In [9]:
```
#Creation
import sqlite3
from contextlib import closing
conn=sqlite3.connect('clg.db')
if conn:
    print('database is created')
    with closing(conn.cursor()) as cur:
        sql='''create table AINDS(
        sid int primary key not null,
        sname text not null,
        splace text not null);'''
        cur.execute(sql)
else:
    print('problem in data base creation')
```

```
database is created
```

In [10]:
```
#insertion
import sqlite3
conn = sqlite3.connect('clg.db')
cursor=conn.cursor()
cursor.execute("INSERT INTO AINDS(sid,sname,splace)VALUES(3016,'Jayannth','Nandalur');"
print('inserted into database successfully')
conn.commit()
conn.close()
```

```
inserted into database successfully
```

In [11]:
```
#Multiple Insertion
import sqlite3
conn = sqlite3.connect('clg.db')
cur=conn.cursor()
while(True):
    stop=input("do u want to insert more values??")
    if (stop=='y'):
        sid=int(input("enter student id"))
        sname=input("enter student name")
        splace=input("enter student place")
        sql='''INSERT INTO AINDS VALUES(?,?,?);'''
        cur.execute(sql,(sid,sname,splace));
        conn.commit()
    else:
        break
    print("inserted into database successfully")
```

```
do u want to insert more values??y
enter student id3001
enter student nameabc
enter student placerjpt
inserted into database successfully
do u want to insert more values??y
enter student id3002
enter student namexyz
enter student placepdtr
inserted into database successfully
do u want to insert more values??n
```

In [13]:
```
#Updatation
import sqlite3
conn = sqlite3.connect('clg.db')
cur=conn.cursor()
if conn:
    sid=int(input("enter student id"))
    splace=input("enter student place")
    print("connected to the data base")
    sql='''UPDATE AINDS set splace=? where sid=?;'''
    cur.execute(sql,(splace,sid))
    print("sid\t sname\t splace\t\n")
    for row in cur.execute("SELECT * from ainds"):
        print(row[0],'\t',row[1],'\t',row[2],'\n')
    conn.commit()
else:
    print("problem in connection")
```

```
enter student id3002
enter student placeamerica
connected to the data base
sid      sname   splace

3016     Jayannth        Nandalur

3001     abc     kadapa

3002     xyz     america
```

In [3]:
```
#Deletion
import sqlite3
conn = sqlite3.connect('clg.db')
cur=conn.cursor()
if conn:
    sid=int(input("enter student id"))
    print("connected to the data base")
    sql='''DELETE from AINDS where sid=?;'''
    cur.execute(sql,(sid,))
    print("sid\t sname\t splace\t\n")
    for row in cur.execute("SELECT * from ainds"):
        print(row[0],'\t',row[1],'\t',row[2],'\n')
    conn.commit()
else:
    print("problem in connection")
```

```
enter student id3016
connected to the data base
sid      sname   splace

3001     abc     kadapa

3002     xyz     america
```

objects.py

```
class Movie:
    def __init__(self,id=0,name=None,year=0,minutes=0,category=None):
        self.id=id
        self.name=name
        self.year=year
        self.minutes=minutes
        self.category=category
class Category:
    def __init__(self,id=0,name=None):
        self.id=id
        self.name=name
```

db.py:

```
import sqlite3
from contextlib import closing
from objects import Category
from objects import Movie
conn=None
def connect():
    global conn
    db_file="D:/AJP/chinook/mydb.db"
    conn=sqlite3.connect(db_file)
    conn.row_factory=sqlite3.Row

def close():
    if conn:
        conn.close()

def make_category(row):
    return Category(row['categoryID'],row['categoryName'])

def make_movie(row):
    return
Movie(row['movieID'],row['name'],row['year'],row['minutes'],make_category(row))

def get_categories():
    query='''select categoryID,name as categoryName from Category'''
    with closing(conn.cursor()) as c:
        c.execute(query)
        results=c.fetchall()
    categories=[]
    for row in results:
        categories.append(make_category(row))
    return categories
```

```
def get_category(category_id):
    query='''select categoryID,name as categoryName from Category where
categoryID=?'''
    with closing(conn.cursor()) as c:
        c.execute(query,(category_id,))
        row=c.fetchone()
    category=make_category(row)
    return category

def get_movies_by_category(category_id):
    query='''select movieID,Movie.name,year,minutes,Movie.categoryID as
categoryID,Category.name as categoryName
                from Movie join Category on Movie.categoryID=Category.categoryID
where Movie.categoryID=?'''
    with closing(conn.cursor()) as c:
        c.execute(query,(category_id,))
        results=c.fetchall()
    movies=[]
    for row in results:
        movies.append(make_movie(row))
    return movies

def get_movies_by_year(year):
    query='''select movieID,Movie.name,year,minutes,Movie.categoryID as
categoryID,Category.name as categoryName
                from Movie JOIN Category ON Movie.categoryID=Category.categoryID
                where year=?'''
    with closing(conn.cursor()) as c:
        c.execute(query,(year,))
        results=c.fetchall()
    movies=[]
```

```
    for row in results:
        movies.append(make_movie(row))
    return movies

def add_movie(movie):
    sql='''insert into Movie(categoryID,name,year,minutes) values(?,?,?,?)'''
    with closing(conn.cursor()) as c:
        c.execute(sql,(movie.category.id,movie.name,movie.year,movie.minutes))
        conn.commit()

def delete_movie(movie_id):
    sql='''delete from Movie where movieID=?'''
    with closing(conn.cursor()) as c:
        c.execute(sql,(movie_id,))
        conn.commit()
```

user.py:

```
import db
from objects import Movie

def display_title():
    print("The Movie List Program")
    print()
    display_menu()

def display_menu():
    print("COMMAND MENU")
    print("cat - view movies by category")
    print("year - view movies by year")
    print("add - Add a movie")
    print("del - Delete a movie")
    print("exit - Exit program")
    print()

def display_categories():
    print("CATEGORIES")
    categories=db.get_categories()
    for category in categories:
        print(str(category.id)+"."+category.name)
    print()

def display_movies(movies,title_term):
    print("MOVIES-"+title_term)
    line_format="{:3s} {:37s} {:6s} {:5s} {:10s}"
    print(line_format.format("ID","Name","year","Mins","Category"))
    print("-"*64)
    for movie in movies:
```

```
print(line_format.format(str(movie.id),movie.name,str(movie.year),str(movie.min
utes),movie.category.name))
    print()

def display_movies_by_category():
    category_id=int(input("Category ID:"))
    category=db.get_category(category_id);
    if category==None:
        print("There is no Category with that id.\n")
    else:
        print()
        movies=db.get_movies_by_category(category_id)
        display_movies(movies,category.name.upper())



def display_movies_by_year():
    year=int(input("year:"))
    print()
    movies=db.get_movies_by_year(year)
    display_movies(movies,str(year))

def add_movie():
    name=input("Name:")
    year=int(input("Year:"))
    minutes=int(input("Minutes:"))
    category_id=int(input("Category ID;"))
    category=db.get_category(category_id)
    if category==None:
        print("There is no category id.Movie not added")
    else:
        movie=Movie(name=name,year=year,minutes=minutes,category=category)
```

```
        db.add_movie(movie)
        print(name+" was added to data base\n")

def delete_movie():
    movie_id=int(input("Movie ID:"))
    db.delete_movie(movie_id)
    print("Movie ID "+str(movie_id) +" was deleted from data base\n")

db.connect()
display_title()
display_categories()
while True:
    command=input("Comand:")
    if command=="cat":
        display_movies_by_category()
    elif command=="year":
        display_movies_by_year()
    elif command=="add":
        add_movie()
    elif command=="del":
        delete_movie()
    elif command=="exit":
        break
    else:
        print("Not a valid command Please try again\n")
        display_menu()

db.close()
print("Goood Bye!")
```

OUTPUT:

```
The Movie List Program

COMMAND MENU
cat - view movies by category
year - view movies by year
add - Add a movie
del - Delete a movie
exit - Exit program

CATEGORIES
3001.Animation
3002.history
3003.comedy

Comand:add
Name:Akhanda
Year:2022
Minutes:90
Category ID;3002
Akhanda was added to data base

Comand:cat
Category ID:3002

MOVIES-HISTORY
ID  Name                          year   Mins  Category
-----------------------------------------------------------------
1242 Dhoni                         2000   82    history
1243 Akhanda                       2022   90    history

Comand:year
year:1985

MOVIES-1985
ID  Name                          year   Mins  Category
-----------------------------------------------------------------

Comand:year
year:2000

Comand:del
Movie ID:1243
Movie ID 1243 was deleted from data base

Comand:year
year:2002
```

MOVIES-2002

ID	Name	year	Mins	Category

Comand:exit
Goood Bye!

17

How to work with a database

In chapter 7, you learned how to work with programs that store data in files. In the real world, though, most programs store data in databases. That's because storing data in a database provides many advantages over storing data in a file.

In this chapter, you'll learn how to work with a database. More specifically, you'll learn how to use Python to work with a SQLite database. Note, however, that most of these skills also apply to working with other types of databases such as MySQL, Oracle, and SQL Server databases.

An introduction to relational databases 462
How a database table is organized 462
How the tables in a database are related 464
How the columns in a table are defined 466

How to use the SQL statements for data manipulation 468
How to select data from a single table 468
How to select data from multiple tables 470
How to insert, update, and delete rows 472

How to use SQLite Manager to work with a database 474
How to connect to a SQLite database 474
How to execute SQL statements 476

How to use Python to work with a database 478
How to connect to a SQLite database 478
How to execute SELECT statements 480
How to get the rows in a result set 482
How to execute INSERT, UPDATE, and DELETE statements 484
How to test the database code 486
How to handle database exceptions 486

The Movie List program 488
The user interface 488
The business tier 488
The database tier 490
The presentation tier 494

Perspective 498

An introduction to relational databases

In 1970, Dr. E. F. Codd developed a model for a new type of database called a *relational database*. This type of database eliminated some of the problems that were associated with standard files and other database designs. By using the relational model, you can reduce data redundancy, which saves disk storage and leads to efficient data retrieval. You can also view and manipulate data in a way that is both intuitive and efficient. Today, relational databases are the de facto standard for database applications.

To facilitate the use of relational databases, a *database management system* (*DBMS*) like MySQL or SQL Server is used. The DBMS not only lets you create and maintain databases, it also maintains the integrity of the databases and helps them run as efficiently as possible.

How a database table is organized

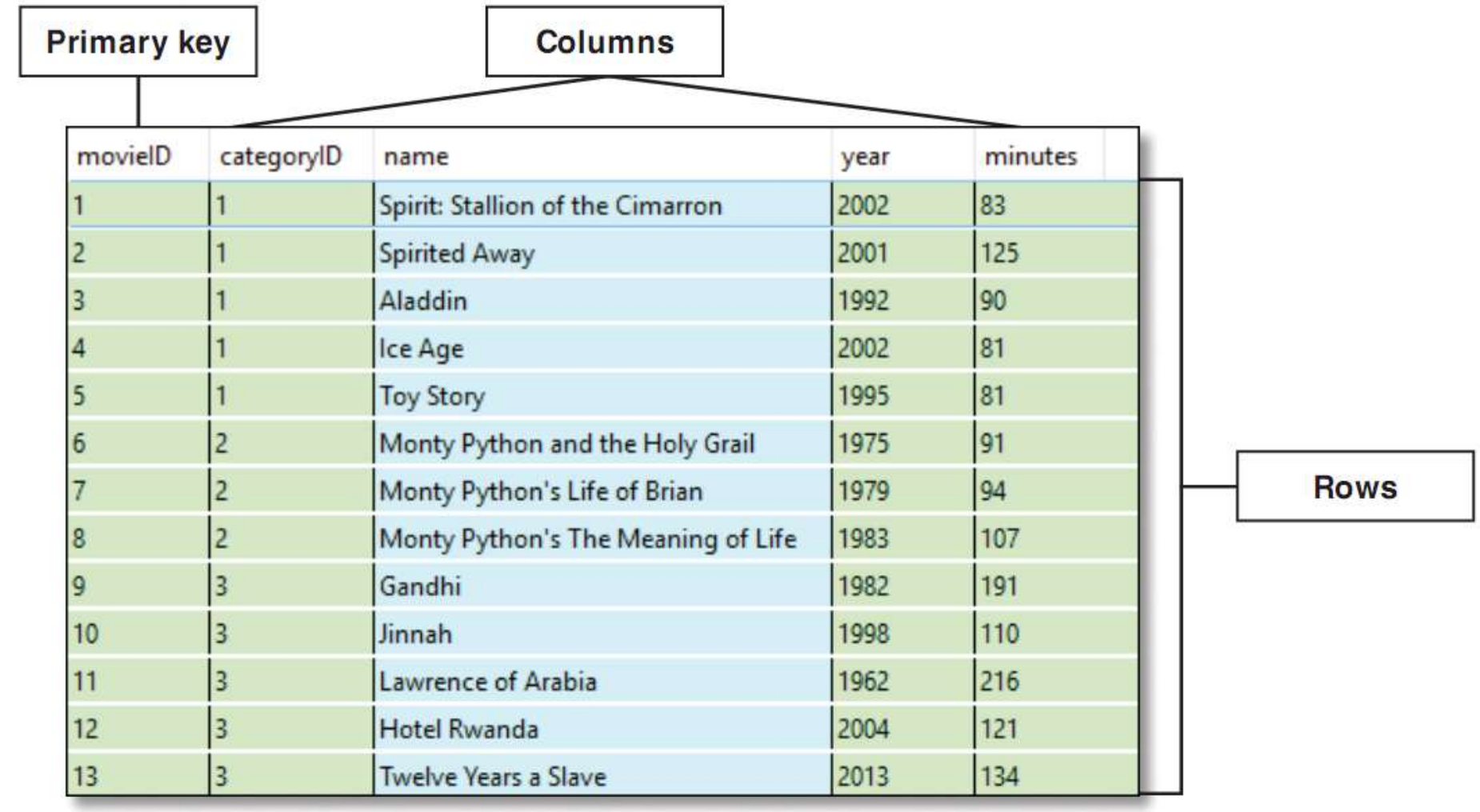
In a relational database, data is stored in one or more *tables* that consist of *rows* and *columns*. This is illustrated by the relational Movie table in figure 17-1. Rows and columns can also be referred to as *records* and *fields*, and a value that is stored at the intersection of each row and column is sometimes called a *cell*.

In this example, each row contains information about a single movie. Then, each column in the row provides data about that movie, including name, year, and running time in minutes. In addition, the first column in this table provides a unique ID for each movie, and the second column provides an ID that associates each movie row with a movie category. For example, the third row in this table is for a movie named Aladdin that was released in 1992 and has a running time of 90 minutes. It also has a movie ID of 3 and a category ID of 1.

Most tables in a relational database have a *primary key* column that uniquely identifies each row in the table. In this example, the movieID column is the primary key for the table. In other words, two movies in the table can't have the same ID. Often, the DBMS generates the primary keys for the records as they are added to the table, which is the case for this table. Also, the DBMS won't allow a record to be added to the table if it has the same (a duplicate) primary key.

In general, each table in a database is modeled after a real-word entity such as a product, customer, or movie. Then, if the entities for a table provide their own unique keys like product number or invoice number, those columns can be used for the primary key so they don't need to be generated. A primary key can also consist of two or more columns.

The Movie table



movieID	categoryID	name	year	minutes
1	1	Spirit: Stallion of the Cimarron	2002	83
2	1	Spirited Away	2001	125
3	1	Aladdin	1992	90
4	1	Ice Age	2002	81
5	1	Toy Story	1995	81
6	2	Monty Python and the Holy Grail	1975	91
7	2	Monty Python's Life of Brian	1979	94
8	2	Monty Python's The Meaning of Life	1983	107
9	3	Gandhi	1982	191
10	3	Jinnah	1998	110
11	3	Lawrence of Arabia	1962	216
12	3	Hotel Rwanda	2004	121
13	3	Twelve Years a Slave	2013	134

Concepts

- A *relational database* consists of *tables*. Tables consist of *rows* and *columns*, which can also be referred to as *records* and *fields*.
- A table is typically modeled after a real-world entity, such as a product or customer, but it can also be modeled after an abstract concept, such as the data for a game.
- A column represents an attribute of the entity, such as a movie's name.
- A row contains a set of values for one instance of the entity, such as one movie.
- Most tables have a *primary key* that uniquely identifies each row in the table.
- The primary key is usually a single column, but it can also consist of two or more columns.

Figure 17-1 How a database table is organized

How the tables in a database are related

The tables in a relational database can be related to other tables by values in specific columns. The two tables shown in figure 17-2 illustrate this concept. Here, each row in the Category table is related to one or more rows in the Movie table. This is called a *one-to-many relationship*. In other words, a category can have many movies, but a movie can only belong to one category.

Typically, relationships exist between the primary keys in one table and *foreign keys* in another table. A foreign key is simply one or more columns in a table that refer to a primary key in another table.

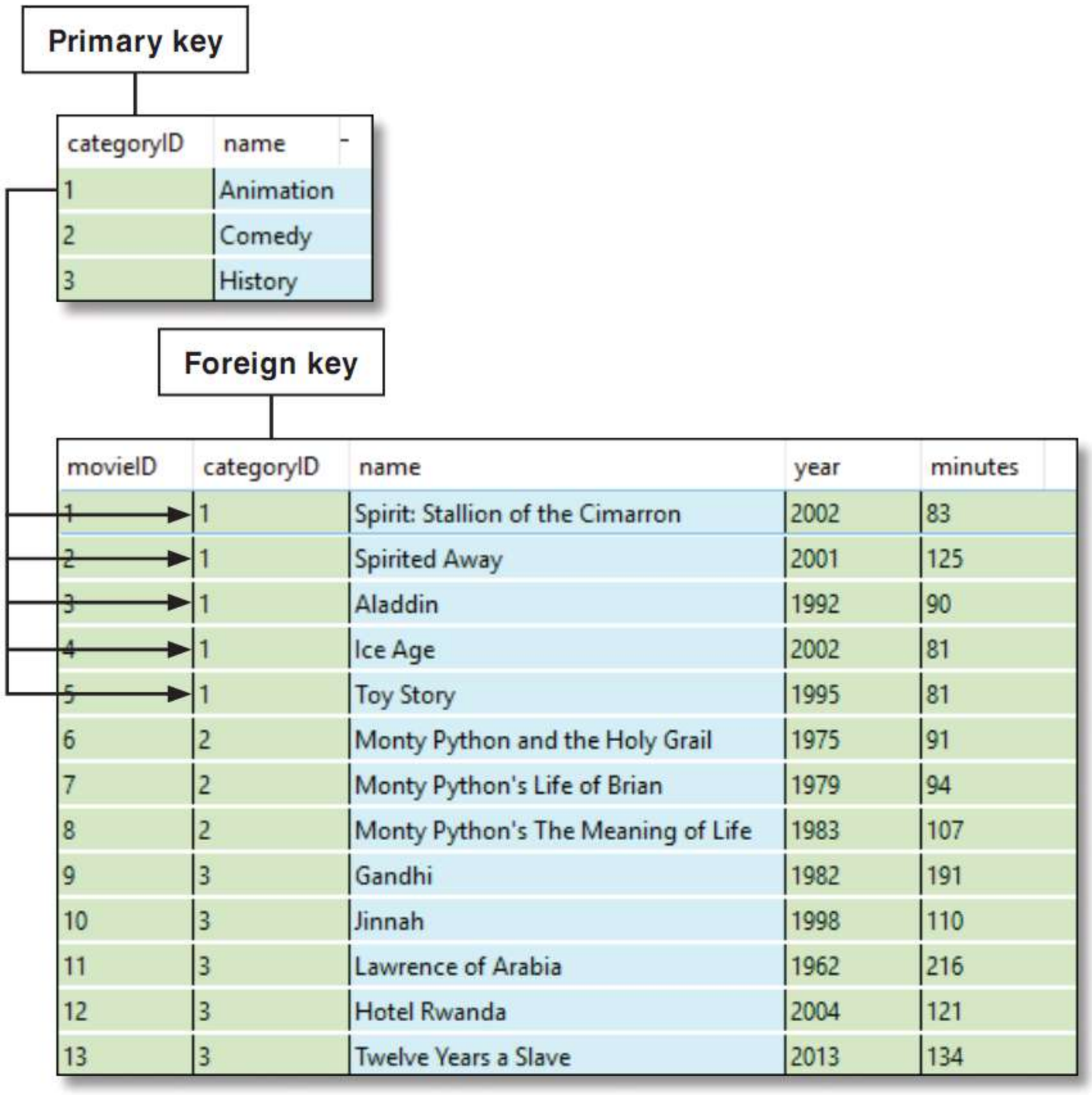
In this example, the categoryID column in the Movie table is a foreign key that refers to the categoryID column in the Category table, which is the primary key for that table. For example, the first row in the Movie table has a categoryID of 1 that relates to the first row in the Category table, which is the Animation category.

You should know that the Movie table in this figure is unrealistic because its rows are in sequence by both movieID and categoryID. As soon as records are added to the Movie table, though, the cateogeryIDs won't be in sequence. If, for example, a 14th record is added to the table with a categoryID of 2, the categoryID sequence will be broken.

Although a one-to-many relationship is the most common type of relationship, two tables can also have a one-to-one or many-to-many relationship. If a table has a *one-to-one relationship* with another table, the data in the two tables could be stored in a single table. Because of that, one-to-one relationships are used infrequently.

In contrast, a *many-to-many relationship* is usually implemented by using an intermediate table that has a one-to-many relationship with the two tables. In other words, a many-to-many relationship can usually be broken down into two or more one-to-many relationships.

The relationship between a Category table and a Movie table



categoryID	name
1	Animation
2	Comedy
3	History

movieID	categoryID	name	year	minutes
1	1	Spirit: Stallion of the Cimarron	2002	83
2	1	Spirited Away	2001	125
3	1	Aladdin	1992	90
4	1	Ice Age	2002	81
5	1	Toy Story	1995	81
6	2	Monty Python and the Holy Grail	1975	91
7	2	Monty Python's Life of Brian	1979	94
8	2	Monty Python's The Meaning of Life	1983	107
9	3	Gandhi	1982	191
10	3	Jinnah	1998	110
11	3	Lawrence of Arabia	1962	216
12	3	Hotel Rwanda	2004	121
13	3	Twelve Years a Slave	2013	134

Concepts

- The tables in a relational database are related to each other through their key columns. For example, the Category and Movie tables in this figure use the categoryID column to create the relationship between the two tables.
- The categoryID column in the Movie table is called a *foreign key* because it identifies a related row in the Category table. A table may contain one or more foreign keys.
- When you define a foreign key, you can't add rows to the table with the foreign key unless there's a matching primary key in the related table.
- The relationships between the tables in a database correspond to the relationships between the entities they represent. The most common type of relationship is a *one-to-many relationship* as illustrated by the Category and Movie tables.
- A table can also have a *one-to-one relationship* or a *many-to-many relationship* with another table.

Figure 17-2 How the tables in a relational database are related

How the columns in a table are defined

When you define a column in a table, you assign properties to it as indicated by the design of the Movie table in figure 17-3. First, you need to define a name for the column, which should generally reflect the type of information that the column stores. Then, you need to define a *data type* for the column, which determines the type of data that the column stores. In this table, all of the columns use the INTEGER type to store integer numbers, except for the name column, which uses the TEXT type to store the name of the movie.

In addition to choosing a data type, you must identify whether the column can store a *null value*. The NULL keyword represents a value that's unknown, unavailable, or not applicable. If you don't allow null values, then you must provide a value for the column or you can't store the row in the table. In this figure, none of the columns can be NULL.

You can also assign a *default value* to a column. Then, the column uses the default value if no value is provided when adding the row. In this example, the default value is NULL for all columns. When a default value of NULL is combined with a column that doesn't allow null values, it forces the user to provide a value for that column when saving a row.

When you define a column as the primary key, its value must be unique. One way to assure that is to let the DBMS generate the primary key when a new row is added to the database.

When you define a column as a foreign key, you also specify which column it refers to in another table. In this example, the categoryID column in the Movie table refers to the categoryID column in the Category table. When a column is a foreign key, its value must exist in a row of the table that it references. In addition, the DBMS won't let you delete a row from a table if other tables contain foreign key values that refer to that row.

If, for example, there are any movies in the Movie table that belong to category 1, the DBMS won't let you delete the category 1 row from the Category table. If the DBMS didn't enforce this constraint, you would end up with movies that have foreign keys that refer to a category that doesn't exist. These keys can be referred to as *orphaned keys*.

The columns of the Category table

Name	Data Type	Not Null?	Default Value	Primary Key?	Foreign Key?
categoryID	INTEGER	Y	NULL	Y	N
name	TEXT	Y	NULL	N	N

The columns of the Movie table

Name	Data Type	Not Null?	Default Value	Primary Key?	Foreign Key?
movieID	INTEGER	Y	NULL	Y	N
categoryID	INTEGER	Y	NULL	N	Y
name	TEXT	Y	NULL	N	N
year	INTEGER	Y	NULL	N	N
minutes	INTEGER	Y	NULL	N	N

Common SQLite data types

Type	Description
TEXT	A variable-length Unicode string.
INTEGER	Integer values of various sizes.
REAL	Decimal values that can contain an integer portion and a decimal portion.
BLOB	The data is stored exactly as entered. Can contain binary data such as images.

Description

- The *data type* that's assigned to a column determines the type of data that can be stored in the column.
- Each column definition also indicates whether or not it can contain *null values*. The NULL keyword indicates that the value of the column is unknown.
- A column can also be defined with a *default value*. Then, that value is used if another value isn't provided when a row is added to the table.
- If a column is defined as a primary key, some databases automatically generate its value when a new row is added to the table.
- When you define a column as a foreign key, you specify the column in another table that the key refers to.
- To avoid creating *orphaned keys*, a database typically prevents you from deleting a row if one of its values is being used as a foreign key in another table.

Figure 17-3 How the columns in a table are defined

How to use the SQL statements for data manipulation

Structured Query Language, or *SQL*, is a standard language for working with databases. In conversation, SQL is pronounced "S-Q-L" or "sequel". In this book, we use "sequel" as the pronunciation so we refer to *a* SQL statement, instead of *an* SQL statement.

The SELECT, INSERT, UPDATE, and DELETE statements that you'll learn about next make up SQL's *Data Manipulation Language* (*DML*). These statements work with the data in a database, and they are the statements that programmers use every day in their programs. As a result, these are the statements that you'll learn how to use in this chapter.

In contrast, the CREATE DATABASE, DROP DATABASE, CREATE TABLE, and DROP TABLE statements are part of SQL's *Data Definition Language* (*DDL*). These statements are typically used by database administrators to create and delete databases and tables, and they aren't presented in this book.

How to select data from a single table

The *SELECT statement* is the most commonly used *SQL statement*. It is used to retrieve data from one or more tables in a database. When you run a SELECT statement, it is commonly referred to as a *query*. The result of a query is a table known as a *result set*, or *result table*.

Figure 17-4 shows the syntax for a SELECT statement that gets all of the columns of a table, and it shows the syntax for a SELECT statement that gets only selected columns. In the syntax summaries for SQL statements, the capitalized words are SQL keywords, and the lowercase words are the ones that you supply. Also, the brackets indicate optional components of a statement, and the bar (|) indicates a choice between two options.

The first example in this figure shows how to get all columns and selected rows from the Movie table. Here, the SELECT clause uses the asterisk (*) wildcard to indicate that all of the columns in the table should be retrieved, and the FROM clause specifies that these columns should be retrieved from the Movie table. Then, the WHERE clause indicates that each row should have a categoryID column that's equal to 2. As a result, this query returns three rows and five columns.

The second example shows how to get selected columns and rows from the Movie table. This time, the SELECT clause identifies the name and minutes columns, and the FROM clause identifies the Movie table. Then, the WHERE clause specifies that each row that's returned by the statement should have a running time that's less than 90 minutes.

Last, the ORDER BY clause indicates that the retrieved rows should be sorted in ascending order by the minutes column. This sorts the rows from the shortest movie to the longest movie. However, if you wanted to switch the sort order, you could use the DESC keyword instead of the ASC keyword. Then, the rows would be sorted in descending sequence instead of ascending sequence.

The syntax for a SELECT statement that gets all columns

```
SELECT *
FROM table
[WHERE selection-criteria]
[ORDER BY column-1 [ASC|DESC] [, column-2 [ASC|DESC] ...]]
```

A SELECT statement that gets all columns

```
SELECT * FROM Movie
WHERE categoryID = 2
```

movieID	categoryID	name	year	minutes
6	2	Monty Python and the Holy Grail	1975	91
7	2	Monty Python's Life of Brian	1979	94
8	2	Monty Python's The Meaning of Life	1983	107

The syntax for a SELECT statement that gets selected columns

```
SELECT column-1[, column-2] ...
FROM table
[WHERE selection-criteria]
[ORDER BY column-1 [ASC|DESC][, column-2 [ASC|DESC] ...]]
```

A SELECT statement that gets selected columns and rows

```
SELECT name, minutes
FROM Movie
WHERE minutes < 90
ORDER BY minutes ASC
```

name	minutes
Ice Age	81
Toy Story	81
Spirit: Stallion of the Cimarron	83

Description

- A *SELECT statement* is a SQL statement that returns a *result set* (or *result table*) that consists of the specified rows and columns.
- To specify the columns to select, use the SELECT clause.
- To specify the table that the data should be retrieved from, use the FROM clause.
- To specify the rows to select, use the WHERE clause.
- To specify how the result set should be sorted, use the ORDER BY clause. Within this clause, use the ASC keyword to sort a column in ascending order or the DESC keyword to sort a column in descending order.

Figure 17-4 How to select data from a single table

How to select data from multiple tables

Figure 17-5 shows how to use the SELECT statement to retrieve data from two tables. This is commonly known as a *join*. The result of any join is a single result set.

An *inner join* is the most common type of join. When you use an inner join, the data from the rows in two tables are included in the result set only if their related columns match. In this figure, the SELECT statement joins the data from the rows in the Movie and Category tables, but only if the value of the categoryID column in the Movie table is equal to the categoryID column in the Category table. In other words, if there isn't any data in the Movie table for a category, that category isn't added to the result set. This join results in a user-friendly result set because the category now has a name instead of a numerical ID.

To code an inner join, you use the JOIN clause to specify the second table and the ON clause to specify the columns to be used for the join. Then, if a column in one table has the same name as a column in the other table, you must qualify the column name by coding the table name, a dot, and the column name. This is illustrated by both columns in the ON clause. This is also illustrated by the first two columns in the SELECT clause.

When a column in one table of a join has the same name as a column in the other table, you can use the AS clause to create an *alias* for the column name. In this example, an alias of categoryName is assigned to the Category table's name column. So that's the name that's displayed at the top of the column in the result set, and that's the name that you can use in your Python code to access this column.

Although there are other types of joins besides inner joins, inner joins are so common that they are the default type of join. That's why you don't have to code the INNER keyword before the JOIN keyword when you code your SQL statement.

Although this figure only shows how to join data from two tables, you can extend this syntax to join data from additional tables. If, for example, you want to create a result set that includes data from three tables named Category, Movie, and Actor, you can code the FROM clause of the SELECT statement like this:

```
FROM Movie
    JOIN Category ON Category.categoryID = Movie.categoryID
    JOIN Actor ON Actor.movieID = Movie.movieID
```

Then, you can include any of the columns from the three tables in the column list of the SELECT statement.

The syntax for a SELECT statement that joins two tables

```
SELECT column-1 [AS alias-1] [[, column-2] [AS alias-2]]...
FROM table-1
   [INNER ]JOIN table-2 ON table-1.column-1 = table-2.column-2
```

A statement that gets data from two related tables

```
SELECT Movie.name, Category.name AS categoryName, minutes
FROM Movie
    JOIN Category ON Category.categoryID = Movie.categoryID
WHERE minutes < 90
ORDER BY minutes ASC
```

name	categoryName	minutes
Ice Age	Animation	81
Toy Story	Animation	81
Spirit: Stallion of the Cimarron	Animation	83

Description

- To return a result set that contains data from two tables, you *join* the tables. To do that, you can use a JOIN clause.
- Most of the time, you'll want to code an *inner join* so that rows are only included when the key of a row in the first table is equal to (matches) the key of a row in the second table.
- If you don't specify the join type you want, an inner join is used by default.
- If the two tables you want to join contain duplicate column names, you have to qualify the column by prefixing each column name with its table name. Otherwise, the column name is ambiguous, and the database doesn't know which column to retrieve.
- When you qualify a name, you often want to provide a new name, or *alias*, for the column name so each column in the result set has a unique name. To do that, you can code the AS keyword followed by the aliases for the columns.
- When you're getting started with joins, you may want to prefix every column with its table name so it's clear which table the column is coming from. However, this is only required for the column names that are the same in both tables.

Figure 17-5 How to select data from multiple tables

How to insert, update, and delete rows

Figure 17-6 shows how to use the INSERT, UPDATE, and DELETE statements to add, update, or delete one or more rows in a database. Note that none of these statements return a result set. Instead, they change the data in the database.

The *INSERT statement* is used to add a row to a database table. As the syntax and examples show, you can do that in two ways. Here, the INTO clause in the first INSERT statement provides a list of the columns that data will be provided for. Then, the VALUES clause provides the values for those columns in the same sequence as the columns. Note here that you don't have to provide values for columns like the MovieID column that will have their values generated by the DBMS. And you don't have to provide values for columns that have default values. For the Movie table, though, none of the columns provide default values.

The second INSERT statement shows that the column list in the INTO clause is optional. In other words, you don't need to supply the list of column names after the table name. In this case, though, you must code the values in the order in which the columns are defined for the table, and you must supply a value for every column, even ones that SQLite generates automatically.

The syntax and example for the *UPDATE statement* show how to use that statement to update the data in one or more columns of one or more rows. Here, the SET clause specifies the columns and values that are to be updated, and the WHERE clause specifies which rows should be updated. In the example, the UPDATE statement updates the minutes column in the row where the movieID column is equal to 4, so only one row is updated.

Last, the syntax and examples for the *DELETE statement* show how to delete one or more rows in a table. Here, the WHERE clause identifies the row or rows to be deleted. As a result, the first DELETE statement deletes the row from the Movie table where the movieID equals 12, so only one row is deleted. But the second DELETE statement deletes all rows in the Movie table that have a year that's equal to 1979, so more than one row is deleted.

When you use DELETE statements, you need to be aware of how dangerous they can be. That's because a coding error in the WHERE clause can delete far more rows than you intended. So use DELETE statements with care, especially when you're deleting rows based on columns that don't contain unique values.

When you specify a value within a SQL statement, you must enclose string values in single quotes. However, you aren't required to enclose numeric values in single quotes. This is illustrated by the INSERT, UPDATE, and DELETE statements in this figure, but it's also true for SELECT statements.

The syntax for the INSERT statement

```
INSERT INTO table-name [(column-list)]
VALUES (value-list)
```

A statement that uses a column list to add one row

```
INSERT INTO Movie (name, year, minutes, categoryID)
VALUES ('Juno', 2007, 96, 2)
```

A statement that doesn't use a column list to add one row

```
INSERT INTO Movie
VALUES (14, 2, 'Juno', 2007, 96)
```

The syntax for the UPDATE statement

```
UPDATE table-name
SET expression-1 [, expression-2] ...
WHERE selection-criteria
```

A statement that updates a column in one row

```
UPDATE Movie
SET minutes = 84
WHERE movieID = 4
```

The syntax for the DELETE statement

```
DELETE FROM table-name
WHERE selection-criteria
```

A statement that deletes one row from a table

```
DELETE FROM Movie
WHERE movieID = 14
```

A statement that deletes multiple rows from a table

```
DELETE FROM Movie
WHERE year = 1979
```

Description

- The *INSERT*, *UPDATE*, and *DELETE statements* modify the data that's stored in a database. These statements don't return a result set.
- Be careful when deleting or updating based on columns that might not be unique, since you could inadvertently delete or update more rows than intended.
- When you specify a value within a SQL statement, you must enclose string values in single quotes. However, you aren't required to enclose numeric values in single quotes.

Figure 17-6 How to insert, update, and delete data

How to use SQLite Manager to work with a database

SQLite is a popular open-source relational database that can be embedded into programs. Many software products use SQLite including the Firefox web browser, the Civilization V game, and many other games. This chapter shows how to use SQLite because it's easy to set up and because Python includes built-in support for working with SQLite.

To work with a SQLite database, you can use a Firefox add-on called *SQLite Manager*. If you haven't already installed it, the appendix for your operating system will show you how.

How to connect to a SQLite database

Figure 17-7 begins by showing how to use SQLite Manager to connect to a database and view one of its tables. First, you start Firefox and then you start SQLite Manager. If you've added the SQLite Manager icon to the Firefox toolbar, you can just click on the icon to start the SQLite Manager in a separate window.

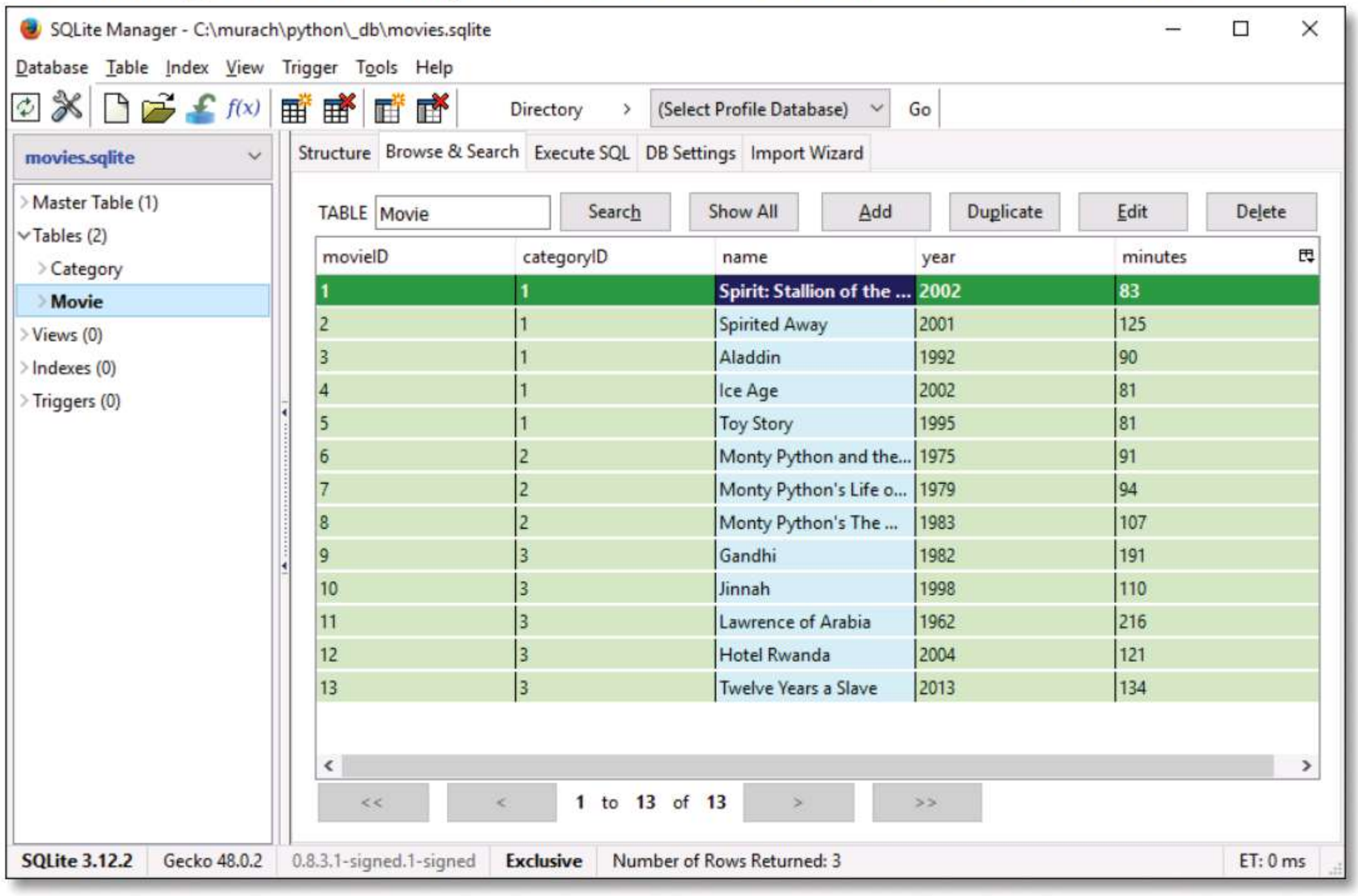
Then, to connect to the database you want to work with, you select Database→Connect Database from the Manager's menu. In the resulting dialog box, you just navigate to the file for the SQLite database. In this figure, SQLite Manager has opened the SQL database that you get when you install the source code for this book.

Once you connect to the database, you can view one of its tables by clicking the Browse & Search tab in SQLite Manager's main panel. Then, in the left panel, you can expand the Tables node and click the table you want to view. This displays the table in the main panel, along with all of its rows and columns. In this figure, SQLite Manager shows the Movie table and its data.

As you view a table, you can click on a row to select it, and you can use the buttons at the top of the table to add, edit, or delete rows. In this figure, the first row is selected. As a result, you could click the Edit button to edit its data, or you could click the Delete button to delete the row.

After you expand the Tables node, you can work with a table by right-clicking on it (Ctrl-clicking on Mac) and selecting a command from the resulting menu. For example, you could select the Rename command to rename the table, or you could select the Delete command to delete the table.

SQLite Manager after using the Browse & Search tab to view a table



How to connect to a SQLite database and view a table

1. Start Firefox.
2. If you added the SQLite Manager icon to the Firefox toolbar, as shown in the appendix, you can click on the icon to start SQLite Manager. Otherwise, you can use Tools→SQLite Manager to start SQLite Manager, or you can press Alt+T (Option+T on Mac) to access the Tools menu and then select SQLite Manager.
3. Select Database→Connect Database. Then, use the dialog box to select the SQLite database you want to open. The database for this chapter should be located here:
   `python\_db\movies.sqlite`
4. In the main panel, click the Browse & Search tab.
5. In the left panel, expand the Tables node and click the table you want to view.

Description

- To install the Firefox browser and the SQLite Manager, use the instructions in the appendix for your operating system.
- To connect to a database and view one of its tables, use the procedure above.
- As you view a table, you can click on a row to select it, and you can use the buttons at the top of the table to add, edit, or delete rows.
- After you expand the Tables node, you can right-click (Ctrl-click on Mac) on a table's name. Then, you can use the resulting menu to copy, rename, or delete the table.

Figure 17-7 How to use SQLite Manager to connect to and work with a database

How to execute SQL statements

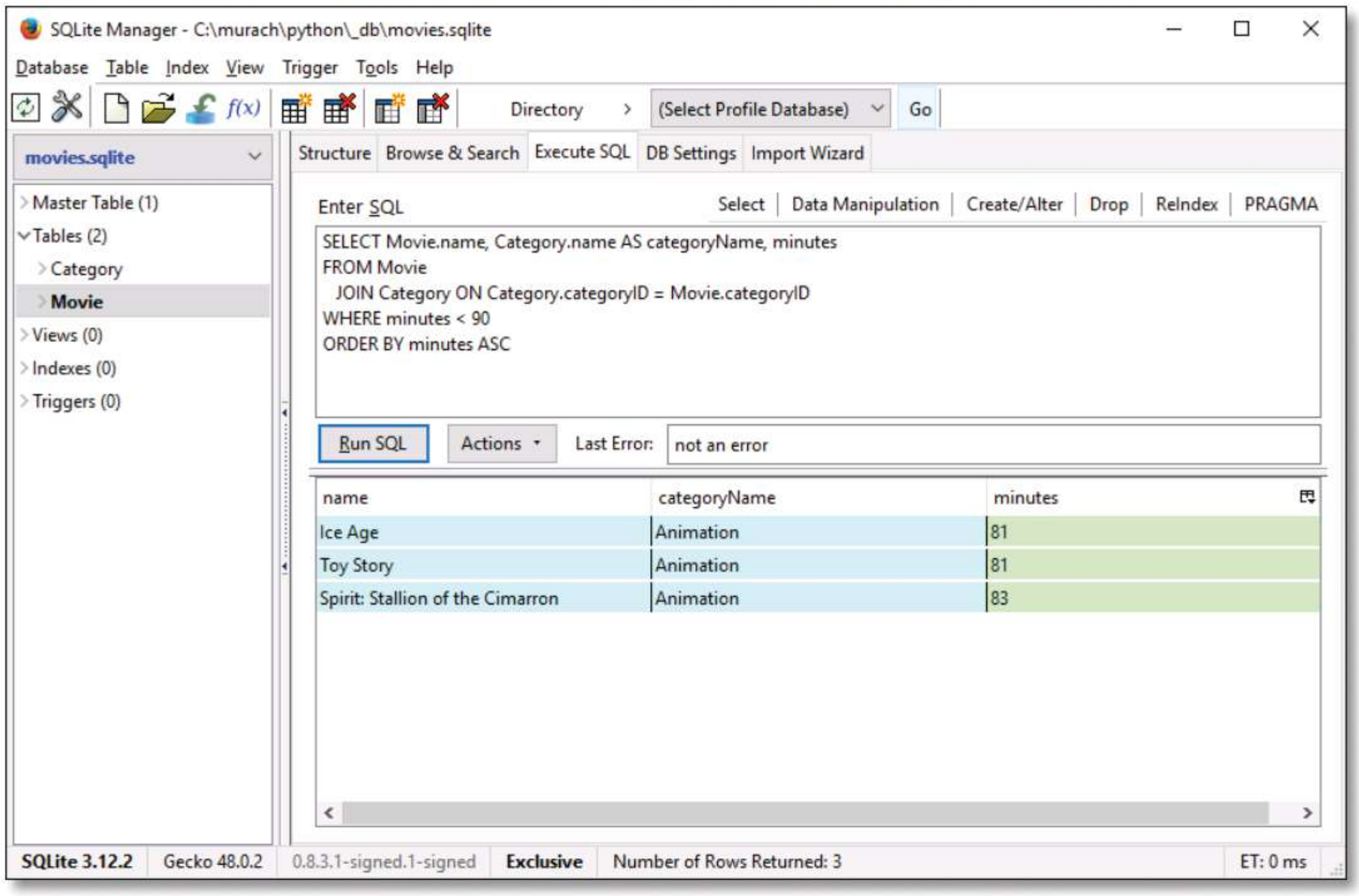
After you use SQLite Manager to connect to a database, you can use it to execute SQL statements against that database. This can help you test your SQL statements before you use them in your Python code, and it can help you debug SQL statements that aren't working correctly.

Figure 17-8 shows how to use SQLite Manager to execute a SQL statement. To start, you click on the Execute SQL tab. Then, you enter a SQL statement and click the Run SQL button. When you do, SQLite Manager displays the result set just below the Run SQL button. In this example, SQLite Manager has executed a SELECT statement successfully and displayed a result set. However, if the SQL statement isn't valid, SQLite Manager displays an error message that can help you find and fix the problem.

Of course, the INSERT, UPDATE, or DELETE statements, don't return a result set. So, for those statements, SQLite Manager doesn't display a result set. However, it still executes the statement, which modifies the data in the database. It also displays an error message if the statement causes an error to occur.

To make it easy to identify the keywords in a SQL statement, the examples in this book capitalize the keywords. You should realize, however, that this capitalization is optional. So, if you want to use lowercase letters in the keywords when you type them into SQLite Manager, you can do that.

The Execute SQL tab after a SQL statement has been executed



Description

- To execute a SQL statement, click the Execute SQL tab. Then, enter the SQL statement and click the Run SQL button.
- To sort the columns in the result set, click on the column name.

Figure 17-8 How to use SQLite Manager to run a SQL statement

How to use Python to work with a database

Now that you know how to code SQL statements and how to use SQLite Manager to test those statements, you're ready to learn how you can use those statements in Python as you develop programs that work with a database.

How to connect to a SQLite database

Figure 17-9 shows you how to connect to a SQLite database from Python. To begin, you import the sqlite3 module as shown in the first example. This module allows you to work with SQLite databases.

Once you've imported the sqlite3 module, you need to open a connection to the database. To do that, you call the connect() method from the sqlite3 module to create a *connection object*. This method accepts an argument that specifies the path to the database file. If the database file is in the working directory, which is usually the same directory as the program file, you only need to specify the name of the database file as shown in the second example. Here, the database file has a name of movies.sqlite.

Although this technique is adequate for programs like the ones in this book, it may not be adequate for real-world programs. That's why the next example shows how to connect to a database that isn't in the working directory. To start, this example imports the sys and os modules. Then, it uses the sys module to determine whether the platform is Windows, Mac OS X, or Linux. If the platform attribute is "win32", the operating system is Windows. That's true even if you are running on a 64-bit version of Windows. If the platform attribute isn't "win32", it's safe to assume that the operating system is OS X or some variant of Linux or Unix.

On a Windows system, this example connects to the movie database that's in the _db directory by supplying a complete path to its file. However, on OS X and Linux, normal users can't store files outside of their own home directory. As a result, if you are running as a normal user (and you should be for security reasons), the database file is most likely installed in the home directory.

Then, to get the user's home directory, this code accesses the environ dictionary that's available from the os module and uses the HOME key to get the home directory of the user who is running the program. Next, it appends the path to the database file to the path to the home directory. The last statement in this example gets a connection object by passing the path to the database file to the connect() method of the sqlite module.

When your program is through with a connection object, your program should close it so its resources are returned to the system. That last example in this figure shows how to use the close() method of a connection object to do that. Here, the if statement checks whether the connection object named conn exists. If it does, the close() method is called.

How to import the module that supports SQLite databases

```
import sqlite3
```

The syntax for connecting to a database and returning a connection object

```
conn = sqlite3.connect(path_to_database_file)
```

How to connect to a database that's in the working directory

```
conn = sqlite3.connect("movies.sqlite")
```

How to connect to a database that's not in the working directory

```
import sys
import os

if sys.platform == "win32":                                    # Windows
    DB_FILE = "/murach/python/_db/movies.sqlite"
else:                                                          # Mac OS X and Linux
    HOME = os.environ["HOME"]
    DB_FILE = HOME + "/Documents/murach/python/_db/movies.sqlite"
conn = sqlite3.connect(DB_FILE)
```

How to use the close() method to close a connection object

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

Description

- To work with a SQLite database in Python, you need to import the sqlite3 module.
- To connect to a database, call the connect() method on the sqlite3 module with the path to the database. This returns a *connection object* that you can use to access the database.
- If the database file is in the working directory, which is usually the same directory as the program, the path to the database is just the name of the database file. Otherwise, you can specify a complete path to the database file.
- If you need to get the home directory of the current user, you can import the os module and use its environ dictionary.
- Because a SQLite database runs locally and isn't accessible over a network, you don't need a username or password to connect to it.
- When you're through with a database connection, you need to close it so it frees up the resources that it's using.

Figure 17-9 How to use Python to connect to a SQLite database

How to execute SELECT statements

Figure 17-10 shows how to execute SELECT statements with Python. But first, in order to execute any SQL statement on a database, you need to get a *cursor object* for the database. To do that, you call the cursor() method from the connection object. Then, to execute a SQL statement, you call the execute() method from the cursor object. If the method requires any values as arguments, you pass a tuple argument that supplies the values to the method.

The first example in this figure shows how to get a cursor object named c. Then, the second example shows how to execute a simple SELECT statement for the database that's represented by the cursor object. Here, the first statement creates a variable named query that stores a SELECT statement that gets all rows from the Movie table. Then, the next statement runs the query by calling the execute() method from the cursor object and passing the query variable as the first argument.

When you code a SQL statement in Python, you can enclose it in single or double quotes. However, since SQL statements often span multiple lines, it's often better to use three single quotes as shown in this example. That way, you don't have to concatenate multiple lines of code.

The third example in this figure shows how to run a query based on input provided by the user of the program. Here, the query selects all movies with a running time of less than a parameter value that will get passed to the query. In a query like this, a question mark (?) is used as a placeholder for each parameter that's needs to be supplied by the calling statement. Then, when you call the execute() method, you pass a tuple that contains the required placeholder values as the second argument for the method.

When you use this technique, remember that you must include a comma at the end of a tuple that contains only one value. In this example, the SELECT statement has only one placeholder so the tuple has a comma after the value 90. That means that the statement will return a result set that contains all movies with a running time of less than 90 minutes.

When your program is through with a cursor, it should close it so its resources are released to the system. That's why the last example in this figure shows how to do that. First, you need to import the closing() function from the Python contextlib module. Then, you need to code a with statement for the closing() function that contains the statements that use the cursor. This code will close the cursor object whether or not an exception is thrown.

The cursor() method of the connection object

Method	Description
`cursor()`	Returns a cursor object that you can use to execute SQL statements.

The execute() method of the cursor object

Method	Description
`execute(sql [params_tuple])`	Executes the SQL statement. If the statement includes question mark placeholders, you supply the values for the placeholders in a tuple parameter.

How to get a cursor object from the connection object

```
c = conn.cursor()
```

How to execute a SELECT statement that doesn't have parameters

```
query = '''SELECT * FROM Movie'''
c.execute(query)
```

How to execute a SELECT statement that has a parameter

```
query = '''SELECT * FROM Movie
           WHERE minutes < ?'''
c.execute(query, (90,))
```

How to automatically close the cursor object

The code for importing the closing() function

```
from contextlib import closing
```

The syntax for automatically closing the cursor object

```
with closing(resource) as name:
    statements
```

How to automatically close the cursor object

```
with closing(conn.cursor()) as c:
    query = '''SELECT * FROM Movie'''
    c.execute(query)
```

Description

- To execute a SELECT statement in Python, you use the cursor() method of the connection object to get a *cursor object*. Then, you use the execute() method of the cursor object to execute the SQL statement.
- To mark a variable in a SQL statement, you code a question mark (`?`) placeholder. Then, you provide a tuple parameter for the execute() method that provides the values for the placeholders. But remember that a tuple with only one item must end with a comma.
- When you are done with a cursor, you should close it to make sure its resources are released. To make sure that happens, even if an exception occurs, you can code the database operations in a with statement that uses the closing() function.

Figure 17-10 How to use Python to execute SELECT statements

How to get the rows in a result set

After you execute a query, the cursor object contains a result set with the rows returned by the query. Then, to access a row or rows, you can use the fetchone() or fetchall() methods of the cursor object. These methods are summarized and illustrated in figure 17-11.

The first example begins by getting a cursor and executing a query that selects the movie in the Movie table that has an ID of 5. Then, it uses the fetchone() method to get the row that's in the result set and store it in a variable named movie. Because this code is within a with statement for the closing() function, the cursor is closed after the statements are executed.

After a row is stored in the movie variable, you can access the columns of the row using an index. For instance, the second example prints the name, year, and running time for the movie by specifying indexes of 2, 3, and 4 for those columns. This works because movieID and categoryID are in the columns with the indexes 0 and 1.

Because indexes don't clearly identify the data that's stored in the columns, the third example shows how to access the columns by name. To do that, you set the row_factory attribute of the connection object to sqlite3.Row. Once that's done, you can access the columns of a row by name, which makes your code easier to read and less prone to errors. In addition, your code won't break if you add a column to the table later on that changes the indexes.

If you want to access all of the rows in a result, you can use the fetchall() method as shown in the last example. Here again, the query returns all rows in the Movie table that have a running time of less than 90 minutes. Then, the fetchall() method stores all of these rows in a list named movies. At this point, you can use a for loop to access each movie in the list. In this example, this loop prints the movie's name, year of release, and running time.

The fetchone() and fetchall() methods of the cursor object

Method	Description
`fetchone()`	Returns a tuple containing the next row from the result set. If there is no next row in the result set, this method returns None.
`fetchall()`	Returns a list containing all of the rows in the result set.

How to use the fetchone() method to get a row from of a table

```
with closing(conn.cursor()) as c:
    query = '''SELECT * FROM Movie
               WHERE movieID = ?'''
    c.execute(query, (5,)
    movie = c.fetchone()
```

How to access columns by index

```
print("Name:    " + movie[2])
print("Year:    " + str(movie[3]))
print("Minutes: " + str(movie[4]))
```

How to access columns by name

How to use the row_factory attribute to enable name access

```
conn.row_factory = sqlite3.Row      # Row is a SQLite constant
```

How to access columns by name

```
print("Name:    " + movie["name"])
print("Year:    " + str(movie["year"]))
print("Minutes: " + str(movie["minutes"]))
```

How to use the fetchall() method to retrieve all rows in the cursor

```
with closing(conn.cursor()) as c:
    query = '''SELECT * FROM Movie
               WHERE minutes < ?'''
    c.execute(query, (90,))
    movies = c.fetchall()
```

How to loop through all rows

```
for movie in movies:
    print(movie["name"], "|", movie["year"], "|", movie["minutes"])
```

The console

```
Spirit: Stallion of the Cimarron | 2002 | 83
Ice Age | 2002 | 81
Toy Story | 1995 | 81
```

Description

- If you access columns by index, your code can break if the database structure changes. When you access columns by name, your code is more durable and easier to read.
- Before you can access columns by name, you need to set the the row_factory attribute of the connection object to the SQLite constant named Row (sqllite.Row).

Figure 17-11 How to use Python to get the rows in a result set

How to execute INSERT, UPDATE, and DELETE statements

Figure 17-12 shows how to use Python to execute INSERT, UPDATE, and DELETE statements. This works much like executing a SELECT statement, but these statements don't return a result set. Instead, they modify the data in the database.

Also, when you execute one of these SQL statements, the database is changed but it isn't saved to the database. To save the changes, you must call the commit() method of the connection object. Otherwise, the changes are lost.

The first example in figure 17-12 shows how to use an INSERT statement to add a new movie to the database. First, it sets four variables to the name, year, running time, and category ID of the movie to be added to the database. Next, it uses a with statement to open a cursor object named c.

Inside the with statement, the code creates a string named sql that stores the INSERT statement. This statement uses question mark placeholders for four values. Then, the execute() method is called with a tuple argument that supplies the values for these placeholders. As a result, the INSERT statement that the database actually runs becomes:

```
INSERT INTO Movie (name, year, minutes, categoryID)
VALUES ('Juno', 2007, 96, 2)
```

After this statement is executed, the commit() method is called to save the new row to the database.

The second example shows an UPDATE statement that changes the running time to 84 minutes for the movie with an ID of 4. Here again, note the use of the parameterized SQL statement, and the tuple that supplies the values for the two placeholders in the SQL statement.

The last example shows a DELETE statement that deletes the movie with an ID of 12 from the database. This SQL statement is similar to the previous two statements, but note the comma in the tuple argument that passes just one value to the DELETE statement.

The commit() method of the connection object

Method	Description
`commit()`	Commits the changes to the database.

How to execute an INSERT statement

```
name = "Juno"
year = 2007
minutes = 96
categoryID = 2

with closing(conn.cursor()) as c:
    sql = '''INSERT INTO Movie (name, year, minutes, categoryID)
             VALUES (?, ?, ?, ?)'''
    c.execute(sql, (name, year, minutes, categoryID))
    conn.commit()
```

How to execute an UPDATE statement

```
id = 4
minutes = 84

with closing(conn.cursor()) as c:
    sql = '''UPDATE Movie
             SET minutes = ?
             WHERE movieID = ?'''
    c.execute(sql, (minutes, id))
    conn.commit()
```

How to execute a DELETE statement

```
id = 12

with closing(conn.cursor()) as c:
    sql =  '''DELETE FROM Movie
              WHERE movieID = ?'''
    c.execute(query, (id,))
    conn.commit()
```

Description

- As with the SELECT statement, you use question marks (?) to identify the parameters in the INSERT, UPDATE, and DELETE statements. Then, you use the execute() method to supply the values for those parameters.
- After you execute an INSERT, UPDATE, or DELETE statement, you need to *commit* the changes by calling the commit() method from the connection object. If you don't, the changes aren't saved in the database.

Figure 17-12 How to execute INSERT, UPDATE, and DELETE statements

How to test the database code

To give you a better idea of how the Python skills that you've just learned work, figure 17-13 presents an example that works with the movies database. To start, this code imports the sqlite3 module for working with a SQLite database. Then, it imports the closing() function that automatically closes the cursor object when you're done with it.

The next statement connects to the movies database. Here, the code only specifies the name of the file for the database, not its complete path. As a result, the database file must be in the working directory. So, if you use IDLE to open the db_tester.py file and run it, it should connect to the movies database.

This is followed by a statement that sets the row factory attribute for the connection object. As a result, the statements that follow can use names instead of indexes to refer to the columns in the result sets.

The next block of code executes a SELECT statement that gets all columns from the Movie table for all movies that are less than 90 minutes long. Then, it uses a loop to print the name, year, and minutes for these movies to the console. After the SELECT statement, this code executes an INSERT statement that adds a new movie to the database, and a DELETE statement that removes that movie from the database.

When you're new to database programming, experimenting with code like this is a good way to get started. As you experiment, you will not only see how the statements work, but also what errors can occur.

How to handle database exceptions

When you write the code that works with a database, the database may throw exceptions if it can't execute the code successfully. For example, SQLite raises an exception if you try to execute a query on a table that isn't in the database, or if you attempt to add a row to a table that has the same primary key as another row in the table.

However, unlike a traditional database like MySQL or SQL Server, SQLite doesn't raise exceptions for some types of operations that may result in bad data such as orphaned keys. For example, SQLite doesn't raise an exception if you attempt to insert a row that has an invalid foreign key value, or if you attempt to delete a row that has a foreign key value that's being used by other rows.

If you need to handle database exceptions, you can use try statements to handle the exceptions just as you would handle any other exception. In this figure, for example, the code that executes the SELECT statement may raise an OperationalError. That might happen, for example, if the connect() method doesn't find the movies.sqlite database. In that case, SQLite creates a new database that's empty and doesn't have a Movie table. Then, the except clause prints an error message and sets the movies variable to None. That way, the code that follows can check whether the SELECT statement executed successfully.

Before you add exception handling to your programs, though, you should try to write your code so it avoids the operations that could throw exceptions. That's always a good practice, and that's illustrated by the program in the next figure.

Code that tests the database

```
# import the sqlite module and closing function
import sqlite3
from contextlib import closing

# connect to the database and set the row factory
conn = sqlite3.connect("movies.sqlite")
conn.row_factory = sqlite3.Row

# execute a SELECT statement - with exception handling
try:
    with closing(conn.cursor()) as c:
        query = '''SELECT * FROM Movie
                   WHERE minutes < ?'''
        c.execute(query, (90,))
        movies = c.fetchall()
except sqlite3.OperationalError as e:
    print("Error reading database -", e)
    movies = None

# display the results
if movies != None:
    for movie in movies:
        print(movie["name"], "|", movie["year"], "|", movie["minutes"])
    print()

# execute an INSERT statement
name = "A Fish Called Wanda"
year = 1988
minutes = 108
categoryID = 1
with closing(conn.cursor()) as c:
    sql = '''INSERT INTO Movie (name, year, minutes, categoryID)
             VALUES (?, ?, ?, ?)'''
    c.execute(sql, (name, year, minutes, categoryID))
    conn.commit()
print(name, "inserted.")

# execute a DELETE statement
with closing(conn.cursor()) as c:
    sql =  '''DELETE FROM Movie
              WHERE name = ?'''
    c.execute(sql, (name,))
    conn.commit()
print(name, "deleted.")
```

The console

```
Spirit: Stallion of the Cimarron | 2002 | 83
Ice Age | 2002 | 81
Toy Story | 1995 | 81

A Fish Called Wanda inserted.
A Fish Called Wanda deleted.
```

Figure 17-13 How to test the database code

The Movie List program

In chapter 7, you learned how to code a Movie List program that stores its data in a text file. However, a real-world program would store this information in a database. That's why this chapter finishes by presenting a three-tier version of the Movie List program that stores its data in a SQLite database.

The user interface

Figure 17-14 begins with the user interface for another version of the Movie List program. In contrast to the Movie List program in chapter 7, though, this program groups movies by category and stores more data for each movie. It also stores its data in a SQLite database.

When this program starts, it displays a list of commands followed by a list of categories that includes the IDs for the categories. Then, the users have to use these IDs as they use some of the commands. For instance, the cat command requires the ID of the category so it can display just the movies in that category. And the add command requires a category ID that specifies the type of movie that's being added.

The business tier

The business tier of the Movie List program defines two business objects: a Movie object and a Category object. For simplicity, both of these objects use public attributes to store their data. These attributes correspond to the columns of the Movie and Category tables in the movies database.

The Movie class begins with a constructor that assigns default values to all of its public attributes. Here, the category attribute is assigned a default value of None. This indicates that this attribute is designed to store a Category object, not an ID for the category.

The Category class works like the Movie class. However, it only provides two public attributes: id and name.

In both classes, the constructors provide default values for all of the arguments. This isn't necessary, but this makes these classes easier to use. For example, when the code adds a movie to the database, the Movie object doesn't need to have an ID because the database generates the ID automatically. However, when the code creates a Movie object from data that it retrieves from the database, it needs to set the ID to the value that's stored in the database. By providing default values for all of the arguments, you provide for both cases.

The user interface

```
The Movie List program

COMMAND MENU
cat   - View movies by category
year  - View movies by year
add   - Add a movie
del   - Delete a movie
exit  - Exit program

CATEGORIES
1. Animation
2. Comedy
3. History

Command: add
Name: The Lion King
Year: 1994
Minutes: 89
Category ID: 1
The Lion King was added to database.

Command: cat
Category ID: 1

MOVIES - ANIMATION
ID  Name                                    Year   Mins  Category
-------------------------------------------------------------------
1   Spirit: Stallion of the Cimarron        2002   83    Animation
2   Spirited Away                           2001   125   Animation
3   Aladdin                                 1992   90    Animation
4   Ice Age                                 2002   81    Animation
5   Toy Story                               1995   81    Animation
14  The Lion King                           1994   89    Animation

Command: exit
Bye!
```

The objects module for the business tier

```
class Movie:
    def __init__(self, id=0, name=None, year=0, minutes=0, category=None):
        self.id = id
        self.name = name
        self.year = year
        self.minutes = minutes
        self.category = category

class Category:
    def __init__(self, id=0, name=None):
        self.id = id
        self.name = name
```

Figure 17-14 The Movie List program (part 1)

The database tier

Part 2 of figure 17-14 presents the code for the database tier, which is stored in a module named db. To begin, this code imports the modules, functions, and classes that this code uses. This includes the Category and Movie classes from the objects module. Then, it sets a global variable for a connection object to None. This is followed by six functions.

The connect() function sets the global conn variable to the connection object. To do that, the first statement declares that the conn variable is global, which is required if this variable is going to be modified. Then, the first if statement creates a new connection object if the conn variable doesn't already contain one.

The if statement within the outer if statement is needed because the database file isn't in the same directory as the program file. As a result, the nested if statement sets a different path to the database file depending on the operating system that the program is running on. Once that's done, the connection object is created and the row factory attribute is set to allow the use of column names when accessing the result sets.

The close() function that follows closes the connection. To do that, this function first checks if the conn variable refers to a connection object. If so, it calls the close() method of the conn variable. Because this code only accesses the conn object and doesn't change it, it doesn't need to declare the conn variable as global.

The make_category() function is a utility function that accepts a row from a result set as an argument. This function contains a single line of code that creates a Category object from the data that's in the row and returns that object. This function is called by the three functions that follow

The make_movie() function works similarly, but it creates a Movie object from a row. However, the category attribute of the Movie object stores a Category object. As a result, this code calls the make_category() function to create a new Category object that's set as the category attribute of the Movie object.

The get_categories() function returns a list of all of the available categories in the database. To do that, it creates a query that selects the category ID and name from the Category table, using an alias of categoryName for the column that contains the name of the category. This is necessary because the make_category() function uses this alias to get the name of the category. After defining the query, this function executes the query and fetches all of the results. Then, it creates an empty list called categories, and loops over each row in the the result set. Within this loop, the code appends a Category object for each row to the list. To do that, it calls the make_category() function.

The get_category() function works similarly, but it only gets one Category object that corresponds with a single category ID. After this function accepts an argument for the category ID, it uses the fetchone() method to fetch the row for the specified category. Then, it calls the make_category() function to make a Category object from the row, and it returns the Category object.

The db module for the database tier

```
import sys
import os
import sqlite3
from contextlib import closing

from objects import Category
from objects import Movie

conn = None

def connect():
    global conn
    if not conn:
        if sys.platform == "win32":
            DB_FILE = "/murach/python/_db/movies.sqlite"
        else:
            HOME = os.environ["HOME"]
            DB_FILE = HOME + "/Documents/murach/python/_db/movies.sqlite"
        conn = sqlite3.connect(DB_FILE)
        conn.row_factory = sqlite3.Row

def close():
    if conn:
        conn.close()

def make_category(row):
    return Category(row["categoryID"], row["categoryName"])

def make_movie(row):
    return Movie(row["movieID"], row["name"], row["year"], row["minutes"],
            make_category(row))

def get_categories():
    query = '''SELECT categoryID, name as categoryName
               FROM Category'''
    with closing(conn.cursor()) as c:
        c.execute(query)
        results = c.fetchall()
        categories = []
        for row in results:
            categories.append(make_category(row))
        return categories

def get_category(category_id):
    query = '''SELECT categoryID, name AS categoryName
               FROM Category WHERE categoryID = ?'''
    with closing(conn.cursor()) as c:
        c.execute(query, (category_id,))
        row = c.fetchone()

    category = make_category(row)
    return category
```

Figure 17-14 The Movie List program (part 2)

The get_movies_by_category() function accepts a category ID as an argument. Then, it creates a query that joins the Movie and Category tables on the categoryID column that's in both tables. Because both tables have a column named name, this code uses an alias of categoryName for the name column of the Category table. This prevents a name collision between the name columns that are in both the Movie and Category tables.

After defining the query, this code executes the query and fetches all of the rows in the result set. Then, it creates an empty list named movies, and loops over each row in the result set. Within the loop, the code uses the make_movie() function to create a Movie object from each row, and it appends that object to the movies list. After the loop, this function returns the list of Movie objects.

The get_movies_by_year function works similarly, except that it selects the movie by year instead of by category_id. As a result, the only difference between these two functions is the WHERE clause in the SELECT statement, and the argument that the code supplies to the execute() method of the cursor object.

The add_movie() function accepts a Movie object as an argument. Within this function, the first statement creates a parameterized SQL statement that adds a row to the Movie table. Then, this code executes the SQL statement and uses the Movie object to provide the parameters for the SQL statement. Finally, this function commits the changes to the database.

The delete_movie() function accepts a movie ID as an argument. Within this function, the first statement creates a SQL statement that deletes a row from the Movie table. Then, this code executes the SQL statement with the movie ID of the record to be deleted as the argument. This function also finishes by committing the changes to the database.

The db module for the database tier (continued)

```
    def get_movies_by_category(category_id):
        query = '''SELECT movieID, Movie.name, year, minutes,
                          Movie.categoryID as categoryID,
                          Category.name as categoryName
                   FROM Movie JOIN Category
                          ON Movie.categoryID = Category.categoryID
                   WHERE Movie.categoryID = ?'''
        with closing(conn.cursor()) as c:
            c.execute(query, (category_id,))
            results = c.fetchall()

        movies = []
        for row in results:
            movies.append(make_movie(row))
        return movies

    def get_movies_by_year(year):
        query = '''SELECT movieID, Movie.name, year, minutes,
                          Movie.categoryID as categoryID,
                          Category.name as categoryName
                   FROM Movie JOIN Category
                          ON Movie.categoryID = Category.categoryID
                   WHERE year = ?'''
        with closing(conn.cursor()) as c:
            c.execute(query, (year,))
            results = c.fetchall()

        movies = []
        for row in results:
            movies.append(make_movie(row))
        return movies

    def add_movie(movie):
        sql = '''INSERT INTO Movie (categoryID, name, year, minutes)
                 VALUES (?, ?, ?, ?)'''
        with closing(conn.cursor()) as c:
            c.execute(sql, (movie.category.id, movie.name, movie.year,
                            movie.minutes))
            conn.commit()

    def delete_movie(movie_id):
        sql = '''DELETE FROM Movie WHERE movieID = ?'''
        with closing(conn.cursor()) as c:
            c.execute(sql, (movie_id,))
            conn.commit()
```

Figure 17-14 The Movie List program (part 3)

The presentation tier

Part 4 of figure 17-14 presents the code for the presentation tier, which is stored in the ui module. This class begins by importing the db module and by importing the Movie class from the objects module. Then, the first two functions display a title and a command menu. This is followed by four more functions.

The display_categories() function prints a list of categories, along with their IDs. To do that, this function calls the get_categories() function from the db module to get a list of Category objects. Then, it loops over each Category object in the list and print its id and name to the console. Here, the str() function converts the category ID from an integer to a string.

The display_movies() function accepts an argument for a list of Movie objects and an argument for a title term. Within this function, the first statement prints a title that begins with "MOVIES -" and ends with the title term. The second statement defines a format string that's used to align the data in columns. The third statement prints the column names for the result set. And the fourth statement prints a separator line that's 64 characters long. Then, the code loops through each Movie object in the list. Within this loop, the statement aligns and prints the data that's stored in each Movie object.

The display_movies_by_category() function prompts the user to enter a category ID and converts the entry to an int value. Then, it passes the category ID to the get_category() function of the db module, which returns a Category object for the specified category. Next, an if statement checks whether a Category object has been returned. If so, it passes the category ID to the get_movies_by_category() function of the db module, which returns a list of Movie objects. Last, this code passes this list and the uppercase name of the specified category to the display_movies() function.

The display_movies_by_year() function works much like the display_movies_by_category() function. However, it accepts the movie's year as an argument instead of the category. Then, it passes this argument to the get_movies_by_year() function of the db module. This returns a list of Movie objects. Finally, this code calls the display_movies() function and passes it the list of Movie objects and the year argument after it has been converted to a string.

The ui module for the presentation tier

```
#!/usr/bin/env/python3

import db
from objects import Movie

def display_title():
    print("The Movie List program")
    print()
    display_menu()

def display_menu():
    print("COMMAND MENU")
    print("cat   - View movies by category")
    print("year  - View movies by year")
    print("add   - Add a movie")
    print("del   - Delete a movie")
    print("exit  - Exit program")
    print()

def display_categories():
    print("CATEGORIES")
    categories = db.get_categories()
    for category in categories:
        print(str(category.id) + ". " + category.name)
    print()

def display_movies(movies, title_term):
    print("MOVIES - " + title_term)
    line_format = "{:3s} {:37s} {:6s} {:5s} {:10s}"
    print(line_format.format("ID", "Name", "Year", "Mins", "Category"))
    print("-" * 64)
    for movie in movies:
        print(line_format.format(str(movie.id), movie.name,
                                 str(movie.year), str(movie.minutes),
                                 movie.category.name))
    print()

def display_movies_by_category():
    category_id = int(input("Category ID: "))
    category = db.get_category(category_id)
    if category == None:
        print("There is no category with that ID.\n")
    else:
        print()
        movies = db.get_movies_by_category(category_id)
        display_movies(movies, category.name.upper())

def display_movies_by_year():
    year = int(input("Year: "))
    print()
    movies = db.get_movies_by_year(year)
    display_movies(movies, str(year))
```

Figure 17-14 The Movie List program (part 4)

The add_movie() function begins by prompting the user for the data for a new movie including its name, year, minutes, and category ID. Then, this code calls the get_category() function of the db module to get the Category object for the specified category ID. Next, an if statement checks to make sure a Category object has been returned. If it hasn't been returned, an error message is displayed to the user and the function ends.

If it has been returned, this code creates a Movie object and passes it the appropriate data, including the Category object. But note that this code doesn't assign a value to the id attribute of the Movie object. This isn't necessary since the database generates the ID value automatically when it adds the movie to the database.

After the Movie object has been created, this code calls the add_movie() function of the db module and passes it the new Movie object. Then, it displays a message that indicates that the movie was added.

The delete_movie() function is much simpler. To start, it prompts the user for a movie ID. Then, it calls the delete_movie() function of the db module with the movie ID as the argument, and it prints a message that indicates that the movie was deleted. This code could be improved by first making sure that the ID entered by the user is actually in the database table. But when you're using SQLite, if the ID isn't in the table, nothing is done.

The main() function begins by calling the connect() function in the db module to open a connection to the database. Then, it displays the title, the command menu, and a list of categories, including the category IDs.

After displaying this information, the function enters a loop where it waits for the user to enter a command. Once the user enters a command, this code calls the appropriate function depending on the command entered by the user. For example, if the user enters the cat command, the code calls the display_movies_by_category() function. But if the user enters the exit command, this code executes a break statement, which causes the loop to end. Then, this code calls the close() function of the db module to close the database connection. Last, it prints a message that indicates that the program is exiting.

The ui module for the presentation tier (continued)

```
def add_movie():
    name        = input("Name: ")
    year        = int(input("Year: "))
    minutes     = int(input("Minutes: "))
    category_id = int(input("Category ID: "))

    category = db.get_category(category_id)
    if category == None:
        print("There is no category with that ID. Movie NOT added.\n")
    else:
        movie = Movie(name=name, year=year, minutes=minutes,
                      category=category)
        db.add_movie(movie)
        print(name + " was added to database.\n")

def delete_movie():
    movie_id = int(input("Movie ID: "))
    db.delete_movie(movie_id)
    print("Movie ID " + str(movie_id) + " was deleted from database.\n")

def main():
    db.connect()
    display_title()
    display_categories()
    while True:
        command = input("Command: ")
        if command == "cat":
            display_movies_by_category()
        elif command == "year":
            display_movies_by_year()
        elif command == "add":
            add_movie()
        elif command == "del":
            delete_movie()
        elif command == "exit":
            break
        else:
            print("Not a valid command. Please try again.\n")
            display_menu()
    db.close()
    print("Bye!")

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

Figure 17-14 The Movie List program (part 5)