# 11

- Базы данных DBM
- Базы данных SQL

# Программирование приложений баз данных

Большинство разработчиков программного обеспечения под термином *база данных* подразумевают СУРБД (система управления реляционными базами данных). Для хранения данных эти системы используют таблицы (по своему строению подобные электронным таблицам), строки которых соответствуют записям, а столбцы – полям. Манипулирование данными, хранящимися в этих таблицах, производится с помощью инструкций, написанных на языке SQL (Structured Query Language – язык структурированных запросов). Язык Python включает в себя API (Application Programming Interface – прикладной программный интерфейс) для работы с базами данных SQL и обычно распространяется с поддержкой базы данных SQLite 3.

Существует еще одна разновидность баз данных – *DBM* (Database Manager – система управления базами данных), в которой данные хранятся в виде произвольного числа элементов ключ-значение. В стандартной библиотеке Python имеется несколько интерфейсов для работы с разными реализациями DBM, включая характерные для операционной системы UNIX. Базы данных DBM работают по принципу словарей в языке Python, за исключением того, что обычно они хранятся на диске, а не в памяти, а их ключами и значениями всегда являются объекты типа `bytes`, размер которых может ограничиваться. В первом разделе этой главы рассматривается модуль `shelve`, предоставляющий удобный интерфейс DBM, позволяя использовать строковые ключи и любые (поддающиеся консервированию) объекты в качестве значений.

Если имеющихся в наличии баз данных DBM и SQLite окажется недостаточно, в каталоге пакетов Python Package Index, *pypi.python.org/pypi*, можно найти множество пакетов, предназначенных для работы

с различными базами данных, включая bsddb DBM («Berkeley DB»), объектно-реляционные отображения, такие как SQLAlchemy (*www.sqlalchemy.org*), и интерфейсы к популярным клиент/серверным базам данных, таким как DB2, Informix, Ingres, MySQL, ODBC и PostgreSQL.

В этой главе мы реализуем две версии программы ведения списка фильмов на дисках DVD, в котором будут храниться название фильма, год выхода, продолжительность в минутах и имя режиссера. Первая версия программы для хранения информации использует базу данных DBM (с помощью модуля `shelve`), а вторая версия – базу данных SQLite. Обе программы могут также загружать и сохранять информацию в простом формате XML, обеспечивая возможность, например, экспортировать сведения о фильмах из одной программы и импортировать их в другую. Версия, использующая базу данных SQL, обладает немного более широкими возможностями, чем версия, использующая базу данных DBM, и имеет более ясную организацию.

## Базы данных DBM

Модуль `shelve` представляет собой обертку вокруг DBM, позволяя нам взаимодействовать с базой данных DBM, как с обычным словарем, в котором в качестве ключей допускается использовать только строки, а в качестве значений – объекты, допускающие возможность консервирования. За кулисами модуль `shelve` преобразует ключи и значения в объекты типа `bytes` и обратно.

Тип данных bytes, стр. 344

Поскольку модуль `shelve` основан на использовании лучшей из доступных баз данных DBM, есть вероятность, что файл DBM, сохраненный на одной машине, не будет читаться на другой, если на другой машине отсутствует поддержка той же самой DBM. Наиболее типичное решение такой проблемы состоит в том, чтобы обеспечить возможность импорта и экспорта данных в формате XML для файлов, которые должны быть переносимыми с машины на машину. Именно это мы и реализуем в программе *dvds-dbm.py* в этом разделе.

В качестве ключей мы будем использовать названия фильмов на дисках DVD, а в качестве значений – кортежи, в которых будут храниться имя режиссера, год и продолжительность фильма. Благодаря модулю `shelve` нам не придется выполнять каких-либо преобразований данных, и мы можем воспринимать объект DBM как обычный словарь.

Так как по своей структуре программа похожа на интерактивные программы, управляемые с помощью меню, которые мы уже видели ранее, мы сосредоточимся исключительно на аспектах, связанных с программированием DBM. Ниже приводится фрагмент из функции `main()`, где был опущен программный код, выполняющий обработку меню:

```
db = None
try:
    db = shelve.open(filename, protocol=pickle.HIGHEST_PROTOCOL)
    ...
finally:
    if db is not None:
        db.close()
```

Здесь открывается (или создается, если он еще не существует) указанный файл DBM в режиме для чтения и для записи. Значение каждого элемента сохраняется в файле в виде объекта, законсервированного с использованием указанного протокола консервирования. Существующие элементы можно будет прочитать, даже если они были сохранены с использованием меньшего номера протокола, поскольку интерпретатор в состоянии определять правильный номер протокола при чтении законсервированных объектов. В конце функции файл DBM закрывается, в результате происходит очистка внутреннего кэша DBM, производится запись всех изменений на диск и собственно закрытие файла.

Программа предоставляет возможность добавлять, редактировать, просматривать, импортировать и экспортировать данные. Мы пропустим процедуры импортирования и экспортирования данных в формате XML, поскольку они очень похожи на те, что мы рассматривали в главе 7. Точно так же мы опустим большую часть программного кода реализации пользовательского интерфейса, кроме операции добавления, потому что мы видели его прежде в других контекстах.

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    db[title] = (director, year, duration)
    db.sync()
```

Этой функции, как и любой другой, вызываемой из меню программы, передается в качестве единственного параметра объект DBM (`db`). Большая часть функции связана с получением данных о диске DVD и только в последней строке производится сохранение элемента ключ-значение в файле DBM, где в качестве ключа используется название фильма, а в качестве значения – кортеж с именем режиссера, годом выпуска и продолжительностью (который консервируется средствами модуля `shelve`).

Для сохранения непротиворечивости, свойственной языку Python, механизмы DBM предоставляют тот же самый API, что и словари, поэтому нам не придется осваивать новый синтаксис помимо функции `shelve.open()`, которую мы уже видели выше, и метода `shelve.Shelf.sync()`, который используется для очистки внутреннего кэша модуля `shelve` и синхронизации данных, находящихся в дисковом файле с последними изменениями, – в данном случае просто добавляется новый элемент.

```
def edit_dvd(db):
    old_title = find_dvd(db, "edit")
    if old_title is None:
        return
    title = Console.get_string("Title", "title", old_title)
    if not title:
        return
    director, year, duration = db[old_title]
    ...
    db[title] = (director, year, duration)
    if title != old_title:
        del db[old_title]
    db.sync()
```

Чтобы отредактировать сведения о диске, пользователь должен сначала выбрать диск, с которым он будет работать. Для этой операции требуется получить только название, так как названия служат ключами, а значения хранят остальные данные. Необходимая для этого функциональность будет востребована и в других местах (например, при удалении DVD), поэтому мы вынесли ее в отдельную функцию `find_dvd()`, которую мы рассмотрим следующей. Если диск найден, мы получаем от пользователя изменения, используя существующие значения как значения по умолчанию, чтобы повысить скорость взаимодействия. (Мы опустили большую часть программного кода, выполняющего взаимодействие с пользователем, так как в большинстве своем он остался тем же, что используется в функции добавления нового диска.) В конце мы сохраняем данные точно так же, как и в функции добавления. Если название не изменялось, эта операция будет иметь эффект перезаписи значения, ассоциированного с ключом, но если название изменилось, будет создана новая пара ключ-значение, и в этом случае необходимо удалить оригинальный элемент.

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    while True:
        matches = []
        start = Console.get_string(message, "title")
        if not start:
            return None
        for title in db:
            if title.lower().startswith(start.lower()):
```

```
                matches.append(title)
        if len(matches) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(matches) == 1:
            return matches[0]
        elif len(matches) > DISPLAY_LIMIT:
            print("Too many dvds start with {0}; try entering "
                  "more of the title".format(len(matches)))
            continue
        else:
            for i, match in enumerate(sorted(matches, key=str.lower)):
                print("{0}: {1}".format(i + 1, match))
            which = Console.get_integer("Number (or 0 to cancel)",
                                        "number", minimum=1, maximum=len(matches))
            return matches[which - 1] if which != 0 else None
```

Чтобы упростить и максимально ускорить поиск названия, пользователю предлагается ввести один или несколько начальных символов названия. Получив начало названия, функция выполняет итерации по данным в DBM и создает список найденных совпадений. Если имеется всего одно совпадение, оно возвращается, а если имеется несколько совпадений (но не более чем целочисленное значение `DISPLAY_LIMIT`, которое устанавливается где-то в другом месте программы), то они выводятся в алфавитном порядке без учета регистра символов, с порядковыми номерами перед ними, чтобы пользователь мог сделать выбор простым вводом числа. (Функция `Console.get_integer()` принимает 0, даже если значение аргумента `minimum` больше нуля, благодаря чему значение 0 может использоваться как признак отмены операции. Эту особенность поведения можно отключить, для чего достаточно передать аргумент `allow_zero=False`. Мы не можем использовать для отмены простое нажатие клавиши Enter, потому что ввод пустой строки рассматривается как ввод значения по умолчанию.)

```
def list_dvds(db):
    start = ""
    if len(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
    print()
    for title in sorted(db, key=str.lower):
        if not start or title.lower().startswith(start.lower()):
            director, year, duration = db[title]
            print("{0} ({1}) {2} minute{3}, by {4}".format(
                  title, year, duration, Util.s(duration), director))
```

Вывод списка всех дисков (или только тех, названия которых начинаются с определенной подстроки) реализуется простым обходом всех элементов в базе данных.

Функция `Util.s()` определена как `s = lambda x: "" if x == 1 else "s"`; здесь она возвращает символ «s», если продолжительность фильма превышает одну минуту.

```
def remove_dvd(db):
    title = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Remove {0}?".format(title), "no")
    if ans:
        del db[title]
        db.sync()
```

Удаление диска заключается в том, чтобы отыскать диск, который пользователь желает удалить, запросить подтверждение и в случае его получения выполнить удаление элемента из DBM.

Теперь мы знаем, как открыть (или создать) файл DBM с помощью модуля `shelve`, как добавлять в него элементы, редактировать элементы, выполнять итерации по элементам и удалять элементы.

К сожалению, в нашей базе данных имеется один недостаток. Имена режиссеров могут повторяться, что легко может приводить к несоответствиям, например, имя режиссера Danny DeVito для одного фильма может быть введено, как «Danny De Vito», а для другого, как «Danny deVito». Одно из решений этой проблемы состоит в том, чтобы создавать два файла DBM. Главный – с названиями в качестве ключей и значениями (год, продолжительность, идентификатор режиссера) и файл с режиссерами, ключами в котором являются идентификаторы режиссеров (например, целые числа), а значениями – имена. Мы устраним этот недостаток в следующем разделе, где версия программы, работающей с базой данных SQL, будет использовать две таблицы: в одной будет храниться информация о дисках, а во второй – о режиссерах.

## Базы данных SQL

Интерфейсы к наиболее популярным базам данных SQL доступны в виде модулей сторонних разработчиков, а по умолчанию в составе Python поставляется модуль `sqlite3` (и база данных SQLite 3), поэтому к созданию приложений баз данных можно приступать сразу же. База данных SQLite – это облегченная база данных SQL, в которой отсутствуют многие особенности, которые имеются, например, в PostgreSQL, но ее очень удобно использовать для создания прототипов, и во многих случаях предоставляемых ею возможностей оказывается вполне достаточно.

С целью упростить миграцию с одной базы данных на другую, в PEP 249 (Python Database API Specification v2.0) дается спецификация API, которая называется DB-API 2.0, которой должны следовать интерфейсы к базам данных; модуль `sqlite3`, к примеру, следует этой специфи-

кации, но не все модули сторонних разработчиков соблюдают ее. Спецификацией API определяются два основных объекта – объект соединения и объект курсора, а API, который они должны поддерживать, приводится в табл. 11.1 и в табл. 11.2. В случае с модулем `sqlite3` его объекты соединения и курсора предоставляют множество дополнительных атрибутов и методов сверх требований, предъявляемых спецификацией DB-API 2.0.

Версия программы, использующая базу данных SQL, находится в файле *dvds-sql.py*. Программа предусматривает хранение имен режиссеров отдельно от информации о дисках, чтобы избежать повторений, и предлагает дополнительный пункт меню, дающий пользователю получить список режиссеров. Структура двух таблиц показана на рис. 11.1. Размер этой программы составляет чуть меньше 300 строк, тогда как размер программы *dvds-dbm.py* из предыдущего раздела составляет чуть меньше 200 строк. Такое различие в основном обусловлено необходимостью использовать запросы SQL вместо простых операций со словарем, а также необходимостью создавать таблицы в базе данных при первом запуске программы.

*Таблица 11.1. Методы объекта соединения в соответствии со спецификацией DB-API 2.0*

| Синтаксис | Описание |
|---|---|
| `db.close()` | Закрывает соединение с базой данных (представленной объектом `db`, который возвращается вызовом функции `connect()`) |
| `db.commit()` | Подтверждает любую, ожидающую подтверждения, транзакцию в базе данных и ничего не делает, если база данных не поддерживает транзакции |
| `db.cursor()` | Возвращает объект курсора базы данных, посредством которого могут выполняться запросы |
| `db.rollback()` | Откатывает любую, ожидающую подтверждения, транзакцию в базе данных до состояния, в котором база данных находилась на момент начала транзакции, и ничего не делает, если база данных не поддерживает транзакции |

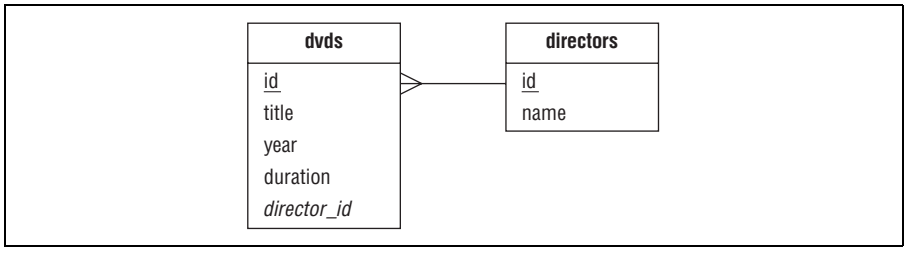


***Рис. 11.1.** Структура базы данных дисков DVD*

*Таблица 11.2. Методы и атрибуты объекта курсора в соответствии со спецификацией DB-API 2.0*

| Синтаксис | Описание |
|---|---|
| `c.arraysize` | Число строк (доступно для чтения/записи), которое будет возвращено методом `fetchmany()`, если аргумент `size` не определен |
| `c.close()` | Закрывает курсор `c` — эта операция выполняется автоматически, когда поток управления покидает область видимости курсора |
| `c.description` | Последовательность (только для чтения), представленная кортежем из 7 элементов (`name, type_code, display_size, internal_size, precision, scale, null_ok`), описывающая очередной столбец курсора `c` |
| `c.execute(sql, params)` | Выполняет запрос SQL, находящийся в строке `sql`, замещая каждый символ-заполнитель соответствующим параметром из последовательности или отображения *params*, если имеется |
| `c.executemany (sql, seq_of_params)` | Выполняет запрос SQL по одному разу для каждого элемента в последовательности последовательностей или отображений *seq_of_params*; этот метод не должен использоваться для выполнения операций, создающих наборы результатов (таких как инструкции `SELECT`) |
| `c.fetchall()` | Возвращает последовательность всех строк, которые еще не были извлечены (это могут быть все строки) |
| `c.fetchmany(size)` | Возвращает последовательность строк (каждая строка сама по себе является последовательностью); по умолчанию аргумент *size* принимает значение `c.arraysize` |
| `c.fetchone()` | Возвращает следующую строку из набора результатов, полученных в результате запроса, или `None`, если все результаты были исчерпаны. Возбуждает исключение, если набор результатов отсутствует |
| `c.rowcount` | Доступный только для чтения счетчик строк для последней операции (такой как `SELECT`, `INSERT`, `UPDATE` или `DELETE`) или -1, если счетчик недоступен или не имеет смысла |

Функция `main()` напоминает одноименную функцию из предыдущей версии программы, только на этот раз она вызывает нашу функцию `connect()`, чтобы установить соединение с базой данных.

```
def connect(filename):
    create = not os.path.exists(filename)
    db = sqlite3.connect(filename)
    if create:
        cursor = db.cursor()
        cursor.execute("CREATE TABLE directors ("
            "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
```

```
                "name TEXT UNIQUE NOT NULL)")
            cursor.execute("CREATE TABLE dvds ("
                "id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL, "
                "title TEXT NOT NULL, "
                "year INTEGER NOT NULL, "
                "duration INTEGER NOT NULL, "
                "director_id INTEGER NOT NULL, "
                "FOREIGN KEY (director_id) REFERENCES directors)")
            db.commit()
        return db
```

Функция `sqlite3.connect()` возвращает объект базы данных, открывает указанный файл базы данных или создает пустой файл базы данных, если файл не существует. Вследствие этого перед вызовом функции `sqlite3.connect()` необходимо проверить существование файла, чтобы знать, будет ли создаваться новая база данных, потому что в этом случае необходимо создать таблицы, которые используются программой. Все запросы к базе данных выполняются с помощью объекта курсора, который можно получить вызовом метода `cursor()` объекта базы данных.

Обратите внимание, что в обеих таблицах присутствует поле ID с ограничением `AUTOINCREMENT` – это означает, что SQLite автоматически будет записывать в поля ID уникальные числа, поэтому при добавлении новых записей можно переложить заботу о заполнении этих полей на SQLite.

База данных SQLite поддерживает ограниченный диапазон типов данных – по сути, только логические значения, числа и строки, но этот диапазон может быть расширен с помощью «адаптеров», либо предопределенных – для таких типов данных, как даты и время, либо создаваемых самостоятельно, которые могут использоваться для представления любых желаемых типов данных. В нашей программе этого не требуется, но в случае необходимости вы можете обратиться к описанию модуля `sqlite3`, где описываются все подробности. Синтаксис определения внешнего ключа, который мы использовали, возможно, не совпадает с синтаксисом определения внешнего ключа в других базах данных, но в любом случае – это просто описание наших намерений, поскольку база данных SQLite, в отличие от многих других, соблюдает ссылочную целостность. Еще одна особенность `sqlite3`, которая действует по умолчанию, заключается в неявной поддержке транзакций, поэтому в модуле отсутствует явный метод «запуска транзакции».

```
def add_dvd(db):
    title = Console.get_string("Title", "title")
    if not title:
        return
    director = Console.get_string("Director", "director")
    if not director:
        return
```

```
    year = Console.get_integer("Year", "year", minimum=1896,
                               maximum=datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor = db.cursor()
    cursor.execute("INSERT INTO dvds "
                   "(title, year, duration, director_id) "
                   "VALUES (?, ?, ?, ?)",
                   (title, year, duration, director_id))
    db.commit()
```

Эта функция начинается так же, как эквивалентная ей функция из программы *dvds-dbm.py*, отличия начинаются после сбора всей необходимой информации. Имя режиссера, введенное пользователем, может присутствовать, а может отсутствовать в таблице `directors`, поэтому мы предусмотрели функцию `get_and_set_director()`, которая добавляет имя режиссера, если оно еще отсутствует в базе данных, и в любом случае возвращает его идентификатор для вставки в таблицу `dvds`. Собрав все данные, функция выполняет инструкцию `INSERT` языка SQL. Здесь не требуется указывать идентификатор записи, потому что база данных SQLite подставит его автоматически.

Знаки вопроса в тексте запроса используются в качестве символов-заполнителей. Каждый знак `?` замещается соответствующим значением из последовательности, следующей за строкой с инструкцией SQL. Имеется также возможность использовать именованные символы-заполнители, она будет продемонстрирована в функции редактирования записи. Несмотря на то, что существует возможность отказаться от использования символов-заполнителей простым форматированием строки SQL, внедряя в нее необходимые данные, тем не менее мы рекомендуем всегда использовать символы-заполнители, а бремя корректного кодирования и экранирования служебных символов в элементах данных перекладывать на модуль базы данных. Еще одно преимущество использования символов-заполнителей состоит в том, что они повышают уровень безопасности, предотвращая возможность инъекции в запрос злонамеренного кода SQL.

```
def get_and_set_director(db, director):
    director_id = get_director_id(db, director)
    if director_id is not None:
        return director_id
    cursor = db.cursor()
    cursor.execute("INSERT INTO directors (name) VALUES (?)",
                   (director,))
    db.commit()
    return get_director_id(db, director)
```

Эта функция возвращает идентификатор указанного имени режиссера и в случае необходимости добавляет новую запись в таблицу `directors`.

Если была добавлена новая запись, идентификатор режиссера извлекается с помощью повторного вызова функции `get_director_id()`.

```
def get_director_id(db, director):
    cursor = db.cursor()
    cursor.execute("SELECT id FROM directors WHERE name=?",
                   (director,))
    fields = cursor.fetchone()
    return fields[0] if fields is not None else None
```

Функция `get_director_dvd()` возвращает идентификатор для заданного имени режиссера или `None`, если такого имени в базе данных не существует. Здесь используется метод `fetchone()`, потому что для данного имени может существовать либо одна запись, либо ни одной. (Дубликатов записей не может существовать, потому что поле `name` в таблице `directors` имеет ограничение `UNIQUE`, и в любом случае, прежде чем добавить новое имя режиссера в таблицу, мы проверяем его существование.) Методы извлечения записей всегда возвращают последовательность полей (или `None`, если все записи были извлечены), даже если, как в данном случае, выполняется попытка извлечь единственное поле.

```
def edit_dvd(db):
    title, identity = find_dvd(db, "edit")
    if title is None:
        return
    title = Console.get_string("Title", "title", title)
    if not title:
        return
    cursor = db.cursor()
    cursor.execute("SELECT dvds.year, dvds.duration, directors.name "
                   "FROM dvds, directors "
                   "WHERE dvds.director_id = directors.id AND "
                   "dvds.id=:id", dict(id=identity))
    year, duration, director = cursor.fetchone()
    director = Console.get_string("Director", "director", director)
    if not director:
        return
    year = Console.get_integer("Year", "year", year, 1896,
                               datetime.date.today().year)
    duration = Console.get_integer("Duration (minutes)", "minutes",
                                   duration, minimum=0, maximum=60*48)
    director_id = get_and_set_director(db, director)
    cursor.execute("UPDATE dvds SET title=:title, year=:year, "
                   "duration=:duration, director_id=:director_id "
                   "WHERE id=:id", locals())
    db.commit()
```

Чтобы отредактировать запись с информацией о диске, ее сначала необходимо отыскать. Если запись будет найдена, пользователю предоставляется возможность изменить название фильма. Затем извлекаются значения остальных полей, которые будут использоваться в качест-

ве значений по умолчанию, чтобы минимизировать ввод с клавиатуры, так как пользователю достаточно будет просто нажать клавишу Enter, чтобы принять значение по умолчанию. В этой функции используются именованные символы-заполнители (в форме *:name*), и поэтому значения для них должны поставляться в виде отображения. Для инструкции SELECT был использован вновь созданный словарь, а для инструкции UPDATE — словарь, полученный вызовом функции locals(). В обоих случаях можно было бы создавать новый словарь, и тогда для инструкции UPDATE вместо вызова функции locals() можно было бы указать словарь dict(title=title, year=year, duration=duration, director_id=director_id, id=identity).

Как только у нас будут значения всех полей и пользователь внесет все необходимые изменения, из базы данных извлекается идентификатор режиссера (при этом в случае необходимости вставляется новая запись) и затем выполняется запись обновленных данных в базу. Мы предприняли упрощенный подход, выполняя обновление сразу всех полей записи, вместо того чтобы выявлять и обновлять только те, которые действительно изменились.

При использовании базы данных DBM название диска использовалось в качестве ключа, поэтому при изменении названия создавался новый элемент ключ-значение, а прежний элемент удалялся. В данном же случае запись имеет уникальный идентификатор (поле id), который определяется в момент ее добавления, поэтому можно без всяких ограничений изменять значение любого другого поля без необходимости выполнять какие-либо дополнительные действия.

```
def find_dvd(db, message):
    message = "(Start of) title to " + message
    cursor = db.cursor()
    while True:
        start = Console.get_string(message, "title")
        if not start:
            return (None, None)
        cursor.execute("SELECT title, id FROM dvds "
                       "WHERE title LIKE ? ORDER BY title",
                       (start + "%"))
        records = cursor.fetchall()
        if len(records) == 0:
            print("There are no dvds starting with", start)
            continue
        elif len(records) == 1:
            return records[0]
        elif len(records) > DISPLAY_LIMIT:
            print("Too many dvds ({0}) start with {1}; try entering "
                  "more of the title".format(len(records), start))
            continue
        else:
            for i, record in enumerate(records):
```

```
            print("{0}: {1}".format(i + 1, record[0]))
        which = Console.get_integer("Number (or 0 to cancel)",
                                    "number", minimum=1, maximum=len(records))
        return records[which - 1] if which != 0 else (None, None)
```

Эта функция играет ту же роль, что и функция `find_dvd()` из программы *dvds-dbm.py*, и возвращает кортеж из двух элементов (название, идентификатор диска) или (`None`, `None`) в зависимости от того, была ли найдена требуемая запись. Вместо того чтобы выполнять итерации по всем данным, здесь используется оператор шаблонного символа SQL (`%`), поэтому из базы данных извлекаются только необходимые записи. А поскольку ожидается, что число записей будет невелико, выполняется извлечение сразу всех записей в последовательность последовательностей. Если будет найдено более одной записи, соответствующей условию поиска, или не настолько много, чтобы все они одновременно не поместились на экране, функция выводит их, предваряя каждую запись порядковым номером, чтобы пользователь смог сделать выбор вводом числа, как это делалось в программе *dvds-dbm.py*.

```
def list_dvds(db):
    cursor = db.cursor()
    sql = ("SELECT dvds.title, dvds.year, dvds.duration, "
           "directors.name FROM dvds, directors "
           "WHERE dvds.director_id = directors.id")
    start = None
    if dvd_count(db) > DISPLAY_LIMIT:
        start = Console.get_string("List those starting with "
                                   "[Enter=all]", "start")
        sql += " AND dvds.title LIKE ?"
    sql += " ORDER BY dvds.title"
    print()
    if start is None:
        cursor.execute(sql)
    else:
        cursor.execute(sql, (start + "%",))
    for record in cursor:
        print("{0[0]} ({0[1]}) {0[2]} minutes, by {0[3]}".format(
              record))
```

Чтобы получить сведения о каждом диске, создается запрос `SELECT`, выполняющий объединение двух таблиц. Если число записей в базе данных (возвращается функцией `dvd_count()`) оказывается больше, чем может поместиться на экране, в предложение `WHERE` добавляется второй элемент, вводящий дополнительное ограничение. Затем запрос выполняется и производится обход всех записей в результирующем наборе данных. Каждая запись представляет собой последовательность, в которой порядок следования полей определяется запросом `SELECT`.

```
def dvd_count(db):
    cursor = db.cursor()
```

```
    cursor.execute("SELECT COUNT(*) FROM dvds")
    return cursor.fetchone()[0]
```

Эти строки программного кода были оформлены в виде отдельной функции, потому что они требуются в нескольких функциях.

Мы опустили программный код функции `list_directors()`, так как по своей структуре она очень похожа на функцию `list_dvds()`, только гораздо проще, потому что она выводит список записей, состоящих из единственного поля (`name`).

```
def remove_dvd(db):
    title, identity = find_dvd(db, "remove")
    if title is None:
        return
    ans = Console.get_bool("Remove {0}?".format(title), "no")
    if ans:
        cursor = db.cursor()
        cursor.execute("DELETE FROM dvds WHERE id=?", (identity,))
        db.commit()
```

Эта функция вызывается, когда пользователь выбирает операцию удаления записи, и она очень похожа на эквивалентную функцию в программе *dvds-dbm.py*.

На этом мы заканчиваем обзор программы *dvds-sql.py*, и теперь мы знаем, как создавать таблицы в базе данных, как выбирать записи, как выполнять итерации по выбранным записям, а также как вставлять, изменять и удалять записи. С помощью метода `execute()` можно выполнять любые инструкции SQL, какие только поддерживаются используемой базой данных.

База данных SQLite обладает гораздо более широкими возможностями, чем было использовано здесь, включая режим автоматического подтверждения транзакций (и другие операции управления транзакциями), и возможность создавать функции, которые могут выполняться внутри запросов SQL. Имеется также возможность создавать фабричные функции, управляющие представлением возвращаемых записей (например, в виде словарей или собственных типов данных, вместо последовательностей полей). Дополнительно имеется возможность создавать базы данных SQLite в памяти, для чего достаточно передать строку `":memory"` в качестве имени файла.

## В заключение

В главе 7 были продемонстрированы различные способы сохранения на диск и загрузки данных с диска, и в этой главе было показано, как можно взаимодействовать с типами данных, сохраняющих информацию на диске, а не в памяти.

В случае использования файлов DBM очень удобным в использовании оказывается модуль `shelve`, так как он позволяет хранить элементы данных в виде строка-объект. В случае необходимости иметь более полный контроль имеется возможность прямого взаимодействия с используемыми базами данных DBM. Одна замечательная особенность баз данных DBM вообще и модуля `shelve` в частности состоит в том, что они используют API словаря, обеспечивая простоту получения, добавления, редактирования и удаления элементов, и позволяют легко перевести программу, применяющую словари, на использование DBM. Одно маленькое неудобство применения DBM заключается в том, что в случае реляционных данных каждая таблица элементов ключ-значение должна храниться в отдельном файле DBM, тогда как база данных SQLite хранит все данные в одном файле.

База данных SQLite прекрасно подходит для разработки прототипов программ, которые будут работать с базами данных SQL, и во многих случаях она может использоваться как основная база данных, особенно благодаря тому, что она включается в состав дистрибутива Python. В этой главе было продемонстрировано, как получать объект базы данных с помощью функции `connect()` и как выполнять запросы SQL (такие как `CREATE TABLE`, `SELECT`, `INSERT`, `UPDATE` и `DELETE`) с помощью метода `execute()` объекта курсора.

Язык Python предлагает широкий диапазон средств хранения данных на диске или в памяти, начиная от двоичных файлов, файлов XML и законсервированных объектов и заканчивая базами данных DBM и SQL, что позволяет выбирать нужное средство в зависимости от ситуации.

## Упражнение

Напишите интерактивную консольную программу обслуживания списка закладок. Для каждой закладки должны храниться два элемента данных: адрес URL и имя. Ниже приводится пример сеанса работы с программой:

```
Bookmarks (bookmarks.dbm)
(1) Programming in Python 3..... http://www.qtrac.eu/py3book.html
(2) PyQt........................ http://www.riverbankcomputing.com
(3) Python...................... http://www.python.org
(4) Qtrac Ltd................... http://www.qtrac.eu
(5) Scientific Tools for Python.... http://www.scipy.org

(A)dd (E)dit (L)ist (R)emove (Q)uit [l]: e
Number of bookmark to edit: 2
URL [http://www.riverbankcomputing.com]:
Name [PyQt]: PyQt (Python bindings for GUI library)
```

Программа должна давать пользователю возможность добавлять, редактировать, выводить список и удалять закладки. Чтобы обеспечить максимальную простоту выбора закладки при выполнении операций

редактирования и удаления, закладки должны выводиться в виде списка с порядковыми номерами и пользователю должна предлагаться возможность ввести номер закладки для редактирования или удаления. Данные должны сохраняться в файле DBM с применением модуля `shelve`, где имена должны играть роль ключей, а адреса URL – значений. По своей структуре программа очень похожа на программу *dvds-dbm.py*, за исключением функции `find_bookmark()`, которая получится намного проще, чем функция `find_dvd()`, так как она будет получать от пользователя только порядковый номер закладки и выполнять поиск имени по этому номеру.

В качестве дополнительного удобства, если пользователь явно не указывает протокол, добавляемые или редактируемые адреса URL следует предварять строкой `http://`.

Вся программа может уместиться менее чем в 100 строк (предполагается, что будут использоваться функции `Console.get_string()` и подобные ей из модуля `Console`). Решение приводится в файле *bookmarks.py*.