s1)
    session.save(s2)
    return (s1, s2)

def test_query_results_with_index():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student)[0]
    assert f == s1
```

The methods `all()`, `first()`, and `one()` are used to immediately select portions of a result set. `all()` returns all the results as a collection:

```
def test_query_results_all():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).all()
    assert f == [s1, s2]
```

The `first()` method returns the first element from a result set. It is equivalent to using `[0]`, but more expressive.

```
def test_query_results_first():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).first()
    assert f == s1
```

If only one result is returned by a query, then the `one()` method behaves like `first()`:

```
def test_query_results_one_with_one_result():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter_by(username="jeff").one()
    assert f == s1
```

If the query does not return precisely one result, then `one()` raises an `InvalidRequestError`.

```
from nose.util import assert_raises
from sqlalchemy.orm.exceptions import InvalidRequestError
...
def test_query_results_one_raises_error_with_multiple_results():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    assert_raises(InvalidRequestError, session.query(Student).one)
```

Filtering with SQL Queries

In the simplest cases, the `filter()` method works analogously to SQLObject's `select()` method. In these cases, it takes one argument that is a query expression written as a Python expression. These column names come from either the mapped class or the column listings in the corresponding table object's `c` attribute. The two queries in the following test are equivalent:

```
from sqlalchemy_ex import schema, Student, student_table
...
def test_simple_filter_expressions():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter(Student.username == "jeff")
    g = session.query(Student).\
      filter(student_table.c.username == "jeff")
    assert list(f) == list(g) == [s1]
```

SQLObject requires importing SQL expression constructors from the SQLBuilder library. Under SQLAlchemy, these sorts of operators are directly accessible from columns:

```
def test_sql_filter_expressions():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter(Student.username.like('%ef%'))
    assert list(f) == [s1]
```

`filter()` also accepts raw SQL where expressions. This allows you to produce hand-tuned queries when needed. These queries can contain variables that are expanded. The two string queries in the following test are equivalent:

```
def test_simple_literal_sql_filter_expressions():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter("username = 'jeff'")
    g = session.query(Student).\
       filter("username = :un").params(un="jeff")
    assert list(f) == list(g) == [s1]
```

In the second query, the string :un is expanded to jeff. This expansion performs the appropriate escapes. If you have to substitute variables into a query, then always use this form to stave off SQL injection attacks.

Hand-tuning goes even further with the from_statement() method. It accepts complete SQL from statements:

```
def test_from_statement():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    query = "SELECT * FROM student WHERE username like :match"
    f = session.query(Student).from_statement(query).\
      params(match='%ou%').one()
    assert f == s2
```

Keyword Queries

The filter_by() method is directly analogous to SQLObject's selectBy() method. The keywords are column names, and the values will be exactly matched. The usage has already been demonstrated, but here's a reminder:

```
def test_filter_by():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter_by(username="jeff").one()
    assert f == s1
```

Chaining

The query() method produces a query object. Each filter method produces another query object as its result, so filter expressions can be chained together. The chained expressions are combined with a logical and.

```
def test_chained_filters():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="jeffs", full_name="Jeff Smith")
    session.save(s1)
    session.save(s2)
    f = session.query(Student).\
      filter(Student.full_name.like('Jeff%')).\
      filter_by(username="jeffs").one()
    assert f == s2
```

Printing query objects shows the generated SQL:

```
>>> print session.query(Student).filter_by(username="fool")
```

```
SELECT student.id AS student_id, student.username AS➥
student_username, student.full_name AS student_full_name
FROM student
WHERE student.username = :student_username_1 ORDER BY student.id
```

One-to-Many Relationships

One-to-many relationships and the conjugate many-to-one relationships are specified from the one-to-many side. While foreign keys in SQLObject automatically yield instances of the appropriate type, in SQLAlchemy only the key value is available until the connection is established from the associated table. The relationship is declared through mappers. The schema used for this example is shown in Figure 9-6.

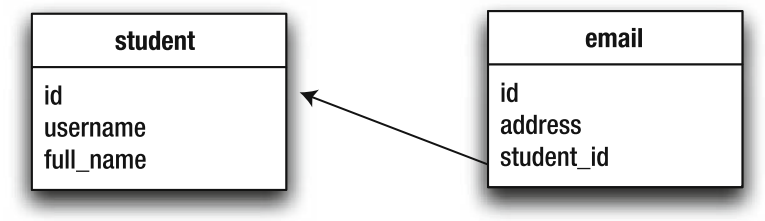

Figure 9-6. *The email table points back to the student table.*

You add the `email` table here, but the one-to-many relationship is not established yet:

```
from sqlalchemy import Column, ForeignKey, Integer, MetaData, \
  Table, String
from sqlalchemy.orm import mapper

schema = MetaData()

student_table = Table('student', schema,
                      Column('id', Integer, primary_key=True),
                      Column('username', String(16), nullable=False),
                      Column('full_name', String(64), nullable=False),
)

email_table = Table('email', schema,
                    Column('id', Integer, primary_key=True),
                    Column('address', String(255), nullable=False),
                    Column('student_id', Integer, \
                           ForeignKey('student.id'), nullable=False),
)
```

```
class Student(object):

    def __init__(self, username, full_name):
        self.username = username
        self.full_name = full_name


class Email(object):

    def __init__(self, address):
        self.address = address


mapper(Student, student_table)

mapper(Email, email_table)
```

The following test shows that the foreign key does not engender an attribute pointing back to the associated `Student` instance:

```
def test_email_doesnt_have_student_attribute():
    e1 = Email(address="jeff@not.real.com")
    assert_raises(AttributeError, getattr, e1, 'student')
```

The `mapper()` directive for the `student` table establishes the one-to-many relationship between `Student` and `Email`:

```
from sqlalchemy.orm import mapper, relation
...
mapper(Student, student_table, properties={
           'emails': relation(Email, backref="student")
})
```

The previous test now fails, and it is changed to one that verifies the existence of this attribute:

```
def test_email_now_has_student_attribute():
    e1 = Email(address="jeff@not.real.com")
    assert e1.student is None
```

The relationship can now be established from either end of the connection. On the `Student` end, the one-to-many connection appears as a list, just as with SQLObject—but there it was a read-only attribute. Here it is writable, though, and adding an object to it sets the appropriate foreign key in the added object:

```
def test_email_adding_via_one_to_many_side():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    e1 = Email(address="jeff@not.real.com")
    session.save(s1)
    s1.emails.append(e1)
```

```
    session.flush()
    assert s1.emails == [e1]
    assert e1.student == s1
```

As stated earlier, it works from the other direction, too:

```
def test_email_adding_via_many_to_one_side():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    e1 = Email(address="jeff@not.real.com")
    session.save(s1)
    e1.student = s1
    assert s1.emails == [e1]
    assert e1.student == s1
```

Elements can also be deleted from the joined attribute as if it is a normal list. Were the foreign key in the `email` table `nullable`, then the following test would work:

```
def test_email_removing_via_many_to_one_side():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    e1 = Email(address="jeff@not.real.com")
    session.save(s1)
    s1.emails.append(e1)
    session.flush()
    del s1.emails[0]
    session.flush()
    assert s1.emails == []
```

Many-to-Many Relationships

As with one-to-many relationships, creating the simplest many-to-many relationships is a little more involved than with SQLObject, as the intermediate tables must be explicitly described.

The schema described here is pictured in Figure 9-7.

```
course_table = Table('course', schema,
                     Column('id', Integer, primary_key=True),
                     Column('name', String(64), nullable=False),
)

enrolled_assc_table = \
  Table('enrolled_assc', schema,
    Column('student_id', Integer, ForeignKey('student.id')),
    Column('course_id', Integer, ForeignKey('course.id')),
)
...
```

```
class Course(object):

    def __init__(self, name):
        self.name = name
...
mapper(Course, course_table, properties={
    'enrolled': relation(Student, secondary=enrolled_assc_table)
})
```

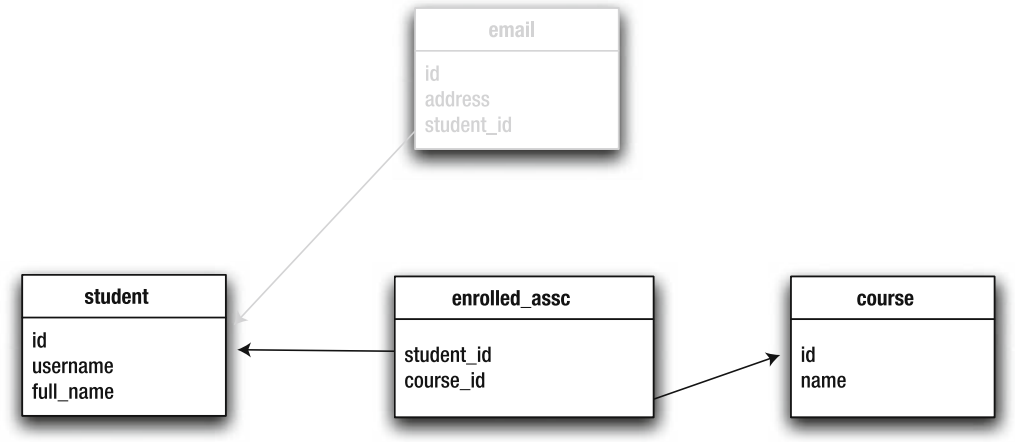

Figure 9-7. *Students are enrolled in courses.*

The keyword `secondary` indicates that this is a many-to-many relationship. Unlike SQLObject, it is not necessary to describe the details of this intermediate class when the relationship is declared, as this has already been done in the table definition.

The preceding declaration only creates a link from the `Course` object to the `Student` object. To make this link from the `Student` object back to the `Course` object, the `backref` keyword must be used:

```
mapper(Course, course_table, properties={
    'enrolled': relation(Student,
                secondary=enrolled_assc_table,
                backref='enrolled')
})
```

The relationship can now be viewed from either side, as shown in the next test. The next few tests require a pair of `Course` instances, so you'll create a method to prepare the needed test data.

```
from sqlalchemy_ex import Course, Email, schema, Student, student_table
...
def prepare_two_courses(session):
    c1 = Course('Modern Algebra')
    c2 = Course('Biochemistry')
```

```
    session.save(c1)
    session.save(c2)
    return (c1, c2)

def test_enrolled_adding():
    session = session_from_new_db()
    (s1, unused_s2) = prepare_two_students(session)
    (c1, c2) = prepare_two_courses(session)
    s1.enrolled.append(c1)
    c2.enrolled.append(s1)
    session.flush()
    assert Set(s1.enrolled) == Set([c1, c2])
    assert c1.enrolled == [s1]
    assert c2.enrolled == [s1]
```

Querying Relations

SQLAlchemy provides better support for querying relations than does SQLObject. This is done through the `join()` query method:

```
def test_select_student_by_course():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    (c1, c2) = prepare_two_courses(session)
    s1.enrolled.append(c1) # Course "Modern Algebra"
    s2.enrolled.append(c2) # Course "Biochemistry"
    session.flush()
    f = session.query(Student).join('enrolled').\
             filter(Course.name=="Biochemistry").one()
    assert f == s2
```

Deleting

Deletions are scheduled using the session's `delete()` method:

```
def test_delete_student():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    session.flush()
    session.delete(s1)
    students = session.query(Student).all()
    assert students == [s2]
```

By default, deletes don't cascade. Deleting an entity related to others leaves orphans and dangling foreign keys. Cascading deletes are a property of a relation, and are declared through the `cascade` keyword:

```
def test_delete_cascade():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    e1 = Email(address="jeff@not.real.com")
    session.save(s1)
    s1.emails.append(e1)
    session.flush()
    session.delete(s1)
    session.flush()
    email = session.query(Email).all()
    assert email == []
```

When setting up delete cascades with bidirectional relations, you should pay close attention to which object is the desired parent.

Going Further with SQLAlchemy

I've tried to cover the absolute minimum that you need to know to work with SQLAlchemy. An entire book could easily be written about it.

Fortunately, it has wonderful documentation. It is concise and clear, but also voluminous. Topics worth examining further include the following:

- Building SQL expressions
- Using raw connections as a better DBI layer
- Returning custom collections from relations
- Loading strategies
- Mapping classes onto arbitrary select results
- Mapping classes onto multiple tables
- Table inheritance

Building the Database

Agile development entails short iterations. Updating databases with short iterations requires automation. There is no time to hand-tweak changes or adjust between the development environments, the QA environments, and the production environments. The only sensible way to do this is to use the same system throughout.

The database consists of the schema and the test data. Database setup must include both—however, the schema and the test data must be separable. The schema will go to QA and production, but the test data will not. The same mechanisms used to produce a database in development must be used to produce the database in QA and production. Otherwise, the changes produced in the first part of the development pipeline will have to be adapted to the mechanisms and processes used in later parts. This is a recipe for error, delay, and wasted effort.

The database schema is part of the source code. The code depends upon having a specific schema version. If application code can't access the database, then the data stored there may be worthless. Without a database running the appropriate schema version, the code is useless, and without appropriate code, the information in the database is useless; the two are inextricably intertwined. Since the database schema is part of the code base, it is owned collectively, and it is versioned.

Even though many more modifications happen to the database, using the same migration mechanisms throughout means that less operational effort must be expended. Martin Fowler has mentioned a project that had 30 developers and 1 DBA. He has also stated that many projects reduce their number of dedicated DBAs by substituting developers with database skills.

Much of the work involved with agile database development has focused on organizations in which the database is closely coupled to a single application. Some production environments have databases that serve many applications. Old financial institutions are one such case, as are potentially medical records systems. In these cases, the database schema may not travel along with a single application. There has been less work done in these areas, but the techniques developed to manage incremental database change can still be applied to them.

Testing

Programs that interact with databases have common elements, and testing each requires a different approach. These elements include

- Application–mapping layer interactions
- Mapping layer–database interactions
- Functional interactions between the application and the database
- Embedded code
- Database migrations

Application–mapping layer interactions are often addressed most easily through normal unit-testing isolation techniques; mapped objects can frequently be replaced with impostors. When testing larger subsystems, fake databases may be useful, although the existence of embedded databases and in-memory databases lessens the need for these.

Mapping layer–database interactions often benefit from using a real database of some sort. Behavioral differences between various kinds of target databases can be identified or verified before integration. Doing this requires running instances to be available to each and every developer. These days, a multi-CPU desktop has more than enough horsepower to run several virtual machines hosting "real" databases such as Oracle, Microsoft SQL Server, or Sybase. For basic sanity checking of the mapping layer, it is usually sufficient to use something like SQLite.

■ **Warning** No development work should ever be done against production databases. This is a recipe for disaster. Mistakes can easily destroy production data; successful modifications remove the database from a known state, and this has the potential to ruin automation. If a problem can't be replicated in development or QA, that suggests there is something wrong with the development and QA resources.

Functional interactions between the application and the database are addressed by using real databases. As noted previously, VMs are useful for this. Indeed, I know of several organizations in which a scaled-down version of the entire production network runs on each developer's desktop. Over the next few years, I expect this sort of full environment simulation to become more common.

Embedded code refers to code that runs in the database. Triggers and stored procedures are the most frequently used kinds of embedded code. The only way to test this code is with a live database.

Database migrations must be tested against their target databases. Those migrations must be tested against both the schemas and realistic sets of data. How this is done constitutes much of the remainder of this chapter.

Refactorings

Refactorings are modifications to the database that improve its structure. As with code refactorings, these are incremental changes. They can be applied to the existing database to achieve a desired final state. Unlike source refactorings, they have to take data into account, too.

If you're lucky, then your database can be brought down completely while your software is upgraded and the current database is refactored. If you work in a 24/7 environment or you have many applications working with a single database, then this is often not a possibility. In these cases, refactorings have to be performed in two steps.

The first step adjusts the database so that both the old and new versions of the application continue to work with the new database. This is referred to as the transition period. During this time, the applications depending on the old schema are upgraded.

Once the applications have been upgraded, compatibility with the old schema is no longer required. It is safe to transform the database to the completed state.

Consider renaming a column from `Foo` to `Bar`. When the change is applied, a new column, `Bar`, is added. Either the application code or database machinery mirrors changes between `Bar` and `Foo` so that old code continues to operate. Once all running applications reference `Bar`, the mirroring mechanism is turned off, and the column `Foo` is removed.

Almost all live production database environments are upgraded incrementally, so refactorings fit into the DBA's natural operational model.

"But I run a Java cluster on server Foo, and it switches deployed versions all at once!" No it doesn't. Every Java clustering mechanism I have seen switches over to the new version after the old connections terminate. Connections to each machine will terminate at different rates. This means that every member of the cluster switches to the new version of your application at different times. The result is that, often for a few seconds, some proportion of your cluster is running at version n, and some proportion of your cluster is running at version n+1.

Migrations

All developer databases should be synchronized. This can be done in one of two ways: either each developer's instance is updated from a central location, or the database is reconstructed by the build. These two are not mutually exclusive, either.

I prefer the second option, in which the database is reconstructed by the build. The migration of the central source of truth needs to be done, and so it will need to be in a replicable

manner. If it can be done once, then it can be done many times, and only one mechanism will be required.

Over the last few years, a consensus has started to emerge about how database migrations should be performed. The database records what schema version it is running. Developers write a set of instructions describing how to get from one version to the next. A tool consults the database version and the desired version, it determines which set of instructions must be applied, and then it applies them.

These instructions are never applied by hand—only by automation. They are applied to production en masse at release time. Generally, reverse scripts are supplied so that the production database may be rolled back if needed.

The Instructions

There are two broad systems for generating the migration instructions:

> With *explicit migrations*, the developer declares which steps will be taken. In some cases, this is done with XML declarations. In others, it is done with a DSL (Rails Migrations is an example). Sometimes it is done in the application language code, and often it is done in raw SQL. The first three techniques have the advantage of being database independent. The last is not, but it has the advantage of giving direct access to many database features, including applying security.
>
> With *derived migrations*, the instructions are derived from the difference between the desired schema and the current schema. Sometimes the desired state is encoded in the full schema. More often, it is in a DSL, with XML being one of the more common formats. In some systems, the difference is derived by comparing the desired schema with the database itself.

Derived migrations are appealing, but in practice experienced developers seem to be very wary of them. Even if the derived migrations work, data migration code must still often be written. A framework for running this code must be supplied, and this is the same framework that is necessary for running explicit migrations. If you have one, why complicate it with the other? Explicit migrations also give the opportunity for migration procedure code reviews.

Numbering Migrations and Playing Them Back

In the simplest case, the migration scripts are numbered in a monotonically increasing integer sequence (1, 2, 3, etc.). The database records the most recently applied script. All scripts below that are assumed to have been applied, and those above have not. When a migration happens, all the scripts between the current version and the desired version are played back, and the last one is recorded.

The simple numbered sequence works for small groups working on a single codeline. When those two assumptions break down, so does the sequence numbering. The numbers are now a scarce resource, and developers must arbitrate access to them. When merges happen, duplicates collide and the migrations have to be renumbered, and heaven help you if they have been applied to shared resources. The potential for error quickly becomes large.

The solution is to use numbers with a low chance of collision: timestamps. They are monotonically increasing integers, and there is a small chance that two people will choose exactly the same second to create a migration file. The migration system also needs to know

nothing about what they represent, as they're just a fancy kind of integer. A typical timestamp might be `20080204080953` (Feb, 4, 2008, 08:09:53.)

That doesn't completely solve the merge problem, though. Consider the case in which one branch has already been applied to a database, and another branch is merged in. Both were under development at the same time, so they have migrations with intermingled timestamps. The odds are that these migrations are independent, so you should be able to play them back successfully.

Problems happen when some of the newly merged migrations are below the current version. These will not be played back when a new migration is attempted. This can be solved by tracking all of the applied revisions instead of just the most recently applied revision. When a migration happens, the desired version is determined, the applied migration list is consulted, and then all unapplied migrations below the desired version are applied to the database.

Where to Put the Migration Mechanism

Applying migrations can be viewed as part of the installer, part of the application, or the duty of a special application that just manages database upgrades. It all depends on the application that you're using and its intended purpose. I don't have firm feelings except in the case of clustered applications. In these environments, coordinating deployment is a major headache, and I feel that migration duties belong with the application or with a special-purpose database migration management application. No matter where you put it, a mechanism is required to manage migration attempts.

DBMigrate: A Migration Mechanism

Writing the migration mechanism itself is painful. While not seeming terribly complicated, it has lots of edge cases that need to be addressed. This problem has been tackled in the Ruby world with Rails Migrations. In the Java world, PatchDB is a notable example, and there are many others. The Python world has had no such mechanism . . . until now. In the course of working for my employer I had to write one, as did an acquaintance of mine. I've taken aspects of my code and his, and I've published DBMigrate.

Using DBMigrate

DBMigrate is installed via `easy_install dbmigrate`. Migrations and test data are written as Python packages. These migrations can be applied through the following:

- Command-line tools
- Setuptools directives
- Embedding the engine within your program
- Unit tests

The tool supports test data importation. Applied migrations are tracked individually rather than using a single counter. The application supports bootstrapping and complete tear-down of a database. Migrations are explicit, and they are written as raw SQL, Python functions, or a mixture of both. Different kinds of databases can have different migrations;

MySQL may have one set of migrations while SQLite has another. In this case, MySQL might set user permissions. SQLite does not have user permissions, so these are skipped.

Starting from Scratch

The database must be created before it can be used. With SQLite, this isn't a problem—the file is the database. However, with other databases, DBMigrate must be able to connect as a user that has permission to create databases, and in many cases, to grant privileges. DBMigrate calls this user the *admin user*; the admin user has an associated admin password. This is distinct from the application user that will attach to the database in production or during testing.

Throughout the application, the following keywords are referenced:

`scheme`: The database connection scheme (e.g., `sqlite` or `mysql`).

`admin_user`: The database user with rights to create a database, create users, and grant rights.

`admin_pw`: The admin user's password.

`user`: The database user that the application connects as. This user may not exist until the migrations are run. If this is omitted, then it is assumed to be the same as `admin_user`.

`pw`: The application user's password. If this is omitted, then it is assumed to be the same as `admin_pw`.

`db`: The name of the database to be created.

`host`: The name of the host on which the database server runs.

`port`: The port number on which the database server listens.

`socket`: The path to the socket that the database listens on.

`versiontable`: The name of the table containing the applied revisions.

These values are passed to the migration scripts in a dictionary. This is the set of *expansions* for a database.

Creating Migrations

The first migration must always set up the records table. Migrations are stored in a table.

```
$ python ./setup.py make_migration --package apptest.db.schema➥
--name create_db
```

```
Migration apptest.db.schema.migrate_20080218151301_create_db created.
```

```
$ more apptest/db/schema/migrate_20080218151301_create_db.py
```

```
# Migration template created by DBMigrate at 2008/02/18 at 15:13:01 UTC.

migration = []
```

Migrations are specified as a list of atoms. The atoms are tuples. The first component of the atom is a single SQL statement that performs an upgrade. The second component is a single SQL statement that undoes the first operation. Either one may be an empty string or `None`.

The atoms are applied in order from first to last when upgrading. They are applied in reverse order when downgrading.

A sample migration to create the `student` table from earlier in this chapter follows. This is one of those cases where I feel that breaking convention for readability is worth it.

```
migration = [("""
        CREATE TABLE student (
           id INTEGER PRIMARY KEY AUTO_INCREMENT,
           full_name VARCHAR(64) NOT NULL,
           username VARCHAR(16) NOT NULL
        )
    """, """
        DROP TABLE student
    """),
]
```

Migration strings are expanded before they are executed. This is done with Python string expansion using named parameters such as `%(`*foo*`)s`. The precise set of expansions depends upon the kind of database being constructed. SQLite uses only the minimum set of expansions: `scheme` and `versiontable`. Databases with multiple accounts will always expand `user` and `pw`. Database servers with multiple databases also expand `db`.

```
migration = [("""
        CREATE TABLE %(db)s.student (
           id INTEGER PRIMARY KEY AUTO_INCREMENT,
           full_name VARCHAR(64) NOT NULL,
           username VARCHAR(16) NOT NULL
        )
    """, """
        DROP TABLE %(db)s.student
    """),
]
```

Migrations can also be functions. These functions receive a SQLAlchemy connection argument. The function optionally accepts an expansions dictionary. Migration functions and migration strings can be freely intermixed.

```
def my_data_migration_up(connection, expansions):
    # This just happens to accept an expansions dictionary
    pass

def my_data_migration_down(connection):
   # This doesn't
   pass

migration = [(my_data_migration_up, my_data_migration_down)]
```

Different migrations can be specified for different database schemes. These schemes correspond to the schemes used in SQLAlchemy database URIs. In this case, `migration` is a dictionary instead of a list. The keys correspond to the database's URI scheme. For example, if the URI for the database was `mysql://localhost/db`, then the scheme would be `myqsl`. The default migration is used when there is no matching migration. It is keyed with `_`.

```
generic_migration = [("""
        CREATE DATABASE %(db)s
    """, """
        DROP DATABASE %(db)s
    """),
   ("""
        CREATE TABLE %(db)s.%(versiontable)s (
           id INTEGER PRIMARY KEY AUTO_INCREMENT,
           package VARCHAR(64) NOT NULL,
           revision INTEGER UNSIGNED NOT NULL
        )
    """, """
        DROP TABLE %(db)s.%(versiontable)s
    """),
]

sqlite_migration = [("""
        CREATE TABLE %(versiontable)s (
           id INTEGER PRIMARY KEY AUTOINCREMENT,
           package VARCHAR(64) NOT NULL,
           revision INTEGER UNSIGNED NOT NULL
        )
    """, """
        DROP TABLE %(versiontable)s
    """),
],

migration = {'_': generic_migration, 'sqlite': sqlite_migration}
```

The first migration always creates the database and migration schema. Templates for these migrations are available in `db.migrate.templates`. Currently, there are templates for SQLite, MySQL, and PostgreSQL, and one suitable for use under Pylons.

Manually Migrating a Database

Databases are manually migrated using `setup.py`. Different parameters are supplied depending on the database being migrated. The following command shows a MySQL database being upgraded to the most recent version:

```
$ python ./setup.py dbmigrate --scheme mysql --admin_user root➥
--admin_pw ROOT_USER_PW --user dbu --pw dbpw➥
--host localhost -v --db mydb my.app.schema
```

```
Upgrade from revision 0 to revision 20080219120356
Applying my.app.schema.migrate_20080218151301_create_db
Applying my.app.schema.migrate_20080218192701_create_student
Applying my.app.schema.migrate_20080219120356_create_email
```

The database can be torn down using the `--revision` flag:

```
$ python ./setup.py dbmigrate --scheme mysql --admin_user root➥
--admin_pw ROOT_USER_PW --user dbu --pw DB_PW➥
--host localhost -v --db mydb --revision 0 my.app.schema
```

```
Downgrade from 20080219120356 to revision 0
Applying my.app.schema.migrate_20080219120356_create_email
Applying my.app.schema.migrate_20080218192701_create_student
Applying my.app.schema.migrate_20080218151301_create_db
```

As with any Setuptools command, commonly used options can be stored in the `setup.cfg` file. The section heading is `[dbmigrate]`. For the preceding command, the corresponding `setup.cfg` is the following:

```
[dbmigrate]
    scheme=mysql
    admin_user=root
    admin_pw=ROOT_USER_PW
    db=mydb
    user=dbu
    pw=DB_PW
    host=localhost
```

Minus the verbose flag, `-v`, the previous installation commands are now the following:

```
$ python ./setup.py dbmigrate my.app.schema
$ python ./setup.py dbmigrate --revision 0 my.app.schema
```

When more than two packages are passed to DBMigrate, the migrations are interleaved based on their timestamps. The ordering between identical timestamps is undefined; they may be applied in any order.

```
$ python ./setup.py dbmigrate -v my.app.schema my.test.schema
```

```
Upgrade from revision 0 to revision 20080219120534
Applying my.app.schema.migrate_20080218151301_create_db
Applying my.app.schema.migrate_20080218192701_create_student
Applying my.app.testdata.migrate_20080218192734_populate_student
Applying my.app.schema.migrate_20080219120356_create_email
Applying my.app.testdata.migrate_20080219120534_populate_email
```

Running DBMigrate with Unit Tests

Running unit tests requires a minimal `setup.cfg`. `scheme` is the only required parameter for all databases; in general, only the minimum amount of information required to create an administrative connection is needed. For SQLite, nothing is required; for MySQL, only `scheme=mysql`, `admin_user`, and `admin_pw` are required.

A `setup.cfg` for MySQL might read as follows:

```
[dbmigrate]
    scheme=mysql
    admin_user=root
    admin_pw=ROOT_USER_PW
    host=localhost
```

The values `user`, `pw`, and `db` are ignored by the unit tests, as DBMigrate creates random values for them. This results in a unique database; these randomly chosen values are still passed to the migration scripts in the expansions dictionary.

Within a unit test, migrations are applied using `dbmigrate.DBTestCase` in the case of unittest. The method `connect_application()` is called from `setUp()`, and the method `disconnect_application()` is called from `tearDown()`. If these methods are not supplied, then no error results.

```
from dbmigrate import DBTestCase

class MyTestCase(DBTestCase):
    """Database test case using migrations framework"""

     # One or more sets of migrations
    migrations = ['my.app.schema', 'my.app.testdata']

    def connect_application(self, uri, expansions):
        # The function should connect your application code
        MyApp.connect(uri)

    def disconnect_application(self, uri, expansions):
        # This function should disconnect your application code
        MyApp.disconnect()

    def test_method(self):
        # this would be a test method if this were real code.
        MyApp.run()
```

Running DBMigrate from Your Program

Your program may need to control the setup or tear-down of a database at install time or runtime. In these cases, the migration framework can be embedded and run from your application code. The migration engine is run with a dictionary of connection parameters and a list of packages containing schema files.

```
from dbmigrate import MigrationEngine
...
def install_database():
    config = read_your_app_config()
    params = {'scheme': 'mysql',
              'admin_user': config['db.admin_user'],
              'admin_pw': config['db.admin_pw'],
              'user': config['db.user'],
              'pw': config['db.pw'],
              'host': config['db.host'],
              'port': config['db.port'],
              'db': config['db.name'],
    }
    MigrationEngine().run(params, ['my.app.schema'])
```

The preceding code shows how the connection parameter dictionary `params` might be constructed from an application's configuration, and then how the migration engine is run.

Specific revisions may also be requested using the `revisions` keyword:

```
MigrationEngine().run(params, ['my.app.schema'],
                      revision=20080215135324)
```

The database is upgraded or downgraded to achieve the desired revision.

Summary

Database technology has become steadily cheaper and more widely available over the last ten years. Only recently has database development started to adapt to these changing realities. The result is an agile approach to database development sometimes called evolutionary database development. It is built around the assumption that databases, like software, will change.

Agile development's focus on short iterations forces this change. Traditional DBA organizations are unable to meet the rapid turnaround without large increases in staff. The only way to shorten cycles is through automation, and automation by nature pervades the entire development cycle.

As a result, database development is viewed as part of the overall software development process. DBAs and developers work closely together to understand each other's concerns as early in the development cycle as possible. The developers themselves often compose the migration scripts for the database.

ORMs are valuable tools for isolating application code from the details of the underlying database. They map between application objects and database structures such as tables, columns, and rows. Because database access is mediated by normal objects, testing can be performed using normal techniques. The two most common Python ORMs are SQLObject and SQLAlchemy.

In an agile environment, database upgrades must be performed through automation. This automation must be consistent and testable, and is often done through schema migration systems. These store the schema version in the database and compare it to the version required by the code. If the two disagree, then a series of migration scripts are applied to bring the database into conformance.

Agile database development is very much on the cutting edge of agile development. There is room for many new tools and new practices. It is problematic because the development involves external applications and additional special-purpose languages working in conjunction with the application code.

Another cutting edge area is web development, which has similar issues. Today's rich web applications are built around JavaScript running in the user's browser. Compatibility must be maintained against a wide variety of client platforms. Somehow this code and its interactions with the Python application must be verified—which is the subject of the next chapter.