17

Базы данных и постоянное хранение

«Дайте мне приказ стоять до конца, но сохранить данные»

До сих пор в этой книге мы использовали Python в системном программировании, для разработки графических интерфейсов и создания сценариев для Интернета – трех наиболее типичных областях применения Python, где он наиболее ярко проявляется как прикладной язык программирования в целом. В следующих четырех главах мы бросим беглый взгляд на другие важные темы программирования на языке Python: постоянное хранение данных, приемы работы со структурами данных, обработку текста и интеграцию языков Python/C.

Все эти темы имеют отношение не к прикладному программированию как таковому, а к смежным областям. Например, знания о базах данных, полученные в этой главе, можно применять при создании веб-приложений, при разработке настольных приложений с графическим интерфейсом и так далее. Умение обрабатывать текст также является универсальным навыком. Кроме того, хотя ни одна из четырех тем не освещается исчерпывающим образом (каждой из них вполне можно посвятить отдельную книгу), мы представим примеры работы Python в этих областях и подчеркнем основные идеи и инструменты. Если какая-либо из этих глав вызовет у вас интерес, то следует обратиться к дополнительным источникам.

Возможности постоянного хранения данных в Python

В этой главе наше внимание будет сосредоточено на *постоянно хранящихся* данных, которые продолжают существовать после завершения создавшей их программы. По умолчанию это не так для объектов, создаваемых сценариями: такие объекты, как списки, словари и даже экземпляры классов, находятся в памяти компьютера и исчезают, как только сценарий завершает работу. Чтобы заставить данные жить дольше, требуется предпринять особые меры. В программировании на языке Python есть по крайней мере шесть традиционных способов сохранения информации между запусками программы:

Плоские файлы

Обеспечивают хранение текста и байтов непосредственно на компьютере

Файлы DBM

Обеспечивают доступ к строкам, хранящимся в файлах, напоминающих словари, по ключу

Сериализованные объекты

Сериализованные объекты могут сохраняться в файлах и потоках

Файлы хранилищ (shelve)

Обеспечивают хранение сериализованных объектов в файлах DBM

Объектно-ориентированные базы данных (OODB)

Обеспечивают сохранение объектов в хранилищах, имеющих структуру словарей (ZODB, Durus)

Реляционные базы данных SQL (RDBMS)

Хранилища в виде таблиц, поддерживающие запросы SQL (SQLite, MySQL, PostgreSQL и другие)

Объектно-реляционные отображения (ORM)

Промежуточные механизмы, обеспечивающие отображение классов Python в реляционные таблицы (SQLObject, SQLAlchemy)

В некотором смысле интерфейсы Python к сетевым протоколам передачи объектов, таким как SOAP, XML-RPC и CORBA, также предоставляют возможность сохранения данных, но их описание выходит далеко за рамки этой главы. Интерес для нас здесь представляют приемы, позволяющие программам сохранять данные непосредственно и, обычно, на локальном компьютере. Некоторые серверы баз данных могут действовать на удаленном компьютере в сети, однако в целом это не имеет значения для большинства приемов, которые мы будем изучать здесь.

Мы изучали интерфейсы Python простых (или «плоских») файлов в главе 4 и с того момента пользуемся ими. Python предоставляет стандарт-

ный доступ к файловой системе `stdio` (через встроенную функцию `open`), а также на низком уровне – через дескрипторы файлов (с помощью встроенного модуля `os`). Для простых задач хранения данных многим сценариям ничего другого не требуется. Чтобы сохранить данные для использования при следующих запусках программы, нужно записать их в открытый файл на компьютере, а потом прочитать их обратно из этого файла. Как мы видели, в более сложных задачах Python поддерживает также другие интерфейсы, сходные с файлами, такие как каналы, очереди и сокеты.

Так как мы уже знакомы с плоскими файлами, я не буду больше рассказывать здесь о них. Оставшаяся часть главы знакомит с другими темами из списка, приведенного в начале раздела. В конце мы также познакомимся с программой с графическим интерфейсом для просмотра содержимого файлов хранилищ и файлов DBM. Но прежде нам нужно разобраться, что это за звери.

Примечание к четвертому изданию: В предыдущем издании этой книги использовался интерфейс `mysql-python` к системе управления реляционными базами данных MySQL, а также система управления объектно-ориентированными базами данных ZODB. Когда я занимался обновлением этой главы в июне 2010 года, ни один из них еще не был доступен в Python 3.X – версии Python, используемой в этом издании. По этой причине большая часть информации о ZODB была убрана из главы, а примеры работы с базами данных SQL были переориентированы на использование интерфейса SQLite к базе данных внутри процесса, который входит в состав стандартной библиотеки Python 3.X. Примеры использования ZODB и MySQL и обзоры из предыдущего издания по-прежнему доступны в пакете примеров к книге, как будет описано ниже. Однако благодаря переносимости интерфейса к базам данных SQL в языке Python, программный код, использующий SQLite, практически без изменений сможет работать с большинством других баз данных.

Файлы DBM

Плоские файлы удобно использовать для простых задач постоянного хранения данных, но обычно они связаны с последовательным режимом обработки. Несмотря на возможность произвольно перемещаться по файлам с помощью вызовов метода `seek`, плоские файлы мало что вносят в структуру данных помимо понятий байтов и текстовых строк.

Файлы DBM, стандартный инструмент в библиотеке Python для управления базами данных, улучшают это положение, предоставляя доступ к хранящимся строкам текста по ключу. Они реализуют представление хранящихся данных с произвольным доступом и одним ключом. Например, информация, относящаяся к объектам, может храниться в файле DBM с использованием уникального ключа для каждого объекта

и позднее может быть получена обратно с помощью того же самого ключа. Файлы DBM реализуются с помощью целого ряда базовых модулей (в том числе одного, написанного на языке Python), но если у вас есть Python, значит, есть и поддержка DBM.

Работа с файлами DBM

Хотя файловые системы DBM должны проделать некоторую работу по отображению сохраняемых данных в ключи для быстрого извлечения (технически для сохранения данных в файлах они обычно используют прием, называемый *хешированием*), сценариям не приходится беспокоиться о том, что происходит за кулисами. В действительности файлы DBM являются одним из простейших способов сохранения информации в Python – файлы DBM ведут себя настолько сходно со словарями, размещаемыми в памяти, что можно забыть о том, что в самом деле вы работаете с файлом. Например, если есть объект файла DBM:

- Операция индексирования по ключу извлекает данные из файла.
- Операция присвоения по индексу сохраняет данные в файле.

Объекты файлов DBM поддерживают также стандартные методы словарей, такие как выборка и проверка по списку ключей и удаление по ключу. Сама библиотека DBM скрыта за этой простой моделью. Ввиду простоты перейдем прямо к интерактивному примеру, в котором создается файл DBM и демонстрируется, как работает интерфейс:

```
C:\...\PP4E\Dbase> python
>>> import dbm                  # получить интерфейс: bsddb, gnu, ndbm, dumb
>>> file = dbm.open('movie', 'c')  # создать файл DBM с именем 'movie'
>>> file['Batman'] = 'Pow!'        # сохранить строку с ключом 'Batman'
>>> file.keys()                    # получить список ключей в файле
[b'Batman']
>>> file['Batman']                 # извлечь значение по ключу 'Batman'
b'Pow!'

>>> who = ['Robin', 'Cat-woman', 'Joker']
>>> what = ['Bang!', 'Splat!', 'Wham!']
>>> for i in range(len(who)):
...     file[who[i]] = what[i]     # добавить еще 3 "записи"
...
>>> file.keys()
[b'Cat-woman', b'Batman', b'Joker', b'Robin']
>>> len(file), 'Robin' in file, file['Joker']
(4, True, b'Wham!')
>>> file.close()                   # иногда требуется закрывать явно
```

На практике при импорте стандартного модуля `dbm` автоматически загружается тот интерфейс DBM, который доступен вашему интерпретатору Python (опробуются различные альтернативы в строго определенном порядке), а при открытии нового файла DBM создается один или более внешних файлов с именами, начинающимися со строки `'movie'`

(о деталях будет сказано чуть ниже). Но после импорта и открытия файл DBM фактически неотличим от словаря.

В сущности, объект с именем `file` в этом примере можно представить себе как словарь, отображаемый во внешний файл с именем `movie`. Единственные очевидные отличия состоят в том, что ключами могут быть только строки (а не произвольные неизменяемые объекты), и для доступа к данным необходимо открывать, а после изменения данных – закрывать файл.

Однако в отличие от обычных словарей, содержимое `file` сохраняется в перерыве между запусками программы Python. Если мы потом вернемся и заново запустим Python, наш словарь будет по-прежнему доступен. Файлы DBM похожи на словари, которые требуется открывать:

```
C:\...\PP4E\Dbase> python
>>> import dbm
>>> file = dbm.open('movie', 'c')  # открыть существующий файл DBM
>>> file['Batman']
b'Pow!'

>>> file.keys()                    # метод keys возвращает список ключей
[b'Cat-woman', b'Batman', b'Joker', b'Robin']

>>> for key in file.keys(): print(key, file[key])
...
b'Cat-woman' b'Splat!'
b'Batman' b'Pow!'
b'Joker' b'Wham!'
b'Robin' b'Bang!'
```

Обратите внимание, что при обращении к методу `keys` файлов DBM возвращается действительный список – здесь не показано, но метод `values` этих файлов, напротив, возвращает итерируемое представление, как при обращении к обычным словарям. Кроме того, файлы DBM всегда хранят ключи и значения в виде объектов `bytes` – интерпретация их в виде текста Юникода остается за клиентским приложением. При обращении к хранимым ключам или значениям в нашем программном коде мы можем использовать строки `bytes` или `str` – использование строк `bytes` позволяет сохранять ключи и значения в произвольных кодировках Юникода, тогда как при использовании объектов `str` они будут кодироваться в объекты `bytes` внутренней реализацией DBM с применением кодировки UTF-8.

Однако при необходимости мы всегда можем декодировать строки `bytes` в строки `str` для отображения, и подобно словарям файлы DBM имеют итератор по ключам. Кроме того, операции присваивания и удаления по ключу изменяют содержимое файла DBM, и после внесения изменений необходимо закрывать файл (это гарантирует сохранение изменений на диске):

```
>>> for key in file: print(key.decode(), file[key].decode())
...
Cat-woman Splat!
Batman Pow!
Joker Wham!
Robin Bang!

>>> file['Batman'] = 'Ka-Boom!'  # изменить значение ключа Batman
>>> del file['Robin']            # удалить запись Robin
>>> file.close()                 # закрыть после изменений
```

За исключением необходимости импортировать интерфейс и открывать/закрывать файл DBM, программам на языке Python не требуется ничего знать собственно о DBM. Модули DBM добиваются такой интеграции, перегружая операции индексирования и переадресуя их более простым библиотечным инструментам. Но глядя на этот программный код, вы никогда бы этого не узнали – файлы DBM выглядят как обычные словари Python, хранящиеся во внешних файлах. Произведенные в них изменения сохраняются неограниченно долго:

```
C:\...\PP4E\Dbase> python
>>> import dbm                   # открыть файл DBM еще раз
>>> file = dbm.open('movie', 'c')
>>> for key in file: print(key.decode(), file[key].decode())
...
Cat-woman Splat!
Batman Ka-Boom!
Joker Wham!
```

Как видите, проще уже некуда. В табл. 17.1 перечислены наиболее частые операции с файлами DBM. Если открыт такой файл, он обрабатывается так, как если бы был словарем Python, находящимся в памяти. Выборка элементов осуществляется индексированием объекта файла по ключу, а запись – путем присвоения по ключу.

Таблица 17.1. Операции с файлами DBM

Программный код Python	Действие	Описание
`import dbm`	Импорт	Получить реализацию DBM
`file=dbm.open('filename', 'c')`	Открытие	Создать или открыть существующий файл DBM для ввода-вывода
`file['key'] = 'value'`	Запись	Создать или изменить запись с ключом `key`
`value = file['key']`	Выборка	Загрузить значение из записи с ключом `key`
`count = len(file)`	Размер	Получить количество записей
`index = file.keys()`	Индекс	Получить список (не представление) ключей

Программный код Python	Действие	Описание
`found = 'key' in file`	Запрос	Проверить наличие записи с ключом `key`
`del file['key']`	Удаление	Удалить запись с ключом `key`
`for key in file:`	Итерации	Выполнить итерации по имеющимся ключам
`file.close()`	Закрытие	Закрыть вручную, требуется не всегда

Особенности DBM: файлы, переносимость и необходимость закрытия

Несмотря на интерфейс, напоминающий словари, файлы DBM в действительности отображаются в один или более внешних файлов. Например, при использовании интерфейса `dbm` в Python 3.1 для Windows создаются два файла – *movie.dir* и *movie.dat* – когда создается файл DBM с именем `movie`, а при последующих операциях открытия создается еще файл *movie.bak*. Если Python имеет доступ к другому базовому интерфейсу файлов с доступом по ключу, на компьютере могут появиться другие внешние файлы.

Технически модуль `dbm` является интерфейсом к той файловой системе типа DBM, которая имеется в Python.

- При открытии существующего файла DBM модуль `dbm` пытается определить создавшую его систему с помощью функции `dbm.whichdb`. Вывод делается на основе содержимого самой базы данных.
- При создании нового файла `dbm` пытается загрузить интерфейсы доступа к файлам по ключу в строго определенном порядке. Согласно документации он пытается загрузить интерфейс `dbm.bsd`, `dbm.gnu`, `dbm.ndbm` или `dbm.dumb` и использовать первый, который будет благополучно загружен. При их отсутствии Python автоматически использует универсальную реализацию с именем `dbm.dumb`, которая, в действительности, конечно же не является «тупой» («dumb»), но не отличается такой производительностью и надежностью, как другие реализации.

В будущих версиях Python этот порядок выбора реализации может измениться, и могут даже появиться дополнительные альтернативы. Однако обычно о таких вещах не приходится беспокоиться, если не удалять файлы, которые создает ваша система DBM, или не перемещать их между компьютерами с различными конфигурациями. Если вас волнует проблема *переносимости* файлов DBM (как будет показано дальше, то же самое относится к файлам хранилищ `shelve`), вам необходимо позаботиться о настройке компьютеров, чтобы на них были установлены одни и те же интерфейсы DBM, или положиться на интерфейс `dumb`. На-

пример, пакет поддержки Berkeley DB (он же bsddb), используемый интерфейсом dbm.bsd, достаточно широко распространен и обладает высокой степенью переносимости.

Обратите внимание, что файлы DBM иногда требуется закрывать явно, в формате, приведенном в последней строке в табл. 17.1. Некоторые файлы DBM не требуют вызова метода закрытия, но другим он нужен, чтобы записать изменения из буфера на диск. В таких системах файл может оказаться поврежден, если не выполнить закрытие. К несчастью, используемая по умолчанию поддержка DBM в старых версиях Python для Windows, dbhash (она же bsddb), является как раз той системой DBM, которая требует вызова метода закрытия во избежание потери данных. Как правило, всегда следует закрывать файлы DBM явно после внесения изменений и перед выходом из программы, чтобы обойти возможные проблемы – по сути, это операция подтверждения изменений («commit»). Данное правило распространяется и на хранилища shelve, с которыми мы познакомимся далее в этой главе.

Последние изменения: Не забывайте также передавать строку 'c' во втором аргументе в вызов dbm.open, чтобы заставить интерпретатор создать файл, если он еще не существует. Прежде этот аргумент подразумевался по умолчанию, но теперь это не так. Аргумент 'c' не требуется передавать при открытии файлов-хранилищ, создаваемых модулем shelve, обсуждаемых далее, – для них по-прежнему по умолчанию используется режим 'c' «открыть или создать», если отсутствует аргумент, определяющий режим открытия. Модулю dbm можно передавать также другие строки, определяющие режим открытия, включая 'n', в котором всегда создается новый файл, и 'r', определяющий доступ к файлу только для чтения, – по умолчанию используется режим создания нового файла. За дополнительной информацией обращайтесь к руководству по стандартной библиотеке Python.

Кроме того, в Python 3.X все элементы, ключи и значения сохраняются как строки bytes, а не str, как мы уже видели (что, как оказывается, удобно при работе с сериализованными объектами в хранилищах shelve, обсуждаемых ниже). В этой версии интерпретатора более не доступен компонент bsddb, бывший ранее стандартным, однако он доступен как независимое стороннее расширение, которое можно загрузить из Интернета, а при его отсутствии Python переходит на использование собственной реализации файлов DBM. Поскольку правила выбора базовой реализации DBM могут изменяться со временем, вы всегда должны обращаться за дополнительной информацией к руководствам по библиотеке Python, а также к исходному программному коду стандартного модуля dbm.

Сериализованные объекты

Вероятно, наибольшим ограничением файлов DBM является тип хранимых в них данных: данные, записываемые под ключом, должны быть

простой текстовой строкой. При необходимости сохранять в файле DBM объекты Python иногда можно вручную преобразовывать их в строки и обратно при записи и чтении (например, с помощью функций `str` и `eval`), но это полностью не решает проблемы. Для объектов Python произвольной сложности, таких как экземпляры классов, требуется нечто другое. Объекты экземпляров классов, к примеру, обычно нельзя впоследствии воссоздать из стандартного строкового представления. Кроме того, при реализации собственных методов преобразования в строку и воссоздания объекта из строки легко допустить ошибку, и такие инструменты не являются универсальным решением.

Модуль Python `pickle`, входящий в стандартную поставку системы Python, обеспечивает требуемое преобразование. Это своего рода универсальный инструмент, осуществляющий прямое и обратное преобразование, – модуль `pickle` может преобразовывать практически любые объекты Python, находящиеся в памяти, в формат одной линейной строки, пригодной для хранения в плоских файлах, пересылки через сокеты по сети и так далее. Это преобразование объекта в строку часто называют преобразованием в последовательную форму, или *сериализацией* – произвольные структуры данных, размещенные в памяти, отображаются в последовательный строковый формат.

Строковое представление объектов иногда также называют потоком байтов в соответствии с его линейным форматом. При сериализации сохраняются содержимое и структура оригинального объекта в памяти. При последующем воссоздании объекта из такой строки байтов создается новый объект, идентичный оригиналу по структуре и содержимому, однако он будет размещен в другой области памяти.

В результате при воссоздании объекта фактически создается его *копия* – говоря на языке Python, соблюдается условие `==`, но не `is`. Поскольку воссоздание объектов происходит, как правило, в совершенно новом процессе, это отличие обычно не имеет большого значения (хотя, как мы видели в главе 5, именно это обстоятельство обычно препятствует использованию сериализованных объектов для непосредственной передачи информации о состоянии между процессами).

Операция сериализации может применяться практически к любым типам данных в языке Python – числам, спискам, словарям, экземплярам классов, вложенным структурам и другим – благодаря чему она является универсальным способом сохранения данных. Поскольку в последовательной форме сохраняются фактические объекты Python, в библиотеке отсутствует какой-либо прикладной интерфейс баз данных для них; объекты сохраняются и при последующем извлечении обрабатываются с применением обычного синтаксиса языка Python.

Применение сериализации объектов

Сериализация может показаться сложной, когда сталкиваешься с ней впервые, но отрадно узнать, что Python скрывает все сложности преоб-

разования объектов в строки. На самом деле интерфейсы модуля pickle чрезвычайно просты в употреблении. Например, чтобы преобразовать объект в последовательную форму, можно создать объект Pickler и вызывать его методы или использовать функции из модуля для достижения того же эффекта:

`P = pickle.Pickler(`*file*`)`

Создаст новый объект Pickler для вывода в последовательном виде в открытый выходной файл *file*.

`P.dump(`*object*`)`

Запишет объект *object* в файл/поток объекта Pickler.

`pickle.dump(`*object*`, `*file*`)`

То же, что два последних вызова вместе: сериализует объект *object* и выведет его в открытый файл *file*.

`string = pickle.dumps(`*object*`)`

Вернет объект *object* в сериализованном представлении, в виде строки символов.

Воссоздание объекта из строки с сериализованным представлением выполняется похожим образом; доступны простой функциональный и объектно-ориентированный интерфейсы:

`U = pickle.Unpickler(`*file*`)`

Создаст объект Unpickler, осуществляющий обратное преобразование сериализованной формы объекта из открытого входного файла *file*.

`object = U.load()`

Прочитает объект из файла/потока объекта Unpickler.

`object = pickle.load(`*file*`)`

То же, что два последних вызова вместе: восстановит объект из открытого файла.

`object = pickle.loads(`*string*`)`

Прочитает объект из строки символов вместо файла.

Pickler и Unpickler — это экспортируемые классы. Во всех этих вызовах аргумент file является либо объектом открытого файла, либо любым объектом, в котором реализованы те же атрибуты, что и у объектов файла:

- Объект Pickler вызывает метод write файла, передавая в качестве аргумента строку.
- Объект Unpickler вызывает метод read файла со счетчиком байтов и метод readline без аргументов.

Любой объект, имеющий эти атрибуты, может быть передан в качестве параметра file. В частности, аргумент file может быть экземпляром

класса Python, предоставляющего методы read/write (то есть, поддерживающим *интерфейс* файлов). Благодаря этому можно произвольно отображать сериализованные потоки в объекты, находящиеся в памяти с классами. Например, класс io.BytesIO, имеющийся в стандартной библиотеке и обсуждавшийся в главе 3, предоставляет интерфейс, отображающий обращения к файлам на строки байтов в памяти, благодаря чему его экземпляры могут служить альтернативой использованию строковых функций dumps/loads из модуля pickler.

Можно также пересылать объекты Python через сеть, заключая сокеты в оболочку, выглядящую как файл и вызывающую методы сериализации у отправителя и методы обратного преобразования у получателя (подробнее об этом рассказывается в разделе «Придание сокетам внешнего вида файлов и потоков ввода-вывода» в главе 12). Фактически передача сериализованных объектов Python по сети между доверенными узлами является более простой альтернативой использованию сетевых транспортных протоколов, таких как SOAP и XML-RPC, при условии что с обоих концов соединения находится Python (сериализованные объекты передаются не в виде XML, а в формате, специфическом для Python).

Последние изменения: В Python 3.X сериализованные объекты всегда представлены в виде строк bytes, а не str независимо от запрошенного уровня протокола (даже при запросе старейшего протокола ASCII возвращается строка байтов). Вследствие этого файлы, используемые для сохранения сериализованных объектов Python, всегда должны открываться в двоичном режиме. Кроме того, в 3.X также автоматически выбирается и используется оптимизированная реализация модуля _pickle, если она присутствует. Подробнее обе эти темы описываются ниже.

Сериализация в действии

Несмотря на то, что сериализованные объекты могут переправляться самыми необычными способами, тем не менее, в наиболее обычном случае для сериализации объекта в плоский файл нужно открыть файл в режиме записи и вызвать функцию dump:

```
C:\...\PP4E\Dbase> python
>>> table = {'a': [1, 2, 3],
             'b': ['spam', 'eggs'],
             'c': {'name':'bob'}}
>>>
>>> import pickle
>>> mydb = open('dbase', 'wb')
>>> pickle.dump(table, mydb)
```

Обратите внимание на наличие здесь вложенных объектов – объект Pickler способен обрабатывать объекты произвольной структуры. От-

метьте также, что файл открывается в двоичном режиме. В Python 3.X это является обязательным условием, потому что сериализованные объекты всегда представлены в виде строки bytes. Чтобы выполнить обратное преобразование в другом сеансе или при последующих запусках приложения, достаточно просто открыть файл и вызвать load:

```
C:\...\PP4E\Dbase> python
>>> import pickle
>>> mydb = open('dbase', 'rb')
>>> table = pickle.load(mydb)
>>> table
{'a': [1, 2, 3], 'c': {'name': 'bob'}, 'b': ['spam', 'eggs']}
```

Восстановленный объект имеет то же самое содержимое и ту же структуру, что и оригинал, но он создается в другой области памяти[1]. Это относится и к объектам, воссозданным в том же процессе, и к объектам, воссозданным в другом процессе. Напомню, что для воссозданного объекта выполняется условие ==, но не is:

```
C:\...\PP4E\Dbase> python
>>> import pickle
>>> f = open('temp', 'wb')
>>> x = ['Hello', ('pickle', 'world')] # список с вложенным кортежем
>>> pickle.dump(x, f)
>>> f.close()                          # закрыть, чтобы записать на диск
>>>
>>> f = open('temp', 'rb')
>>> y = pickle.load(f)
>>> y
['Hello', ('pickle', 'world')]
>>>
>>> x == y, x is y                     # то же значение, но разные объекты
(True, False)
```

Чтобы еще больше упростить этот процесс, модуль в примере 17.1 заключает вызовы прямого и обратного преобразований объектов в функции, которые также открывают файлы, хранящие сериализованную форму объекта.

Пример 17.1. PP4E\Dbase\filepickle.py

```
"Утилиты сохранения и восстановления объектов из плоских файлов"
import pickle

def saveDbase(filename, object):
    "сохраняет объект в файле"
    file = open(filename, 'wb')
    pickle.dump(object, file)  # сохранить в двоичный файл
```

[1] При некоторых условиях не исключена вероятность, что объект будет воссоздан в той же области памяти, но это совпадение будет чисто случайным. – *Прим. перев.*

```
    file.close()                # подойдет любой объект, похожий на файл

def loadDbase(filename):
    "загружает объект из файла"
    file = open(filename, 'rb')
    object = pickle.load(file)  # загрузить из двоичного файла
    file.close()                # воссоздать объект в памяти
    return object
```

Теперь, чтобы сохранить и извлечь объект, просто вызывайте функции из этого модуля. В примере ниже они используются для манипулирования довольно сложной структурой со множественными ссылками на одни и те же вложенные объекты – вложенный список с именем `L` сохраняется в файле в единственном экземпляре:

```
C:\...\PP4E\Dbase> python
>>> from filepickle import *
>>> L = [0]
>>> D = {'x':0, 'y':L}
>>> table = {'A':L, 'B':D}       # присутствуют две ссылки на список L
>>> saveDbase('myfile', table)   # сериализовать в файл

C:\...\PP4E\Dbase> python
>>> from filepickle import *
>>> table = loadDbase('myfile')  # загрузить/воссоздать
>>> table
{'A': [0], 'B': {'y': [0], 'x': 0}}
>>> table['A'][0] = 1            # изменить совместно используемый объект
>>> saveDbase('myfile', table)   # перезаписать в файл

C:\...\PP4E\Dbase> python
>>> from filepickle import *
>>> print(loadDbase('myfile'))   # изменились оба списка L, как и ожидалось
{'A': [1], 'B': {'y': [1], 'x': 0}}
```

Помимо встроенных типов, таких как списки, кортежи и словари, использовавшихся в примерах до сих пор, сериализовать можно также *экземпляры классов*. Тем самым обеспечивается естественный способ связать поведение с хранимыми данными (методы классов обрабатывают атрибуты экземпляров) и простой способ миграции (изменения в классе автоматически будут подхвачены хранимыми экземплярами). Ниже приводится короткая демонстрация в интерактивной оболочке:

```
>>> class Rec:
        def __init__(self, hours):
            self.hours = hours
        def pay(self, rate=50):
            return self.hours * rate

>>> bob = Rec(40)
>>> import pickle
>>> pickle.dump(bob, open('bobrec', 'wb'))
```

```
>>>
>>> rec = pickle.load(open('bobrec', 'rb'))
>>> rec.hours
40
>>> rec.pay()
2000
```

Принцип действия этого механизма мы подробно рассмотрим, когда будем исследовать хранилища, создаваемые модулем `shelve`, далее в этой главе – как мы увидим позже, модуль `pickle` может использоваться непосредственно, но он также является базовым механизмом баз данных `shelve` и ZODB.

В целом Python может сериализовать почти все, что угодно, за исключением:

- Объектов компилированного программного кода: функций и классов, когда при сериализации известны только их имена, без имен модулей, что не позволяет позднее повторно импортировать их и автоматически подхватить изменения в файлах модулей.
- Экземпляры классов, не выполняющие правила импортируемости: если говорить кратко, классы должны быть доступны для импортирования при загрузке объекта (подробнее об этом будет рассказываться ниже, в разделе «Файлы хранилищ shelve»).
- Экземпляров некоторых встроенных и определяемых пользователем типов, написанных на языке C или зависящих от преходящих состояний операционной системы (например, объекты открытых файлов не могут быть сериализованы).

Если объект не может быть сериализован, возбуждается исключение `PicklingError`. Напомню, что мы еще вернемся к проблеме сериализуемости объектов и классов, когда будем знакомиться с хранилищами, создаваемыми модулем `shelve`.

Особенности сериализации: протоколы, двоичные режимы и модуль _pickle

В последних версиях Python в операцию сериализации было введено понятие *протоколов* – форматов хранения сериализованных данных. Чтобы определить желаемый протокол, необходимо передать дополнительный параметр функциям сериализации (его не требуется передавать функциям обратного преобразования: протокол определяется автоматически по сериализованным данным):

```
pickle.dump(object, file, protocol) # или именованный аргумент protocol=N
```

Сериализация данных может быть выполнена с применением текстового или двоичного протокола – двоичный протокол позволяет получить более эффективный формат, но при этом создаются нечитаемые человеком файлы. По умолчанию в Python 3.X используется исключительно

двоичный формат (известный также, как протокол 3). В текстовом режиме (протокол 0) сериализованные данные представляют собой печатаемый текст ASCII, который может читаться человеком (по сути, он представляет собой последовательность инструкций для машины стека), но в Python 3.X в любом случае получается объект bytes. Другие протоколы (протоколы 1 и 2) также создают сериализованные данные в двоичном формате.

В Python 3.X независимо от номера протокола сериализованные данные представляют собой объект bytes, а не str, и именно поэтому при сохранении и чтении их в плоских файлах требуется использовать двоичный режим (причины описываются в главе 4, если вы забыли). Аналогично, имитируя интерфейс объектов файлов, мы должны использовать объекты bytes:

```
>>> import io, pickle
>>> pickle.dumps([1, 2, 3])                    # по умолчанию=двоич. протокол
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
>>> pickle.dumps([1, 2, 3], protocol=0)        # протокол формата ASCII
b'(lp0\nL1L\naL2L\naL3L\na.'

>>> pickle.dump([1, 2, 3], open('temp','wb')) # даже если protocol=0, ASCII
>>> pickle.dump([1, 2, 3], open('temp','w'))  # при чтении необх. режим 'rb'
TypeError: must be str, not bytes
>>> pickle.dump([1, 2, 3], open('temp','w'), protocol=0)
TypeError: must be str, not bytes

>>> B = io.BytesIO()                # использовать двоичные потоки/буферы
>>> pickle.dump([1, 2, 3], B)
>>> B.getvalue()
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'

>>> B = io.BytesIO()                # bytes и для формата ASCII
>>> pickle.dump([1, 2, 3], B, protocol=0)
>>> B.getvalue()
b'(lp0\nL1L\naL2L\naL3L\na.'

>>> S = io.StringIO()               # это не объект str
>>> pickle.dump([1, 2, 3], S)       # даже если protocol=0, ASCII
TypeError: string argument expected, got 'bytes'
>>> pickle.dump([1, 2, 3], S, protocol=0)
TypeError: string argument expected, got 'bytes'
```

За дополнительными сведениями о сериализации обращайтесь к руководству по библиотеке Python – там вы найдете описание дополнительных интерфейсов, которые могут использоваться классами для переопределения поведения этой операции, которые мы не будем рассматривать здесь ради экономии места. Обратите также внимание на модуль marshal, который тоже сериализует объекты, но может обрабатывать только простые типы объектов. Модуль pickle является более универсальным, чем модуль marshal, и обычно более предпочтителен.

Имеется еще один родственный модуль, _pickle, написанная на языке C оптимизация модуля pickle, который автоматически используется модулем pickle, если доступен – его не требуется выбирать вручную или использовать непосредственно. Модуль shelve наследует эту оптимизацию автоматически. Я еще не рассказывал о модуле shelve, но сейчас сделаю это.

Файлы shelve

Сериализация позволяет сохранять произвольные объекты в файлах и подобных им объектах, но это все же весьма неструктурированный носитель – он не обеспечивает непосредственного простого доступа к членам совокупностей сериализованных объектов. Можно добавлять структуры более высокого уровня, но они не являются внутренне присущими:

- Порой имеется возможность определить собственную организацию файлов сериализации более высокого уровня с помощью базовой файловой системы (например, можно записать каждый сериализованный объект в файл, имя которого уникально идентифицирует объект), но такая организация не является частью механизма сериализации и должна управляться вручную.
- Можно также записывать словари произвольно большого размера в файл сериализации и обращаться к ним по ключу после загрузки обратно в память, но при этом обратное восстановление из файла загружает весь словарь, а не только тот элемент, в котором мы заинтересованы.

Хранилища, создаваемые модулем shelve, позволяют определить некоторую структуру для совокупностей сериализованных объектов. В файлах этого типа, являющихся стандартной частью системы Python, произвольные объекты Python сохраняются по ключу для извлечения в дальнейшем. В действительности здесь не так много нового – файлы-хранилища являются простой комбинацией файлов DBM и объектов сериализации:

- Чтобы *сохранить* находящийся в памяти объект по ключу, модуль shelve сначала сериализует его в строку с помощью модуля pickle, а затем записывает эту строку в файл DBM по ключу с помощью модуля dbm.
- Чтобы *загрузить* обратно объект по ключу, модуль shelve сначала загружает по ключу строку с сериализованным объектом из файла DBM с помощью модуля dbm, а затем преобразует ее обратно в исходный объект с помощью модуля pickle.

Поскольку внутренняя реализация модуля shelve использует модуль pickle, она может сохранять те же объекты, что и pickle: строки, числа, списки, словари, рекурсивные объекты, экземпляры классов и другие. Поскольку внутренняя реализация модуля shelve использует модуль dbm,

она наследует все черты этого модуля, в том числе и его ограничения, касающиеся переносимости.

Использование хранилищ

Иными словами, модуль `shelve` служит всего лишь посредником – он сериализует и десериализует объекты, чтобы их можно было поместить в файлы DBM. В конечном итоге хранилища позволяют записывать в файлы по ключу почти любые объекты Python и позднее загружать их обратно по тому же ключу.

Однако сами сценарии никогда не видят всех этих взаимодействий. Подобно файлам DBM хранилища предоставляют интерфейс, напоминающий словарь, который нужно открыть. Фактически хранилища – это всего лишь постоянно хранимые словари с постоянно хранимыми объектами Python: содержимое словарей-хранилищ отображается в файлы на компьютере, благодаря чему они сохраняются между запусками программы. На словах все это звучит довольно сложно, но в программном коде оказывается просто. Чтобы получить доступ к хранилищу, импортируйте модуль и откройте свой файл:

```
import shelve
dbase = shelve.open("mydbase")
```

Модуль `shelve` откроет файл DBM с именем *mydbase* или создаст его, если он еще не существует (по умолчанию он использует режим `'c'` открытия файлов DBM). Операция присваивания по ключу сохраняет объект:

```
dbase['key'] = object  # сохранить объект
```

Внутренне эта операция преобразует объект в сериализованный поток байтов и запишет его по ключу в файл DBM. Обращение к хранилищу по ключу загружает сохраненный объект:

```
value = dbase['key']   # извлечь объект
```

Внутренне эта операция обращения по индексу загрузит по ключу строку из файла DBM и развернет ее в объект в памяти, совпадающий с исходным объектом. Здесь также поддерживается большинство операций со словарями:

```
len(dbase)             # количество хранящихся элементов
dbase.keys()           # список ключей хранящихся элементов
```

И, за исключением некоторых тонких моментов, это все, что касается использования модуля `shelve`. Хранилища объектов обрабатываются с использованием обычного синтаксиса словарей Python, поэтому не нужно изучать новый интерфейс базы данных. Более того, объекты, сохраняемые в хранилищах и извлекаемые из них, являются обычными объектами Python. Для сохранения не требуется, чтобы они были экземплярами особых классов или типов. То есть система постоянного

хранения объектов Python является внешней по отношению к самим сохраняемым объектам. В табл. 17.2 сведены вместе эти и другие часто используемые операции с хранилищами.

Таблица 17.2. Операции с файлами хранилищ

Программный код Python	Действие	Описание
`import shelve`	Импорт	Получить интерфейс `bsddb`, `gdbm` и так далее в зависимости от того, что установлено
`file=shelve.open('filename')`	Открытие	Создать или открыть существующий файл DBM хранилища
`file['key'] = anyvalue`	Запись	Создать или изменить запись с ключом `key`
`value = file['key']`	Выборка	Загрузить значение из записи с ключом `key`
`count = len(file)`	Размер	Получить количество записей
`index = file.keys()`	Индекс	Получить список ключей (итерируемое представление)
`found = 'key' in file`	Запрос	Проверить наличие записи с ключом `key`
`del file['key']`	Удаление	Удалить запись с ключом `key`
`for key in file:`	Итерации	Выполнить итерации по имеющимся ключам
`file.close()`	Закрытие	Закрыть вручную, требуется не всегда

Так как хранилища тоже экспортируют интерфейс, подобный словарю, эта таблица почти идентична таблице операций с файлами DBM. Однако здесь имя модуля `dbm` заменяется на `shelve`, вызовы `open` не требуют второго аргумента `'c'`, а сохраняемые значения могут быть объектами практически любых типов, а не просто строками. Однако ключами все еще могут быть только строки (технически ключами могут быть только объекты `str`, которые автоматически преобразуются в тип `bytes` и обратно с применением кодировки UTF-8), и для надежности по-прежнему необходимо явно закрывать хранилища после проведенных изменений: хранилища внутренне используют модуль `dbm`, а некоторые базовые модули поддержки DBM требуют выполнять закрытие, чтобы избежать потери или повреждения данных.

Последние изменения: Функции открытия хранилища в модуле `shelve` теперь принимают необязательный аргумент `writeback` – если в нем передать значение `True`, все записи будут кэшироваться в памяти и записываться обратно на диск только при выполнении опе-

рации закрытия. Это устраняет необходимость вручную повторно присваивать модифицированные изменяемые объекты, чтобы вытолкнуть их на диск, но может приводить к непроизводительным потерям при большом количестве записей – кэш может занять огромный объем памяти, и операция закрытия в таких случаях будет выполняться достаточно медленно, так как необходимо будет записать в файл все извлеченные записи (интерпретатор не способен определить, какие из записей изменялись).

Помимо того что значениями могут быть не только простые строки, но и любые объекты, интерфейс хранилищ в Python 3.X имеет еще два тонких отличия от интерфейса файлов DBM. Во-первых, метод `keys` возвращает итерируемый объект *представления* (а не физический список). Во-вторых, значениями *ключей* всегда являются строки типа `str`, а не `bytes` – при извлечении, записи, удалении и в других операциях используемые ключи типа `str` кодируются в строки `bytes`, которые ожидает получить реализация DBM, с использованием кодировки UTF-8. Это означает, что в отличие от модуля `dbm`, нельзя в качестве ключей хранилища `shelve` использовать строки `bytes`, чтобы использовать произвольные кодировки.

Кроме того, ключи хранилищ декодируются из типа `bytes` в тип `str` с помощью кодировки UTF-8 всякий раз, когда они возвращаются функциями модуля `shelve` (например, при выполнении итераций по ключам). Хранимые *значения* всегда представлены объектами `bytes`, создаваемыми модулем `pickle` при сериализации объектов. Мы увидим эти особенности в действии, далее в этом разделе.

Сохранение объектов встроенных типов в хранилищах

Запустим интерактивный сеанс и поэкспериментируем с интерфейсами хранилищ. Как уже отмечалось, хранилища, по сути, являются постоянно хранимыми словарями объектов, которые необходимо открывать и закрывать:

```
C:\...\PP4E\Dbase> python
>>> import shelve
>>> dbase = shelve.open("mydbase")
>>> object1 = ['The', 'bright', ('side', 'of'), ['life']]
>>> object2 = {'name': 'Brian', 'age': 33, 'motto': object1}

>>> dbase['brian'] = object2
>>> dbase['knight'] = {'name': 'Knight', 'motto': 'Ni!'}
>>> dbase.close()
```

Здесь мы открываем хранилище и сохраняем две довольно сложных структуры данных в виде словаря и списка, просто присваивая их ключам хранилища. Поскольку модуль `shelve` внутренне использует модуль `pickle`, здесь можно использовать почти все – деревья вложенных объектов автоматически сериализуются в строки для хранения. Чтобы загрузить их обратно, достаточно просто открыть хранилище и обратиться к индексу:

```
C:\...\PP4E\Dbase> python
>>> import shelve
>>> dbase = shelve.open("mydbase")
>>> len(dbase)                          # количество записей
2

>>> dbase.keys()                        # индекс
KeysView(<shelve.DbfilenameShelf object at 0x0181F630>)

>>> list(dbase.keys())
['brian', 'knight']

>>> dbase['knight']                     # извлечь
{'motto': 'Ni!', 'name': 'Knight'}

>>> for row in dbase.keys():            # использовать метод .keys() необяз.
...     print(row, '=>')
...     for field in dbase[row].keys():
...         print('  ', field, '=', dbase[row][field])
...
brian =>
   motto = ['The', 'bright', ('side', 'of'), ['life']]
   age = 33
   name = Brian
knight =>
   motto = Ni!
   name = Knight
```

Вложенные друг в друга циклы в конце этого сеанса выполняют обход вложенных словарей – внешний просматривает хранилище, а внутренний – объекты, хранящиеся в нем (в обоих можно было бы использовать итераторы по ключам и опустить вызовы `.keys()`). Важно отметить, что для записи и выборки этих постоянных объектов, так же как для их обработки после загрузки, используется обычный синтаксис Python. Это данные Python, постоянно хранящиеся на диске.

Сохранение экземпляров классов в хранилищах

Более полезным типом объектов, которые можно хранить в хранилищах, являются экземпляры классов. Поскольку в их атрибутах записывается состояние, а унаследованные методы определяют поведение, постоянно хранимые объекты классов, в сущности, выполняют роль как *записей* базы данных, так и *программ* обработки баз данных. Для сериализации и сохранения экземпляров классов в плоских файлах и в других подобным им объектах (например, в сетевых сокетах) можно также использовать базовый модуль `pickle`, но высокоуровневый модуль `shelve` обеспечивает также возможность использовать доступ к хранилищу по ключу. Рассмотрим простой класс, представленный в примере 17.2, с помощью которого моделируются записи с информацией о сотрудниках гипотетического предприятия.

Пример 17.2. PP4E\Dbase\person.py (версия 1)

```
"объект с информацией о сотруднике: поля + поведение"

class Person:
    def __init__(self, name, job, pay=0):
        self.name = name
        self.job  = job
        self.pay  = pay            # действительные данные экземпляра

    def tax(self):
        return self.pay * 0.25     # вычисляется при вызове

    def info(self):
        return self.name, self.job, self.pay, self.tax()
```

Ничто в этом классе не говорит о том, что его экземпляры будут использоваться в качестве записей в базе данных – его можно импортировать и использовать независимо от внешнего хранилища. Однако его удобно использовать для создания записей в базе данных: из этого класса можно создавать постоянно хранимые объекты, просто создавая экземпляры как обычно и сохраняя их по ключу в открытом хранилище shelve:

```
C:\...\PP4E\Dbase> python
>>> from person import Person
>>> bob = Person('bob', 'psychologist', 70000)
>>> emily = Person('emily', 'teacher', 40000)
>>>
>>> import shelve
>>> dbase = shelve.open('cast')  # создать новое хранилище
>>> for obj in (bob, emily):   # сохранить объекты
...     dbase[obj.name] = obj  # использовать значение name в качестве ключа
...
>>> dbase.close()              # необходимо для bsddb
```

Здесь в качестве ключей в базе данных мы использовали атрибуты `name` экземпляров объектов. Когда позднее мы снова загрузим эти объекты в интерактивном сеансе Python или сценарии, они воссоздадутся в памяти в том виде, какой у них был в момент сохранения:

```
C:\...\PP4E\Dbase> python
>>> import shelve
>>> dbase = shelve.open('cast')  # открыть хранилище
>>>
>>> list(dbase.keys())           # в хранилище присутствуют оба объекта
['bob', 'emily']
>>> print(dbase['emily'])
<person.Person object at 0x0197EF70>
>>>
>>> print(dbase['bob'].tax())    # вызов: метода tax объекта с именем bob
17500.0
```

Обратите внимание, что вызов метода `tax` для объекта с именем «Bob» работает, несмотря на то, что мы не импортировали класс `Person`. Python достаточно сообразителен, чтобы снова связать объект с исходным классом после восстановления из сериализованной формы и сделать доступными все методы загруженных объектов.

Изменение классов хранимых объектов

Технически Python повторно импортирует класс для воссоздания его сохраненных экземпляров при их загрузке и восстановлении. Ниже описано, как это действует:

Запись

Когда Python сериализует экземпляр класса, чтобы сохранить его в хранилище, он сохраняет атрибуты и ссылку на класс экземпляра. Фактически сериализованные экземпляры класса в предыдущем примере записывают атрибуты `self`, присваивание которым выполняется в классе. В действительности Python сериализует и записывает словарь атрибутов `__dict__` экземпляра вместе с информацией об исходном файле модуля класса, чтобы позднее иметь возможность отыскать модуль класса – имена класса экземпляров и модуля, вмещающего этот класс.

Выборка

Когда Python восстанавливает экземпляр класса, извлеченный из хранилища, он воссоздает в памяти объект экземпляра, повторно импортируя класс, используя сохраненные строки с именами класса и модуля; присваивает сохраненный словарь атрибутов новому пустому экземпляру класса и связывает экземпляр с классом. Эти действия выполняются по умолчанию, но имеется возможность изменить этот процесс, определив специальные методы, которые будут вызываться модулем `pickle` при извлечении и сохранении информации об экземпляре (за подробностями обращайтесь к руководству по библиотеке Python).

Главное здесь то, что класс и хранимые экземпляры отделены друг от друга. Сам класс не хранится вместе со своими экземплярами, а находится в исходном файле модуля Python и импортируется заново, когда загружаются экземпляры.

Недостаток этой модели заключается в том, что класс должен быть доступен для импортирования, чтобы обеспечить возможность загрузки экземпляров из хранилища (подробнее об этом чуть ниже). А преимущество в том, что, модифицировав внешние классы в файлах модулей, можно изменить способ, которым интерпретируются и используются данные сохраненных объектов, не изменяя сами хранящиеся объекты. Это похоже на то, как если бы класс был программой, обрабатывающей хранящиеся записи.

Для иллюстрации предположим, что класс Person из предыдущего раздела был изменен, как показано в примере 17.3.

Пример 17.3. PP4E\Dbase\person.py (версия 2)

```
"""
объект с информацией о сотруднике: поля + поведение
изменения: метод tax теперь является вычисляемым атрибутом
"""


class Person:
    def __init__(self, name, job, pay=0):
        self.name = name
        self.job = job
        self.pay = pay              # действительные данные экземпляра

    def __getattr__(self, attr):    # в person.attr
        if attr == 'tax':
            return self.pay * 0.30  # вычисляется при попытке обращения
        else:
            raise AttributeError()  # другие неизвестные атрибуты

    def info(self):
        return self.name, self.job, self.pay, self.tax
```

В этой версии устанавливается новая ставка налога (30%), вводится метод __getattr__ перегрузки операции доступа к атрибуту класса и убран оригинальный метод tax. Поскольку при загрузке экземпляров из хранилища будет импортирована эта новая версия класса, они автоматически приобретут новую реализацию поведения – попытки обращения к атрибуту tax будут перехватываться и в ответ будет возвращаться вычисленное значение:

```
C:\...\PP4E\Dbase> python
>>> import shelve
>>> dbase = shelve.open('cast')     # открыть хранилище
>>>
>>> print(list(dbase.keys()))       # в хранилище присутствуют оба объекта
['bob', 'emily']
>>> print(dbase['emily'])
<person.Person object at 0x019AEE90>
>>>
>>> print(dbase['bob'].tax)         # больше не требуется вызывать tax()
21000.0
```

Так как класс изменился, к атрибуту tax теперь можно обращаться как к простому свойству, не вызывая его как метод. Кроме того, поскольку ставка налога в классе изменена, Бобу придется на этот раз платить больше. Конечно, этот пример искусственный, но при правильном использовании такое разделение классов и постоянно хранимых экземпляров может избавить от необходимости использовать традиционные программы обновления баз данных – чтобы добиться нового поведения,

в большинстве случаев можно просто изменить класс, а не каждый хранящийся экземпляр.

Ограничения модуля shelve

Обычно работа с хранилищами не вызывает затруднений, однако существуют некоторые шероховатости, о которых следует помнить.

Ключи должны быть строками (str)

Во-первых, несмотря на то, что сохранять можно любые объекты, ключи все же должны быть строками. Следующая инструкция не будет выполнена, если сначала вручную не преобразовать целое число 42 в строку 42:

```
dbase[42] = value      # ошибка, но str(42) работает
```

Этим хранилища отличаются от словарей, размещаемых в памяти, которые допускают использование в качестве ключей любых неизменяемых объектов, и обусловлено использованием файлов DBM. Как мы уже видели, в Python 3.X ключи могут быть только строками `str`, а не `bytes`, потому что внутренняя реализация хранилищ во всех случаях пытается их кодировать.

Объекты уникальны только по ключу

Хотя модуль `shelve` достаточно сообразителен, чтобы обнаруживать множественные случаи вложенного объекта и воссоздавать только один экземпляр при загрузке, это относится только к данной ситуации:

```
dbase[key] = [object, object]  # OK: сохраняется и извлекается
                               # только одна копия

dbase[key1] = object
dbase[key2] = object           # плохо?: две копии объекта в хранилище
```

После извлечения объектов по ключам `key1` и `key2` они будут указывать на независимые копии оригинального общего объекта – если этот объект относится к категории изменяемых, изменения в одном из них не будут отражены в другом. В действительности это обусловлено тем, что каждое присвоение ключу запускает независимую операцию сериализации – механизм сериализации обнаруживает повторяющиеся объекты, но только в рамках одного обращения к модулю `pickle`. Это может не коснуться вас на практике и преодолевается введением дополнительной логики, но следует помнить, что объект может дублироваться, если он соответствует нескольким ключам.

Обновления должны выполняться в режиме «загрузить-модифицировать-сохранить»

Так как объекты, загруженные из хранилища, не знают, что они были извлечены оттуда, то операции, изменяющие части загруженного объ-

екта, касаются только экземпляра, находящегося в памяти, а не данных в хранилище:

```
dbase[key].attr = value        # данные в хранилище не изменились
```

Чтобы действительно изменить объект в хранилище, нужно загрузить его в память, изменить и записать обратно в хранилище целиком, выполнив присваивание по ключу:

```
object = dbase[key]  # загрузить
object.attr = value  # модифицировать
dbase[key] = object  # записать обратно – хранилище изменилось
                     # (если при открытии не был указан аргумент writeback)
```

Как отмечалось выше, если методу `shelve.open` передать необязательный аргумент `writeback`, то выполнение последнего шага здесь не потребуется, – за счет автоматического кэширования извлеченных объектов и сохранения их на диске при закрытии хранилища. Но это может повлечь за собой существенный расход памяти и замедлить операцию закрытия.

Возможность одновременных обновлений не поддерживается

В настоящее время модуль `shelve` не поддерживает возможность одновременного обновления. Одновременное чтение допускается, но для записи программа должна получить исключительный доступ к хранилищу. Хранилище можно разрушить, если несколько процессов будут выполнять запись в него одновременно, а это, например, в серверных сценариях может происходить часто. Если изменять данные в хранилищах может потребоваться сразу нескольким процессам, оберните операции обновления данных в вызов функции `os.open` из стандартной библиотеки, чтобы заблокировать доступ к файлу и обеспечить исключительный доступ.

Переносимость базового формата DBM

Для сохранения объектов модуль `shelve` создает файлы с помощью базовой системы DBM, которые необязательно будут совместимы со всеми возможными реализациями DBM или версиями Python. Например, файл, созданный с помощью `gdbm` в Linux или библиотекой `bsddb` в Windows, может не читаться при использовании Python, установленного с другими модулями DBM.

Это все та же проблема переносимости, которую мы рассматривали при обсуждении файлов DBM выше. Как вы помните, когда создается файл DBM (а соответственно и файл хранилища модуля `shelve`), модуль `dbm` пытается импортировать все возможные модули системы DBM в предопределенном порядке и использует первый найденный модуль. Когда позднее модуль `dbm` открывает существующий файл, он пытается определить по содержимому файла, какая система DBM использовалась при его создании. При создании файла сначала делается попытка использо-

вать систему bsddb, доступную в Windows и во многих Unix-подобных системах, поэтому ваш файл DBM будет совместим со всеми платформами, где установлена версия Python с поддержкой BSD. То же относится к платформам, где установлена версия Python, использующая собственную реализацию dbm.dumb при отсутствии поддержки других форматов DBM. Однако если для создания файла DBM использовалась система, недоступная на целевой платформе, использовать этот файл будет невозможно.

Если обеспечение переносимости файлов DBM имеет большое значение для вас, примите меры, чтобы все версии Python, под управлением которых будет выполняться чтение ваших данных, использовали совместимые модули DBM. Если это невозможно, используйте для сохранения данных модуль pickle и плоские файлы (в обход модулей shelve и dbm) или одну из объектно-ориентированных баз данных, с которыми мы познакомимся далее в этой главе. Такие базы данных зачастую способны предложить полную поддержку *транзакций* в виде методов подтверждения изменений и автоматической отмены в случае ошибки.

Ограничения класса Pickler

Помимо указанных ограничений хранилищ модуля shelve, сохранение экземпляров классов в таких хранилищах вводит ряд правил, о которых необходимо знать. В действительности они налагаются модулем pickle, а не shelve, поэтому следуйте им, даже когда будете сохранять экземпляры классов непосредственно с помощью модуля pickle.

Классы должны быть доступны для импортирования

Объект класса Pickler при сериализации объекта экземпляра сохраняет только атрибуты экземпляра и затем заново импортирует класс, чтобы воссоздать экземпляр. По этой причине, когда объекты восстанавливаются из сериализованной формы, должна обеспечиваться возможность импортировать классы сохраненных объектов – они должны быть определены как невложенные, на верхнем уровне файла модуля, который должен находиться в пути поиска модулей в момент загрузки (например, в PYTHONPATH, или в файле *.pth*, или в текущем рабочем каталоге, или быть самим сценарием верхнего уровня).

Далее, при сериализации экземпляров классы должны ассоциироваться с действительным импортируемым модулем, а не со сценарием верхнего уровня (с именем модуля __main__), если только они не будут использоваться в сценарии верхнего уровня. Кроме того, нужно следить за тем, чтобы модули классов не перемещались после сохранения экземпляров. При восстановлении экземпляра Python должен иметь возможность найти модуль класса в пути поиска модулей по имени исходного модуля (включая префиксы путей в пакетах) и загрузить класс из этого модуля, используя первоначальное имя класса. Если модуль или класс будут перемещены или переименованы, интерпретатор не сможет отыскать класс.