17

Python Database Application Programming Interface (DB-API)

Objectives

- To understand the relational database model.
- To understand basic database queries using Structured Query Language (SQL).
- To use the methods of the **`MySQLdb`** module to query a database, insert data into a database and update data in a database.

It is a capital mistake to theorize before one has data.
Arthur Conan Doyle

Now go, write it before them in a table, and note it in a book, that it may be for the time to come for ever and ever.
The Holy Bible: The Old Testament

Let's look at the record.
Alfred Emanuel Smith

True art selects and paraphrases, but seldom gives a verbatim translation.
Thomas Bailey Aldrich

Get your facts first, and then you can distort them as much as you please.
Mark Twain

I like two kinds of men: domestic and foreign.
Mae West

Outline

17.1 Introduction

In Chapter 14, File Processing and Serialization, we discussed sequential-access and random-access file processing. Sequential-file processing is appropriate for applications in which most or all of the file's information is to be processed. On the other hand, random-access file processing is appropriate for applications in which only a small portion of a file's data is to be processed. For instance, in transaction processing it is crucial to locate and, possibly, update an individual piece of data quickly. Python provides solid capabilities for both types of file processing.

A *database* is an integrated collection of data. Many companies maintain databases to organize employee information, such as names, addresses and phone numbers. There are many different strategies for organizing data to facilitate easy access and manipulation of the data. A *database management system (DBMS)* provides mechanisms for storing and organizing data in a manner consistent with the database's format. Database management systems allow for the access and storage of data without worrying about the internal representation of databases.

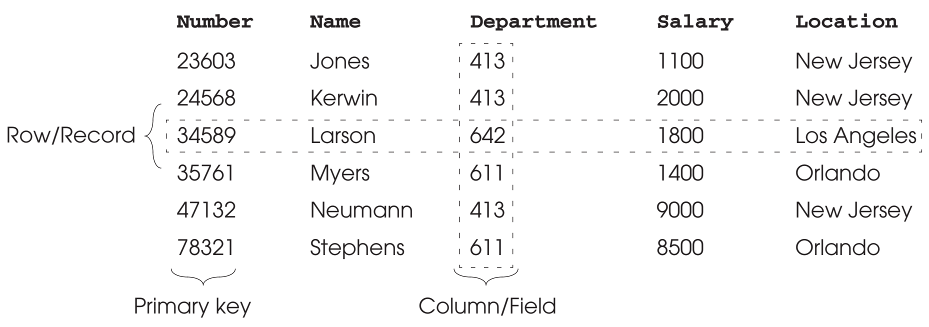
Today's most popular database systems are *relational databases*, which consist of data that correspond to one another. A language called *Structured Query Language* (*SQL*—pronounced as its individual letters or as "sequel") is used almost universally with relational database systems to perform *queries* (i.e., to request information that satisfies given criteria) and to manipulate data. [*Note*: The writing in this chapter assumes that SQL is pronounced as its individual letters. For this reason, we often precede SQL with the article "an" as in "an SQL database" or "an SQL statement."]

Some popular enterprise-level relational database systems include Microsoft SQL Server, Oracle, Sybase, DB2, Informix and MySQL. In this chapter, we present examples using MySQL. Section 17.5 describes MySQL and other innovative databases. All examples in this chapter use MySQL version 3.23.41. [*Note*: The Deitel & Associates, Inc. Web site (**`www.deitel.com`**) provides step-by-step instructions for installing MySQL and helpful MySQL commands for creating, populating and deleting tables.]

A programming language connects to, and interacts with, relational databases via an *interface*—software that facilitates communications between a database management system and a program. Python programmers communicate with databases using modules that conform to the Python *Database Application Programming Interface (DB-API)*. Section 17.6 discusses the DB-API specification.

17.2 Relational Database Model

The *relational database model* is a logical representation of data that allows the relationships between the data to be considered independent of the actual physical structure of the data. A relational database is composed of *tables*. Figure 17.1 illustrates a sample table that might be used in a personnel system. The table *name* is **`Employee`**, and its primary purpose is to illustrate the attributes of an employee and how they are related to a specific employee. Any particular row of the table is called a *record* (or *row*). This table consists of six records. The **`Number`** *field* (or *column*) of each record in the table is used as the *primary key* for referencing data in the table. A primary key is a field (or fields) in a table that contain(s) unique data, which cannot be duplicated in other records. A table's primary key uniquely identifies each record in the table. This guarantees each record can be identified by a unique value. Good examples of primary fields are a social security number, an employee ID and a part number in an inventory system. The records of Fig. 17.1 are *ordered* by primary key. In this case, the records are in increasing order, but they also could be sorted in decreasing order.



`Number`	**`Name`**	**`Department`**	**`Salary`**	**`Location`**
23603	Jones	413	1100	New Jersey
24568	Kerwin	413	2000	New Jersey
34589	Larson	642	1800	Los Angeles
35761	Myers	611	1400	Orlando
47132	Neumann	413	9000	New Jersey
78321	Stephens	611	8500	Orlando

Row/Record
Primary key
Column/Field

Fig. 17.1 Relational database structure of an **`Employee`** table.

Software Engineering Observation 17.1

Tables in a database normally have primary keys.

Each column of the table represents a different field (or column or *attribute*). Records normally are unique (by primary key) within a table, but particular field values may be duplicated between records. For example, three different records in the **Employee** table's **Department** field contain the number 413. The primary key can be composed of more than one column (or field) in the database.

Different users of a database often are interested in different data and different relationships among those data. Some users require only subsets of the table columns. To obtain table subsets, SQL statements specify the data to *select* from a table. SQL enables programmers to define complex queries that select data from a table by providing a complete set of commands, including ***SELECT***. The results of a query are commonly called *result sets* (or *record sets*). For example, an SQL statement might select data from the table in Fig. 17.1 to create a new result set that shows where departments are located. This result set is shown in Fig. 17.2. SQL queries are discussed in Section 17.4.

Department	**Location**
413	New Jersey
611	Orlando
642	Los Angeles

Fig. 17.2 Result set formed by selecting data from a table.

17.3 Relational Database Overview: The Books Database

This section gives an overview of SQL in the context of a sample **Books** database we created for this chapter. Before we discuss SQL, we overview the tables of the **Books** database. We use this to introduce various database concepts, including the use of SQL to obtain useful information from the database and to manipulate the database. We provide the database in the examples directory for this chapter on the CD that accompanies this book. Note that when using MySQL on windows, it is case insensitive (i.e., the **Books** database and the **books** database refer to the same database). However, when using MySQL on Linux, it is case sensitive (i.e., the **Books** database and the **books** database refer to different databases).

The database consists of four tables: **Authors**, **Publishers**, **AuthorISBN** and **Titles**. The **Authors** table (described in Fig. 17.3) consists of three fields (or columns) that maintain each author's unique ID number, first name and last name. Figure 17.4 contains the data from the **Authors** table of the **Books** database.

Field	Description
`AuthorID`	Author's ID number in the database. In the **`Books`** database, this integer field is defined as an *autoincremented* field. For each new record inserted in this table, the database increments the **`AuthorID`** value, to ensure that each record has a unique **`AuthorID`**. This field represents the table's primary key.
`FirstName`	Author's first name (a string).
`LastName`	Author's last name (a string).

Fig. 17.3 **`Authors`** table from **`Books`**.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Ted	Lin
6	Praveen	Sadhu
7	David	McPhie

Fig. 17.4 Data from the **`Authors`** table of **`Books`**.

The **`Publishers`** table (described in Fig. 17.5) consists of two fields representing each publisher's unique ID and name. Figure 17.6 contains the data from the **`Publishers`** table of the **`Books`** database.

Field	Description
`PublisherID`	Publisher's ID number in the database. This autoincremented integer is the table's primary-key field.
`PublisherName`	Name of the publisher (a string).

Fig. 17.5 **`Publishers`** table from **`Books`**.

PublisherID	PublisherName
1	Prentice Hall
2	Prentice Hall PTG

Fig. 17.6 Data from the **`Publishers`** table of **`Books`**.

The **`AuthorISBN`** table (described in Fig. 17.7) consists of two fields that maintain each ISBN number and its corresponding author's ID number. This table helps associate the names of the authors with the titles of their books. Figure 17.8 contains the data from the **`AuthorISBN`** table of the **`Books`** database. ISBN is an abbreviation for "International Standard Book Number", a numbering scheme that publishers worldwide use to give every book a unique identification number. [*Note*: To save space, we have split the contents of this table into two columns, each containing the **`AuthorID`** and **`ISBN`** fields.]

Field	Description
`AuthorID`	Author's ID number, which allows the database to associate each book with a specific author. The integer ID number in this field must also appear in the **`Authors`** table.
`ISBN`	ISBN number for a book (a string).

Fig. 17.7 **`AuthorISBN`** table from **`Books`**.

AuthorID	ISBN	AuthorID	ISBN
1	0130895725	2	0130161438
1	0132261197	2	0130856118
1	0130895717	2	0130125075
1	0135289106	2	0138993947
1	0139163050	2	0130852473
1	013028419x	2	0130829277
1	0130161438	2	0134569555
1	0130856118	2	0130829293
1	0130125075	2	0130284173
1	0138993947	2	0130284181

Fig. 17.8 Data from the **`AuthorISBN`** table of **`Books`** (part 1 of 2).

AuthorID	ISBN	AuthorID	ISBN
1	0130852473	2	0130895601
1	0130829277	3	013028419x
1	0134569555	3	0130161438
1	0130829293	3	0130856118
1	0130284173	3	0134569555
1	0130284181	3	0130829293
1	0130895601	3	0130284173
2	0130895725	3	0130284181
2	0132261197	4	0130895601
2	0130895717	5	0130284173
2	0135289106	6	0130284173
2	0139163050	7	0130284181
2	013028419x		

Fig. 17.8 Data from the **`AuthorISBN`** table of **`Books`** (part 2 of 2).

The **`Titles`** table (described in Fig. 17.9) consists of six fields that maintain general information about each book in the database, including the ISBN number, title, edition number, copyright year, publisher's ID number, name of a file containing an image of the book cover and finally, the price. Figure 17.10 contains the data from the **`Titles`** table.

Field	Description
`ISBN`	ISBN number of the book (a string).
`Title`	Title of the book (a string).
`EditionNumber`	Edition number of the book (an integer).
`Copyright`	Copyright year of the book (an integer).
`PublisherID`	Publisher's ID number (an integer). This value must correspond to an ID number in the **`Publishers`** table.
`ImageFile`	Name of the file containing the book's cover image (a string).
`Price`	Retail price of the book (a real number). [*Note*: The prices shown in this book are for example purposes only.]

Fig. 17.9 **`Titles`** table from **`Books`**.

ISBN	Title	Edition-Number	Copyright	PublisherID	ImageFile	Price
0130895725	C How to Program	3	2001	1	`chtp3.jpg`	69.95
0132261197	C How to Program	2	1994	1	`chtp2.jpg`	49.95
0130895717	C++ How to Program	3	2001	1	`cpphtp3.jpg`	69.95
0135289106	C++ How to Program	2	1998	1	`cpphtp2.jpg`	49.95
0139163050	The Complete C++ Training Course	3	2001	2	`cppctc3.jpg`	109.95
013028419x	e-Business and e-Commerce How to Program	1	2001	1	`ebechtp1.jpg`	69.95
0130161438	Internet and World Wide Web How to Program	1	2000	1	`iw3htp1.jpg`	69.95
0130856118	The Complete Internet and World Wide Web Programming Training Course	1	2000	2	`iw3ctc1.jpg`	109.95
0130125075	Java How to Program (Java 2)	3	2000	1	`jhtp3.jpg`	69.95
0138993947	Java How to Program (Java 1.1)	2	1998	1	`jhtp2.jpg`	49.95
0130852473	The Complete Java 2 Training Course	3	2000	2	`javactc3.jpg`	109.95
0130829277	The Complete Java Training Course (Java 1.1)	2	1998	2	`javactc2.jpg`	99.95

Fig. 17.10 Data from the **`Titles`** table of **`Books`** (part 1 of 2).

ISBN	Title	Edition -Number	Copy-right	Publi-sherID	ImageFile	Price
0134569555	Visual Basic 6 How to Program	1	1999	1	**vbhtp1.jpg**	69.95
0130829293	The Complete Visual Basic 6 Training Course	1	1999	2	**vbctc1.jpg**	109.95
0130284173	XML How to Program	1	2001	1	**xmlhtp1.jpg**	69.95
0130284181	Perl How to Program	1	2001	1	**perlhtp1.jpg**	69.95
0130895601	Advanced Java 2 Platform How to Program	1	2002	1	**advjhtp1.jpg**	69.95

Fig. 17.10 Data from the **Titles** table of **Books** (part 2 of 2).

Figure 17.11 illustrates the relationships between the tables in the **Books** database. The first line in each table is the table's name. The field name in green in each table is that table's primary key. Every record must have a unique value in the primary-key field. This is known as the *Rule of Entity Integrity*.

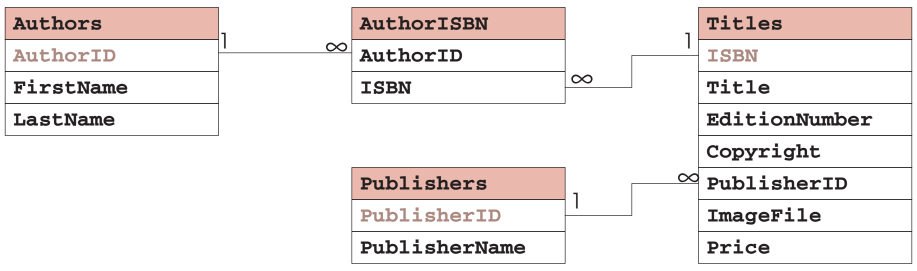


Fig. 17.11 Table relationships in **Books**.

The lines connecting the tables in Fig. 17.11 represent the *relationships* between the tables. Consider the line between the **Publishers** and **Titles** tables. On the **Publishers** end of the line, there is a **1**, and on the **Titles** end, there is an infinity (∞) symbol. This line indicates a *one-to-many relationship* in which every publisher in the **Publishers** table can have an arbitrarily large number of books in the **Titles** table. Note that the relationship line links the **PublisherID** field in the **Publishers** table to the **PublisherID** field in the **Titles** table. The **PublisherID** field in the **Titles** table is a *foreign key*—a field for which every entry has a unique value in another table and where the field in the other table is the primary key for that table (i.e., **PublisherID** in the **Publishers** table). Foreign keys (sometimes called *constraints*) are specified when creating a table. The foreign key helps maintain the *Rule of Referential Integrity*: Every foreign key-field value must appear in another table's primary-key field. Foreign keys enable information from multiple tables to be *joined* together for analysis purposes. There is a one-to-many relationship between a primary key and its corresponding foreign key. This means that a foreign key-field value can appear many times in its own table, but it can only appear once as the primary key of another table. The line between the tables represents the link between the foreign key in one table and the primary key in another table. Notice that table **AuthorISBN** does not have a primary key because both **AuthorID** and **ISBN** are foreign keys.

Common Programming Error 17.1

Not providing a value for a primary-key field in every record breaks the Rule of Entity Integrity and causes the DBMS to report an error.

Common Programming Error 17.2

Providing duplicate values for the primary-key field in multiple records causes the DBMS to report an error.

Common Programming Error 17.3

Providing a foreign-key value that does not appear as a primary-key value in another table breaks the Rule of Referential Integrity and causes the DBMS to report an error.

The line between the **AuthorISBN** and **Authors** tables indicates that for each author in the **Authors** table, there can be an arbitrary number of ISBNs for books written by that author in the **AuthorISBN** table. The **AuthorID** field in the **AuthorISBN** table is a foreign key of the **AuthorID** field (the primary key) of the **Authors** table. Note again that the line between the tables links the foreign key of table **AuthorISBN** to the corresponding primary key in table **Authors**. The **AuthorISBN** table links information in the **Titles** and **Authors** tables.

Finally, the line between the **Titles** and **AuthorISBN** tables illustrates a one-to-many relationship; a title can be written by any number of authors. In fact, the sole purpose of the **AuthorISBN** table is to represent a many-to-many relationship between the **Authors** and **Titles** tables; an author can write any number of books and a book can have any number of authors.

17.4 Structured Query Language (SQL)

This section provides an overview of SQL in the context of a sample database called **Books**. You will be able to use the SQL queries discussed here in the examples later in the chapter.

The SQL keywords of Fig. 17.12 are discussed in the context of complete SQL queries in the next several subsections—other SQL keywords are beyond the scope of this text. [*Note*: For more information on SQL, please refer to the World Wide Web resources in Section 17.10 and the bibliography at the end of this chapter.]

SQL keyword	Description
SELECT	Select (retrieve) fields from one or more tables.
FROM	Tables from which to select fields. Required in every **SELECT**.
WHERE	Criteria for selection that determine the rows to be retrieved.
ORDER BY	Criteria for ordering records.
INSERT INTO	Insert data into a specified table.
UPDATE	Update data in a specified table.
DELETE FROM	Delete data from a specified table.

Fig. 17.12 Some SQL query keywords.

17.4.1 Basic SELECT Query

Let us consider several SQL queries that extract information from database **Books**. A typical SQL query selects information from one or more tables in a database. Such selections are performed by ***SELECT*** queries. The simplest format of a **SELECT** query is

```
SELECT * FROM tableName
```

In this query, the asterisk (*****) indicates that all rows and columns from table *tableName* of the database should be selected. For example, to select the entire contents of the **Authors** table (i.e., all the data in Fig. 17.13), use the query

```
SELECT * FROM Authors
```

To select specific fields from a table, replace the asterisk (*****) with a comma-separated list of field names. For example, to select only the fields **AuthorID** and **LastName** for all rows in the **Authors** table, use the query

```
SELECT AuthorID, LastName FROM Authors
```

This query returns the data in Fig. 17.13.

AuthorID	LastName
1	Deitel
2	Deitel
3	Nieto
4	Santry
5	Lin
6	Sadhu

Fig. 17.13 AuthorID and **LastName** from the **Authors** table (part 1 of 2).

AuthorID	LastName
7	McPhie

Fig. 17.13 AuthorID and LastName from the Authors table (part 2 of 2).

Good Programming Practice 17.1

If a field name contains spaces, the name must be enclosed in quotation marks (`""`) in the query. For example, if the field name is `First Name`, the field name should appear in the query as `"First Name"`.

Good Programming Practice 17.2

For most SQL statements, the asterisk (`*`) should not be used to specify field names to select from a table (or several tables). In general, programmers process result sets by knowing in advance the order of the fields in the result set. For example, selecting `AuthorID` and `LastName` from table `Authors` guarantees that the fields will appear in the result set with `AuthorID` as the first field and `LastName` as the second field.

Software Engineering Observation 17.2

Specifying the field names to select from a table (or several tables) guarantees that the fields are always returned in the specified order, even if the actual order of the fields in the database table(s) changes or if new fields are added to the table(s).

Common Programming Error 17.4

If a program assumes that the fields in a result set are always returned in the same order from an SQL statement that uses the asterisk (`*`) to select fields, the program may process the result set incorrectly. If the field order in the database table(s) changes, the order of the fields in the result set would change accordingly.

17.4.2 WHERE Clause

In most cases, it is necessary to locate records in a database that satisfy certain *selection criteria*. Only records that match the selection criteria are selected. SQL uses the optional ***WHERE** clause* in a **SELECT** query to specify the selection criteria for the query. The simplest format of a **SELECT** query with selection criteria is

```
SELECT fieldName1, fieldName2, … FROM tableName WHERE criteria
```

For example, to select the **Title**, **EditionNumber** and **Copyright** fields from table **Titles** where the **Copyright** is greater than **2000**, use the query

```
SELECT Title, EditionNumber, Copyright
   FROM Titles
   WHERE Copyright > 2000
```

Figure 17.14 shows the results of the preceding query.

Title	EditionNumber	Copyright
C How to Program	3	2001
C++ How to Program	3	2001
The Complete C++ Training Course	3	2001
e-Business and e-Commerce How to Program	1	2001
XML How to Program	1	2001
Perl How to Program	1	2001
Advanced Java 2 Platform How to Program	1	2002

Fig. 17.14 Titles published after 2000 from table **Titles**.

Performance Tip 17.1

Using selection criteria improves performance typically selecting a smaller portion of the database. Working with a portion of the data is more efficient than working with the entire set of data stored in the database.

The **WHERE** clause condition can contain operators **<**, **>**, **<=**, **>=**, **=**, **<>** and **LIKE**. *Operator **LIKE*** is used for *pattern matching* with wildcard characters *percent (**%**)* and *underscore (_)*. Pattern matching allows SQL to search for similar strings that "match a pattern."

A pattern that contains a percent character (**%**) searches for strings that have zero or more characters at the percent character's position in the pattern. For example, the following query locates the records of all the authors whose last names start with the letter **D**:

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   WHERE LastName LIKE 'D%'
```

The preceding query selects the two records shown in Fig. 17.15, because two of the four authors in our database have last names starting with the letter **D** (followed by zero or more characters). The **%** in the **WHERE** clause's **LIKE** pattern indicates that any number of characters can appear after the letter **D** in the **LastName** field. Notice that the pattern string (like all strings in SQL) is surrounded by single-quote characters.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel

Fig. 17.15 Authors whose last names start with **D** from the **Authors** table.

Portability Tip 17.1

See the documentation for your database system to determine if SQL is case sensitive on your system and to determine the syntax for SQL keywords (i.e., should they be all uppercase letters, all lowercase letters or some combination of the two?).

Portability Tip 17.2

Not all database systems support the **`LIKE`** *operator, so be sure to read your system's documentation carefully.*

Portability Tip 17.3

Some databases use the **`*`** *character in place of the* **`%`** *in a* **`LIKE`** *expression.*

Portability Tip 17.4

Some databases allow regulation expression patterns.

Portability Tip 17.5

In some databases, string data is case sensitive.

Good Programming Practice 17.3

By convention, on systems that are not case sensitive, SQL keywords should be all uppercase letters to emphasize them in an SQL statement.

An underscore (_) in the pattern string indicates a single character at that position in the pattern. For example, the following query locates the records of all authors whose last names start with any character (specified with _), followed by the letter **`i`**, followed by any number of additional characters (specified with **`%`**):

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   WHERE LastName LIKE '_i%'
```

The preceding query produces the two records in Fig. 17.16, because two authors in our database have last names in which **`i`** is the second letter.

AuthorID	FirstName	LastName
3	Tem	Nieto
5	Ted	Lin

Fig. 17.16 The authors from the **`Authors`** table whose last names contain **`i`** as the second letter.

Portability Tip 17.6

Some databases use the **`?`** *character in place of the _ in a* **`LIKE`** *expression.*

17.4.3 ORDER BY Clause

The results of a query can be arranged in ascending or descending order using the optional ***ORDER BY*** clause. The simplest form of an **ORDER BY** clause is

```
SELECT fieldName1, fieldName2, … FROM tableName ORDER BY field ASC
SELECT fieldName1, fieldName2, … FROM tableName ORDER BY field DESC
```

where **ASC** specifies ascending order (lowest to highest), **DESC** specifies descending order (highest to lowest) and *field* specifies the field on which the sort is based. Without an **ORDER BY** clause, rows are returned in an unpredictable order.

For example, to obtain the list of authors in ascending order by last name, use the query

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName ASC
```

The default sorting order is ascending, so **ASC** is optional. Figure 17.17 shows the results.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
5	Ted	Lin
7	David	McPhie
3	Tem	Nieto
6	Praveen	Sadhu
4	Sean	Santry

Fig. 17.17 Authors from table **Authors** in ascending order by **LastName**.

To obtain the same list of authors in descending order by last name (Fig. 17.18), use the query

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName DESC
```

AuthorID	FirstName	LastName
4	Sean	Santry
6	Praveen	Sadhu

Fig. 17.18 Authors from table **Authors** in descending order by **LastName** (part 1 of 2).

AuthorID	FirstName	LastName
3	Tem	Nieto
7	David	McPhie
5	Ted	Lin
1	Harvey	Deitel
2	Paul	Deitel

Fig. 17.18 Authors from table **Authors** in descending order by **LastName** (part 2 of 2).

Multiple fields can be used for ordering purposes with an **ORDER BY** clause of the form

```
ORDER BY field1 sortingOrder, field2 sortingOrder, …
```

where *sortingOrder* is either **ASC** or **DESC**. The *sortingOrder* does not have to be identical for each field. The query

```
SELECT AuthorID, FirstName, LastName
   FROM Authors
   ORDER BY LastName, FirstName
```

sorts in ascending order all the authors by last name, then by first name. If any authors have the same last name, their records are returned in sorted order by their first names (Fig. 17.19).

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
5	Ted	Lin
7	David	McPhie
3	Tem	Nieto
6	Praveen	Sadhu
4	Sean	Santry

Fig. 17.19 Authors from table **Authors** in ascending order by **LastName** and by **FirstName**.

The **WHERE** and **ORDER BY** clauses can be combined in one query. For example, the query

```
SELECT ISBN, Title, EditionNumber, Copyright
   FROM Titles
   WHERE Title LIKE '%How to Program'
   ORDER BY Title ASC
```

returns the **ISBN**, **Title**, **EditionNumber** and **Copyright** date of each book in the **Titles** table that has a **Title** ending with "**How to Program**" and sorts them in ascending order by **Title**. The results of the query are shown in Fig. 17.20.

ISBN	Title	Edition-Number	Copyright
0130895601	Advanced Java 2 Platform How to Program	1	2002
0130895725	C How to Program	3	2001
0132261197	C How to Program	2	1994
0130895717	C++ How to Program	3	2001
0135289106	C++ How to Program	2	1997
013028419x	e-Business and e-Commerce How to Program	1	2001
0130161438	Internet and World Wide Web How to Program	1	2000
0130284181	Perl How to Program	1	2001
0134569555	Visual Basic 6 How to Program	1	1999
0130284173	XML How to Program	1	2001

Fig. 17.20 Books from table **Titles** whose titles end with **How to Program** in ascending order by **Title**.

17.4.4 Merging Data from Multiple Tables: Joining

Often it is necessary to merge data from multiple tables into a single view for analysis purposes. This is referred to as *joining* the tables and is accomplished using a comma-separated list of tables in the **FROM** clause of a **SELECT** query. A join merges records from two or more tables by testing for matching values in a field that is common to both tables. The simplest format of a join is

```
SELECT fieldName1, fieldName2, …
   FROM table1, table2
   WHERE table1.fieldName = table2.fieldName
```

The query's **WHERE** clause specifies the fields from each table that should be compared to determine which records will be selected. These fields normally represent the primary key in one table and the corresponding foreign key in the other table. Foreign keys enable information from multiple tables to be joined together and presented to the user.

For example, the following query produces a list of authors and the ISBN numbers for the books that each author wrote:

```
SELECT FirstName, LastName, ISBN
   FROM Authors, AuthorISBN
```

```
WHERE Authors.AuthorID = AuthorISBN.AuthorID
ORDER BY LastName, FirstName
```

The query merges the **FirstName** and **LastName** fields from the **Authors** table and the **ISBN** field from the **AuthorISBN** table and sorts the results in ascending order by **LastName** and **FirstName**. Notice the use of the syntax *tableName*.*fieldName* in the **WHERE** clause of the query. This syntax (called a *fully qualified name*) specifies the fields from each table that should be compared to join the tables. The "*tableName*." syntax is required if the fields have the same name in both tables. Fully qualified names that start with the database name can be used to perform cross-database queries.

Software Engineering Observation 17.3

If an SQL statement uses fields with the same name from multiple tables, the field name must be fully qualified with its table name and a dot operator (.), as in ***Authors.AuthorID****.*

Common Programming Error 17.5

In a query, not providing fully qualified names for fields with the same name from two or more tables is an error.

As always, the **FROM** clause can be followed by an **ORDER BY** clause. Figure 17.21 shows the results of the preceding query. [*Note*: To save space, we split the results of the query into two columns, each containing the **FirstName**, **LastName** and **ISBN** fields.]

FirstName	LastName	ISBN	FirstName	LastName	ISBN
Harvey	Deitel	0130161438	Paul	Deitel	0130125075
Harvey	Deitel	0130852473	Paul	Deitel	0132261197
Harvey	Deitel	0135289106	Paul	Deitel	0134569555
Harvey	Deitel	0130284173	Paul	Deitel	013028419x
Harvey	Deitel	0130856118	Paul	Deitel	0130895601
Harvey	Deitel	0130895725	Paul	Deitel	0138993947
Harvey	Deitel	0130829277	Paul	Deitel	0130895717
Harvey	Deitel	0139163050	Paul	Deitel	0130829293
Harvey	Deitel	0130284181	Paul	Deitel	0130161438
Harvey	Deitel	0130125075	Paul	Deitel	0130852473
Harvey	Deitel	0132261197	Paul	Deitel	0135289106
Harvey	Deitel	0134569555	Ted	Lin	0130284173
Harvey	Deitel	013028419x	David	McPhie	01300284181
Harvey	Deitel	0130895601	Tem	Nieto	0130829293
Harvey	Deitel	0138993947	Tem	Nieto	0130161438
Harvey	Deitel	0130895717	Tem	Nieto	0130284173

Fig. 17.21 Authors and the ISBN numbers for the books they have written in ascending order by **LastName** and **FirstName** (part 1 of 2).

FirstName	LastName	ISBN
Harvey	Deitel	0130829293
Paul	Deitel	0130284173
Paul	Deitel	0130856118
Paul	Deitel	0130895725
Paul	Deitel	0130829277
Paul	Deitel	0139163050
Paul	Deitel	0130284181
Tem	Nieto	0130856118
Tem	Nieto	0130284181
Tem	Nieto	0134569555
Tem	Nieto	013028419x
Praveen	Sadhu	0130284173
Sean	Santry	0130895601

Fig. 17.21 Authors and the ISBN numbers for the books they have written in ascending order by **LastName** and **FirstName** (part 2 of 2).

17.4.5 INSERT INTO Statement

The ***INSERT INTO*** statement inserts a new record into a table. The simplest form of this statement is

```
INSERT INTO tableName ( fieldName1, fieldName2, …, fieldNameN )
   VALUES ( value1, value2, …, valueN )
```

where *tableName* is the table in which to insert the record. The *tableName* is followed by a comma-separated list of field names in parentheses. (This list is not required if the **INSERT INTO** operation specifies a value for every column of the table in the correct order.) The list of field names is followed by the SQL keyword **VALUES** and a comma-separated list of values in parentheses. The values specified here should match the field names specified after the table name in order and type (i.e., if *fieldName1* is supposed to be the **FirstName** field, then *value1* should be a string in single quotes representing the first name). The **INSERT INTO** statement

```
INSERT INTO Authors ( FirstName, LastName )
   VALUES ( 'Sue', 'Smith' )
```

inserts a record into the **Authors** table. The statement indicates that values will be inserted for the **FirstName** and **LastName** fields. The corresponding values to insert are **'Sue'** and **'Smith'**. [*Note*: The SQL statement does not specify an **AuthorID** in this example, because **AuthorID** is an *autoincrement field* in table **Authors** (Fig. 17.3). For every new record added to this table, MySQL assigns a unique **AuthorID** value that is the next value in the auto-increment sequence (i.e., 1, 2, 3, etc.). In this case, MySQL assigns **AuthorID** number 8 to Sue Smith.] Figure 17.22 shows the **Authors** table after the **INSERT INTO** operation.

`AuthorID`	`FirstName`	`LastName`
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Ted	Lin
6	Praveen	Sadhu
7	David	McPhie
8	Sue	Smith

Fig. 17.22 Table **`Authors`** after an **`INSERT INTO`** operation to add a record.

Common Programming Error 17.6

*In MySQL, SQL statements use the single-quote (`'`) character as a delimiter for strings. To specify a string containing a single quote (such as O'Malley) in an SQL statement, the string must have two single quotes in the position where the single-quote character appears in the string (e.g., **`'O''Malley'`**). The first of the two single-quote characters acts as an escape character for the second. Not escaping single-quote characters in a string that is part of an SQL statement is a syntax error.*

17.4.6 `UPDATE` Statement

An ***`UPDATE`*** statement modifies data in a table. The simplest form for an **`UPDATE`** statement is

```
UPDATE tableName
   SET fieldName1 = value1, fieldName2 = value2, …, fieldNameN = valueN
   WHERE criteria
```

where *tableName* is the table in which to update a record (or records). The *tableName* is followed by keyword ***`SET`*** and a comma-separated list of field name/value pairs in the format *fieldName* **`=`** *value*. The **`WHERE`** clause specifies the criteria used to determine which record(s) to update. The **`UPDATE`** statement

```
UPDATE Authors
   SET LastName = 'Jones'
   WHERE LastName = 'Smith' AND FirstName = 'Sue'
```

updates a record in the **`Authors`** table. The statement indicates that the **`LastName`** is assigned the value **`Jones`** for the record in which **`LastName`** equals **`Smith`** and **`FirstName`** equals **`Sue`**. The **`AND`** keyword indicates that all components of the selection criteria must be satisfied. If we know the **`AuthorID`** in advance of the **`UPDATE`** operation (possibly because we searched for the record previously), the **`WHERE`** clause could be simplified as follows:

```
WHERE AuthorID = 8
```

Figure 17.23 shows the **Authors** table after the **UPDATE** operation.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Ted	Lin
6	Praveen	Sadhu
7	David	McPhie
8	Sue	Jones

Fig. 17.23 Table **Authors** after an **UPDATE** operation to change a record.

17.4.7 DELETE FROM Statement

An SQL ***DELETE*** statement removes data from a table. The simplest form for a **DELETE** statement is

```
DELETE FROM tableName WHERE criteria
```

where *tableName* is the table from which to delete a record (or records). The **WHERE** clause specifies the criteria used to determine which record(s) to delete. The **DELETE** statement

```
DELETE FROM Authors
   WHERE LastName = 'Jones' AND FirstName = 'Sue'
```

deletes the record for Sue Jones in the **Authors** table. If we know the **AuthorID** in advance of the **DELETE** operation, the **WHERE** clause could be simplified as follows:

```
WHERE AuthorID = 8
```

Figure 17.24 shows the **Authors** table after the **DELETE** operation.

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto

Fig. 17.24 Table **Authors** after a **DELETE** operation to remove a record (part 1 of 2).

`AuthorID`	`FirstName`	`LastName`
4	Sean	Santry
5	Ted	Lin
6	Praveen	Sadhu
7	David	McPhie

Fig. 17.24 Table **`Authors`** after a **`DELETE`** operation to remove a record (part 2 of 2).

Common Programming Error 17.7

Missing out a **`WHERE`** *clause may result in all records being deleted from the table.*

17.5 Managed Providers

Although the previous sections may appear somewhat trivial in nature, they form the basis of state-of-the-art database systems. The relational database model of Section 17.2 and the Structured Query Language of Section 17.4 are what fuels exciting technologies like MySQL, SQLServer 2000™, and Oracle9i™.

MySQL is an *open-source* DBMS. The term open-source refers to software that can be freely obtained and customized for corporate, educational and personal requirements. [*Note*: Under certain situations, a commercial license is required for MySQL.] MySQL was written in C/C++ and provides an extremely fast *low-tier* user interface to the database. A low-tier interface is one in which there are few levels of interfacing between users and the *shell* (the command interface to an operating system). Thus, MySQL's *Application Programming Interface (API)* allows programmers to build efficient and robust database applications, especially with programming languages that easily interact with C and C++.

SQLServer 2000 is a Microsoft product that is designed for easy integration with Web applications. In a large *distributed computing system*, like the Internet, many different users share resources across different platforms. To that end, Microsoft has added XML and HTTP support in addition to numerous other features.

Oracle9i is another commercial database system in wide-spread use. The focus, as with SQLServer, is on eBusiness and Internet applications. The database supports all types of content, allows users to modify the data through and interface and ensures security. Oracle9i is a *scalable* product because performance does not decrease as the size of the system and its number of users increase.

17.6 Python DB-API Specification

As mentioned earlier, the code examples in this chapter use the MySQL database system; however, the databases supported by Python are not restricted to MySQL. Modules have been written that can interface with most popular databases, thus hiding database details from the programmer. These modules follow the Python *Database Application Program-*

ming Interface (DB-API), a document that specifies common object and method names for manipulating any database.

Specifically, the DB-API describes a ***Connection*** object that accesses the database (connects to the database). A program then uses the **Connection** object to create the ***Cursor*** object, which manipulates and retrieves data. We discuss the methods and attributes of these objects in the context of a working example throughout the remainder of the chapter.

A **Cursor** provides a way to operate or execute queries (such as inserting rows into a table, deleting rows from a table), as well as manipulate data returned from query execution. Three functions are available to fetch row(s) of a query result set—**fetchone**, **fetchmany** and **fetchall**. Function **fetchone** gets the next row in a result set stored in **Cursor**. Function **fetchmany** takes one argument—the number of rows to be fetched, and gets the next set of rows of a result set. Function **fetchall** gets all rows of a result set. On a large database, a **fetchall** would be impractical.

A benefit of the DB-API is that a program does not need to know much about the database to which the program connects. Therefore, a program can use different databases with few modifications in the Python source code. For example, to switch from the MySQL database to another database, a programmer needs to change three or four lines at the most. However, the switch between databases may require modifications to the SQL code (to compensate for case sensitivity, etc.).

17.7 Database Query Example

Figure 17.25 presents a CGI program that performs a simple query on the **Books** database. The query retrieves all information about the authors in the **Authors** table and displays the data in an XHTML table. The program demonstrates connecting to the database, querying the database and displaying the results. The discussion that follows presents the key DB-API aspects of the program.

```
#!c:\Python\python.exe
# Fig. 17.25: fig17_25.py
# Displays contents of the Authors table,
# ordered by a specified field

import MySQLdb
import cgi

def printHeader( title ):
   print """Content-type: text/html

<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE html PUBLIC
   "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "DTD/xhtml1-transitional.dtd">
<html xmlns = "http://www.w3.org/1999/xhtml"
```

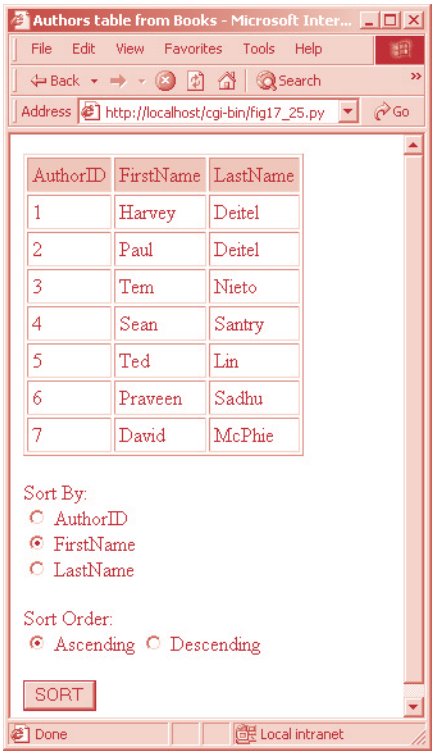
Fig. 17.25 Connecting to and querying a database and displaying the results (part 1 of 4).

```
   xml:lang = "en" lang = "en">
<head><title>%s</title></head>

<body>""" % title

# connect to database and retrieve a cursor
connection = MySQLdb.connect( db = "Books" )
cursor = connection.cursor()

# query field names from Authors table
cursor.execute( "SELECT * FROM Authors" )
allFields = cursor.description

# obtain user query specifications
form = cgi.FieldStorage()

if form.has_key( "sortBy" ):
   sortBy = form[ "sortBy" ].value
else:
   sortBy = allFields[ 0 ][ 0 ]   # first field name

if form.has_key( "sortOrder" ):
   sortOrder = form[ "sortOrder" ].value
else:
   sortOrder = "ASC"

# query all records from Authors table
cursor.execute( "SELECT * FROM Authors ORDER BY %s %s",
   ( sortBy, sortOrder ) )
allRecords = cursor.fetchall()

cursor.close()
connection.close()

# output results in table
printHeader( "Authors table from Books" )
print """\n<table border = "1" cellpadding = "3" >
      <tr bgcolor = "silver" >"""

# create table header
for field in allFields:
   print "<td>%s</td>" % field[ 0 ]

print "</tr>"

# display each record as a row
for author in allRecords:
   print "<tr>"

   for item in author:
      print "<td>%s</td>" % item
```

Fig. 17.25 Connecting to and querying a database and displaying the results (part 2 of 4).

```

   print "</tr>"

print "</table>"

# obtain sorting method from user
print \
   """\n<form method = "post" action = "/cgi-bin/fig17_25.py">
   Sort By:<br />"""

# display sorting options
for field in allFields:
   print """<input type = "radio" name = "sortBy"
      value = "%s" />""" % field[ 0 ]
   print field[ 0 ]
   print "<br />"

print "<br />\nSort Order:<br />
      <input type = "radio" name = "sortOrder"
      value = "ASC" checked = "checked" />
      Ascending"
      <input type = "radio" name = "sortOrder"
      value = "DESC" />"""
      Descending"
      <br /><br />\n<input type = "submit" value = "Sort" />
      </form>\n\n</body>\n</html>"""
```

Fig. 17.25 Connecting to and querying a database and displaying the results (part 3 of 4).

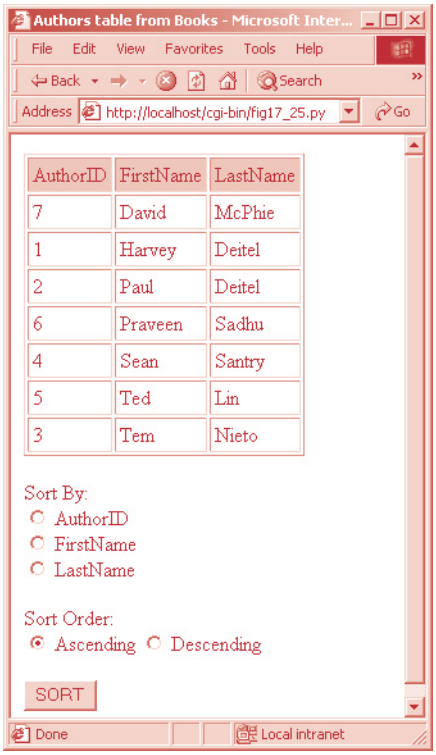

Authors table from Books - Microsoft Inter...
File Edit View Favorites Tools Help
Back Search
Go
SORT
Done
Local intranet

Address http://localhost/cgi-bin/fig17_25.py

AuthorID	FirstName	LastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Ted	Lin
6	Praveen	Sadhu
7	David	McPhie

Sort By:
○ AuthorID
[✓ FirstName]
○ LastName

Sort Order:
[✓ Ascending] ○ Descending


Authors table from Books - Microsoft Inter...
File Edit View Favorites Tools Help
Back Search
Go
SORT
Done
Local intranet

Address http://localhost/cgi-bin/fig17_25.py

AuthorID	FirstName	LastName
7	David	McPhie
1	Harvey	Deitel
2	Paul	Deitel
6	Praveen	Sadhu
4	Sean	Santry
5	Ted	Lin
3	Tem	Nieto

Sort By:
○ AuthorID
○ FirstName
○ LastName

Sort Order:
[✓ Ascending] ○ Descending

Fig. 17.25 Connecting to and querying a database and displaying the results (part 4 of 4).

Line 6 imports *module* ***MySQLdb***, which contains classes and functions for manipulating MySQL databases in Python. Windows users can download **MySQLdb** from **www.cs.fhm.edu/~ifw00065/**, and Linux users can download **MySQLdb** from **sourceforge.net/projects/mysql-python**. For installation instructions, please visit **www.deitel.com**.

Line 23 creates a **Connection** instance called **connection**. This instance manages the connection between the Python program and the database. Function ***MySQLdb.connect*** creates the connection. The function receives the name of the database as the value of keyword argument **db**. If **MySQLdb.connect** fails, the function raises a **MySQLdb** ***OperationalError*** exception.

Line 24 calls **Connection** method ***cursor*** to create a **Cursor** instance for the database. The **Cursor** method ***execute*** takes as an argument a query string to execute against the database (line 27). Our program queries the **Author** table to retrieve the field names and values in the table.

A **Cursor** instance internally stores the results of a database query. The **Cursor** attribute ***description*** contains a tuple of tuples that provides information about the fields obtained by method **execute**. Each record in the return value is represented as a tuple that contains the values of that record's fields (Fig. 17.26). Line 28 assigns the tuple of field name records to variable **allFields**.

```
Python 2.2b1 (#25, Oct 19 2001, 11:44:52) [MSC 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license" for more information.
>>> import MySQLdb
>>> connection = MySQLdb.connect( db = "Books" )
>>> cursor = connection.cursor()
>>> cursor.execute( "SELECT * FROM Authors" )
7L
>>> allFields = cursor.description
>>> print allFields
(('AuthorID', 3, 1, 11, 11, 0, 0), ('FirstName', 253, 7, 20, 20,
0, 0), ('LastName', 253, 6, 30, 30, 0, 0))
>>> allFields[0][0]
'AuthorID'
```

Fig. 17.26 **Cursor** method **fetchall** returns a tuple of tuples. (Copyright © 2001 Python Software Foundation.)

Lines 74–93 create an XHTML form that enables the user to specify how to sort the records of the **Authors** table. Lines 31–41 retrieve and process this form. The records are sorted by the field assigned to variable **sortBy**. By default, the records are sorted by the first field (**LastName**). The user can select a radio button to have the records sorted by another field. Similarly, variable **sortOrder** has either the user-specified value or **"ASC"**.

Lines 44–46 query and retrieve all records from the **Authors** table sorted by variables **sortBy** and **sortOrder**. **Cursor** method ***close*** (line 48) closes the **Cursor** object; line 49 closes the **Connection** object with **Connection** method ***close***. These methods explicitly close the **Cursor** and the **Connection** instances. Although the instances' **close** methods execute when the instances are destroyed at program termination, programmers should explicitly close the instances once they are no longer needed.

Good Programming Practice 17.4

*Explicitly close **Cursor** and **Connection** instances with their respective **close** methods as soon as the program no longer needs the objects.*

The remainder of the program displays the results of the database query in an XHTML table. Lines 57–60 display the **Authors** table's fields using a **for** loop. For each field, the program displays the first entry in that field's tuple. Lines 63–69 display a table row for each record in the **Authors** table using nested **for** loops. The outer **for** loop (line 63) iterates through each record in the table to create a new row. The inner **for** loop (line 66) iterates over each field in the current record and displays each field in a new cell.

17.8 Querying the Books Database

The example of Fig. 17.27 enhances the example of Fig. 17.25 by allowing the user to enter any query into a GUI program. This example introduces database error handling and the

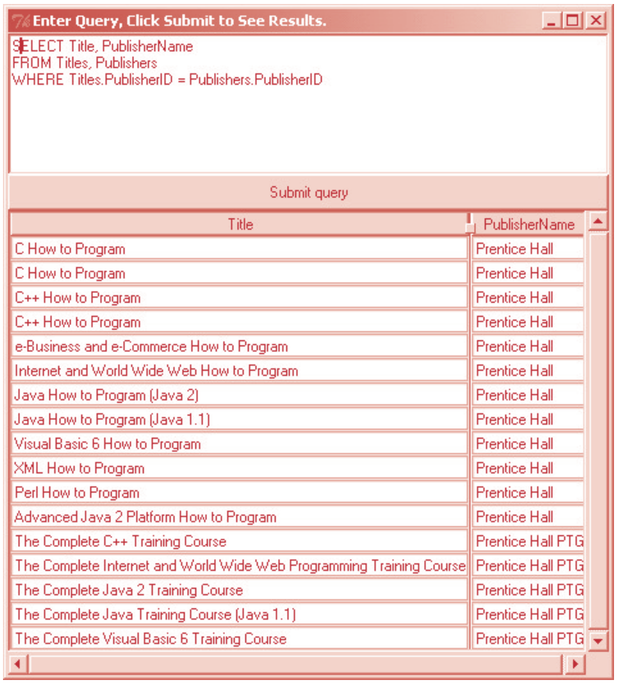
Pmw ScrolledFrame and **PanedWidget** component. Module **Pmw** is introduced in Chapter 11, Graphical User Interface Components: Part 2. The GUI constructor (lines 13–39) creates four GUI elements.

```
1  # Fig. 17.27: fig17_27.py
2  # Displays results returned by a
3  # query on Books database
4
5  import MySQLdb
6  from Tkinter import *
7  from tkMessageBox import *
8  import Pmw
9
10 class QueryWindow( Frame ):
11    """GUI Database Query Frame"""
12
13    def __init__( self ):
14       """QueryWindow Constructor"""
15
16       Frame.__init__( self )
17       Pmw.initialise()
18       self.pack( expand = YES, fill = BOTH )
19       self.master.title( \
20          "Enter Query, Click Submit to See Results." )
21       self.master.geometry( "525x525" )
22
23       # scrolled text pane for query string
24       self.query = Pmw.ScrolledText( self, text_height = 8 )
25       self.query.pack( fill = X )
26
27       # button to submit query
28       self.submit = Button( self, text = "Submit query",
29          command = self.submitQuery )
30       self.submit.pack( fill = X )
31
32       # frame to display query results
33       self.frame = Pmw.ScrolledFrame( self,
34          hscrollmode = "static", vscrollmode = "static" )
35       self.frame.pack( expand = YES, fill = BOTH )
36
37       self.panes = Pmw.PanedWidget( self.frame.interior(),
38          orient = "horizontal" )
39       self.panes.pack( expand = YES, fill = BOTH )
40
41    def submitQuery( self ):
42       """Execute user-entered query agains database"""
43
44       # open connection, retrieve cursor and execute query
45       try:
46          connection = MySQLdb.connect( db = "Books" )
```

Fig. 17.27 GUI application for submitting queries to a database (part 1 of 3). (Copyright © 2001 Python Software Foundation.)

```
            cursor = connection.cursor()
            cursor.execute( self.query.get() )
         except MySQLdb.OperationalError, message:
            errorMessage = "Error %d:\n%s" % \
               ( message[ 0 ], message[ 1 ] )
            showerror( "Error", errorMessage )
            return
         else:   # obtain user-requested information
            data = cursor.fetchall()
            fields = cursor.description   # metadata from query
            cursor.close()
            connection.close()

         # clear results of last query
         self.panes.destroy()
         self.panes = Pmw.PanedWidget( self.frame.interior(),
            orient = "horizontal" )
         self.panes.pack( expand = YES, fill = BOTH )

         # create pane and label for each field
         for item in fields:
            self.panes.add( item[ 0 ] )
            label = Label( self.panes.pane( item[ 0 ] ),
               text = item[ 0 ], relief = RAISED )
            label.pack( fill = X )

         # enter results into panes, using labels
         for entry in data:

            for i in range( len( entry ) ):
               label = Label( self.panes.pane( fields[ i ][ 0 ] ),
                  text = str( entry[ i ] ), anchor = W,
                  relief = GROOVE, bg = "white" )
               label.pack( fill = X )

         self.panes.setnaturalsize()

def main():
   QueryWindow().mainloop()

if __name__ == "__main__":
   main()
```

Fig. 17.27 GUI application for submitting queries to a database (part 2 of 3). (Copyright © 2001 Python Software Foundation.)



```
SELECT Title, PublisherName
FROM Titles, Publishers
WHERE Titles.PublisherID = Publishers.PublisherID
```

Title	PublisherName
C How to Program	Prentice Hall
C How to Program	Prentice Hall
C++ How to Program	Prentice Hall
C++ How to Program	Prentice Hall
e-Business and e-Commerce How to Program	Prentice Hall
Internet and World Wide Web How to Program	Prentice Hall
Java How to Program (Java 2)	Prentice Hall
Java How to Program (Java 1.1)	Prentice Hall
Visual Basic 6 How to Program	Prentice Hall
XML How to Program	Prentice Hall
Perl How to Program	Prentice Hall
Advanced Java 2 Platform How to Program	Prentice Hall
The Complete C++ Training Course	Prentice Hall PTG
The Complete Internet and World Wide Web Programming Training Course	Prentice Hall PTG
The Complete Java 2 Training Course	Prentice Hall PTG
The Complete Java Training Course (Java 1.1)	Prentice Hall PTG
The Complete Visual Basic 6 Training Course	Prentice Hall PTG

Fig. 17.27 GUI application for submitting queries to a database (part 3 of 3). (Copyright © 2001 Python Software Foundation.)

The program display contains two sections. The top section provides a **ScrolledText** component (lines 24–25) for entering a query string. The attribute **text_height** sets the scrolled text area as eight lines high. A **Button** component (lines 28–30) calls the method that executes the query string on the database.

The bottom section contains a ***ScrolledFrame*** component (lines 33–35) for displaying the results of the query. A **ScrolledFrame** component is a scrollable area. The horizontal and vertical scroll bars are displayed because attributes **hscrollmode** and **vscrollmode** are assigned the value **"static"**. The **ScrolledFrame** contains a ***PanedWidget*** component (lines 37–39) for dividing the result records into fields. **Frame** method **interior** specifies that the **PanedWidget** is created within the **ScrolledFrame**. A **PanedWidget** is a subdivided frame that allows the user to change the size of the subdivisions. The **PanedWidget** constructor's ***orient*** argument takes the value ***"horizontal"*** or ***"vertical"***. If the value is **"horizontal"**, the panes are placed left to right in the frame; if the value is **"vertical"**, the panes are placed top to bottom in the frame.

When the user presses the **Submit query** button, method **submitQuery** (lines 41–82) performs the query and displays the results. Lines 45–58 contain a **try/except/else** statement that connects to and queries the database. The **try** structure creates a **Connection** and a **Cursor** instance and uses method **execute** to perform the user-entered query. Function **MySQLdb.connect** fails if the specified database does not exist. The **Cursor** method **execute** fails if the query string contains an SQL syntax error. In either case, **MySQLdb** raises an **OperationalError** exception. Lines 49–53 catch this exception and call **tkMessageBox** function **showerror** with an appropriate error message.

If the user-entered query string successfully executes, the program retrieves the result of the query. The **else** structure (lines 54–58) assigns the queried records to variable **data** and assigns metadata to variable **fields**. *Metadata* are data that describe data. For example, the metadata for a result set may include the field name, field type and if the field can contain null values. The metadata

```
fields = cursor.description
```

contains descriptive information about the result set of the user-entered query (line 56). The **Cursor** attribute ***description*** contains a tuple of tuples that provides information about the fields obtained by method **execute**.

Good Programming Practice 17.5

*The **else** structure (lines 54–58) helps to minimize code in **except** structure. A **except** structure should contain only code that could raise exception.*

PanedWidget method ***destroy*** removes the existing panes to display the query data in new panes (lines 61–64). Lines 67–71 iterate over the field information to display the names of the columns. For each field, method **add** adds a pane to the **PanedWidget**. This method takes a string that identifies the pane. The **Label** constructor adds a label to the pane that contains the name of the field with the **relief** attribute set to **RAISED**. **PanedWidget** method ***pane*** (line 69) identifies the parent of this new label. This method takes the name of a pane and returns a reference to that pane.

Lines 74–80 iterate over each record to create a label that contains the value of each field in the record. Method **pane** specifies the appropriate parent frame for each label. The expression

```
self.panes.pane( fields[ i ][ 0 ] )
```

evaluates to the pane whose name is the field name for the i^{th} value in the record. Once the results have been added to the panes, the **PanedWidget** method ***setnaturalsize*** sets the size of each pane to be large enough to view the largest label in the pane.

17.9 Reading, Inserting and Updating a MySQL Database

The next example (Fig. 17.28) manipulates a simple MySQL **AddressBook** database that contains one table (**addresses**) with 11 columns—**ID** (a unique integer ID number for each person in the address book), **FirstName**, **LastName**, **Address**, **City**, **StateOrProvince**, **PostalCode**, **Country**, **EmailAddress**, **HomePhone** and **FaxNumber**. All fields, except **ID**, are strings. The program provides capabilities for inserting new records, updating existing records and searching for records in the database. [*Note*: The CD that accompanies this book contains an empty **AddressBook** database.]

```
# Fig. 17.28: fig17_28.py
```

Fig. 17.28 Inserting, finding and updating records (part 1 of 6). (Copyright © 2001 Python Software Foundation.)

```
# Inserts into, updates and searches a database

import MySQLdb
from Tkinter import *
from tkMessageBox import *
import Pmw

class AddressBook( Frame ):
   """GUI Database Address Book Frame"""

   def __init__( self ):
      """Address Book constructor"""

      Frame.__init__( self )
      Pmw.initialise()
      self.pack( expand = YES, fill = BOTH )
      self.master.title( "Address Book Database Application" )

      # buttons to execute commands
      self.buttons = Pmw.ButtonBox( self, padx = 0 )
      self.buttons.grid( columnspan = 2 )
      self.buttons.add( "Find", command = self.findAddress )
      self.buttons.add( "Add", command = self.addAddress )
      self.buttons.add( "Update", command = self.updateAddress )
      self.buttons.add( "Clear", command = self.clearContents )
      self.buttons.add( "Help", command = self.help, width = 14 )
      self.buttons.alignbuttons()


      # list of fields in an address record
      fields = [ "ID", "First name", "Last name",
        "Address", "City", "State Province", "Postal Code",
        "Country", "Email Address", "Home phone", "Fax Number" ]

      # dictionary with Entry components for values, keyed by
      # corresponding addresses table field names
      self.entries = {}

      self.IDEntry = StringVar()   # current address id text
      self.IDEntry.set( "" )

      # create entries for each field
      for i in range( len( fields ) ):
         label = Label( self, text = fields[ i ] + ":" )
         label.grid( row = i + 1, column = 0 )
         entry = Entry( self, name = fields[ i ].lower(),
            font = "Courier 12" )
         entry.grid( row = i + 1 , column = 1,
            sticky = W+E+N+S, padx = 5 )

         # user cannot type in ID field
```

Fig. 17.28 Inserting, finding and updating records (part 2 of 6). (Copyright © 2001 Python Software Foundation.)

```
         if fields[ i ] == "ID":
            entry.config( state = DISABLED,
               textvariable = self.IDEntry, bg = "gray" )

         # add entry field to dictionary
         key = fields[ i ].replace( " ", "_" )
         key = key.upper()
         self.entries[ key ] = entry

   def addAddress( self ):
      """Add address record to database"""

      if self.entries[ "LAST_NAME" ].get() != "" and \
         self.entries[ "FIRST_NAME"].get() != "":

         # create INSERT query command
         query = """INSERT INTO addresses (
                 FIRST_NAME, LAST_NAME, ADDRESS, CITY,
                 STATE_PROVINCE, POSTAL_CODE, COUNTRY,
                 EMAIL_ADDRESS, HOME_PHONE, FAX_NUMBER
                 ) VALUES (""" + \
                 "'%s', " * 10 % \
                 ( self.entries[ "FIRST_NAME" ].get(),
                   self.entries[ "LAST_NAME" ].get(),
                   self.entries[ "ADDRESS" ].get(),
                   self.entries[ "CITY" ].get(),
                   self.entries[ "STATE_PROVINCE" ].get(),
                   self.entries[ "POSTAL_CODE" ].get(),
                   self.entries[ "COUNTRY" ].get(),
                   self.entries[ "EMAIL_ADDRESS" ].get(),
                   self.entries[ "HOME_PHONE" ].get(),
                   self.entries[ "FAX_NUMBER" ].get() )
         query = query[ :-2 ] + ")"

         # open connection, retrieve cursor and execute query
         try:
            connection = MySQLdb.connect( db = "AddressBook" )
            cursor = connection.cursor()
            cursor.execute( query )
         except MySQLdb.OperationalError, message:
            errorMessage = "Error %d:\n%s" % \
               ( message[ 0 ], message[ 1 ] )
            showerror( "Error", errorMessage )
         else:
            cursor.close()
            connection.close()
            self.clearContents()

      else:   # user has not filled out first/last name fields
         showwarning( "Missing fields", "Please enter name" )

```

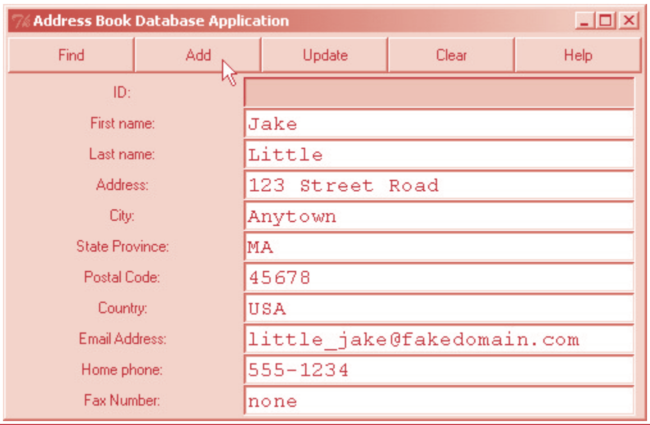
Fig. 17.28 Inserting, finding and updating records (part 3 of 6). (Copyright © 2001 Python Software Foundation.)

```
   def findAddress( self ):
      """Query database for address record and display results"""

      if self.entries[ "LAST_NAME" ].get() != "":

         # create SELECT query
         query = "SELECT * FROM addresses " + \
                 "WHERE LAST_NAME = '" + \
                 self.entries[ "LAST_NAME" ].get() + "'"

         # open connection, retrieve cursor and execute query
         try:
            connection = MySQLdb.connect( db = "AddressBook" )
            cursor = connection.cursor()
            cursor.execute( query )
         except MySQLdb.OperationalError, message:
            errorMessage = "Error %d:\n%s" % \
               ( message[ 0 ], message[ 1 ] )
            showerror( "Error", errorMessage )
            self.clearContents()
         else:   # process results
            results = cursor.fetchall()
            fields = cursor.description

            if not results:   # no results for this person
               showinfo( "Not found", "Nonexistent record" )
            else:             # display information in GUI
               self.clearContents()

               # display results
               for i in range( len( fields ) ):

                  if fields[ i ][ 0 ] == "ID":
                     self.IDEntry.set( str( results[ 0 ][ i ] ) )
                  else:
                     self.entries[ fields[ i ][ 0 ] ].insert(
                        INSERT, str( results[ 0 ][ i ] ) )

            cursor.close()
            connection.close()

      else:   # user did not enter last name
         showwarning( "Missing fields", "Please enter last name" )

   def updateAddress( self ):
      """Update address record in database"""

      if self.entries[ "ID" ].get():

         # create UPDATE query command
         entryItems= self.entries.items()
```

Fig. 17.28 Inserting, finding and updating records (part 4 of 6). (Copyright © 2001 Python Software Foundation.)

```
            query = "UPDATE addresses SET"

            for key, value in entryItems:

               if key != "ID":
                  query += " %s='%s'," % ( key, value.get() )

            query = query[ :-1 ] + " WHERE ID=" + self.IDEntry.get()

            # open connection, retrieve cursor and execute query
            try:
               connection = MySQLdb.connect( db = "AddressBook" )
               cursor = connection.cursor()
               cursor.execute( query )
            except MySQLdb.OperationalError, message:
               errorMessage = "Error %d:\n%s" % \
                  ( message[ 0 ], message[ 1 ] )
               showerror( "Error", errorMessage )
               self.clearContents()
            else:
               showinfo( "database updated", "Database Updated." )
               cursor.close()
               connection.close()

         else:   # user has not specified ID
            showwarning( "No ID specified", """
               You may only update an existing record.
               Use Find to locate the record,
               then modify the information and press Update.""" )

      def clearContents( self ):
         """Clear GUI panel"""

         for entry in self.entries.values():
            entry.delete( 0, END )

         self.IDEntry.set( "" )

      def help( self ):
         "Display help message to user"

         showinfo( "Help", """Click Find to locate a record.
            Click Add to insert a new record.
            Click Update to update the information in a record.
            Click Clear to empty the Entry fields.\n""" )

   def main():
      AddressBook().mainloop()

   if __name__ == "__main__":
      main()
```

Fig. 17.28 Inserting, finding and updating records (part 5 of 6). (Copyright © 2001 Python Software Foundation.)



ID:	
First name:	Jake
Last name:	Little
Address:	123 Street Road
City:	Anytown
State Province:	MA
Postal Code:	45678
Country:	USA
Email Address:	little_jake@fakedomain.com
Home phone:	555-1234
Fax Number:	none

Fig. 17.28 Inserting, finding and updating records (part 6 of 6). (Copyright © 2001 Python Software Foundation.)

Class **`AddressBook`** uses **`Button`** and **`Entry`** components to retrieve and display address information. The constructor creates a list of fields for one address book entry (lines 32–34). Line 38 initializes dictionary data member **`entries`** to hold references to **`Entry`** components. A **`for`** loop then iterates over the length of this list to create an **`Entry`** component for each field (lines 47–48). The loop also adds a reference to the **`Entry`** component to data member entries. Lines 58–60 create a key name for each entry, based on that entry's field name.

Method **`addRecord`** (lines 62–102) adds a new record to the **`AddressBook`** database in response to the **Add** button in the GUI. The method first ensures that the user has entered values for the first and last name fields (lines 65–66). If the user enters values for these fields, the query string inserts a record into the database (lines 69–85). Otherwise, **`tkMessageBox`** function **`showwarning`** reminds the user to enter the information (lines 101–102). Line 74 includes ten string escape sequences whose values are replaced by the values contained in lines 75–84. Line 85 closes the values parentheses in the SQL statement.

Lines 88–99 contain a **`try/except/else`** structure that connects to and updates the database (i.e., inserts the new record in the database). Method **`clearContents`** clears the contents of the GUI (line 99). If an error occurs, **`tkMessageBox`** function **`showerror`** displays the error.

Method **`findAddress`** (lines 104–146) queries the **`AddressBook`** database for a specific record in response to the **Find** button in the GUI. Line 107 tests whether the last name text field contains data. If the entry is empty, the program displays an error. If the user has entered data in the last name text field, a **`SELECT`** SQL statement searches the database for the user-specified last name. We used asterisk (*) in the **`SELECT`** statement because line 126 uses metadata to get field names. Lines 115–131 contain a **`try/except/else`** structure that connects to and queries the database. If these operations succeed, the program retrieves the results from the database (lines 125–126). A message informs the user if the query does not yield results. If the query does yield results, lines 134–140 display the results in the GUI. Each field value is inserted in the appropriate **`Entry`** component. The record's ID must be converted to a string before it can be displayed.

Method **`updateAddress`** (lines 148–183) updates an existing database record. The program displays a message if the user attempts to perform an update operation on a non-existent record. Line 151 tests whether the **`id`** for the current record is valid. Lines 155–162 create the SQL **`UPDATE`** statement. Lines 165–177 connect to and query (i.e., update) the database.

Method **`clearContents`** (lines 185–191) clears the text fields in response to the **Clear** button in the GUI. Method **`help`** (lines 193–199) calls a **`tkMessageBox`** function to display instructions about how to use the program.

17.10 Internet and World Wide Web Resources

This section presents several Internet and World Wide Web resources related to database programming.

`www.mysql.com`
This site offers the free MySQL database for download, the most current documentation and information about open-source licensing.

`ww3.one.net/~jhoffman/sqltut.html`
The *Introduction to SQL* has a tutorial, links to sites with more information about the language and examples.

`www.python.org/topics/databases`
This **`python.org`** page has links to modules like **`MySQLdb`**, documentation, a list of useful books about database programming and the DB-API specification.

`www.chordate.com/gadfly.html`
Gadfly is a free relational database written completely in Python. From this home page, visitors can download the database and view its documentation.

SUMMARY

- A database is an integrated collection of data.
- A database management system (DBMS) provides mechanisms for storing and organizing data in a manner consistent with the database's format. Database management systems allow for the access and storage of data without worrying about the internal representation of databases.
- Today's most popular database systems are relational databases.
- A language called Structured Query Language (SQL—pronounced as its individual letters or as "sequel") is used almost universally with relational database systems to perform queries (i.e., to request information that satisfies given criteria) and to manipulate data.
- A programming language connects to, and interacts with, relational databases via an interface—software that facilitates communications between a database management system and a program.
- Python programmers communicate with databases using modules that conform to the Python Database Application Programming Interface (DB-API).
- The relational database model is a logical representation of data that allows the relationships between the data to be considered independent of the actual physical structure of the data.
- A relational database is composed of tables. Any particular row of the table is called a record (or row).

- A primary key is a field (or fields) in a table that contain(s) unique data, which cannot be duplicated in other records. This guarantees each record can be identified by a unique value.
- A foreign key is a field in a table for which every entry has a unique value in another table and where the field in the other table is the primary key for that table. The foreign key helps maintain the Rule of Referential Integrity—every value in a foreign-key field must appear in another table's primary-key field. Foreign keys enable information from multiple tables to be joined together and presented to the user.
- Each column of the table represents a different field (or column or attribute). Records normally are unique (by primary key) within a table, but particular field values may be duplicated between records.
- SQL enables programmers to define complex queries that select data from a table by providing a complete set of commands.
- The results of a query commonly are called result sets (or record sets).
- A typical SQL query selects information from one or more tables in a database. Such selections are performed by **SELECT** queries. The simplest format of a **SELECT** query is

  SELECT * FROM *tableName*

- An asterisk (*****) indicates that all rows and columns from table *tableName* of the database should be selected.
- To select specific fields from a table, replace the asterisk (*****) with a comma-separated list of field names.
- In most cases, it is necessary to locate records in a database that satisfy certain selection criteria. Only records that match the selection criteria are selected. SQL uses the optional **WHERE** clause in a **SELECT** query to specify the selection criteria for the query. The simplest format of a **SELECT** query with selection criteria is

  SELECT *fieldName1* **FROM** *tableName* **WHERE** *criteria*

- The **WHERE** clause condition can contain operators **<**, **>**, **<=**, **>=**, **=**, **<>** and **LIKE**.
- Operator **LIKE** is used for pattern matching with wildcard characters percent (**%**) and underscore (_). Pattern matching allows SQL to search for similar strings that "match a pattern."
- A pattern that contains a percent character (**%**) searches for strings that have zero or more characters at the percent character's position in the pattern.
- An underscore (_) in the pattern string indicates a single character at that position in the pattern.
- The results of a query can be arranged in ascending or descending order using the optional **ORDER BY** clause. The simplest form of an **ORDER BY** clause is

```
SELECT * FROM tableName ORDER BY field ASC
SELECT * FROM tableName ORDER BY field DESC
```

  where **ASC** specifies ascending order (lowest to highest), **DESC** specifies descending order (highest to lowest) and field specifies the field on which the sort is based.
- Multiple fields can be used for ordering purposes with an **ORDER BY** clause of the form

  ORDER BY *field1* **sortingOrder,** *field2* **sortingOrder, …**

  where **sortingOrder** is either **ASC** or **DESC**. Note that the **sortingOrder** does not have to be identical for each field.

- The **WHERE** and **ORDER BY** clauses can be combined in one query.
- A join merges records from two or more tables by testing for matching values in a field that is common to both tables. The simplest format of a join is

```
SELECT fieldName1, fieldName2, …
 FROM table1, table2
   WHERE table1.fieldName = table2.fieldName
```

- A fully qualified name specifies the fields from each table that should be compared to join the tables. The "*tableName*." syntax is required if the fields have the same name in both tables. The same syntax can be used in a query to distinguish fields in different tables that happen to have the same name. Fully qualified names that start with the database name can be used to perform cross-database queries.
- The **INSERT INTO** statement inserts a new record in a table. The simplest form of this statement is

```
INSERT INTO tableName ( fieldName1, …, fieldNameN )
 VALUES ( value1,…, valueN )
```

  where *tableName* is the table in which to insert the record. The *tableName* is followed by a comma-separated list of field names in parentheses. (This list is not required if the **INSERT INTO** operation specifies a value for every column of the table in the correct order.) The list of field names is followed by the SQL keyword **VALUES** and a comma-separated list of values in parentheses. The values specified here should match the field names specified after the table name in order and type (i.e., if *fieldName1* is supposed to be the **FirstName** field, then *value1* should be a string in single quotes representing the first name).
- An **UPDATE** statement modifies data in a table. The simplest form for an **UPDATE** statement is

```
UPDATE tableName
 SET fieldName1 = value1, …, fieldNameN = valueN
   WHERE criteria
```

  where *tableName* is the table in which to update a record (or records). The *tableName* is followed by keyword **SET** and a comma-separated list of field name/value pairs in the format *fieldName* **=** *value*. The **WHERE** clause specifies the criteria used to determine which record(s) to update.
- An SQL **DELETE** statement removes data from a table. The simplest form for a **DELETE** statement is

```
DELETE FROM tableName WHERE criteria
```

  where *tableName* is the table from which to delete a record (or records). The **WHERE** clause specifies the criteria used to determine which record(s) to delete.
- MySQL is an open-source DBMS. This means that anyone can download and modify the software if necessary. MySQL was written in C/C++ and provides an extremely fast low-tier user interface to the database.
- A low-tier interface is one in which there are fewer levels of interfacing between users and the shell (the command interface to an operating system).
- MySQL's Application Programming Interface (API) allows for fast, powerful database applications to be built, especially with programming languages that easily interact with C and C++.

- Modules have been written that can interface with most popular databases, hiding database details from the programmer. These modules follow the Python Database Application Programming Interface (DB-API), a document that specifies common object and method names for manipulating any database.
- The DB-API describes a **`Connection`** object that programs create to connect to a database.
- A program can use a **`Connection`** object to create a **`Cursor`** object, which the program uses to execute queries against the database.
- The major benefit of the DB-API is that a program does not need to know much about the database to which the program connects. Therefore, the programmer can change the database a program uses without changing vast amounts of Python code. However, changing the DB often requires changes in the SQL code.
- Module **`MySQLdb`** contains classes and functions for manipulating MySQL databases in Python.
- Function **`MySQLdb.connect`** creates the connection. The function receives the name of the database as the value of keyword argument **`db`**. If **`MySQLdb.connect`** fails, the function raises an **`OperationalError`** exception.
- The **`Cursor`** method **`execute`** takes as an argument a query string to execute against the database.
- A **`Cursor`** instance internally stores the results of a database query.
- The **`Cursor`** method **`fetchall`** returns a tuple of records that matched the query. Each record is represented as a tuple that contains the values of that records field.
- The **`Cursor`** method **`close`** closes the **`Cursor`** instance.
- The **`Connection`** method **`close`** closes the **`Connection`** instance.
- A **`PanedWidget`** is a subdivided frame that allows the user to change the size of the subdivisions. The **`PanedWidget`** constructor's **`orient`** argument takes the value **`"horizontal"`** or **`"vertical"`**. If the value is **`"horizontal"`**, the panes are placed left to right in the frame; if the value is **`"vertical"`**, the panes are placed top to bottom in the frame.
- Metadata are data that describe other data. The **`Cursor`** attribute **`description`** contains a tuple of tuples that provides information about the fields of the data obtained by function **`execute`**. The cursor and connection are closed.
- The **`PanedWidget`** method **`pane`** takes the name of a pane and returns a reference to that pane.
- The **`PanedWidget`** method **`setnaturalsize`** sets the size of each pane to be large enough to view the largest label in the pane.

TERMINOLOGY

`AND`
`ASC`
asterisk (**`*`**)
`close` method
column
`Connection` object
`Cursor` object
data attribute
database
database management system (DBMS)
database table
`DELETE`

DESC
escape character
execute method
fetchall method
field
foreign key
FROM
fully qualified name
INSERT INTO
interior method
joining tables
LIKE
MySQL
MySQLdb module
open source
ORDER BY
PanedWidget
pattern matching
percent (**%**) SQL wildcard character
primary key
Python Database Application Programming Interface (DB-API)
query
record
record set
relational database
result set
row
Rule of Referential Integrity
scalability
ScrolledFrame component
SELECT
selection criteria
SET
shell
Structured Query Language (**SQL**)
table
underscore (_) wildcard character
UPDATE
VALUES
WHERE

SELF-REVIEW EXERCISES

17.1 Fill in the blanks in each of the following statements:

a) The most popular database query language is __________.
b) A relational database is composed of __________.
c) A table in a database consists of __________ and __________.
d) The __________ uniquely identifies each record in a table.
e) SQL provides a complete set of commands (including **SELECT**) that enable programmers to define complex __________.

f) SQL keyword __________ is followed by the selection criteria that specify the records to select in a query.
g) SQL keyword __________ specifies the order in which records are sorted in a query.
h) A __________ specifies the fields from multiple tables table that should be compared to join the tables.
i) A __________ is an integrated collection of data which is centrally controlled.
j) A __________ is a field in a table for which every entry has a unique value in another table and where the field in the other table is the primary key for that table.

17.2 State whether the following are *true* or *false*. If *false*, explain why.
a) **`DELETE`** is not a valid SQL keyword.
b) Tables in a database must have a primary key.
c) Python programmers communicate with databases using modules that conform to the DB-API.
d) **`UPDATE`** is a valid SQL keyword.
e) The **`WHERE`** clause condition can not contain operator **`<>`**.
f) Not all database systems support the **`LIKE`** operator.
g) The **`INSERT INTO`** statement inserts a new record in a table.
h) **`MySQLdb.connect`** is used to create a connection to database.
i) A **`Cursor`** object can execute queries in a database.
j) Once created, a connection with database can not be closed.

ANSWERS TO SELF-REVIEW EXERCISES

17.1 a) SQL. b) tables. c) rows, columns. d) primary key. e) queries. f) **`WHERE`**. g) **`ORDER BY`**. h) fully qualified name. i) database. j) foreign key.

17.2 a) False. **`DELETE`** is a valid SQL keyword—it is the function used to delete records. b) False. Tables in a database normally have primary keys. c) True. d) True. e) False. The **`WHERE`** clause can contain operator **`<>`** (not equals). f) True. g) True. h) True. i) True. j) False. **`Connection.close`** can close the connection.

EXERCISES

17.3 Write SQL queries for the **`Books`** database (discussed in Section 17.3) that perform each of the following tasks:
a) Select all authors from the **`Authors`** table.
b) Select all publishers from the **`Publishers`** table.
c) Select a specific author and list all books for that author. Include the title, copyright year and ISBN number. Order the information alphabetically by title.
d) Select a specific publisher and list all books published by that publisher. Include the title, copyright year and ISBN number. Order the information alphabetically by title.

17.4 Write SQL queries for the **`Books`** database (discussed in Section 17.3) that perform each of the following tasks:
a) Add a new author to the **`Authors`** table.
b) Add a new title for an author (remember that the book must have an entry in the **`AuthorISBN`** table). Be sure to specify the publisher of the title.
c) Add a new publisher.

17.5 Modify Fig. 17.25 so that the user can read different tables in the books database.

17.6 Create a MySQL database that contains information about students in a university. Possible fields might include date of birth, major, current grade point average, credits earned, etc. Write a Python program to manage the database. Include the following functionality: sort all students according to GPA (descending), create a display of all students in one particular major and remove all records from the database where the student has the required amount of credits to graduate.

17.7 Modify the **`FIND`** capability in Fig. 17.28 to allow the user to scroll through the results of the query in case there is more than one person with the specified last name in the Address Book. Provide an appropriate GUI.

17.8 Modify the solution from Exercise 17.7 so that the program checks whether a record already exists in the database before adding it.

BIBLIOGRAPHY

(Bl88) Blaha, M. R.; W. J. Premerlani, and J. E. Rumbaugh, "Relational Database Design Using an Object-Oriented Methodology," *Communications of the ACM*, Vol. 31, No. 4, April 1988, pp. 414–427.

(Co70) Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, June 1970.

(Co72) Codd, E. F., "Further Normalization of the Data Base Relational Model," in *Courant Computer Science Symposia*, Vol. 6, *Data Base Systems*. Upper Saddle River, N.J.: Prentice Hall, 1972.

(Co88) Codd, E. F., "Fatal Flaws in SQL," *Datamation*, Vol. 34, No. 16, August 15, 1988, pp. 45–48.

(De90) Deitel, H. M., *Operating Systems, Second Edition*. Reading, MA: Addison Wesley Pubishing, 1990.

(Da81) Date, C. J., *An Introduction to Database Systems*. Reading, MA: Addison Wesley Pubishing, 1981.

(Re88) Relational Technology, *INGRES Overview*. Alameda, CA: Relational Technology, 1988.

(St81) Stonebraker, M., "Operating System Support for Database Management," *Communications of the ACM*, Vol. 24, No. 7, July 1981, pp. 412–418.

(Wi88) Winston, A., "A Distributed Database Primer," *UNIX World*, April 1988, pp. 54–63.