11

Python

If you are not familiar with Python and you do a lot of Perl programming, you definitely want to take a look at it. Python is an object-oriented scripting language that combines the strengths of languages like Perl and Tcl with a clear syntax that lends itself to applications that are easy to maintain and extend. The O'Reilly & Associates, Inc. book *Learning Python, 2nd Edition* by Mark Lutz and David Asher provides an excellent introduction into Python programming. This chapter assumes a working understanding of the Python language.

In order to follow the content of this chapter, you will need to download and install the MySQLdb, the MySQL version of DB-API. You can find the module at *http://dustman.net/andy/python/MySQLdb*. Chapter 23, *The Python DB-API* in the reference section includes directions on how to install MySQLdb.

DB-API

Like Java and Perl, Python has developed a unified API for database access—DB-API. This database API was developed by a Python Special Interest Group (SIG) called the Database SIG. The Database SIG is a group of influential Python developers interested Python access to various databases. On the positive side, DB-API is a very small, simple API. On the negative side, it isn't very good. Part of its problem is that it is very small and thus does not support a lot of the more complex features database programmers expect in a database API. It is also not very good because it realistically does not enable true database independence.

The Database Connection

The entry point into DB-API is really the only part of the API tied to a particular database engine. By convention, all modules supporting DB-API are named after

the database they support with a "db" extension. The MySQL implementation is thus called MySQLdb. Similarly, the Oracle implementation would be called oracledb and the Sybase implementation sybasedb. The module implementing DB-API should contain a `connect()` method that returns a DB-API connection object. This method returns an object that has the same name as the module:

```
import MySQLdb;
conn = MySQLdb.connect(host='carthage', user='test',
                       passwd='test', db='test');
```

The above example connects using the user name/password pair 'test'/'test' to the MySQL database 'test' hosted on the machine 'carthage'. In addition to these four arguments, you can also specify a custom port, the location of a UNIX socket to use for the connection, and finally an integer representing client connection flags. All arguments must be passed to connect() as keyword/value pairs in the example above.

The API for a connection object is very simple. You basically use it to gain access to cursor objects and manage transactions. When you are done, you should close the connection:

```
conn.close();
```

Cursors

Cursors represent SQL statements and their results. The connection object provides your application with a cursor via the `cursor()` method:

```
cursor = conn.cursor();
```

This cursor is the center of your Python database access. Through the `execute()` method, you send SQL to the database and process any results. The simplest form of database access is of course a simple insert:

```
conn = MySQLdb.connect(host='carthage', user='test', passwd='test',
db='test');
cursor = conn.cursor();
cursor.execute('INSERT INTO test (test_id, test_char) VALUES (1, 'test')");
print "Affected rows: ", cursor.rowcount;
```

In this example, the application inserts a new row into the database using the cursor generated by the MySQL connection. It then verifies the insert by printing out the number of rows affected by the insert. For inserts, this value should always be 1.

Query processing is a little more complex. Again, you use the `execute()` method to send SQL to the database. Instead of checking the affected rows, how-

ever, you grab the results from the cursor using one of many fetch methods. Example 11-1 shows a Python program processing a simple query.

Example 11-1. A Simple Query

```
import MySQLdb;

connection = None;
try:
    connection = MySQLdb.connect(host="carthage", user="user",
                                 passwd="pass", db="test");
    cursor = connection.cursor();
    cursor.execute("SELECT test_id, test_val FROM test ORDER BY test_id");
    for row in cursor.fetchall():
        print "Key: ", row[0];
        print "Value: ", row[1];
     connection.close();
 except:
    if connection:
        connection.close();
```

The cursor object actually provides several fetch methods: `fetchone()`, `fetchmany()`, and `fetchall()`. For each of these methods, a row is represented by a Python tuple. In the above example, the `fetchall()` method fetches all of the results from the query into a list of Python tuples This method, like all of the fetch methods, will throw an exception if the SQL was not a query.

Of course, fetching all of the rows at once can be very inefficient for large result sets. You can instead fetch each row one by one using the `fetchone()` method. The `fetchone()` method returns a single row as a tuple where each element represents a column in the returned row. If you have already fetched all of the rows of the result set, `fetchone()` will return `None`.

The final fetch method, `fetchmany()`, is middle ground between `fetchone()` and `fetchall()`. It enables an application to fetch a pre-defined number of rows at once. You can either pass in the number of rows you wish to see returned or instead rely on the value of `cursor.arraysize` to provide a default value.

Parameterized SQL

DB-API includes a mechanism for executing pseudo-prepared statements using the `execute()` method as well as a more complex method called `executemany()`. Parameterized SQL is a SQL statement with placeholders to which you can pass arguments. As with a simple SQL execution, the first argument to `execute()` is a SQL string. Unlike the simple form, this SQL has place holders for parameters specified by the second argument. A simple example is:

```
    cursor.execute('INSERT INTO COLORS (COLOR, ABBR) VALUES (%s, %s)',
                   ('BLUE', 'BL'));
```

In this example, %s is placed in the SQL as placeholders for values that are passed as the second argument. The first %s matches the first value in the paramter tuple and the second %s matches the second value in the tuple.

DB-API actually has several ways of marking SQL parameters. You can specify the format you wish to use by setting `MySQLdb.paramstyle`. The above example is `MySQLdb.paramstyle = "format"`. The "format" value is the default for MySQLdb when a tuple of parameters is passed to `execute()` and is basically the set of placeholders from the ANSI C `printf()` function. Another possible value for `MySQLdb.paramstyle` is "pyformat". This value is the default when you pass a Python mapping as the second argument.

DB-API actually allows several other formats, but MySQLdb does not support them. This lack of support is particularly unfortunate since it is common practice in database applications in other languages to mark placeholders with a ?.

The utility of parameterized SQL actually becomes appearant when you use the `executemany()` method. This method enables you to execute the same SQL statement with multiple sets of parameters. For example, consider this code snippet that adds three rows to the database using `execute()`:

```
cursor.execute("INSERT INTO COLOR (COLOR, ABBREV) VALUES ('BLUE', 'BL')");
cursor.execute("INSERT INTO COLOR (COLOR, ABBREV) VALUES ('PURPLE', 'PPL')");
cursor.execute("INSERT INTO COLOR (COLOR, ABBREV) VALUES ('ORANGE', 'ORN')");
```

That same functionality using `executemany()` looks like this:

```
cursor.executemany("INSERT INTO COLOR ( COLOR, ABBREV ) VALUES (%s, %s )",
                   (("BLUE", "BL"), ("PURPLE", "PPL"), ("ORANGE", "ORN")));
```

As you can see, this one line executes the same SQL three times using different values in place of the placeholders. This can be extremely useful if you are using Python in batch processing.

MySQLdb treats all values as string values, even when their underlying database type is BIGINT, DOUBLE, DATE, etc. Thus, all conversion parameters should be %s even though you might think they should be a %d or %f.

Other Objects

DB-API provides a host of other objects to help encapulate common SQL data types so that they may be passed as parameters to `execute()` and `executemany()` and relieve developers of the burden of formatting them for different databases. These objects include Date, Time, Timestamp, and Binary. MySQLdb

supports these objects up to a point. Specifically, when MySQLdb binds parameters, it converts each paramter to a string (via `__str__`) and places it in the SQL. The Timestamp object, in particular, includes fractional seconds and MySQL considers this illegal input. The following code creates a Date for the current time and updates the database:

```
import time;
d = MySQLdb.DateFromTicks(time.time());
cursor.execute("UPDATE test SET test_date = %s WHERE test_id = 1", (d,));
```

It is important to note that MySQLdb does not properly implement the `Date()`, `Time()`, and `Timestamp()` constructors for their respective objects. You instead have to use the `DateFromTicks()`, `TimeFromTicks()`, and `TimestampFromTicks()` methods to get a reference to the desired object. The argument for each of these methods is the number of seconds since the epoch.

Proprietary Operations

In general, you should stick to the published DB-API specification when writing database code in Python. There will be some instances, however, where you may need access to MySQL-specific functionality.MySQLdb is actually built on top of the MySQL C API, and it exposes that API to programs that wish to use it. This ability is particularly useful for applications that want meta-data about the MySQL database.

Basically, MySQLdb exposes most C methods save those governing result set processing since cursors are a better interface for that functionality. Example 11-2 shows a trivial application that uses the `list_dbs()` and `list_tables()` methods from the C API to loop through all of the tables in all of the databases on the MySQL server and print out the first row from each table. Needless to say, do not run this application against a production machine.

Example 11-2. . A Python Application Using Proprietary Functionality

```
import MySQLdb;

conn = None;
try:
    conn = MySQLdb.connect(host="carthage", user="test",
                           passwd="test", db="test");
    for db in conn.list_dbs():
        for tbl in conn.list_tables(db[0]):
            cursor = conn.cursor();
            cursor.execute("SELECT * FROM " + tbl[0]);
            print cursor.fetchone();
            cursor.close();
except:
```

Example 11-2. . A Python Application Using Proprietary Functionality

```
    if conn:
        conn.close();
```

Chapter 23, *The Python DB-API* lists the proprietary APIs exposed by MySQLdb.

Applied DB-API

So far, we have walked you through the DB-API and showed you its basic functionality. As a comprehensive database access API, it still leaves a lot to be desired and does not compete with more mature APIs like Perl DBI and Java JDBC. You should therefore except significant change in this API over time. Now, we will go through a practical example of a Python database application using the DB-API.

Our example is a batch routine that pulls stale orders from an order database and builds an XML file. Business partners can then download this XML file and import the order information into their database. Example 11-3 shows a sample generated XML file.

Example 11-3. . An XML File Containing Order Information for a Fictitious Manufacturer

```
<?xml version="1.0"?>

<order orderID="101" date="2000" salesRepID="102">
  <customer customerID="100">
    <name>Wibble Retail</name>
    <address>
      <lines>
        <line>
          1818 Harmonika Rd.
        </line>
      </lines>
      <city>Osseo</city>
      <state>MN</state>
      <country>US</country>
      <postalCode>55369</postalCode>
    </address>
  </customer>
  <lineItem quantity="2">
    <unitCost currency="USD">12.99</unitCost>
    <product productID="104">
      <name>Wibble Scarf</name>
    </product>
  </lineItem>
  <lineItem quantity="1">
    <unitCost currency="USD">24.95</unitCost>
    <product productID="105">
      <name>Wibble Hat</name>
    </product>
  </lineItem>
</order>
```

The XML enables the our business partners to trade information about orders without having to know anything about our data model. Every night, a Python script runs to look for orders that have not been converted to XML in the last day. Any such orders are then read from the database in Figure 11-1 and then converted into XML.

FIGURE11-1.BMP

Figure 11-1. . The Data Model for the Manufacturing Database

The Python script, *xmlgen.py*, starts with a few simple imports:

```
import sys, os;
import traceback;
import MySQLdb;
```

Much of the script defines Python objects that encapsulate the business objects in the database. Example 11-4 contains the code for these business objects.

Example 11-4.

```
class Address:
    def __init__(self, l1, l2, cty, st, ctry, zip):
        self.line1 = l1;
        self.line2 = l2;
        self.city = cty;
        self.state = st;
        self.country = ctry;
        self.postalCode = zip;

    def toXML(self, ind):
        xml = ('%s<address>\r\n' % ind);
        xml = ('%s%s  <lines>\r\n' % (xml, ind));
        if self.line1:
            xml = ('%s%s    <line>\r\n%s      %s\r\n%s    </line>\r\n' %
                   (xml, ind, ind, self.line1, ind));
        if self.line2:
            xml = ('%s%s    <line>\r\n%s      %s\r\n%s    </line>\r\n' %
                   (xml, ind, ind, self.line2, ind));
        xml = ('%s%s  </lines>\r\n' % (xml, ind));
        if self.city:
            xml = ('%s%s  <city>%s</city>\r\n' % (xml, ind, self.city));
        if self.state:
            xml = ('%s%s  <state>%s</state>\r\n' % (xml, ind, self.state));
        if self.country:
            xml = ('%s%s  <country>%s</country>\r\n' % (xml,ind,self.country));
        if self.postalCode:
            xml = ('%s%s  <postalCode>%s</postalCode>\r\n' %
                   (xml, ind, self.postalCode));
        xml = ('%s%s</address>\r\n' % (xml, ind));
        return xml;

class Customer:
    def __init__(self, cid, nom, addr):
```

Example 11-4.

```
        self.customerID = cid;
        self.name = nom;
        self.address = addr;

    def toXML(self, ind):
        xml = ('%s<customer customerID="%s">\r\n' % (ind, self.customerID));
        if self.name:
            xml = ('%s%s  <name>%s</name>\r\n' % (xml, ind, self.name));
        if self.address:
            xml = ('%s%s' % (xml, self.address.toXML(ind + '  ')));
        xml = ('%s%s</customer>\r\n' % (xml, ind));
        return xml;

class LineItem:
    def __init__(self, prd, qty, cost):
        self.product = prd;
        self.quantity = qty;
        self.unitCost = cost;

    def toXML(self, ind):
        xml = ('%s<lineItem quantity="%s">\r\n' % (ind, self.quantity));
        xml = ('%s%s  <unitCost currency="USD">%s</unitCost>\r\n' %
               (xml, ind, self.unitCost));
        xml = ('%s%s' % (xml, self.product.toXML(ind + '  ')));
        xml = ('%s%s</lineItem>\r\n' % (xml, ind));
        return xml;

class Order:
    def __init__(self, oid, date, rep, cust):
        self.orderID = oid;
        self.orderDate = date;
        self.salesRep = rep;
        self.customer = cust;
        self.items = [];

    def toXML(self, ind):
        xml = ('%s<order orderID="%s" date="%s" salesRepID="%s">\r\n' %
               (ind, self.orderID, self.orderDate, self.salesRep));
        xml = ('%s%s' % (xml, self.customer.toXML(ind + '  ')));
        for item in self.items:
            xml = ('%s%s' % (xml, item.toXML(ind + '  ')));
        xml = ('%s%s</order>\r\n' % (xml, ind));
        return xml;

class Product:
    def __init__(self, pid, nom):
        self.productID = pid;
        self.name = nom;

    def toXML(self, ind):
        xml = ('%s<product productID="%s">\r\n' % (ind, self.productID));
        xml = ('%s%s  <name>%s</name>\r\n' % (xml, ind, self.name));
```

Example 11-4.

```
        xml = ('%s%s</product>\r\n' % (xml, ind));
        return xml;
```

Each business object defines two basic methods. The first, the constructor, does nothing more than assign values to the object's attributes. The second method, `toXML()`, converts the business object to XML. So far, we have kept all database access separate from our business objects. This is a very critical design element of good database programming.

All of the database access comes in a module method called `executeBatch()`. The purpose of this method is to find out which orders need XML generated and load them from the database into business objects. It then takes those loaded orders and sends the return value of `toXML()` to an XML file. Example 11-5 shows the `executeBatch()` method.

Example 11-5. . Database Access for the XML Generator

```
def executeBatch(conn):
    try:
        cursor = conn.cursor();
        cursor.execute("SELECT ORDER_ID FROM ORDER_EXPORT " +
                       "WHERE LAST_EXPORT <> CURRENT_DATE()");
        orders = cursor.fetchall();
        cursor.close();
    except:
        print "Error retrieving orders.";
        traceback.print_exc();
        conn.close();
        exit(0);

    for row in orders:
        oid = row[0];
        try:
            cursor = conn.cursor();
            cursor.execute("SELECT CUST_ORDER.ORDER_DATE, " +
                           "CUST_ORDER.SALES_REP_ID, " +
                           "CUSTOMER.CUSTOMER_ID, " +
                           "CUSTOMER.NAME, " +
                           "CUSTOMER.ADDRESS1, " +
                           "CUSTOMER.ADDRESS2, " +
                           "CUSTOMER.CITY, " +
                           "CUSTOMER.STATE, " +
                           "CUSTOMER.COUNTRY, " +
                           "CUSTOMER.POSTAL_CODE " +
                           "FROM CUST_ORDER, CUSTOMER " +
                           "WHERE CUST_ORDER.ORDER_ID = %s " +
                           "AND CUST_ORDER.CUSTOMER_ID = CUSTOMER.CUSTOMER_ID",
                           ( oid ) );
            row = cursor.fetchone();
            cursor.close();
            addr = Address(row[4], row[5], row[6], row[7], row[8], row[9]);
```

Example 11-5. . Database Access for the XML Generator

```
            cust = Customer(row[2], row[3], addr);
            order = Order(oid, row[0], row[1], cust);
            cursor = conn.cursor();
            cursor.execute("SELECT LINE_ITEM.PRODUCT_ID, " +
                           "LINE_ITEM.QUANTITY, " +
                           "LINE_ITEM.UNIT_COST, " +
                           "PRODUCT.NAME " +
                           "FROM LINE_ITEM, PRODUCT " +
                           "WHERE LINE_ITEM.ORDER_ID = %s " +
                           "AND LINE_ITEM.PRODUCT_ID = PRODUCT.PRODUCT_ID",
                           oid);
            for row in cursor.fetchall():
                prd = Product(row[0], row[3]);
                order.items.append(LineItem(prd, row[1], row[2]));
        except:
            print "Failed to load order: ", oid;
            traceback.print_exc();
            exit(0);

        try:
            cursor.close();
        except:
            print "Error closing cursor, continuing...";
            traceback.print_exc();

        try:
            fname = ('%d.xml' % oid);
            xmlfile = open(fname, "w");
            xmlfile.write('<?xml version="1.0"?>\r\n\r\n');
            xmlfile.write(order.toXML(''));
            xmlfile.close();
        except:
            print ("Failed to write XML file: %s" % fname);
            traceback.print_exc();

        try:
            cursor = conn.cursor();
            cursor.execute("UPDATE ORDER_EXPORT " +
                           "SET LAST_EXPORT = CURRENT_DATE() " +
                           "WHERE ORDER_ID = %s", ( oid ));
        except:
            print "Failed to update ORDER_EXPORT table, continuing";
            traceback.print_exc();
```

The first try/except block looks in the ORDER_EXPORT table for all orders that have not had XML generated in the last day. If that fails for any reason, the script bails completely.

Each row returned from `fetchall()` represents an order in need of exporting. The script therefore loops through each of the rows ands loads all of the data for the order represented by the row. Inside the `for` loop, the script executes SQL to

pull order and customer data from the ORDER and CUSTOMER tables. With those columns in hand, it can construct Order, Customer, and Address objects for the order, its associated customer, and the customer's address. Because ORDER to LINE_ITEM is a one-to-many relationship, we need a separate query to load the LineItem objects. The next query looks for all of the line items associated with the current order ID and loads business objects for them.

With all of the data loaded into business objects, the script opens a file and writes out their XML conversion to that file. Once the write is successful, the script goes back to the database to note that the XML is now up-to-date with the database. The script then goes to the next order and continues processing until there are no more orders.

The last part missing from the script is the functionality to call executeBatch():

```
if __name__ == '__main__':
    try:
        conn = MySQLdb.connect(host='carthage', user='test', passwd='test',
                               db='Test');
    except:
        print "Error connecting to MySQL:";
        traceback.print_exc();
        exit(0);

    executeBatch(conn);
```

This script as well as the SQL to generate the tables and data behind it are available with the rest of the examples from this book at the O'Reilly Web site.